UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Análisis e implementación de un estimador de movimiento sobre FPGA para compresión de vídeo a bordo de satélites

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos

Autor: D. Jezael Pérez Herrera

Tutores: D. Roberto Sarmiento Rodríguez

D. Yúbal Barrios Alfaro

Fecha: Noviembre 2020

UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Análisis e implementación de un estimador de movimiento sobre FPGA para compresión de vídeo a bordo de satélites

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario/a**

Fdo.: Fdo.:

Fecha: Noviembre 2020

Resumen

Dentro de la codificación de vídeo, la etapa de estimación de movimiento se presenta como un punto crítico, debido al gran volumen de datos que se maneja. Este concepto se basa en la redundancia de información que se encuentra en una secuencia de vídeo, donde la mayor parte de los datos se encuentran desplazados entre imágenes consecutivas. Debido al creciente interés en la integración de sensores de vídeo a bordo de satélites, resulta necesario la implementación de un sistema de codificación de vídeo, reduciendo la cantidad de información enviada a la estación terrena y dotando al sistema a bordo de cierta autonomía. Para ello, las FPGAs se presentan como la mejor opción debido a su reprogramabilidad, eficiencia energética y robustez ante la radiación presente en el espacio (únicamente en el caso de que se encuentren calificadas para su uso en entornos espaciales).

En este Trabajo Fin de Grado se realizará la implementación sobre FPGA de un algoritmo de estimación de movimiento basado en Block Matching, como etapa de un sistema de codificación de vídeo a bordo de satélites. Para llegar a este fin, se empleará la metodología HLS, la cual nos permite partir de una descripción funcional del algoritmo en lenguaje de alto nivel para ser compilada directamente en una implementación RTL, obteniéndose de esta forma los resultados requeridos con mayor facilidad y menor tiempo de desarrollo.

Abstract

Within video coding, the motion estimation stage is a critical point, due to the large volume of data that is handled. This concept is based on the redundancy of information found in a video sequence, where most of the data is just shifted between consecutive images. Because of the growing interest in the integration of video sensors on-board satellites, it is necessary to implement a video coding system for reducing the quantity of information sent to the ground station and providing certain autonomy to the on-board system. FPGAs are presented as the best option for this purpose, thanks to their reprogrammability capabilities, energy efficiency and robustness to radiation present in space (only in the space-qualified devices).

In this work, the implementation of a motion estimation algorithm based on Block Matching will be carried out on FPGA, as a stage of a video coding system on-board satellites. The HLS methodology will be used for achieving this goal, which allows to start from a functional description of the algorithm in a high-level language to be transformed directly into an RTL implementation, thus obtaining the required results reducing the development time.

Agradecimientos

En primer lugar, me gustaría agradecer a mis tutores Roberto Sarmiento Rodríguez y Yubal Barrios Alfaro por su tiempo y dedicación para concluir satisfactoriamente este trabajo, así como por el trato recibido.

Luego, querría honrar el esfuerzo realizado por mis padres por todo el apoyo y recursos que han aportado a lo largo de mi trayectoria estudiantil, desde la guardería hasta la universidad.

Por último, no puedo finalizar este apartado sin hacer mención a todos los recuerdos y amistades que me llevo de esta etapa universitaria, gracias, sin ustedes no habría sido lo mismo.

Índice general

1 Introducción **1**

1.1 Antecedentes 1

1.2 Objetivos 6

1.3 Estructura del documento 6

2 Codificación de vídeo **9**

2.1 Conceptos generales sobre codificación de vídeo 9

2.1.1 Algoritmos de codificación de vídeo 10

2.2 Estimación de movimiento 15

2.2.1 Estimación local vs. estimación global 15

2.2.2 Algoritmos de estimación de movimiento 16

2.2.3 Algoritmos de búsqueda para block-matching 19

2.3 Conclusiones 22

3 Dispositivo hardware y entorno software **23**

3.1 *Hardware* propuesto 23

3.1.1 Dispositivo XC7Z020 24

3.1.2 Mecanismos de comunicación PS-PL 25

3.1.3 Placa de prototipado Pynq-Z1 26

3.2 Entorno Xilinx Vivado 27

3.2.1 Flujo de diseño 28

3.2.2 HLS y herramientas para síntesis de alto nivel 29

3.2.3 Prototipado hardware 32

3.2.4 Desarrollo software mediante Xilinx SDK 34

3.3 Conclusiones 34

4 Diseño del bloque IP **37**

4.1 Diseño del algoritmo de estimación de movimiento 37

4.1.1 Descripción del algoritmo en alto nivel 38

4.1.2 Depurado y verificación mediante simulación de alto nivel . . 44

4.1.3 Optimización mediante partición hardware/software 45

4.2 Síntesis de alto nivel y análisis de resultados 48

4.2.1 Análisis del diseño con extensión de bordes en el bloque IP . . 48

4.2.2 Modelado de interfaces 50

4.2.3 Análisis del diseño sin directivas de optimización 52
4.2.4 Análisis del diseño final aplicando directivas de optimización 53
4.3 Verificación RTL y encapsulado IP 54
4.4 Conclusiones 55

5 Integración y verificación sobre la plataforma Pynq 57
5.1 Arquitectura y estructura de la plataforma 57
5.1.1 Bloques IP utilizados 60
5.1.2 Comunicación y conexionado entre bloques 61
5.2 Diseño del software empotrado 63
5.2.1 Funcionalidad general 63
5.2.2 Inicialización y configuración de los bloques *DMA, Switch y blockMatchingNTS* 64
5.2.3 Desarrollo de la aplicación *software* 65
5.3 Validación del sistema 66
5.3.1 Banco de pruebas 67
5.3.2 Proceso de validación 68
5.3.3 Resultados obtenidos 71
5.4 Conclusiones 75

6 Conclusiones y trabajos futuros 77
6.1 Conclusiones del proyecto 77
6.2 Trabajos futuros 79

7 Presupuesto 81
7.1 Recursos humanos 81
7.2 Recursos materiales 81
7.2.1 Recursos *software* 82
7.2.2 Recursos *hardware* 82
7.3 Redacción del documento 82
7.4 Derechos de visado del COITT 83
7.5 Presupuesto final del proyecto 84

Bibliografía 85

Índice de figuras

1.1 (a) Cámara CCD PC67XC-2; (b) Cámara modular MicroCam TTL basada en sensor CMOS OmniVision 1
1.2 Estructura del ICUBE-1 2
1.3 Cámara modular C3188A 2
1.4 Plataforma SMART [4] 3
1.5 Relación entre flexibilidad y eficiencia entre opciones de implementación 5

2.1 Formatos de submuestreo más comunes 11
2.2 Posibles particiones y subparticiones de un macrobloque en H.264 . . . 12
2.3 Diagrama de bloques de CTU y CU en HEVC 14
2.4 Etapas de un codificador de vídeo con compensación de movimiento . 15
2.5 Ejemplo de vídeo con movimiento local. (a) Frame n. (b) Frame n-1. . 16
2.6 Tipos de recursiones: (a) horizontal; (b) vertical; (c) temporal. 18
2.7 Algoritmo block matching 19
2.8 Patrones de búsqueda del DS: (a) LDSP, (b) SDSP 20
2.9 Ejemplo de posible ruta utilizando el algoritmo DS [25] 20
2.10 Ejemplo de posible ruta utilizando el algoritmo NTSS 21

3.1 Arquitectura SoC empleada en Zynq [28] 24
3.2 Interfaces de comunicación entre el PL y la memoria del PS[27] 26
3.3 Configuración de los piness en la Pynq Z1 [28] 27
3.4 Placa de prototipado Pynq-Z1 27
3.5 Flujo de diseño de alto nivel en *Vivado Design Suite* [29] 28
3.6 Flujo de diseño IP [30] 29
3.7 Flujo de diseño Vivado HLS [32] 31
3.8 Interfaz de usuario Vivado HLS 31
3.9 Análisis temporal del diseño obtenido en la sintesis lógica 32
3.10 Perspectiva del diseño implementado en Vivado IDE 33
3.11 Visualización de señales mediante Hardware Manager 33
3.12 Creación de un nuevo proyecto en SDK 35

4.1 Proceso seguido en el diseño 38
4.2 Ejemplo ilustrativo sobre la creación de píxeles externos al frame . . . 40
4.3 Representación de los distintos movimientos considerados en la primera etapa del algoritmo NTSS 41

4.4 Ejemplo de vectores de movimiento considerados en la segunda etapa del algoritmo NTSS . . . 43
4.5 Ejemplo de vectores de movimiento considerados en la tercera etapa del algoritmo NTSS . . . 43
4.6 Ejemplo de archivo obtenido de la estimación de movimiento de una secuencia de vídeo . . . 45
4.7 Diagrama del bloque IP diseñado . . . 47
4.8 Diagrama del bloque IP modificado . . . 48
4.9 Latencia obtenida en la síntesis de la primera versión . . . 49
4.10 Recursos consumidos en la síntesis de la primera versión . . . 49
4.11 Consumo de memoria BRAM en la síntesis de la primera versión . . . 49
4.12 Diagrama de E/S del bloque IP . . . 51
4.13 Resultados de latencia de la síntesis de alto nivel . . . 52
4.14 Consumo de recursos del diseño sin directivas de optimización . . . 52
4.15 Latencia final de nuestro diseño . . . 54
4.16 Consumo de recursos del diseño final . . . 54
4.17 Resultados de latencia de la cosimulación . . . 55

5.1 Arquitectura del sistema empotrado . . . 58
5.2 Diagrama del bloque IP AXI DMA . . . 58
5.3 Estructura del bloque IP *blockMatchingNTS_0* . . . 61
5.4 Diagrama de bloques de la plataforma . . . 62
5.5 Bloque IP *System ILA* . . . 68
5.6 Configuración del modo *debug* en Xilinx SDK . . . 69
5.7 Forma de onda de la salida del DMA durante la transmisión del primer macrobloque: (a) Transmisión completa (b) Zoom de la señal . . . 70
5.8 Forma de onda a la entrada *currentBlock* del bloque *blockMatchingNTS* durante la transmisión del macrobloque actual . . . 70
5.9 *Slack* obtenido con un reloj de 100MHz . . . 71
5.10 *Slack* obtenido con un reloj de 111MHz . . . 72
5.11 Forma de onda de las dos entradas del bloque IP . . . 72
5.12 Forma de onda de la salida del bloque IP, el vector de movimiento . . . 73
5.13 Tiempo de ejecución de la estimación de movimiento, desde la recepción del primer macrobloque hasta el envío del vector de movimiento . . . 73
5.14 Potencia consumida por el dispositivo . . . 75

Índice de cuadros

2.1 Características de los estándares de compresión de vídeo MPEG-2, H.264 y H.265 11

5.1 Consumo de recursos del bloque IP y la plataforma 74

7.1 Trabajo tarifado por tiempo empleado 81
7.2 Precios y costes de los recursos *hardware* 82
7.3 Suma de las amortizaciones y el tarifado por tiempo empleado 83
7.4 Coste final del proyecto 84

Siglas

ACP Accelerator coherency port. 25

APU Application Processor Unit. 25

ARM Advanced RISC Machines. 5, 25, 46, 55, 57, 78

AXI Advanced eXtensible Interface. 25, 50, 58–61, 63, 65

BRAM Block RAM. 52, 53

BSP Board Support Package. 34, 63, 64

CABAC Context-Adaptive Binary Arithmetic Coding. 13

CAVLC Context-Adaptive Variable Length Coding. 13

CCD Charge-Coupled Device. 1, 2

CMOS Complementary Metal Oxide Semiconductor. 1, 2

COITT Colegio Oficial de Ingenieros Técnicos de Telecomunicación. 83

CPU Central Processing Unit. 25, 60

DMA Direct Memory Access. 51, 58–60, 63–66

DS Diamond Search. 19, 20

FF Flip-Flop. 52–54

FPGA Field-Programmable Gate Array. 5, 28, 32, 34, 38, 52, 53, 55–57, 67, 75, 78

HDL Hardware Description Language. 47

HDMI High-Definition Multimedia Interface. 26

HLS High Level Synthesis. 23, 28, 30, 31, 35, 37, 54, 55, 57, 64

ICUBE (Institute of Space Technology Pakistan cubeSat program. 2

IDE Integrated Design Environment. 23, 29–33, 57, 60

IEC International Electrotechnical Commission. 3

ILA Integrated Logic Analyzer. 32, 33

IOP I/O Peripherals. 25

ISO International Organization for Standardization. 3

ITU International Telecommunication Union. 3

JTAG Joint Test Action Group. 26

LDSP Large Diamond Search Pattern. 19, 20

LUT Look-up Table. 49, 52, 53

MB macrobloque. 18, 39, 41, 42, 46, 47

MPEG Moving Picture Experts Group. 3, 11

NASA National Aeronautics and Space Administration. 2

NTSC National Television System Committee. 10

NTSS New Three Step Search. 19, 21, 37, 38, 40, 41, 43, 44, 47, 50, 53, 55, 58, 59, 61, 66, 74, 75

OCM On-Chip Memory. 25

PAL Phase Alternating Line. 10

PDM Pulse Density Modulation. 26

PL Programmable Logic. 25, 26, 34

PS Process System. 24–26, 34, 55, 59–61, 63, 78

RAM Random Access Memory. 63

RTL Register Transfer Level. 23, 28, 30, 31, 54

SAD Sum of Absolute Difference. 20, 21, 39, 41, 42

SDK Software Development Kit. 23, 34, 57, 63

SDSP Small Diamond Search Pattern. 19, 20

SMART Small Rocket/Spacecraft Technology. 2

SoC System on Chip. 23, 25, 26, 34, 60

SPI Serial Peripheral Interface. 26

UART Universal Asynchronous Receiver-Transmitter. 26

VIO Virtual Input/Output. 32

1 Introducción

Este primer capítulo describe las principales razones que han llevado a la realización de este proyecto y las necesidades que este pretende cubrir, lo que constituyen los objetivos a alcanzar. Finalmente se explica la estructura que del documento.

1.1 Antecedentes

A lo largo de las últimas décadas, la innovación en la codificación de vídeo ha recibido avances significativos que han provocado la aparición de innumerables aplicaciones. El uso de sensores de vídeo a bordo de satélites para aplicaciones de observación de la Tierra ha generado mucho interés en la última década. No obstante, los tiempos de transmisión, la capacidad computacional del *hardware* y el uso de memoria suponen las principales limitaciones de esta aplicación. Por otro lado, se puede destacar la existencia de un estándar de diseño de nanosatélites, **CubeSat** [1], cuyo desarrollo y creación se lleva a cabo tanto en empresas del sector (generalmente, para formar constelaciones) como en entornos académicos. En este tipo de satélites es imposible incluir sensores de alta resolución de vídeo. La simple obtención de imágenes, su almacenamiento y transmisión a tierra mediante sensores de bajo consumo ya es todo un reto. Para ello, en el estado del arte se pueden distinguir dos tipos de sensores: los CCD y los CMOS. Algunas opciones comerciales de este tipo de sensores se pueden encontrar en [2], como las ilustradas en la figura 1.1.

(a) (b)

Fig. 1.1: (a) Cámara CCD PC67XC-2; (b) Cámara modular MicroCam TTL basada en sensor CMOS OmniVision

No existe una regla estricta sobre qué tecnología de adquisición es la más adecuada para misiones espaciales; sin embargo, se ha de obtener un equilibrio entre la

fiabilidad de los datos obtenidos, la resolución y la velocidad a la que se toman las imágenes. Pese a que los sensores CCD tienen una mayor velocidad en la obtención de los datos, estos son más propensos a errores. Además, consumen más energía y es una tecnología cuyo progreso ha llegado a un límite, mientras la tecnología CMOS sigue aún en evolución. Estas son algunas de las razones por las se ha optado por la tecnología CMOS en la mayoría de satélites CubeSat, como es el caso del ICUBE-1 [3], el primer picosatélite del programa espacial ICUBE (*(Institute of Space Technology Pakistan cubeSat program*) cuya estructura se puede apreciar en la figura 1.2. La cámara utilizada en este satélite se trata de un módulo CMOS de baja resolución (664x492 pixeles de tamaño 7.6 μm) y se encuentra ilustrada en la figura 1.3.

Fig. 1.2: Estructura del ICUBE-1

Fig. 1.3: Cámara modular C3188A

Aunque no directamente relacionado, cabe mencionar también una misión espacial desarrollada directamente por la NASA: la plataforma microsatélite SMART (*Small Rocket/Spacecraft Technology*), mostrada en la figura 1.4. En ella se pueden observar hasta tres cámaras: por un lado un sistema *RocketCam* de *Ecliptic Enterprises*, el cual puede poseer a su vez entre 1 y 8 sensores de vídeo con una resolución de 752x582 (PAL) que utiliza para asegurar la confiabilidad del vuelo del cohete; y por otro lado, dos cámaras de color CCD GigE (*Gigabit Ethernet*) de grado industrial, que se

utilizan para probar y validar las interfaces de alta velocidad mediante las que van conectadas al *SpaceCube*, la FPGA instalada en esta plataforma.

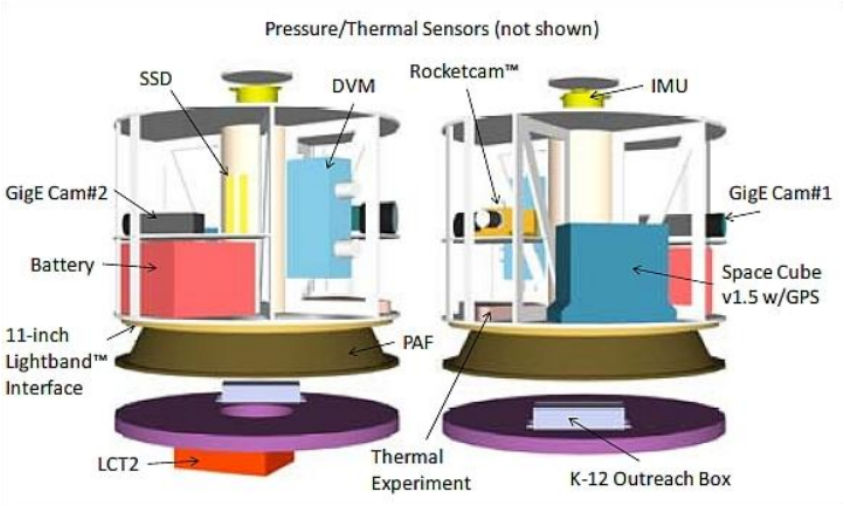

Fig. 1.4: Plataforma SMART [4]

En el entorno de aplicaciones comerciales, es de destacar que el desarrollo dentro de la codificación de vídeo y la tecnología embarcada en satélites ha desembocado en la generación de vídeo en *streaming* desde el espacio por parte de algunas compañías. En 2013, la empresa *Skybox* (posteriormente renombrada Terra Bella y comprada por Google y luego por Planet Labs [5]) publicó las primeras imágenes de su satélite SkySat-1 [6], el primer satélite comercial en producir vídeos de 90 segundos de duración a 30 *frames* por segundo de la Tierra en alta definición (HD) desde el espacio.

En este sentido, resulta necesario mencionar también los avances obtenidos por la *startup* británica *Earth-i* dentro del sector a partir de su constelación de satélites *VividX2* [7]. Estos pequeños satélites están equipados con cámaras a color UHD capaces de capturar hasta 50 *frames* por segundo, siendo los primeros del mundo en proporcionar vídeos a color desde el espacio. Así, produciendo vídeos de hasta 2 minutos de duración se pretenden cubrir numerosas aplicaciones, gracias a que durante ese tiempo se recopila toda la información posible del objetivo que se esté sobrevolando desde varios ángulos diferentes, permitiendo de esta forma la construcción de modelos 3D que resultan más detallados que las imágenes estereoscópicas que se obtienen desde un mismo satélite.

Para uso comercial en tierra, la ITU e ISO/IEC han publicado diversos algoritmos de compresión de vídeo que han cubierto las necesidades que han surgido con el paso de los años [8]. Inicialmente, MPEG 1 y 2 [9, pag 330], que también son funcionales en audio, cubrían aplicaciones de CD-ROM y comunicaciones. Sin embargo, en aplicaciones tan importantes como *Internet*, estos métodos de compresión eran

insuficientes, debido al incremento del ancho de banda y de la resolución del vídeo. De esta forma, comenzaron a desarrollarse nuevos estándares con los que poder ir cubriendo necesidades como la reproducción de multimedia por Internet, videoconferencias y transmisiones de vídeo de alta velocidad, además de dar soporte a las nuevas resoluciones que se han ido estandarizando. A dia de hoy, están naciendo nuevas aplicaciones como la reproducción de vídeo en 8K y la realidad virtual, donde la complejidad algorítmica de los estándares para gestionar este mayor volumen de datos seguirá aumentando para proporcionar una mejor tasa de bit.

Dentro de la codificación de vídeo, la estimación de movimiento es una de las etapas de mayor complejidad debido al volumen de datos, en bits, que se maneja y, por ende, se trata de un punto crítico. Este concepto se basa en la redundancia de información que se encuentra en una secuencia de vídeo, donde la mayor parte de los datos representados en cada imagen son idénticos a los de la imagen anterior, pero con un determinado desplazamiento.

En este proyecto, donde se implementará un algoritmo de estimación de movimiento como etapa de un codificador de vídeo a bordo de satélites, se deberá tener en cuenta la complejidad que presente el algoritmo seleccionado de cara a ser implementado en *hardware* apto para misiones espaciales, ya que esto se verá reflejado en el tiempo de ejecución y en el consumo de recursos del dispositivo utilizado.

Para su implementación, se podría escoger una CPU, GPU o ASIC, optándose finalmente por utilizar una FPGA. En términos de flexibilidad, las CPU y las GPU se presentan como una mejor opción, ya que se pueden utilizar para realizar gran cantidad de tareas y aplicaciones. No obstante, esta flexibilidad se ve reflejada en una menor eficiencia energética en la ejecución de una determinada tarea, ya que el dispositivo en sí no está diseñado y optimizado especificamente para ella. En este contexto, podemos destacar los ASICs y las FPGAs, debido a que se diseñan explícitamente para desarrollar una tarea concreta y, por tanto, todas y cada una de las optimizaciones que se pueden realizar se harán teniendo en cuenta la aplicación a implementar. Esto no quita que las FPGAs puedan realizar también varias tareas en paralelo, siempre y cuando no haya dependencia de datos entre las mismas y haya disponible los recursos lógicos necesarios para implementarlas.

No obstante, hay una característica concreta en la que las FPGAs se imponen a los ASICs: la reprogramabilidad. La gran ventaja que posee una FPGA sobre una ASIC es que puede ser reprogramada internamente para hacer otra tarea completamente diferente a la que estaba haciendo de manera inicial. Esto significa que los elementos lógicos del dispositivo pueden variar su conectividad o, incluso, usarse más o menos cantidad de ellos para añadir funcionalidad extra y/o mejorar las prestaciones del diseño, permitiendo que el dispositivo realice su tarea de una manera mucho más

eficiente en términos computacionales. Además, en caso de que se quiera cambiar dicha funcionalidad, no se estaría incurriendo en un coste adicional, situación que sí se da con los ASICs, no modificables y que requieren comenzar un nuevo proceso de diseño y fabricación en caso de buscar un cambio en las especificaciones del mismo.

Fig. 1.5: Relación entre flexibilidad y eficiencia entre opciones de implementación

Además, teniendo en cuenta que este proyecto está orientado a aplicaciones en la órbita terrestre, otro aspecto a considerar para la elección del dispositivo es la compatibilidad con las condiciones presentes en este entorno espacial; es decir, que el dispositivo electrónico presente cierta robustez a los efectos provocados por la radiación, la cual puede producir fallos temporales en ciertas partes del diseño, corromper los datos o incluso provocar un mal funcionamiento persistente si el dispositivo no es reconfigurado [10]. Por ejemplo, aunque se contemplara la posibilidad de utilizar una GPU para desarrollar este proyecto, sería algo inviable ya que actualmente no hay GPUs tolerantes a la radiación, aunque existen diversos estudios en marcha para su calificación [11][12].

Por otro lado, ya se han utilizado FPGAs para misiones espaciales en multitud de ocasiones desde hace muchos años, tal y como se puede apreciar en un informe realizado por la misma Agencia Espacial Europea [13]. Además, teniendo en cuenta el tema que se aborda en este trabajo, se han encontrado algunas implementaciones reales de codificadores de vídeo en FPGA [14], donde se han obtenido resultados prometedores que demuestran el potencial que presentan estos dispositivos para la implementación de las distintas etapas de las que consta un codificador de video; por ejemplo, la obtención de un *speed-up* en la codificación de video para una resolución de 640x480 píxeles del orden de $90\times$ respecto a la obtenida por la plataforma ARM. Asímismo, ya se ha experimentado concretamente con implementaciones de algoritmos de estimación de movimiento sobre FPGA [15, 16] casos muy semejantes a nuestro proyecto donde se demuestra una vez más las ventajas que presenta contar con estas arquitecturas *hardware* de alto rendimiento y bajo coste.

En lo que respecta a las metodologías de diseño para FPGAs, desde hace varias décadas se han adoptado técnicas de diseño RTL *(Register Transfer Level)*, en las que los lenguajes de descripción de *hardware* (como VHDL) y la síntesis lógica han contribuido a reducir la dificultad y los tiempos de problemas muy complejos.

Sin embargo, dada la creciente complejidad del diseño de los sistemas en chip, ha surgido la necesidad de elevar el nivel de abstracción más allá del nivel de transferencia de registros. Por ello, ha surgido la metodología HLS (High-Level Synthesis), la cual desempeña un papel fundamental en la automatización del diseño ESL *(Electronic System Level)*. Este proceso de diseño automatizado permite, a partir de una especificación de alto nivel del problema mediante una descripción algorítmica en lenguaje de alto nivel (C/C++ o, incluso, usando Python), la creación de una arquitectura a medida del procesador en un lenguaje de descripción hardware (HDL), reduciendo de esta forma tanto el tiempo de diseño como el Time-To-Market del producto final [17].

1.2 Objetivos

Este proyecto tiene como objetivo principal la implementación sobre FPGA de un algoritmo de estimación de movimiento basado en *Block Matching*, como etapa de un sistema de codificación de vídeo a bordo de satélites. Para lograr este fin, se han fijado los siguientes objetivos específicos:

- Revisión del estado del arte, elección y modelado del algoritmo de estimación de movimiento que mejor se adecúe a las necesidades de nuestra aplicación.
- Adaptación del algoritmo seleccionado a una descripción *hardware-friendly*, con el fin de implementarlo siguiendo la metodología HLS.
- Síntesis, optimización e implementación sobre una FPGA de Xilinx mediante *Vivado Design Suite*.
- Validación del sistema con secuencia de vídeo reales, y obtención y análisis de resultados.

1.3 Estructura del documento

En este primer capítulo se describen los antecedentes e ideas fundamentales para dar al lector una comprensión general del contexto en el que se ubica el trabajo realizado, además de exponer los objetivos a alcanzar y cómo se estructura el documento. En el capítulo 2 se exponen los principales estándares de codificación de vídeo que se han desarrollado, para posteriormente explicar concretamente la etapa de estimación de movimiento, así como las principales técnicas para su aplicación.

Por otro lado, el tercer y cuarto capítulo se centran en la plataforma y entorno de diseño que se utilizarán para la implementación del algoritmo escogido. Luego, en el capítulo 5 tiene lugar la definición de la solución propuesta que se ha considerado como óptima para la aplicación objetivo.

En el capítulo 6 se definen las tareas realizadas durante el proceso de diseño y síntesis de alto nivel del bloque IP desarrollado para, posteriormente en el capítulo 7, explicar como ha sido el proceso de integración en la plataforma, explicando el resto de módulos empleados y exponiendo los pasos seguidos para asegurar el propósito y el correcto funcionamiento del sistema final.

Finalmente, la validación del sistema es explicada en el capítulo 8, junto con la recogida de los resultados para su análisis y formulación de las conclusiones finales del trabajo, que se encontrarán en el capítulo 9.

2 Codificación de vídeo

En este capítulo se tratan los principios básicos de la codificación de vídeo, etapa clave en el procesamiento de este tipo de contenido multimedia, reduciendo su tasa de bits, sin incurrir en pérdidas perceptibles por el ojo humano. El principal problema que presenta es la cantidad de bits que se manejan, lo que aumenta la complejidad de los algoritmos.

2.1 Conceptos generales sobre codificación de vídeo

En un sistema de compresión de vídeo es necesario que exista, además de información real, información redundante. Esta información redundante se podrá eliminar en el proceso de compresión, codificando solamente la información útil, para posteriormente reconstruirla en el proceso de descompresión a través de la información útil de la secuencia. Por tanto, la codificación es de vital importancia en la cadena de procesamiento de una secuencia de vídeo. Esta redundancia puede ser de tres tipos: espacial, temporal y estadística.

Por un lado, la **redundancia espacial** tiene lugar dentro de cada fotograma o *frame*. En una imagen, se suelen encontrar diversos objetos con superficies uniformes, donde la información entre píxeles contiguos es casi idéntica. Por tanto, este hecho permite transmitir o almacenar un píxel representativo de un conjunto de píxeles adyacentes que tengan características similares y las diferencias que tengan respecto a este píxel representado. De esta forma, se puede obtener una codificación con menor número de bits, lo que supone una reducción del tamaño de la secuencia transmitida/almacenada. Un ejemplo de codificación aprovechando la redundancia espacial es *Run Length Encoding*, el cual, en lugar de codificar cada dígito de una cadena binaria con varios valores repetidos, codifica el dígito que se repite y número de veces que se repite [18].

Por otro lado, **redundancia temporal** hace referencia a la relación de píxeles homólogos entre *frames* consecutivos. En una secuencia de vídeo de, por ejemplo, 30 fotogramas por segundo, la sensación de movimiento se produce gracias a que las diferencias entre imágenes consecutivas son muy escasas. Por ello, se encuen-

tra un importante exceso de información que puede ser eliminado en la etapa de codificación para obtener una mejor compresión.

Una técnica que se puede utilizar para eliminar esta redundancia temporal, es la codificación diferencial de pulsos modulados (DPCM), la cual codifica el valor de la diferencia de una muestra respecto a la anterior, en lugar de codificar el valor absoluto de estas [19].

Por último, la **redundancia estadística** se puede encontrar en aplicaciones donde determinados valores tienden a repetirse más que otros. Una de las técnicas más utilizadas para su detección es la codificación de longitud variable (VLC, *Variable Length Code*). Este método consiste en analizar estadísticamente los datos que se reciben, para posteriormente realizar una codificación óptima a cada valor. El principio básico de esta codificación es asignar códigos más largos en aquellas cadenas de bits que se repiten con menor frecuencia y códigos más cortos en aquellos bits que se repiten con mayor frecuencia.

Todas estas técnicas se han visto utilizadas a lo largo de las últimas décadas, formando parte de estándares de codificación de vídeo como los que veremos a continuación.

2.1.1 Algoritmos de codificación de vídeo

El diseño básico de los principales estándares de codificación de vídeo desde **H.261** (en 1990) sigue un enfoque de codificación basado en bloques, donde las diversas imágenes que componen una secuencia de vídeo son particionadas en agrupaciones de píxeles. Cada bloque que compone una imagen se encuentra codificado completamente dentro de la imagen (codificación *intra*), sin relacionarse con otros frames de la secuencia de vídeo, o se predice temporalmente (codificación *inter*), realizando una predicción sobre un bloque desplazado de una imagen ya codificacada. Esta última es también conocida como predicción mediante **compensación de movimiento** y representa un concepto clave para utilizar la gran cantidad de redundancia temporal presente en las secuencias de vídeo. Además, teniendo en cuenta las tres componentes que componen el color fuente de cada vídeo en formato YUV, luminancia (Y) y las dos crominancias (Cb y Cr), se considerará en todo momento un formato de submuestreo 4:2:0. Este formato hace referencia a la distribución de las distintas componentes en cada frame, indicando cada número la resolución utilizada para cada una de las componentes [20]. Entonces, en el caso del formato de submuestreo 4:2:0, se utiliza la máxima resolución de luminancia debido a que el primer valor es 4, en referencia a que la frecuencia de muestreo era aproximadamente cuatro veces la frecuencia subportadora de color de los estándares PAL o NTSC. Por otro lado, con

el segundo y tercer valor se indica que los componentes de crominancia Cb y Cr son muestreados con un factor de 2 horizontal y 2 vertical. En la figura 2.1 se pueden observar los formatos de submuestreo más comunes que facilitan el entendimiento de este concepto.

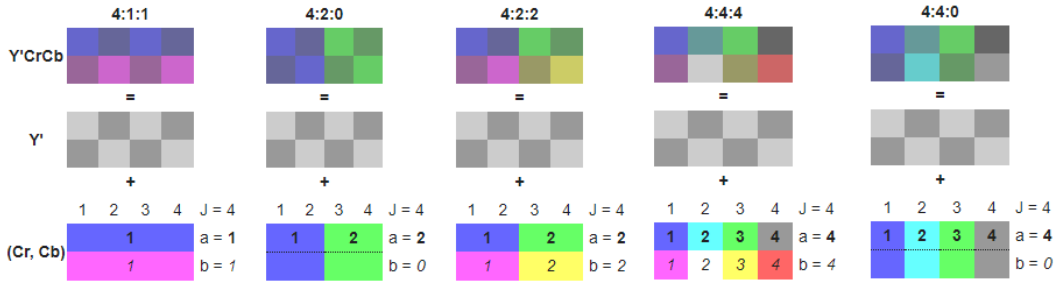

Fig. 2.1: Formatos de submuestreo más comunes

Aunque los principales estándares siguen el mismo diseño básico, difieren en varios aspectos, lo que finalmente resulta en una eficiencia de codificación significativamente mejorada de una generación de estándares a la siguiente. En la tabla 2.1 se pueden ver reflejados los principales algoritmos de compresión de vídeo con sus respectivas diferencias.

Tab. 2.1: Características de los estándares de compresión de vídeo MPEG-2, H.264 y H.265

	MPEG-2	**H.264**	**H.265**
Estructura de codificación	16x16 (Macrobloque)	16x16 (Macrobloque)	*8x8 - 64x64(Coding Tree Block)*
Tamaño de bloques	8x8	16x16 – 4x4	64x64 – 16x16
Predicción INTRA	No	Dominio espacial mediante 2-8 predictores direccionales según el tamaño de los bloques	Dominio espacial mediante 33 predictores direccionales para todos los posibles tamaños
Transformada	DCT de tamaño 8x8	DCT entera de tamaños 8x8, 4x4	DCT/DST de tamaños 32x32 – 4x4
Cuantificación	Escalar	Escalar	Escalar
Codificación de la entropía	VLC	CAVLC o CABAC	CABAC
Exactitud de las muestras de la imagen	½ muestra	¼ muestra	¼ muestra
Cuadros de referencia	1 cuadro	Múltiples cuadros	Múltiples cuadros
Modo de predicción bidireccional	Adelante / atrás	Adelante / atrás Atrás / atrás	Adelante / atrás Atrás / atrás
Predicción con peso	No	Sí	Sí
Filtro de desbloqueo	No	Sí	Sí
Tipo de cuadros	I, P, B	I, P, B, SI, SP	I, P, B
Perfiles	5 perfiles	7 perfiles	3 perfiles
Velocidad de transmisión	2 – 15 Mbps	64 kbps – 150Mbps	64 kbps – Gbps
Complejidad del codificador	Mediana	Alta	Alta

En primer lugar, el estándar **MPEG-2/H.262** fue diseñado como proyecto conjunto entre ITU-T VCEG e ISO/IEC MPEG; por ello, ambos estándares son idénticos. Al igual que sus predecesores H.261 y MPEG-1 Video, cada imagen de la secuencia de vídeo es dividada en macrobloques (MBs) que consistirán en un bloque de 16x16 pixeles de luminancia y dos bloques asociados de crominancia de 8x8 (teniendo

en cuenta un formato de submuestreo de 4:2:0). Este estándar define tres tipos de imágenes: *I*, *P* y *B*.

- *I-frames*: imágenes codificadas de manera totalmente independiente, sin referencia a ninguna otra.

- *P-frames*: imágenes codificadas a partir de las diferencias entre la imagen predicha y una de referencia, que puede ser otra imagen I o P.

- *B-frames*: imágenes codificadas de forma similar a las imágenes P; sin embargo, las predicciones son bidireccionales, es decir, pueden realizarse a partir de frames pasados o futuros, tanto I como P.

H.264 / MPEG-4 AVC es el segundo estándar de codificación de vídeo desarrollado conjuntamente por ITU-T VCEG e ISO/IEC MPEG y, aunque sigue utilizando el concepto de macrobloques de 16x16 píxeles, contiene muchas características adicionales que lo hacen más eficiente para una codificación con menor tasa de bits y menor *delay*. Mientras que el estándar MPEG-2 presentaba una precisión en el vector de movimiento de medio píxel, este algoritmo admite compensación de movimiento con vectores de movimiento con una precisión de un cuarto de muestra. Para la obtención de esta, se realiza una interpolación bilineal, tanto a las muestras enteras como a las medias muestras. Una de las diferencias más obvias respecto a los estándares más antiguos es su mayor flexibilidad para la codificación inter. En el caso de la predicción con compensación de movimiento, cada MB puede dividirse en bloques cuadrados o rectángulares con tamaños que varían de 4x4 a 16x16 muestras de luminancia, como se puede ver en la figura [2.2].

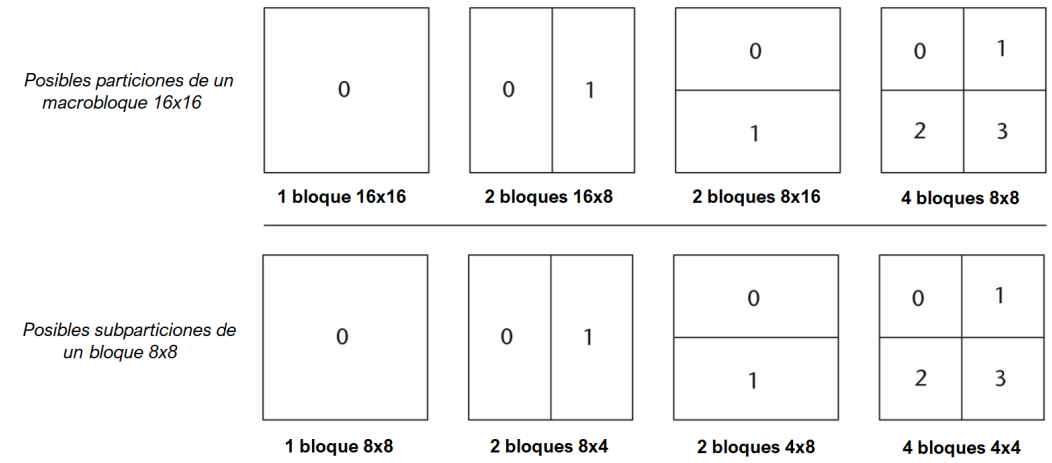

Fig. 2.2: Posibles particiones y subparticiones de un macrobloque en H.264

Además, el concepto de las imágenes B (bidireccionales) se generaliza y el tipo de codificación de las imágenes se desacopla del orden de codificación y del uso como

imagen de referencia. En lugar de las imágenes I, P y B, el estándar especifica I, P y B como secciones o *slices*. Una imagen puede contener *slices* de diferentes tipos y usarse como referencia para la predicción de imágenes posteriores, independientemente del tipo de codificación de las secciones. Esta generalización permitió el uso de estructuras de predicción tales como imágenes B jerárquicas [21], que muestran una eficiencia de codificación mejorada respecto a los anteriores estándares de codificación.

AVC explota la redundancia espacial mucho mejor que en estándares como MPEG-2, utilizando predicción *intraframe*. Un macrobloque es codificado como un intra MB cuando la predicción temporal es imposible (primer frame del video) o ineficiente (cambios de escena). Mientras en estándares anteriores algunos de los coeficientes DCT pueden predecirse a partir de bloques intra contiguos, la predicción intra en AVC se realiza haciendo referencia a muestras vecinas de bloques decodificados previamente. La componente de luminancia de un macrobloque puede predecirse como un solo bloque de 16x16 o puede dividirse en bloques de 8x8 o 4x4, prediciéndose cada bloque por separado. Hay nueve modos de predicción para los tamaños de bloque 4x4 y 8x8, y cuatro modos de predicción para el bloque 16x16. En el primer caso, los modos definen distintas direcciones de predicción y en el segundo caso, el resultado de la predicción es el promedio de los píxeles vecinos.

Para la codificación de entropía (para reducir la redundancia estadística) en H.264 se especifican 2 métodos. El primero, conocido como **CAVLC** (*Context-Adaptive Variable Length Coding*), utiliza un único conjunto de palabras de código para todos los elementos de sintaxis, exceptuando los coeficientes de la transformada. El enfoque para codificar los coeficientes de la transformada es el mismo que en anteriores estándares, básicamente utilizando el concepto de codificación a nivel de ejecución. No obstante, en MPEG-2 se utilizaban tablas VLC fijas para representar cada elemento, mientras que en H.264 se consigue una mejor eficiencia cambiando entre tablas VLC, dependiendo de los valores de los elementos de sintaxis transmitidos previamente. El segundo método, conocido como **CABAC** (*Context-Adaptive Binary Arithmetic Coding*), mejora significativamente la eficiencia de codificación en relación con CAVLC. Las estadísticas de símbolos previamente codificados se utilizan para estimar probabilidades condicionales para símbolos binarios, que se transmiten mediante codificación aritmética. Las dependencias entre símbolos se explotan cambiando entre varios modelos de probabilidad estimada basados en símbolos previamente decodificados en bloques vecinos.

Por último, el estándar **H.265**, conocido también como HEVC (*High Efficiency Video Coding*), fue otro proyecto de estandarización conjunta entre ITU-T VCEG e ISO/IEC MPEG, que nació como sucesor del estándar H.264. Presenta una mejora en respecto a los estándares previos, logrando un avance en la calidad percibida y permitiendo

una mayor eficiencia en entornos de baja tasa binaria. Entre las diferentes innovaciones que presenta este estándar se puede destacar el uso de una nueva estructura de codificación que diverge de las anteriores unidades denominadas macrobloques, con dimensiones de 16x16. En HEVC una imagen se divide en **CTBs** (*Coding Tree Blocks*), cuyo tamaño puede ser elegido de acuerdo con la arquitectura utilizada y las necesidades de su aplicación. Un CTB de luminancia cubre un área rectangular de N × N muestras de la componente Y, mientras que los CTB de crominancia correspondientes cubren cada uno (N/2) × (N/2) muestras de cada una de las dos componentes de crominancia. El valor de N se indica dentro del flujo de bits y puede ser 16, 32 o 64. Los tres CTBs (luminancia más dos crominancias), junto con la sintaxis asociada, forman una **CTU** (*Coding Tree Unit*). La CTU es la unidad básica de procesamiento del estándar para especificar el proceso de decodificación. Los bloques especificados como CTBs se pueden dividir a su vez en múltiples bloques de codificación (CBs), cuyo tamaño puede variar desde el mismo que el CTB hasta un mínimo de 8x8 muestras de luminancia. De esta manera, el CB de luminancia y el de crominancia forman a su vez una unidad de codificación (CU). En la figura 2.3 se pueden entender mejor visualmente estos conceptos.

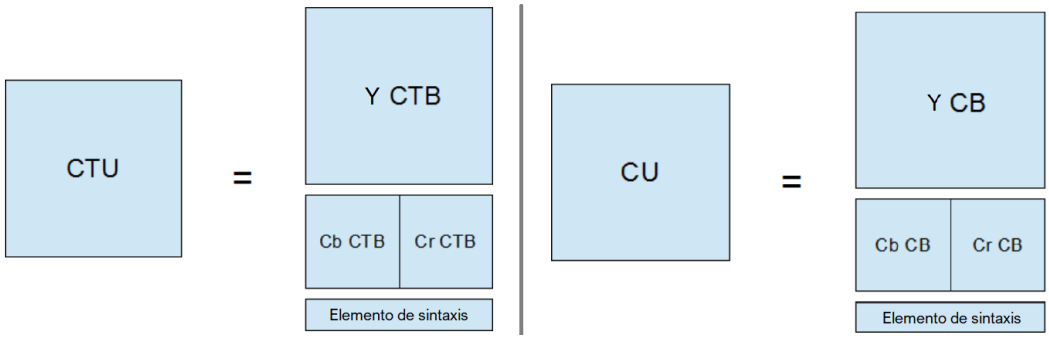

Fig. 2.3: Diagrama de bloques de CTU y CU en HEVC

Además, se define una nueva unidad de predicción **PU**, que puede tener dimensiones desde 32x32 hasta 4x4, lo que representa la mitad de la menor de las unidades de codificación. Estas unidades de predicción pueden aplicarse de modo *intra-frame* o *inter-frame*, al igual que en H.264. Para la predicción *intra-frame* aparece un nuevo método en el que se definen 33 predicciones direccionales para todos los posibles tamaños de PU. Para la predicción *inter-frame*, la estimación de movimiento se lleva a cabo con una precisión de 1/4 de píxel, incorporando además la **AMVP (*Advanced Motion Vector Prediction*)**. En este algoritmo, se logran señalizar los vectores de movimiento a partir del cálculo de los vectores de movimiento más probables obtenidos desde los bloques vecinos [22].

2.2 Estimación de movimiento

Dentro de las etapas de las que consta un codificador de vídeo y que se resumen en la figura 2.4, la estimación de movimiento resulta un punto crítico.

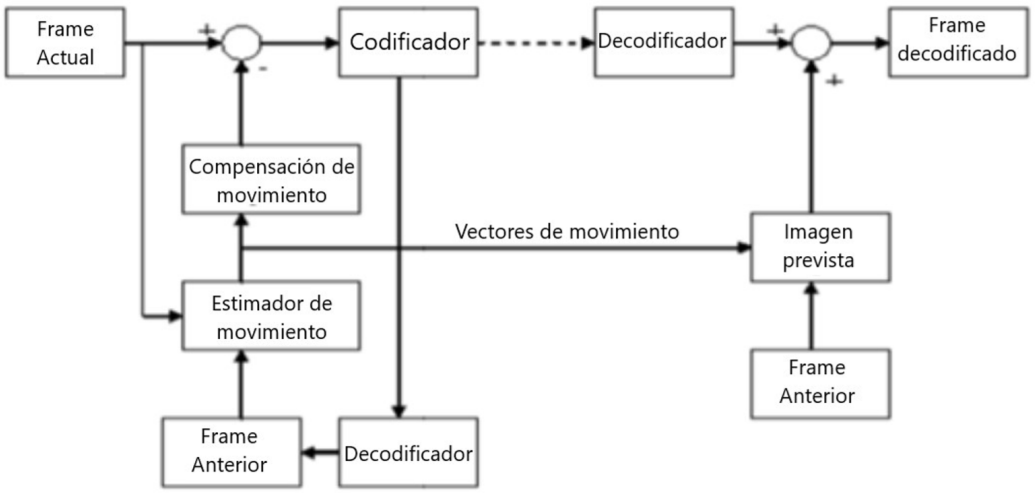

Fig. 2.4: Etapas de un codificador de vídeo con compensación de movimiento

La estimación de movimiento consiste en el análisis del movimiento de los objetos en una secuencia de imágenes para obtener vectores que representan el movimiento estimado; por tanto, la mayor parte de los datos representados en cada *frame* deberán ser equivalentes a los obtenidos en el *frame* anterior, pero situados en píxeles distintos.

2.2.1 Estimación local vs. estimación global

En una secuencia de video real, los distintos objetos que componen cada *frame*, tienen velocidades y movimientos muy diferentes. El algoritmo de codificación utilizado debe tener una tasa de bits y una complejidad muy elevadas para llevar a cabo la codificación del **movimiento local**, ocasionado por cada una de las partes que forman cada *frame*. Por ejemplo, en la figura 2.5 se puede apreciar cómo el niño situado a la derecha de la imagen está realizando un lanzamiento con la pelota, ejerciendo así un movimiento local y diferenciándose así del movimiento ejecutado por el resto de partes que conforman el vídeo. Por tanto, para poder estimar el movimiento de la pelota, se deberá analizar exhaustivamente cada uno de los píxeles que componen esa zona, comparando sus posiciones en cada *frame*. Sin embargo, hay situaciones en las que el movimiento de los píxeles se produce de una forma muy regular, es decir, que todos los píxeles realizan el mismo desplazamiento y, por tanto, se podría deducir el movimiento de los píxeles sin necesidad de hacer un análisis de todos y cada uno de ellos. Este fenómeno es llamado **movimiento global**

y se produce principalmente en situaciones donde el movimiento de la imagen es producido por el desplazamiento de la propia cámara, mientras los distintos objetos de la imagen no realizan ningún movimiento propio.

a b

Fig. 2.5: Ejemplo de vídeo con movimiento local. (a) Frame n. (b) Frame n-1.

Debido a que este proyecto está orientado a aplicaciones a bordo de satélites, el movimiento de los objetos entre *frames* será mínimo, siendo más determinante el movimiento global de los píxeles ocasionado por el movimiento de la cámara, en nuestro caso el desplazamiento del propio satélite en su órbita.

2.2.2 Algoritmos de estimación de movimiento

Para llevar a cabo esta predicción de movimiento entre *frames*, se han desarrollado diversas técnicas. Dos de los principales métodos para la estimación de movimiento son *pel-recursive* y *block-matching*, que serán explicados a continuación.

Pel-recursive

Junto con el algoritmo *block matching*, que se comentará más adelante, esta técnica se puede considerar, dentro de la comunidad de procesado de la señal, como uno de los principales enfoques para la estimación del desplazamiento bidimensional entre imágenes.

Básicamente, el funcionamiento de este método comienza por calcular un desplazamiento y luego separar los píxeles en segmentos predecibles e impredecibles. El vector de desplazamiento de un píxel se estima minimizando recursivamente una función no lineal de la diferencia entre dos regiones determinadas ubicadas en

dos *frames* consecutivos. Cabe considerar que estas regiones o segmentos hacen referencia a un grupo de píxeles, pero podría ser tan pequeño como un solo píxel.

En esta perspectiva, Netravali y Robbins, pioneros en este algoritmo (1979), definieron el llamado ***Displaced Frame Difference*** **(DFD)**. Esta cantidad, como su mismo nombre indica, representa la diferencia de desplazamiento de un píxel en dos *frames*:

$$DFD(x, y; d_x, d_y) = f_n(x, y) - f_{n-1}(x - d_x, y - d_y), \tag{2.1}$$

donde los subindices *n* y *n-1* hacen referencia a dos instantes de tiempo asociados a 2 *frames* consecutivos, basados en los vectores de movimiento que van a ser estimados; *x, y* son las coordenadas en un plano cartesiano; y d_x, d_y son las dos componentes del vector de desplazamiento, representando las direcciones horizontales y verticales en el plano, respectivamente. Evidentemente, si no hay error en la estimación, es decir, el vector de desplazamiento que está bajo análisis es el correcto, el valor de DFD debe ser 0.

No obstante, como ya se ha mencionado al inicio de este apartado, la estimación de movimiento se realiza mediante el uso de una función no lineal, la cual es simplemente el cuadrado de DFD (DFD^2). De esta forma, se convierte la estimación de movimiento en un problema de minimización que se resuelve mediante el uso de métodos de gradiente [23].

Así, el algoritmo es aplicado al mismo píxel tantas veces como sea necesario y, posteriormente, el mismo procedimiento se repite con el siguiente píxel, donde se utiliza el vector de movimiento estimado en el píxel anterior como estimación inicial de este. Esta recursión puede ser llevada a cabo horizontalmente, verticalmente o temporalmente como se puede ver en la figura 2.6. Con recursión temporal, se refiere a que el vector de desplazamiento estimado puede ser pasado al píxel situado en la misma posición espacial pero en un *frame* adyacente.

Una vez desarrollado el primer algoritmo basado en la técnica *pel-recursive* por Netravali y Robbins [24], se han publicado otros algoritmos basados en la misma técnica con el objetivo de obtener mejores resultados en distintas aplicaciones. Algunos de ellos son el algoritmo de *Bergmann*, el algoritmo de *Cafforio y Rocca* y el algoritmo de *Walker y Rao* [9, pág 281].

No obstante, como ya se ha comentado con anterioridad, el proyecto del que trata el presente documento está destinado a aplicaciones en satélite y, por ello, se mostrará especial atención a la estimación global de los píxeles. Por esta razón, un método como el *pel-recursive*, en el que todos y cada uno de los píxeles posee un vector

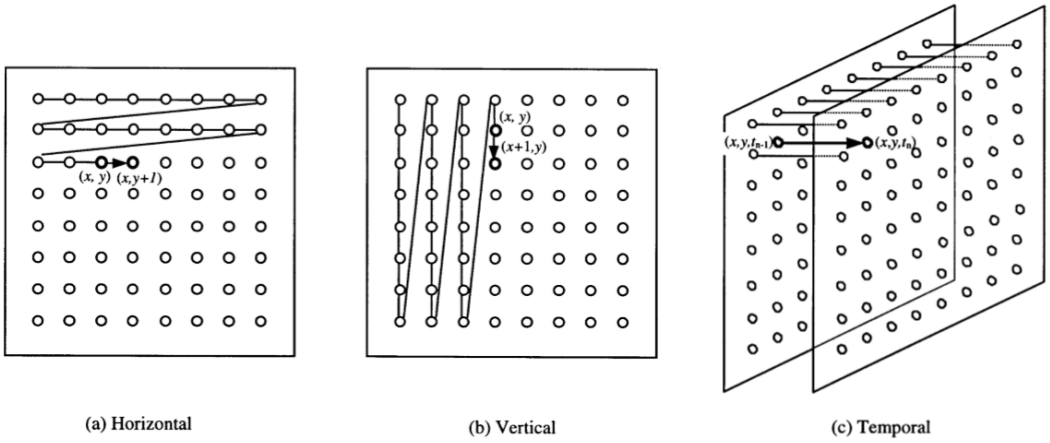

Fig. 2.6: Tipos de recursiones: (a) horizontal; (b) vertical; (c) temporal.

de movimiento propio, no es óptimo. Además, este algoritmo requiere de una alta capacidad de cómputo y presenta una latencia significativa, al probar múltiples combinaciones posibles. En aplicaciones espaciales, los dispositivos electrónicos suelen ser más limitados que los comerciales en cuanto a capacidad de procesamiento se refiere, por lo que este tipo de algoritmos no resultan idóneos.

Block-matching

El algoritmo *Block Matching* (BMA) es la técnica más utilizada para explotar la redundancia temporal en una secuencia de vídeo, ya que ha sido empleada en la mayor parte de los codificadores de vídeo que se han mencionado anteriormente en la sección 2.1.1. Si bien es cierto que la técnica *pel recursive* analizaba individualmente el movimiento de cada píxel, en el caso del BMA se separa cada *frame* en macrobloques (MB) de tamaño *NxN* y se considera que todos los píxeles que componen cada bloque tienen la misma actividad de movimiento; en otras palabras, se genera un único vector de movimiento por cada macrobloque. Para ello, se dispone de una ventana de búsqueda de tamaño P con la que delimitar el alcance máximo que puede alcanzar los vectores de movimiento candidatos en su búsqueda del MB en cuestión. El valor de esta ventana de búsqueda suele tener un valor igual a la mitad del tamaño del MB que se esté utilizando.

En función a lo planteado, es necesaria alguna técnica con la cual poder analizar los posibles desplazamientos de cada macrobloque para la posterior elección del vector de movimiento óptimo. Para ello, una de las técnicas más relevantes es el algoritmo ***Full Search* (FS)**, el cual analiza todas las posibilidades dentro del área de búsqueda en el *frame* para encontrar el vector de movimiento idóneo. La elección de dicho vector se realiza mediante el cálculo de la **suma de absolutas diferencias (SAD)**

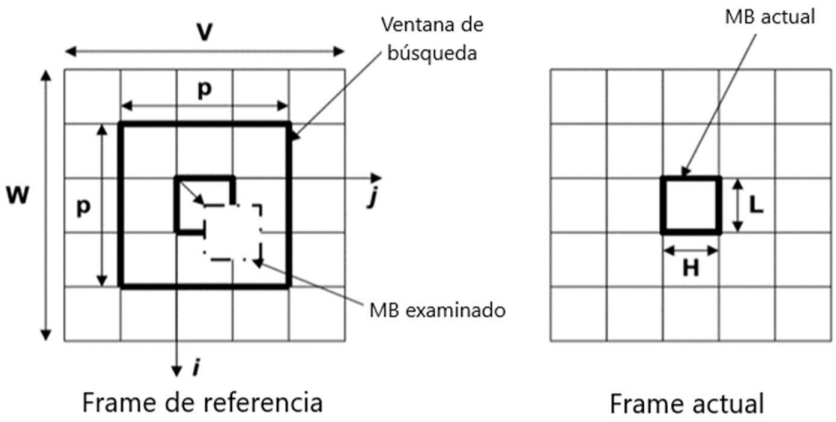

Fig. 2.7: Algoritmo block matching

descrita en la expresión 2.2. Como su nombre indica, este valor se calcula mediante la suma de las diferencias entre cada uno de los píxeles que forman el macrobloque actual y los píxeles análogos en el macrobloque de referencia desplazado por el vector de desplazamiento que esté bajo análisis. De esta forma, cuanto menor sea el SAD, mejor será el vector de movimiento que haya sido utilizado para su cálculo.

$$SAD(m,n) = \sum_{i=0}^{N}\sum_{j=0}^{N} x(i,j) - y(i+m, j+n), \tag{2.2}$$

donde *m* y *n* son los valores del desplazamiento que está bajo análisis, *N* es el tamaño del MB, *x(i,j)* es el píxel actual, e *y(i+m, j+n)* es el píxel de referencia desplazado por el vector de movimiento utilizado.

Como se ha comentado, el algoritmo *Full-Search* analiza todos los posibles vectores de movimiento hasta encontrar el óptimo y, por ello, posee una gran precisión. Sin embargo, su complejidad de cálculo hace que sea difícil de implementar en tiempo real y en soluciones hardware de bajo consumo de recursos. Por esta razón, se han desarrollado otros algoritmos de búsqueda como el *Diamond Search (DS)* [25] o el *New Three Step Search (NTSS)* [26].

2.2.3 Algoritmos de búsqueda para block-matching

Diamond Search

Este algoritmo emplea 2 patrones de búsqueda: *Large Diamond Search Pattern (LDSP)* y *Small Diamond Search Pattern (SDSP)*. Como se puede observar en la figura 2.8, el primer patrón se compone de 9 puntos de verificación que forman una forma de

diamante, mientras el segundo utiliza 5 puntos, formando un diamante de menor dimensión.

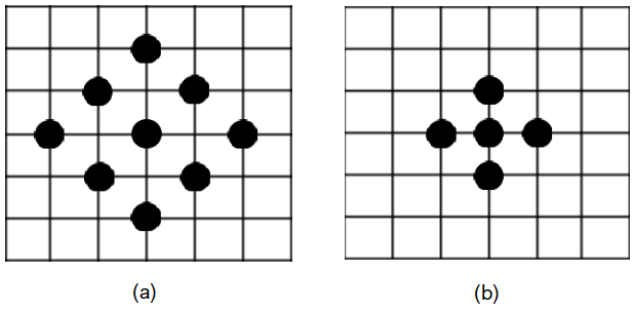

Fig. 2.8: Patrones de búsqueda del DS: (a) LDSP, (b) SDSP

En primer lugar, se utiliza LDSP repetidamente hasta obtener el mejor SAD en el píxel central del "diamante". Cuando esto ocurre, se cambia de LDSP a SDSP y se compara el SAD obtenido en el punto central y en los 4 nuevos puntos. Como resultado de esta comparación, se obtendrá el punto con menor SAD y, por consiguiente, el vector de movimiento óptimo para el MB analizado. Es conveniente tener en cuenta que los puntos de verificación se superponen parcialmente entre pasos adyacentes, especialmente cuando LDSP se aplica continuamente. En otras palabras, en cada iteración del algoritmo no se tendrán que comprobar todos los puntos, ya que se habrán analizado en la iteración anterior. Para poder entender mejor este concepto, se puede observar en la figura 2.9 una posible ruta de búsqueda, donde se utiliza LDSP hasta 4 veces, momento en el que no se encuentra ya menor SAD y se pasa a SDSP como último paso para encontrar el menor SAD, obteniendo el vector de movimiento que mejor se adapta al movimiento de los píxeles entre *frames*.

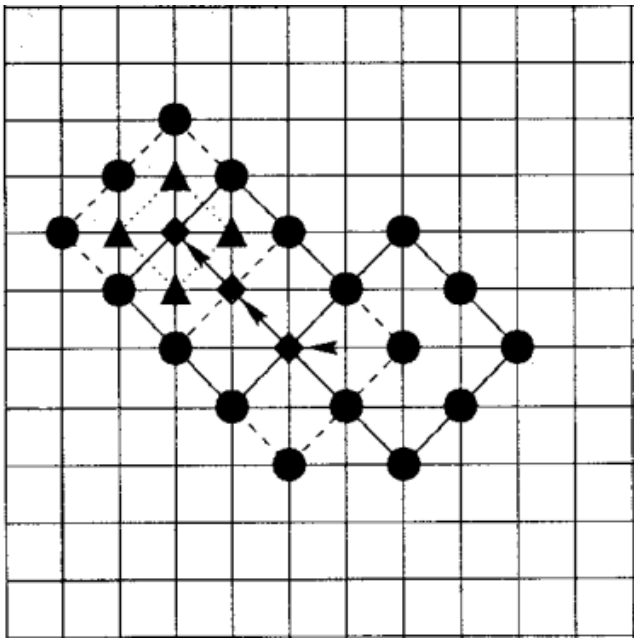

Fig. 2.9: Ejemplo de posible ruta utilizando el algoritmo DS [25]

New Three Step Search

Esta técnica, como bien indica su nombre, se desarrolla en tres pasos. Primero, se comprueban 16 puntos, además del origen, en búsqueda de aquel que produzca un menor SAD. De estas direcciones de búsqueda, 8 se encontrarán a una distancia *d/2*, donde *d* es el diámetro de la ventana de búsqueda; mientras que los otros 8 se encontrarán exactamente a una unidad de distancia del origen. En caso de que el mejor SAD se encuentre en el origen, la búsqueda habrá terminado y el vector de movimiento será (0,0). En cambio, si el mejor resultado se obtiene en alguna de las ubicaciones adyacentes al origen, se traslada el origen a ese punto y se comprueban los valores adyacentes a él, obviando los puntos contiguos que ya se hubieran comprobado en el paso anterior. Una vez hecho esto, se modificará el vector de movimiento a la ubicación que haya obtenido el menor SAD.

Por otro lado, si el mejor resultado se obtiene en aquellos puntos ubicados a una distancia *d/2*, se trasladará el origen a esa posición y se repetirá el proceso anterior. En este caso, se comprobarán 8 puntos situados a una distancia *d/4*, la mitad de distancia que en el primer paso. Una vez encontrado el mejor resultado, se ubicará el origen en ese punto y, comenzando el tercer y último paso, se analizarán los píxeles adyacentes, obteniendo así el vector de movimiento que más se adapte al desplazamiento realizado por el macrobloque bajo observación. En la figura 2.10 se ilustra un ejemplo de una posible ruta seguida a través de los 3 pasos del algoritmo para obtener el vector de desplazamiento óptimo.

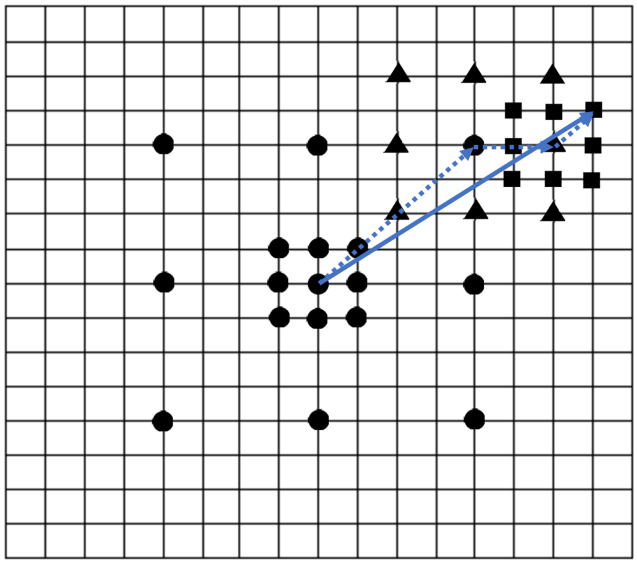

Fig. 2.10: Ejemplo de posible ruta utilizando el algoritmo NTSS

2.3 Conclusiones

En este capítulo se ha hecho un estudio del estado del arte en lo que respecta a la codificación de vídeo y, más concretamente, a la etapa de estimación de movimiento. Así, después del análisis de los principales algoritmos de estimación de movimiento, se ha decidido optar por un BMA ya que, en las aplicaciones espaciales que conciernen a este proyecto, la mayor parte de los píxeles tendrán un movimiento similar (estimación global) y no será necesario un vector de movimiento por cada uno de ellos.

A su vez, dentro de los algoritmos de búsqueda que se pueden emplear en un BMA, se ha descartado el FS, debido a su alta complejidad de cómputo.

Finalmente, se ha optado por la selección de un algoritmo basado en el NTSS por encima del DS pese a que ambas opciones se han considerado bastante similares en términos de complejidad y precisión. Esto es debido a la disponibilidad de un *software* desarrollado dentro del grupo de investigación en el que se utiliza el algoritmo NTSS como estimador de movimiento, lo que permite emplearlo como punto de partida, suponiendo un tiempo menor de desarrollo, pasando directamente a su adaptación a *hardware*. De esta forma, conseguimos también una planificación más ajustada a los tiempos para la realización de este TFG.

3 Dispositivo hardware y entorno software

En este capítulo se describirán las distintas características del dispositivo *hardware* escogido para albergar el sistema diseñado con nuestro algoritmo de estimación de movimiento, desde los recursos disponibles que este presenta, hasta la funcionalidad interna del SoC en cuestión, destacando la comunicación entre los dos bloques centrales del dispositivo, el sistema de procesamiento y la lógica programable.

Además, teniendo en cuenta que se utilizará la metodología *HLS* para desarrollar el sistema que irá implantado sobre la placa de prototipado, se describirán las distintas herramientas que proporciona el entorno completo de *Vivado Design Suite*, con el que pasaremos de un diseño en lenguaje de alto nivel como C/C++ a una descripción RTL mediante Vivado HLS para su posterior encapsulado IP. Luego, se implementará el bloque IP generado dentro de un diagrama de bloques que integrará todos los componentes necesarios para la consolidación del sistema completo mediante Vivado IDE. Así, con esta herramienta, se depurará la plataforma *hardware* diseñada y se generará el *bitstream* necesario en la herramienta Xilinx SDK para el desarrollo de una aplicación *software* a medida que controle el funcionamiento del sistema completo.

3.1 *Hardware* propuesto

Para la implementación de nuestro algoritmo de estimación de movimiento, se ha decidido emplear un dispositivo *hardware* con el que poder obtener resultados de forma rápida y eficiente, sin descuidar las prestaciones en términos de latencia y de consumo de recursos.

Para llegar a este fin, se ha optado por la familia Zynq-7000 [27] de Xilinx, la cual se basa en una arquitectura **SoC** (*System-on-Chip*). Estos dispositivos nos permiten una alta tasa de cómputo gracias a la integración de un sistema de procesamiento (PS); y una gran flexibilidad y posibilidades de reconfiguración debido a la lógica programable (PL) que poseen. En la figura 3.1 se puede observar el diagrama de bloques de esta arquitectura.

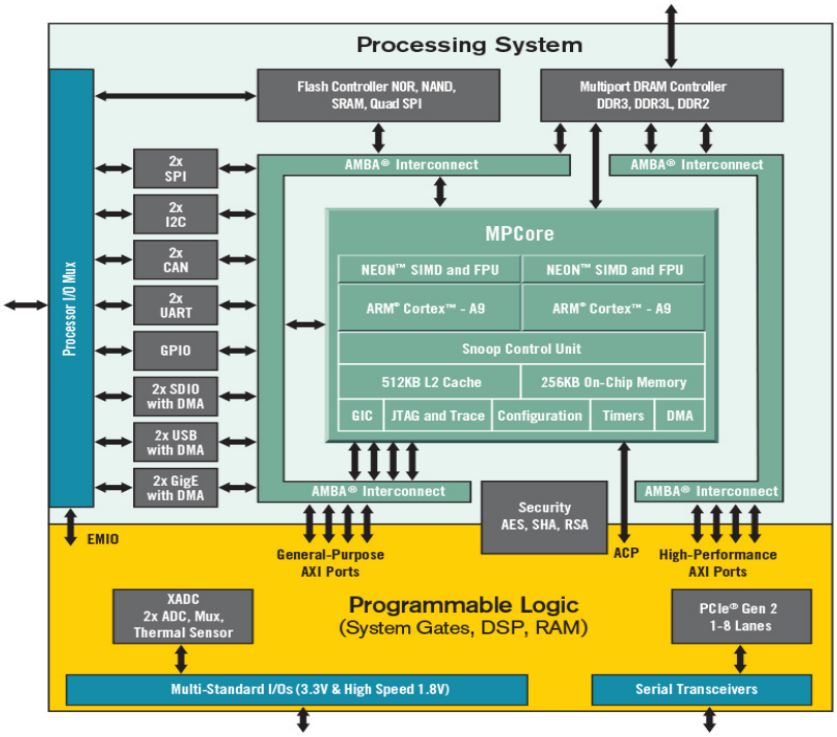

Fig. 3.1: Arquitectura SoC empleada en Zynq [28]

3.1.1 Dispositivo XC7Z020

Entre todos los dispositivos de la familia Zynq que ofrece Xilinx, se empleará el XC7Z020 [27], un SoC que integra una FPGA de la familia Artix-7 y dos núcleos procesadores ARM Cortex-A9 en el PS. Entre las principales características que presenta este dispositivo en lo referente a la lógica programable, se destacan las siguientes:

- PLC (*Programable Logic Cells*): 85000
- LUTs (*Look-Up Tables*): 53200
- *Flip Flops*: 106400
- Block RAM: 4.9 Mb (140 bloques de 36Kb)
- DSP slices: 220

Por otro lado, en la parte parte del PS se pueden destacar las siguientes características:

- L1 cache: 32KB para instrucciones y 32KB para datos

- L2 cache: 512KB
- Memoria *On-Chip*: 256KB
- Memorias externas dinámicas y estáticas soportadas: DDR3, DDR3L, DDR2, LPDDR2, 2x Quad-SPI, NAND, NOR
- Controladores de periféricos: 2x UART, 2x CAN 2.0B, 2x I2C, 2x SPI, 4x 32b GPIO,
- Periféricos incorporados en el DMA: 2x USB 2.0 (OTG), 2x *Tri-mode Gigabit Ethernet*, 2x SD/SDIO

3.1.2 Mecanismos de comunicación PS-PL

Teniendo en cuenta las distintas partes que componen la unidad de procesamiento y la unidad de lógica programable, se hace necesario conocer la forma que que se comunican estos dos dominios entre sí para el correcto intercambio de datos.

El PS consta de 4 bloques principales: *Application Processor Unit (APU)*, las interfaces de memoria, *I/O Peripherals (IOP)* y la infraestructura de buses AMBA AXI. En este último es donde se realizan todas las conexiones entre los distintos bloques de la propia unidad de procesamiento y, además, la comunicación de todos ellos con el PL. Este módulo consta de maestros sensibles a la latencia, como la CPU ARM, que tiene las rutas más cortas a la memoria; y maestros con un ancho de banda crítico, como el PL, que tienen conexiones de alto rendimiento a los esclavos con los que necesita comunicarse. Distintas interfaces del estándar AXI4 permiten transferencias de tamaño variable, alto *throughput* y baja latencia.

En el dispositvo en cuestión, así como en todos los demás SoC de la familia Zynq, se dispone de 4 interfaces AXI de 32 bits (2 maestras y 2 esclavas), además de 4 interfaces esclavas configurables de 32-64 bits para un acceso directo a la memoria DDR y OCM, denominadas como puertos AXI de alto rendimiento. Tal y como se puede ver en la figura 3.2, estos puertos se comunican con el bloque *Interconnect* a través de un controlador FIFO de 1KB de capacidad para cada señal, proporcionando un equilibrio entre las distintas velocidades de funcionamiento.

Además, se encuentra otra interfaz esclava AXI de 64 bits que proporciona acceso a la memoria de la CPU y es conocida como el puerto ACP. Esta fue desarrollada por ARM como una solución *hardware* para facilitar el tratamiento de problemas de

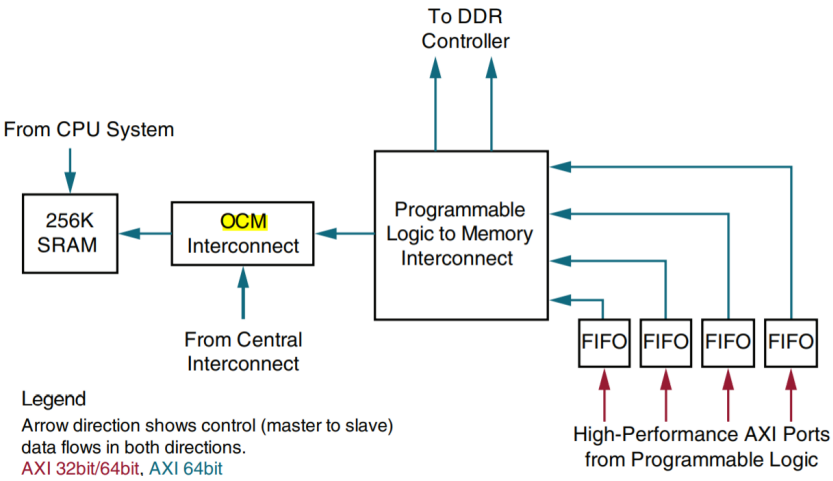

Fig. 3.2: Interfaces de comunicación entre el PL y la memoria del PS[27]

coherencia de caché al introducir nuevos bloques de aceleración en un sistema de múltiples núcleos.

3.1.3 Placa de prototipado Pynq-Z1

Para la implementación de nuestro proyecto se utilizará la placa de prototipado Pynq-Z1 [28], la cual integra el SoC XC7Z020 de la familia Zynq de Xilinx que ya se ha descrito en este capítulo. Además de todas las características que este SoC incorpora, la placa está dotada de diversas herramientas que permiten la comunicación con elementos externos al dispositivo para intercambio de datos o tareas de depuración, entre otras posibilidades. Así, podemos destacar algunos de los componentes más relevantes que el dispositivo incorpora, como la inclusión de un puerto micro-USB como *bridge* UART para comunicaciones seriales o como interfaz JTAG para tareas de depuración; 2 puertos HDMI (uno como entrada y otro como salida), así como un puerto mini jack para salida de audio y un micrófono con interfaz PDM. Adicionalmente, en lo referente a la memoria, la placa de prototipado incorpora una interfaz para memoria RAM de tecnología DDR3 de hasta 512 MB de capacidad, una memoria *flash Quad-SPI* de 16 MB y un lector de tarjetas SD que se puede utilizar tanto para almacenar y recoger datos, como para cargar un sistema operativo en la plataforma.

Siguiendo con las opciones que nos proporciona el dispositivo, se admiten tres modos de arranque diferentes: microSD, *Quad* SPI y JTAG. El modo de arranque, tal y como se puede apreciar en la figura 3.3, se selecciona mediante el *jumper* JP4, que afecta al estado de los pines de configuración de la Zynq después del encendido.

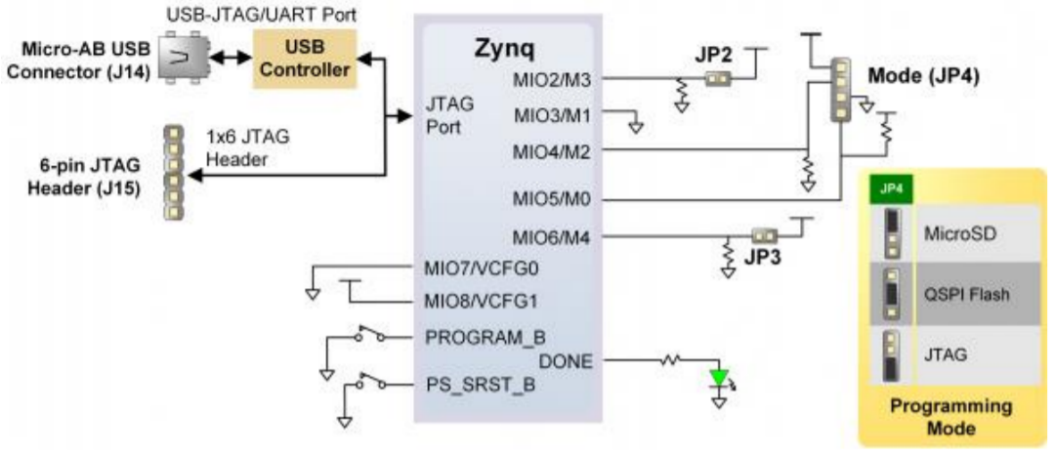

Fig. 3.3: Configuración de los piness en la Pynq Z1 [28]

Por último, a fin de conocer cómo es físicamente el dispositivo, se ilustra una imagen de una placa de prototipado Pynq-Z1 en la figura 3.4, pudiéndose distinguir los distintos componentes a través de las indicaciones grabadas en la superficie de la plataforma.

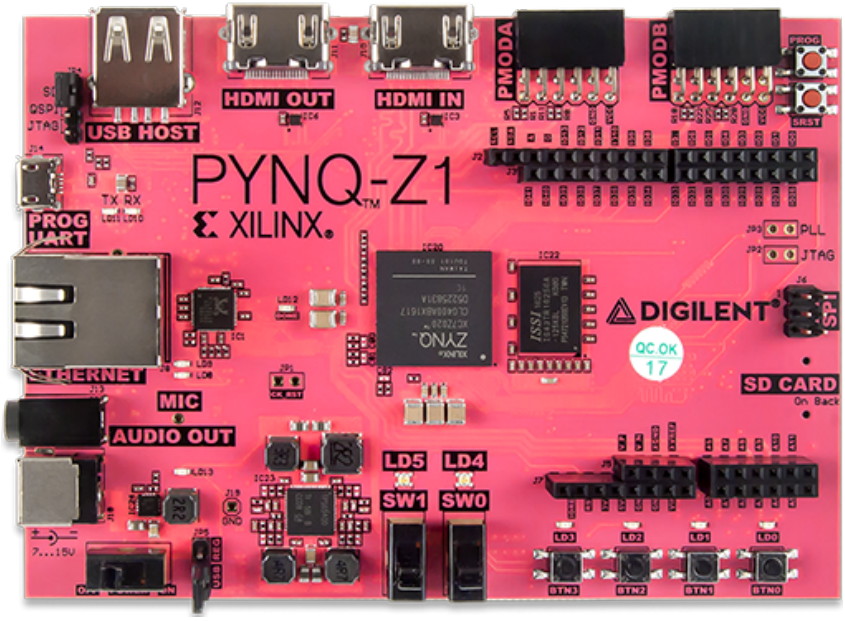

Fig. 3.4: Placa de prototipado Pynq-Z1

3.2 Entorno Xilinx Vivado

Una vez descrito el *hardware* seleccionado para llevar a cabo la implementación de nuestro algoritmo de estimación de movimiento, es preciso definir el entorno *software* mediante el cual diseñaremos el sistema que irá implantado en la placa de prototipado Pynq-Z1. Para ello, se ha optado por el entorno de diseño *Vivado*

Design Suite, ya que cuenta con todo tipo de herramientas para la programación de dispositivos Xilinx, ofreciendo múltiples formas de realizar las tareas involucradas en su diseño, implementación y verificación [29].

3.2.1 Flujo de diseño

El entorno de diseño *Vivado Design Suite* permite la posibilidad de seguir el flujo de diseño tradicional de FPGA *Register Transfer Level (RTL)-to-bitstream* o un flujo de integración a nivel de sistema que permita una reducción del tiempo de diseño. De esta forma, la metodología de diseño puede partir de una descripción RTL o de un diseño basado en modelos mediante Matlab y Simulink, acelerando el camino hacia la programación del dispositivo Xilinx gracias a la generación automática del código. Sin embargo, nos centraremos en el flujo de diseño desde una especificación en un lenguaje de alto nivel como C, C++ o SystemC, teniendo en cuenta que es la opción elegida para el diseño de nuestro sistema.

Tal y como se puede apreciar en la figura 3.5, la descripción RTL generada mediante HLS es integrada en un sistema que contiene los distintos bloques propios de Xilinx a través del *IP Integrator*. Luego, a partir de este diagrama de bloques, se realizaría todo el proceso de síntesis lógica e implementación sobre el dispositivo para posteriormente poder generar el *bitstream* con el que programar y depurar el diseño.

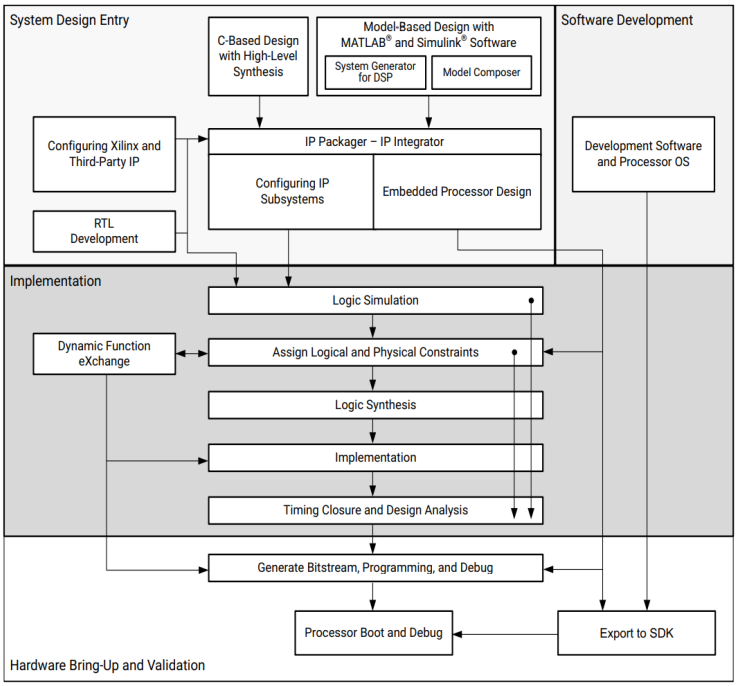

Fig. 3.5: Flujo de diseño de alto nivel en *Vivado Design Suite* [29]

Diagrama de bloques IPs

Una de las partes más importantes dentro del flujo de diseño reside en la integración de nuestro diseño dentro de la plataforma que conforma el sistema completo. Para ello, Vivado IDE dispone de la herramienta IP *Packager* que permite convertir el diseño en un módulo IP reutilizable que luego puede ser añadido al catálogo extensible de Vivado (figura 3.6).

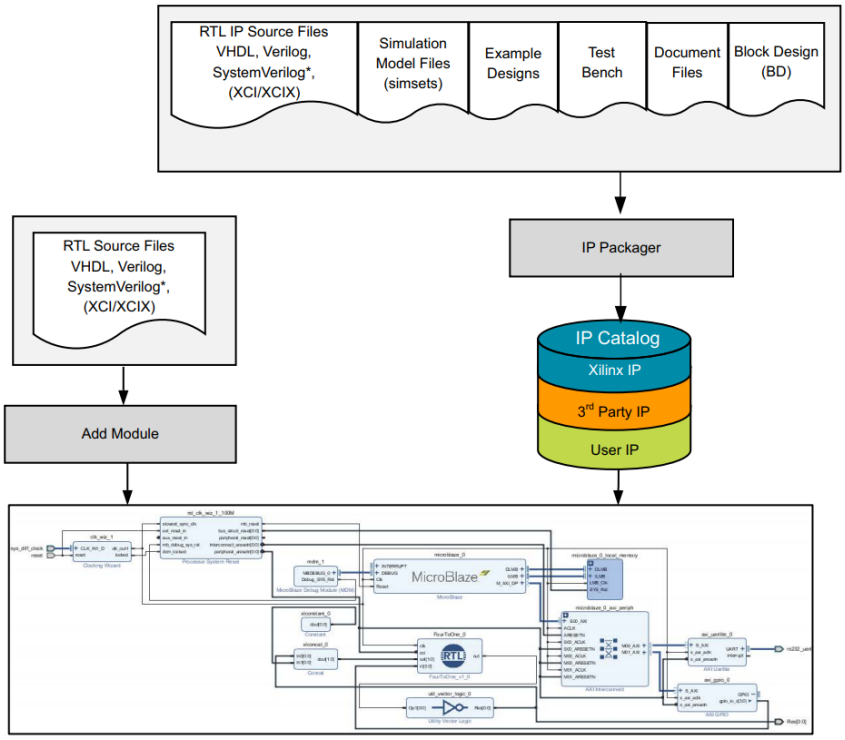

Fig. 3.6: Flujo de diseño IP [30]

Luego, mediante el *IP Integrator* se ofrece la posibilidad de construir un sistema de bloques basado en IPs en el que se disponga de todo el interconexionado entre los distintos módulos que se pueden incluir, tanto los diseñados por el usuario o por terceros, como todos aquellos proporcionados desde el catálogo propio de Xilinx. Al disponer de numerosos IPs estandarizados para integrar en la plataforma, se logra disminuir en gran medida el tiempo necesario para la creación del sistema final, ya que tras añadir los IPs oportunos al diseño, únicamente habría que modificar los parámetros de cada uno de ellos para adaptarlos a la aplicación concreta a desarrollar.

3.2.2 HLS y herramientas para síntesis de alto nivel

La síntesis de alto nivel [31] es un proceso de diseño en el que se desarrolla una descripción funcional de alto nivel y se compila automáticamente en una

implementación RTL que cumple con ciertas restricciones de diseño especificadas por el usuario. El objetivo de HLS es lograr obtener la implementación de una solución determinada con mayor facilidad y menor tiempo de desarrollo. Para ello, utilizando lenguajes como C, C++ o SystemC, se eleva el nivel de abstracción, haciendo innecesario, por ejemplo, describir el comportamiento específico ciclo por ciclo, delegando a las herramientas de HLS la libertad de decidir qué hacer en cada ciclo de reloj, siempre y cuando se cumplan con las restricciones temporales, de área y de consumo de potencia especificadas.

Así pues, la metodología seguida comienza con el diseño de una función algorítmica en lenguaje de alto nivel que desempeñe la funcionalidad deseada del sistema para su posterior síntesis de alto nivel. Con esta, se obtendrá la descripción RTL y se podrá contar con una primera estimación de los recursos lógicos que tomaría implementar dicha funcionalidad en *hardware* sobre el dispositivo seleccionado.

Vivado HLS

Para llevar a cabo esta etapa del flujo de diseño, Xilinx proporciona el *software* Vivado HLS [32]. Gracias a esta herramienta, podremos compilar, simular y depurar el algoritmo C que hayamos diseñado, mediante los bancos de pruebas o *testbenchs* necesarios y, de esta manera, verificar que el diseño desempeña correctamente la función especificada.

Además, como es evidente, permite sintetizar la función C en una implementación RTL a la que se podrán incorporar directivas de optimización indicadas por el usuario, como particiones de *arrays* en memoria o aplicaciones de *pipeline* en bucles concretos.

Una vez realizada la simulación y la síntesis de alto nivel, Vivado HLS permite realizar una verificación de la implementación RTL. Esta *co-simulación* C/RTL utiliza el *testbench* en C creado por el usuario y el RTL generado por la herramienta para confirmar que la simulación RTL coincide con el comportamiento del código fuente en C, además de indicarse la latencia con la que se ejecuta dicha descripción en número de ciclos de reloj.

Por último, el diseño *hardware* obtenido es encapsulado como un bloque IP y exportado para su posterior integración en el sistema completo en Vivado IDE. Todas estas etapas seguidas en el flujo de diseño de Vivado HLS se encuentran ilustradas en la figura 3.7.

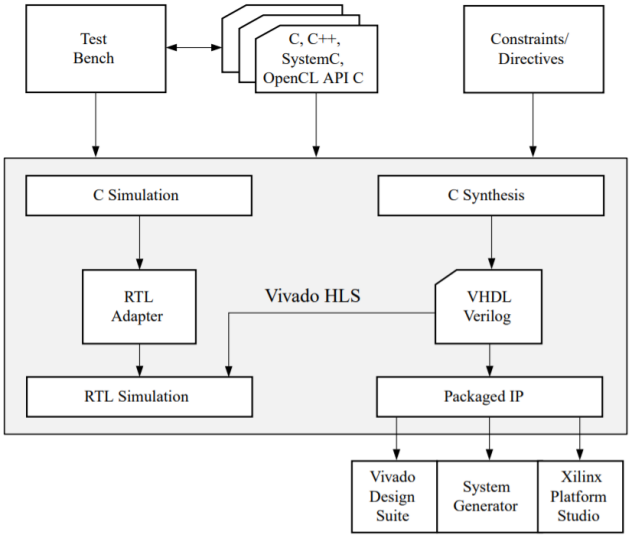

Fig. 3.7: Flujo de diseño Vivado HLS [32]

Adicionalmente, con el objetivo de mostrar cómo es visualmente la herramienta, se ilustra en la figura 3.8 la interfaz de usuario de Vivado HLS, donde se puede ver como presenta las bases del IDE de Eclipse al encontrarse las tres perspectivas de depurado, síntesis y análisis.

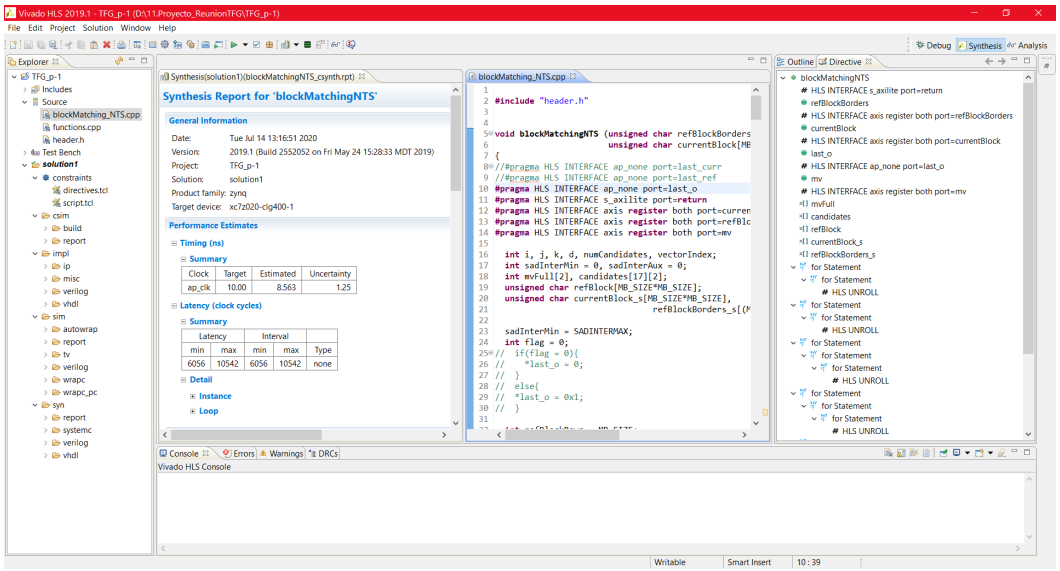

Fig. 3.8: Interfaz de usuario Vivado HLS

Síntesis lógica e implementación mediante Vivado IDE

Además de la síntesis de alto nivel que proporciona el entorno de Vivado HLS, Vivado IDE [33] permite realizar una segunda síntesis, concretamente una síntesis lógica que transforma la descripción RTL en una representación a nivel de puertas lógicas. Tras su finalización, se obtiene un análisis temporal del diseño sintetizado en el que se

especifica el *slack* disponible y las rutas críticas del diseño (figura 3.9), algo de gran utilidad para garantizar que se cumplen adecuadamente las retricciones temporales necesarias para una implementación efectiva. A mayor número de restricciones físicas, mayor precisión se obtendrá en los resultados obtenidos, aunque con un estimación añadida del *delay* por el enrutamiento. No obstante, es importante tener en cuenta que solo el análisis temporal después de la implementación incluye los retrasos reales para el enrutamiento, haciendo el análisis en esta etapa más preciso que el obtenido tras la síntesis lógica.

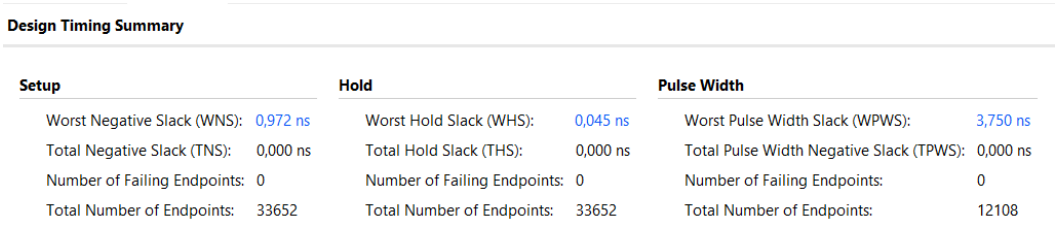

Fig. 3.9: Análisis temporal del diseño obtenido en la sintesis lógica

Una vez realizada la síntesis lógica y generada la *netlist*, el sistema continúa con su etapa de implementación conocida como "*place and route*" [34]. Así, el proceso de implementación transforma una *netlist* lógica junto con las restricciones especificadas en un diseño representado con los recursos *hardware* del dispositivo en cuestión, sobre los que se realizará la distribución y enrutamiento de la *netlist* obtenida.

Por su parte, el IDE de Vivado ofrece una representación de la plataforma física donde se visualizan las ubicaciones y el espacio ocupado por las distintas redes lógicas implementadas, es decir, el consumo de recursos de la FPGA por parte de los distintos componentes del diseño. En la figura 3.10 puede observarse dicho *layout*, además de visualizarse la interfaz tan intuitiva que posee el IDE de Vivado, pudiendo realizar los distintos procesos que ofrece la herramienta desde el *Flow Navigator*, incluyendo el diseño del sistema de bloques IPs hasta la generación del *bitstream*.

3.2.3 Prototipado hardware

Después de implementar con éxito el diseño, el siguiente paso es ejecutarlo en *hardware* mediante la programacion y depurado de la FPGA [35]. Para llevar a cabo este depurado *hardware* se presentan dos opciones: *Virtual Input/Output (VIO)* y el *Integrated Logic Analyzer (ILA)*. En ambos casos, es necesaria la inclusión del bloque IP en cuestión facilitado por Xilinx en el diseño de bloques de nuestro sistema, con el fin de conectarlo a las señales que se deseen monitorizar.

En el presente documento se describirá únicamente la funcionalidad del *System ILA* [36], ya que será el método empleado para el depurado de nuestra plataforma.

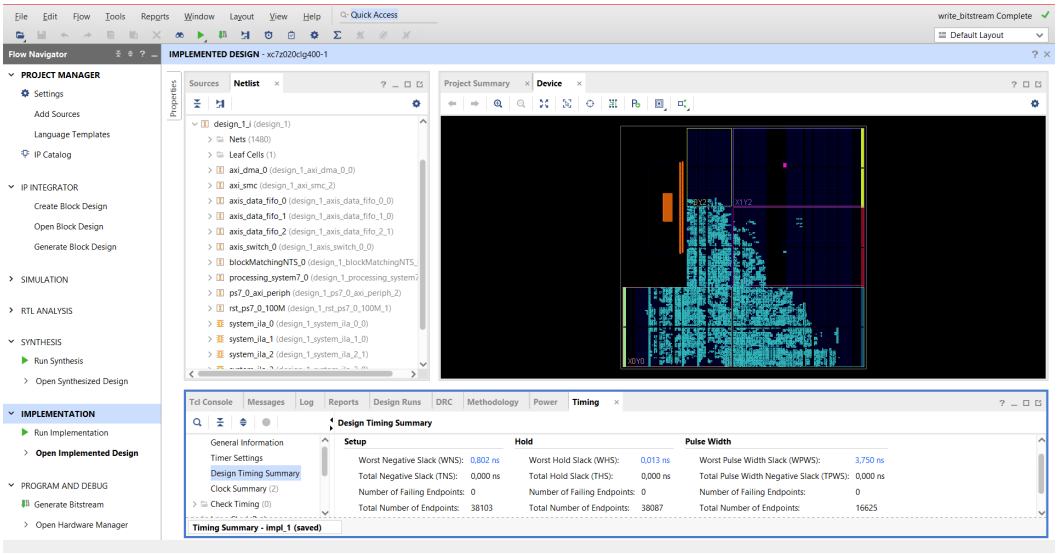

Fig. 3.10: Perspectiva del diseño implementado en Vivado IDE

Este bloque incluye muchas características avanzadas de los analizadores lógicos modernos, incluidas opciones de *trigger*, en las que poder observar ciertos tramos de las señales bajo análisis cuando se cumplan ciertas condiciones predefinidas por el usuario al producirse flancos de subida o de bajada. El *System ILA* permite el estudio de hasta 1024 señales, pudiendo especificar el tramo exacto de señal que se desea capturar, señalando cuál será la muestra inicial y durante cuántos ciclos de reloj se producirá dicha captura de datos.

De esta forma, se proporciona un entorno visual que permite ver las formas de onda de las señales conectadas y así poder comprobar los valores exactos que van tomando a lo largo del tiempo especificado. Para acceder a esta visualización de las señales, Vivado IDE ofrece la herramienta *Hardware Manager*, la cual puede verse en la figura 3.11.

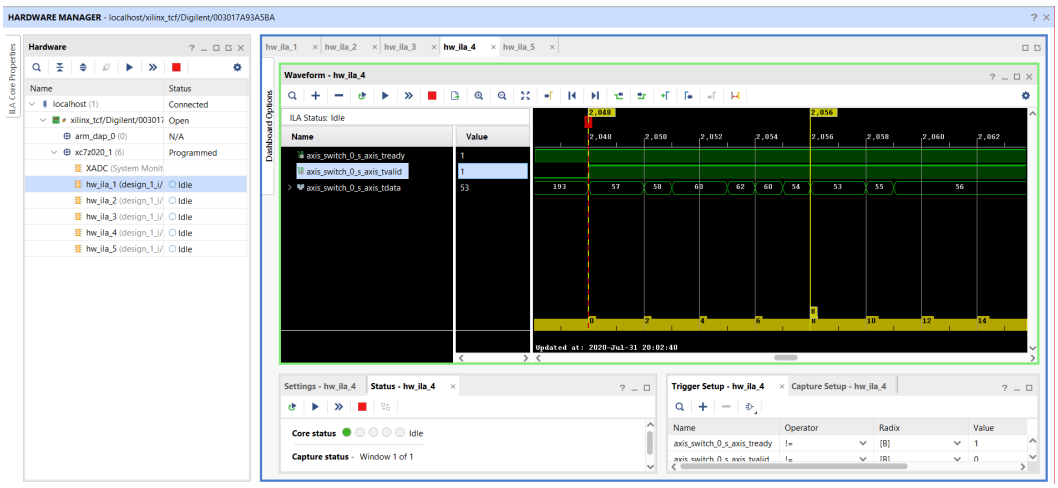

Fig. 3.11: Visualización de señales mediante Hardware Manager

Finalmente, cuando se tiene constancia de que la plataforma funciona correctamente tras haber pasado las distintas etapas del flujo de diseño que se han comentado, se dispone a generar el archivo *bitstream* que será descargado sobre la memoria interna de la FPGA.

3.2.4 Desarrollo software mediante Xilinx SDK

En este punto, tras haber generado el *bitstream* satisfactoriamente, es turno de exportar el *hardware* al entorno Xilinx SDK a fin de comenzar a desarrollar la aplicación *software* que gobernará el funcionamiento de la plataforma [37]. Para ello, desde Xilinx Vivado se selecciona la opción 'exportar *hardware*', incluyendo el *bitstream*, y, una vez hecho, se pulsa la opción *'Launch SDK'* con lo que se ejecutará el entorno de desarrollo *software* Xilinx SDK.

Una vez en el SDK, es necesario crear un proyecto que esté asociado al hardware exportado y que contenga todas las librerías oportunas. Así pues, tal y como se ve en la figura 3.12, se crea un proyecto a partir de nuestro *hardware*, señalando sobre qué microprocesador se ejecutará de entre los dos disponibles en la Zynq empleada y creando, además, un *Board Support Package (BSP)* ligado al proyecto con el que podremos contar con todas las librerías necesarias para cubrir todos los detalles asociados a los diversos módulos que tengamos en nuestro sistema y que deban ser controlados desde la aplicación *software*.

3.3 Conclusiones

En este capítulo se ha proporcionado una descripción de las distintas características que presenta el *System on Chip* XC7Z020 de la familia Zynq utilizado, enfatizando los mecanismos de comunicación PS-PL que tanta importancia tienen para la correcta comunicación entre los distintos componentes de la lógica programable y de la unidad de procesamiento.

Además, se ha analizado la placa de prototipado Pynq-Z1 sobre la que se implementará el sistema completo del proyecto. La elección de este dispositivo se ha considerado la más adecuada entre las opciones de las que se dispone dentro del grupo de investigación debido al número de recursos lógicos disponibles que presenta.

Por último, se ha descrito el flujo de diseño de un sistema *hardware-software* utilizando las herramientas de diseño de Xilinx Vivado, detallando todas las etapas que forman parte del proceso, desde el diseño del módulo funcional en lenguaje de alto

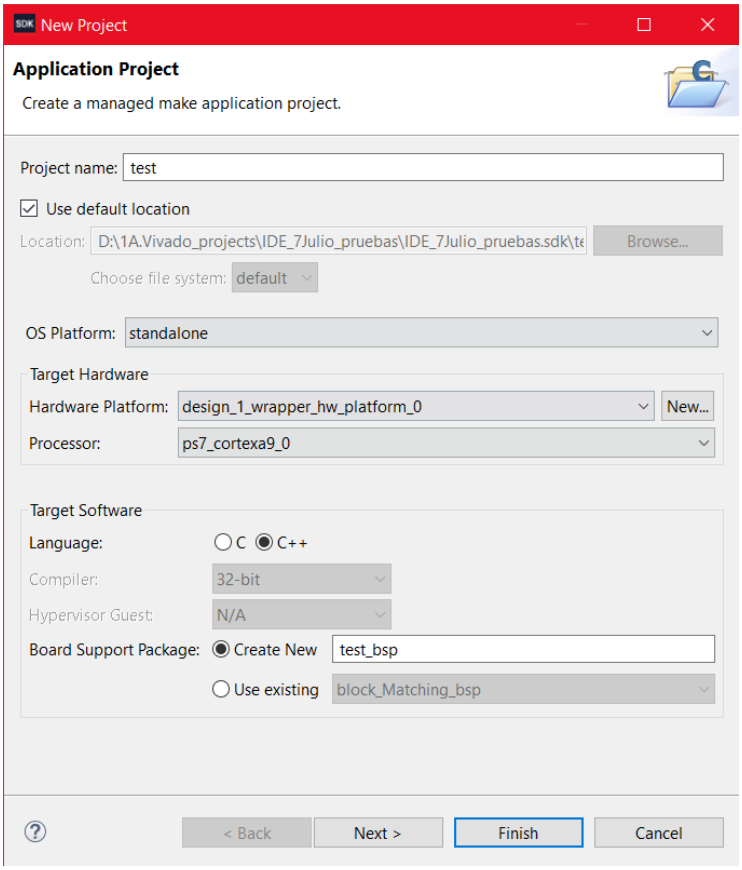

Fig. 3.12: Creación de un nuevo proyecto en SDK

nivel mediante HLS, hasta la exportación *hardware* del proyecto y la programación de la aplicación *software*.

4 Diseño del bloque IP

Tras realizar el estudio de los distintos algoritmos de estimación de movimiento existentes, se ha optado por implementar el algoritmo *New Three Step Search* (descrito en la sección 2.2.3) dado el equilibrio que proporciona entre complejidad algorítmica, tiempo de ejecución y consumo de recursos. Dada la aplicación de este proyecto para secuencias de vídeo obtenidas desde satélites en la órbita terrestre, el movimiento realizado por todos los píxeles se producirá generalmente en la misma dirección (en resumen, debido al movimiento de la cámara a lo largo de la órbita), con la ausencia de movimientos locales destacables en la imagen y, por todo ello, la estimación de movimiento se realizará con una precisión de píxeles enteros, sin necesidad de bajar a un nivel de medio-píxel o cuarto de píxel, que conlleva un aumento tanto de la complejidad como de los tiempos de cálculo.

En el desarrollo de este capítulo se describirá el diseño del bloque IP que contendrá el algoritmo de estimación de movimiento, comenzando por el modelado del código de alto nivel que lo define. Además, se partirá de un exhaustivo análisis de resultados mediante la síntesis de alto nivel que proporciona Vivado HLS, para la elección de las optimizaciones oportunas que mejoren el rendimiento del diseño.

Por último, se expondrá la verificación de la descripción *hardware* generada y su encapsulado IP, dejando el bloque preparado para su integración en la plataforma completa en la siguiente etapa del flujo de diseño.

4.1 Diseño del algoritmo de estimación de movimiento

Como se ha comentado anteriormente, el objetivo del bloque IP a desarrollar será proporcionar los vectores de desplazamiento resultantes de la estimación de movimiento entre los *frames* de una secuencia de vídeo mediante el algoritmo *New Three Step Search*, explicado en la Sección 2.2.3.

Para ello, se sigue la metodología de diseño basada en síntesis de alto nivel, explicada previamente en la Sección 3.2.1, utilizando la herramienta de diseño Xilinx Vivado HLS. De esta forma, utilizando el lenguaje de programación C y C++, se irá

implementando el algoritmo y comprobando su funcionalidad a través del depurado y verificación mediante simulación de alto nivel.

4.1.1 Descripción del algoritmo en alto nivel

En primer lugar, antes de comenzar con la explicación del algoritmo implementado, cabe destacar que, pese a que los más recientes codificadores de vídeo como los comentados en el capítulo 2 son capaces de alcanzar una precisión *half-pixel* e incluso *quarter-pixel*, se ha decidido no bajar a tal nivel de precisión, fijando nuestra estimación de movimiento a una precisión *full-pixel*. Esta decisión ha sido tomada teniendo en cuenta que este proyecto está orientado a la codificación de vídeo en satélite, donde el movimiento local de los píxeles no será tenido en cuenta, primando el movimiento de la cámara a lo largo de la órbita y, por consiguiente, no se producirán situaciones en las que sea necesaria una precisión mayor a la que supone el movimiento píxel a píxel.

Para la implementación del algoritmo se han creado dos archivos principales donde se encontrará toda la funcionalidad del diseño: *blockMatchingNTS.c*, en el que se describirá todo el proceso de la estimación de movimiento, a implementar en la FPGA; y *functions.c*, donde se alojarán las distintas funciones auxiliares utilizadas en el algoritmo. Esta organización permite tener separado el algoritmo principal del resto de las funciones utilizadas.

El flujo de diseño seguido para llevar a cabo la estimación de movimiento de todos y cada uno de los macrobloques que componen cada uno de los frames de la secuencia de vídeo, se encuentra ilustrado de forma resumida en la figura 4.1. Para lograr una clara comprensión sobre el funcionamiento del diseño y contar con una mejor organización de la descripción del código desarrollado, se explicarán primero las funciones secundarias utilizadas, para posteriormente describir la funcionalidad principal del diseño, el algoritmo NTSS.

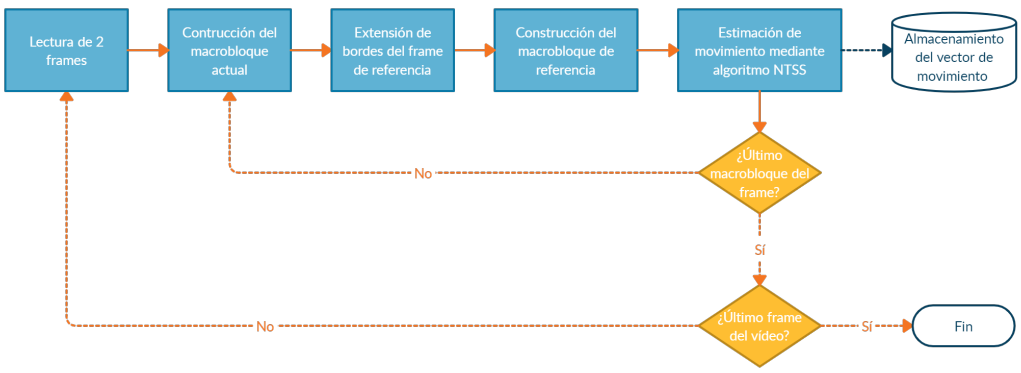

Fig. 4.1: Proceso seguido en el diseño

extendBorders()

Como su nombre indica, esta función se encarga de extender los bordes del *frame* utilizado como referencia; concretamente, el *frame* anterior al *frame* bajo análisis. Esto es algo necesario ya que, cuando se realiza la estimación de movimiento de un macrobloque, se comprueba su semajanza respecto a distintas posiciones que este MB podía tener en el anterior *frame* y, en el caso de los MBs situados en los bordes de la imagen, algunas de los posibles ubicaciones comprobadas serían con píxeles inexistentes, fuera de los bordes de la imagen. Para evitar errores en la estimación de movimiento debido a ello, se crea un *frame* ficticio de referencia, el cual es idéntico al *frame* real pero añadiendo píxeles al exterior de la imagen y aumentando así el tamaño del *frame*. Así, el tamaño del nuevo *frame* de referencia será igual al anterior, añadiendo el valor correspondiente a la ventana de búsqueda a cada lado del rectángulo que forma el *frame*; es decir, 4x8, ya que la ventana de búsqueda es fijada a un valor de 8 píxeles, teniendo en cuenta que es la mitad de los 16 que componen cada MB, como ya se explicó en la sección 2.2.2. Los píxeles creados alrededor del *frame* se obtendrán mediante el valor del píxel real más cercano situado en esa fila o columna.

De esta forma, la función consta de dos parámetros: por un lado, el *frame* de referencia extraido de la secuencia de vídeo; y, por otro lado, el *frame* resultante de esta extensión de bordes. En primer lugar, se utilizan los píxeles de referencia para construir el interior del nuevo *frame* y, posteriormente, se realiza la creación de los píxeles ficticios que se situarán fuera de los bordes de la imagen, partiendo del valor de los píxeles situados en los bordes del *frame* de referencia. En otras palabras, cuando se crean los píxeles situados a la izquierda del *frame* real, los valores recogidos serán los que se encuentren en las respectivas filas de la primera columna del *frame* de referencia. De manera análoga, se realiza la creación de los píxeles situados a la derecha de la imagen, en la parte superior y en la parte inferior. Finalmente, la función se encarga de dar valor a los píxeles situados en las cuatro esquinas de la imagen. Para una mejor comprensión de la función, se incluye la Figura 4.2, donde se puede ver con un sencillo ejemplo cómo se obtienen los valores de los píxeles creados.

sadInter()

Esta función es la encargada de calcular el SAD entre dos macrobloques determinados, que son especificados por parámetros. Se trata de una función bastante sencilla que devuelve el valor resultante de la suma de absolutas diferencias entre cada ubicación de los píxeles de ambos MBs. El valor obtenido de cada píxel es

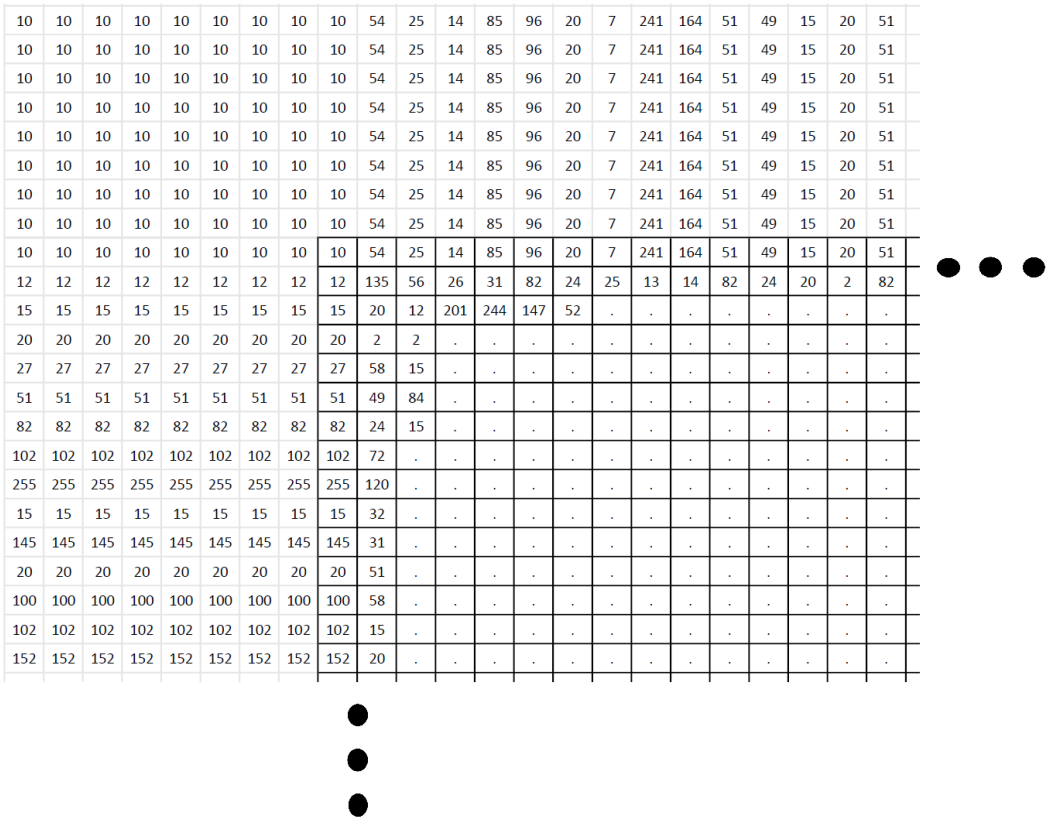

Fig. 4.2: Ejemplo ilustrativo sobre la creación de píxeles externos al frame

representado como un *char* sin signo, que equivale a un número entre 0 y 255. Este rango de valores hace referencia a la luminosidad del píxel en la imagen, teniendo un contraste más oscuro los píxeles con valores más cercanos a 0 y con un contraste más claro los valores cercanos al 255. Todo ello en escala de grises, ya que se trata únicamente de la componente de luminancia de la imagen.

blockMatchingNTS()

Finalmente, es en esta función donde reside el algoritmo *New Three Step Search*, haciendo uso de las funciones previamente explicadas.

Una vez se encuentra preparado el *frame* de referencia con la extensión de bordes realizada mediante la función *extendBorders* para poder construir los macrobloques de referencia, se puede considerar ya el comienzo del proceso de estimación de movimiento. El primer paso del algoritmo *NTSS* comienza con la creación de los vectores de movimiento candidatos a través de un array bidimensional. De estos primeros 17 vectores candidatos, tal y como se aprecia en la figura 4.3, 8 de ellos hacen referencia a un movimiento de un solo píxel en todas las direcciones, otros 8 para marcar un movimiento en esas mismas direcciones pero con una distancia de

recorrido equivalente a la mitad de la ventana de búsqueda (valor de *d*), cuya razón está detalladamente explicada en la sección 2.3.3 (añadir en la explicacion del ntss, el porqué se utiliza este tamaño (por las 3 step, d/2, d/2, 1, llegando así al limite de la ventana de búsqueda)); y un vector (0,0), que indicaría que el macrobloque no presenta movimiento alguno.

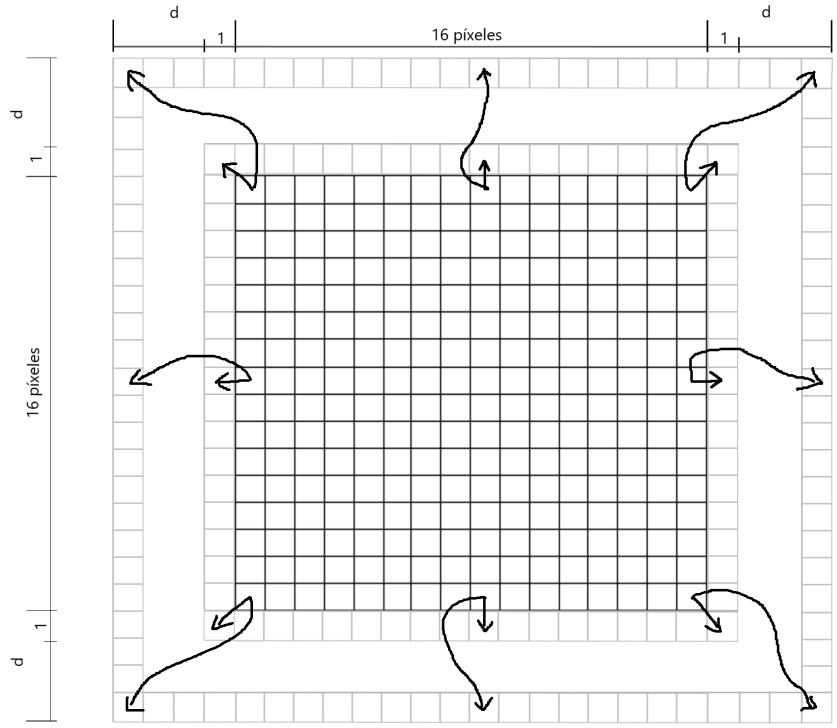

Fig. 4.3: Representación de los distintos movimientos considerados en la primera etapa del algoritmo NTSS

A continuación tiene lugar la elección del vector candidato que mejor se ajusta al movimiento descrito por el macrobloque. En primer lugar, se construye un bloque de referencia para cada vector candidato, utilizando el *frame* de referencia anteriormente construido y los parámetros de entrada *startRow* y *startCol*, para especificar la ubicación de los píxeles que se extraerán del *frame* para formar el MB, añadiendo el desplazamiento ejercido por el vector de movimiento en cuestión.

En segundo lugar, se calcula el SAD entre el bloque actual y el bloque de referencia recién construido mediante la función *sadInter*. El resultado será un valor númerico que denotará la magnitud de la diferencia entre ambos macrobloques, siendo mayor cuanto mayor sea dicho valor, y viceversa.

Para finalizar esta primera etapa del algoritmo, se compara el valor de SAD obtenido con el menor que se haya obtenido con anterioridad (*sadInterMin*). En caso de que el SAD obtenido sea menor que *sadInterMin*, significará que el vector de movimiento testeado resulta ser el óptimo hasta el momento, y por ello se actualiza el SAD

mínimo y la variable que almacena el vector de movimiento que mejor representa el desplazamiento realizado por el MB.

```
// Para cada vector candidato con indice k...
for (k = 0; k < numCandidates; k++)
{
  // Construccion del bloque de referencia
  for (i = 0; i < refBlockRows; i++)
  {
    for (j = 0; j < refBlockCols; j++)
    {
      refBlock[i][j] = refFrameBorders
  [startRow+SEARCH_AREA+1+candidates[k][0]+i]
  [startCol+SEARCH_AREA+1+candidates[k][1]+j];
    }
  }
  // Calculo del SAD entre el bloque actual y de referencia
  sadInterAux = sadInter(currentBlock, refBlock);

  /* Si el valor es menor al obtenido anteriormente, el vector
  actual comprobado es el mas optimo*/
  if (sadInterAux < sadInterMin)
  {
    sadInterMin = sadInterAux;
    mvFull[0] = candidates[k][0];
    mvFull[1] = candidates[k][1];
    vectorIndex = k;
  }
}
```

Listing 4.1: Cálculo del SAD y actualización del vector de desplazamiento

Tras concluir este primer paso, comienza la segunda etapa en la que se pueden dar dos condiciones: que el vector de movimiento que resultaba en un mejor SAD estuviera dirigido a los píxeles contiguos o a una distancia d. En primer lugar, se contemplará este último caso y se presentarán 8 nuevos vectores de movimiento candidatos, que se crearán a partir del vector previamente elegido, añadiendo una nueva distancia de la mitad que la anterior, es decir, 1/4 de la ventana de búsqueda. Con estos vectores candidatos, se realiza el mismo proceso realizado en la primera etapa y se actualiza la variable en la que se almacena el vector de movimiento elegido, así como el SAD mínimo. Estos nuevos vectores considerados se pueden ver ilustrados en la figura 4.4, donde la dirección representada en rojo supone un ejemplo de vector de movimiento resultante de la primera etapa.

De esta forma, comienza el último de los tres pasos en el que se actualiza la d a 1, comprobando los vectores de movimiento que apuntan a los píxeles adyacentes al último vector candidato escogido, tal y como se ilustra en la figura 4.5.

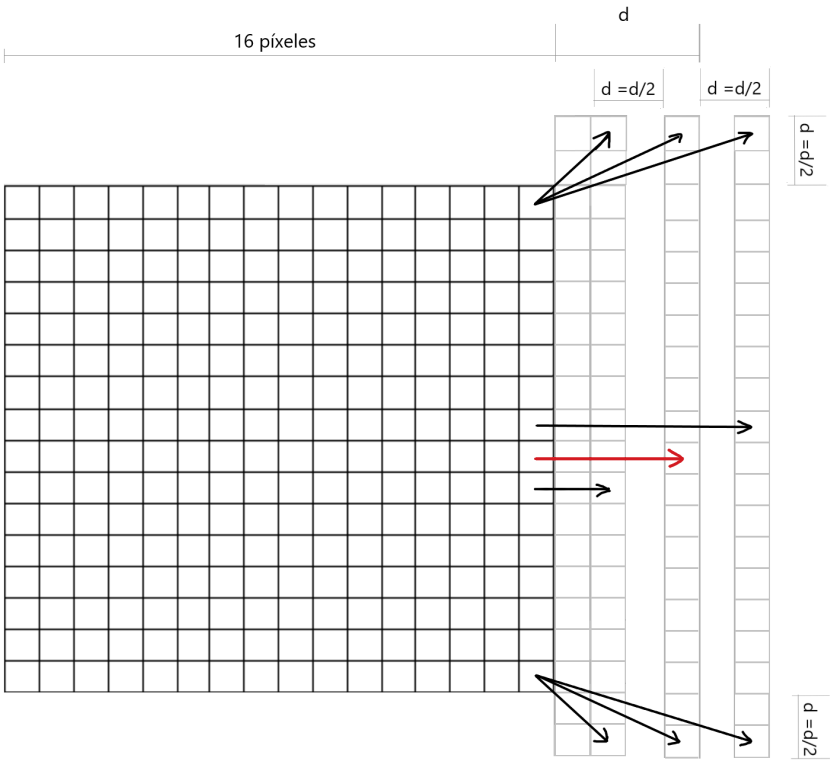

Fig. 4.4: Ejemplo de vectores de movimiento considerados en la segunda etapa del algoritmo NTSS

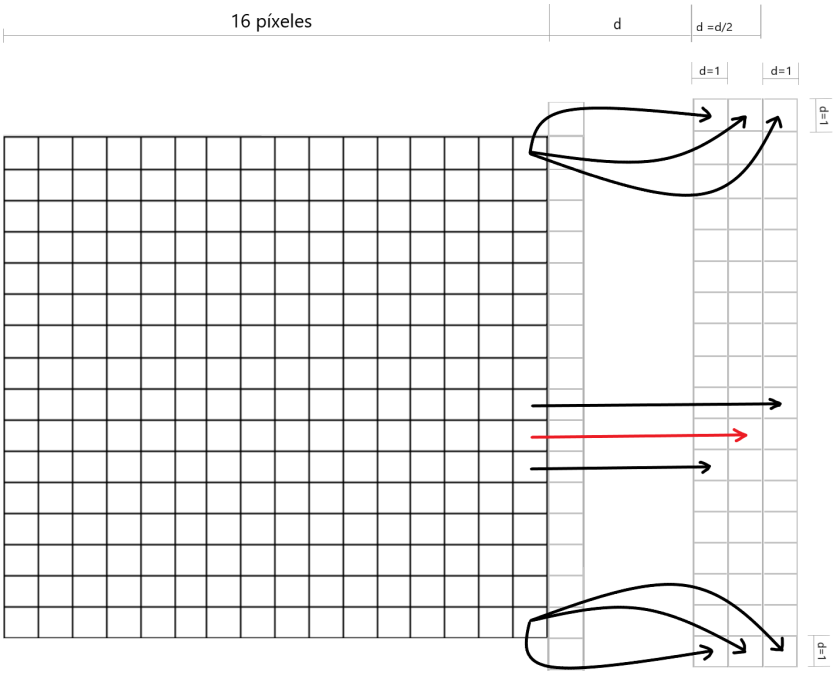

Fig. 4.5: Ejemplo de vectores de movimiento considerados en la tercera etapa del algoritmo NTSS

Así, solo quedaría contemplar el caso en el que el primer vector de movimiento candidato escogido señalara a los píxeles contiguos al macrobloque bajo análisis. En

este caso, se consideran como nuevos vectores candidatos los píxeles adyacentes al vector seleccionado, evitando aquellas posiciones que ya se hayan podido comprobar en la primera etapa.

Una vez desarrollado el algoritmo *New Three Step Search* y las distintas funciones utilizadas, se pasa a la verificación del funcionamiento del sistema, comprobando los resultados obtenidos mediante simulación de alto nivel.

4.1.2 Depurado y verificación mediante simulación de alto nivel

Para la verificación del algoritmo desarrollado, se ha diseñado un *testbench* en el que se seguirán los siguientes pasos:

En primer lugar, para realizar la estimación de movimiento de una secuencia de vídeo, lo primero es obtener dicha secuencia de imágenes. Ese es el objetivo de la función *loadYUV*, obtener la información del vídeo a codificar con formato *.yuv*. Concretamente, la componente de luminancia de los píxeles que constituyen cada *frame*, ya que es la única información necesaria para la estimación de movimiento.

Así, se realiza una llamada a dicha función para la obtención del primer *frame*, almacenándolo en un *array* bidimensional (*refSequenceY*) en el que se encontrará el *frame* de referencia en todo momento. De esta forma, tiene lugar la extensión de bordes de este *frame* de referencia mediante la función *extendBorders* para asegurar la correcta estimación de movimiento de aquellos macrobloques situados en los bordes de la imagen.

A continuación, se realizará la creación de los macrobloques que conformarán el *frame* considerado como actual en cada iteración. Además, se construirán los macrobloques de referencia en función de la posición del MB actual dentro del *frame*. Con ello, podrá ejecutarse la función *blockMatchingNTS* y obtener el vector de movimiento que mejor representa el movimiento seguido por el MB bajo análisis.

Al finalizar la estimación de movimiento de cada macrobloque que compone el *frame* actual, se almacenará el vector de movimiento resultante en una variable que será escrita en un archivo de texto (*.txt*). Este fichero contendrá todos los vectores de movimiento resultantes de la estimación de movimiento, especificándose a qué macrobloque corresponde y el índice de los *frames* que están siendo comparados, tal y como se puede observar en la figura 4.6.

```
**Vectores de movimiento generados gracias al bloque IP blockMatchingNTS**
Current frame 2  Reference frame 1
MB.1:    0    -1    MB.2:    0    -2    MB.3:    0    -2    MB.4:    2    -2    MB.5:    0    -2
MB.23:   0    -2    MB.24:   0    -2    MB.25:   0    -2    MB.26:   0    -2    MB.27:   0    -2
MB.45:   0    -3    MB.46:   0    -2    MB.47:   0    -2    MB.48:   0    -2    MB.49:   0    -2
MB.67:   0    -2    MB.68:   0    -2    MB.69:   0    -2    MB.70:   0    -2    MB.71:   0    -2
MB.89:   0    -2    MB.90:   0    -2    MB.91:   0    -2    MB.92:   0    -2    MB.93:  -1    -2
MB.111:  0    -2    MB.112:  0    -2    MB.113:  0    -2    MB.114: -1    -2    MB.115:  0    -2
MB.133:  0    -2    MB.134: -1    -2    MB.135:  0    -2    MB.136:  0    -2    MB.137:  0    -2
MB.155:  0    -2    MB.156:  0    -2    MB.157:  1    -2    MB.158:  0    -2    MB.159:  0    -2
MB.177:  0    -2    MB.178: -1    -2    MB.179: -1    -2    MB.180: -1    -2    MB.181:  0     0
MB.199:  0    -2    MB.200: -1    -2    MB.201: -1    -2    MB.202:  0     0    MB.203:  0     0
MB.221:  0     0    MB.222:  0    -2    MB.223:  0     0    MB.224:  0     0    MB.225:  0     0
MB.243:  0     0    MB.244:  0     0    MB.245:  0     0    MB.246:  0     0    MB.247:  0     0
MB.265:  0     0    MB.266:  0     0    MB.267:  0     0    MB.268:  0     0    MB.269:  0     0
MB.287:  0    -2    MB.288:  0     0    MB.289:  0     0    MB.290:  0     0    MB.291:  0     0
MB.309:  0    -2    MB.310:  0    -2    MB.311:  0     0    MB.312:  0     0    MB.313:  0     0
MB.331:  0    -2    MB.332:  1    -2    MB.333:  0    -2    MB.334:  0    -2    MB.335:  0     0
MB.353:  0    -2    MB.354:  0    -2    MB.355:  2    -2    MB.356:  0     0    MB.357:  0     0
MB.375:  0    -2    MB.376:  0    -2    MB.377:  0     0    MB.378:  0     0    MB.379:  0     0

Current frame 3  Reference frame 2
MB.1:    0    -3    MB.2:    2    -2    MB.3:    2    -2    MB.4:    2    -2    MB.5:    0    -2
MB.23:   0    -2    MB.24:   0    -2    MB.25:   1    -2    MB.26:   0    -2    MB.27:   0    -2
MB.45:   0    -2    MB.46:   0    -2    MB.47:   0    -2    MB.48:   0    -2    MB.49:   0    -2
MB.67:   0    -2    MB.68:   0    -2    MB.69:   0    -2    MB.70:   0    -2    MB.71:   0    -2
MB.89:   0    -2    MB.90:   0    -2    MB.91:   0    -2    MB.92:  -1    -2    MB.93:   0    -5
MB.111:  0    -2    MB.112:  0    -2    MB.113:  0    -2    MB.114: -1    -2    MB.115:  0    -2
MB.133:  0    -2    MB.134:  0    -2    MB.135:  0    -2    MB.136:  0    -2    MB.137:  0    -2
MB.155:  0    -2    MB.156:  0    -2    MB.157:  0    -2    MB.158:  0    -2    MB.159:  0    -2
MB.177:  0    -2    MB.178:  0    -2    MB.179:  0    -2    MB.180: -1    -2    MB.181:  0     0
MB.199:  0    -2    MB.200:  0    -2    MB.201:  0    -2    MB.202:  1    -1    MB.203:  0     0
MB.221:  0     0    MB.222:  0    -2    MB.223:  0     0    MB.224:  0     0    MB.225:  0     0
MB.243:  0     0    MB.244:  0     0    MB.245:  0     0    MB.246:  0     0    MB.247:  0     0
MB.265:  0    -1    MB.266:  0     0    MB.267:  0     0    MB.268:  0     0    MB.269:  0     0
MB.287:  0    -2    MB.288:  0    -1    MB.289:  0     0    MB.290:  0     0    MB.291:  0     0
MB.309:  0    -2    MB.310:  0    -2    MB.311: -1    -2    MB.312:  0    -1    MB.313:  0     0
MB.331: -1    -2    MB.332:  0    -2    MB.333:  0    -1    MB.334:  0    -2    MB.335:  0     0
MB.353:  0    -2    MB.354:  0    -2    MB.355:  2    -1    MB.356: -1     0    MB.357:  0     0
MB.375:  0    -2    MB.376:  0    -2    MB.377:  0     0    MB.378:  0     0    MB.379:  0     0
```

Fig. 4.6: Ejemplo de archivo obtenido de la estimación de movimiento de una secuencia de vídeo

Estos vectores de movimiento recogidos en el archivo *motionVectors.txt* serán comparados con los vectores de movimiento resultantes de un *software* de referencia sobre un algoritmo de súper-resolución desarrollado por el grupo de investigación [38] en el que se encuentra una etapa de estimación de movimiento que genera un fichero similar al creado en el presente trabajo.

Gracias al modo *debug* que nos proporciona la herramienta de Vivado HLS, ha sido posible ir comprobando línea a línea que el código desarrollado funcionaba correctamente, desde la lectura de los *frames* y construcción de los distintos macrobloques hasta el proceso de estimación de movimiento seguido en el algoritmo NTSS y generación del vector de movimiento resultante.

4.1.3 Optimización mediante partición hardware/software

Una vez se ha comprobado el correcto funcionamiento de nuestro algoritmo de estimación de movimiento, el siguiente paso será llevar a cabo una partición *hardware/software* mediante la cual poder explotar las posibilidades ofrecidas por los dispositivos Zynq, ejecutando una parte del diseño sobre la FPGA (posibilidad de paralelismo si no existe dependencia de datos) y la otra parte sobre uno de los

procesadores ARM disponibles en el dispositivo (mayor frecuencia de reloj, óptimo para ejecución de instrucciones secuenciales).

De esta forma, desde el dominio *software* (en HLS, el *testbench*, que simulará el proceso ejecutado en el sistema final por uno de los procesadores del ARM) se leerán los *frames* de una secuencia de vídeo, se realizará la extensión de bordes y, a partir de ellos, se irán construyendo tanto los macrobloques considerados como actuales, como los respectivos bloques de referencia. Estos últimos se encuentran situados en una posición análoga al MB actual, pero dentro del *frame* de referencia y añadiendo un número de píxeles a cada lado del bloque igual al tamaño de la ventana de búsqueda. Ambos MBs son facilitados a la función que realiza la estimación de movimiento y, una vez realizada esta estimación en el dominio *hardware*, se van incluyendo los vectores de movimiento obtenidos en un archivo de texto que es comparado con los obtenidos por un *software* de referencia.

```
    for (r = 0; r < numBlocksRows; r++)
    {
      startRow = r*MB_SIZE;
      for(c = 0; c < numBlocksCols; c++)
      {
        startCol = c*MB_SIZE;
        int actualBlock = (r*numBlocksCols + c + 1);
        for(i = 0; i < MB_SIZE; i++)
        {
          for(j = 0; j < MB_SIZE; j++)
          {
            // Build the current macroblock
            currentBlock[MB_SIZE*i + j] =
currentSequenceY[COLS_FRAME*(i + startRow) + j + startCol];
          }
        }
        extendBorders(refSequenceY, refFrameBorders);
        for(i = 0; i < refBlockBordersRows; i++)
        {
          for(j = 0; j < refBlockBordersCols; j++)
          {
            // Build the current macroblock with borders
            refBlockBorders[refBlockBordersCols*i + j] =
refFrameBorders[refFrameBordersCols*(i + startRow) + j +
startCol];
          }
        }
        blockMatchingNTS(refBlockBorders, currentBlock, mv);
        fprintf(motionVectors, "MB.%d:\t%3d\t%3d\t", actualBlock,
mv[0], mv[1]);
      }
      fprintf(motionVectors, "\n");
    }
    fprintf(motionVectors, "\n");
```

Listing 4.2: Código principal del testbench diseñado

Por otro lado, el dominio *hardware* estaría representado por la función *blockMatchingNTS*, bloque IP que albergará el algoritmo de estimación de movimiento. Este bloque estará constituido por dos entradas, el macrobloque de referencia extendido por la ventana de búsqueda (*(MBsize+2*SearchWindow)x(MBsize+2*SearchWindow)*) y el macrobloque actual; y por una salida, el vector de desplazamiento resultante de la estimación de movimiento. El diseño del bloque quedaría de esta manera ilustrado en la figura 4.7.

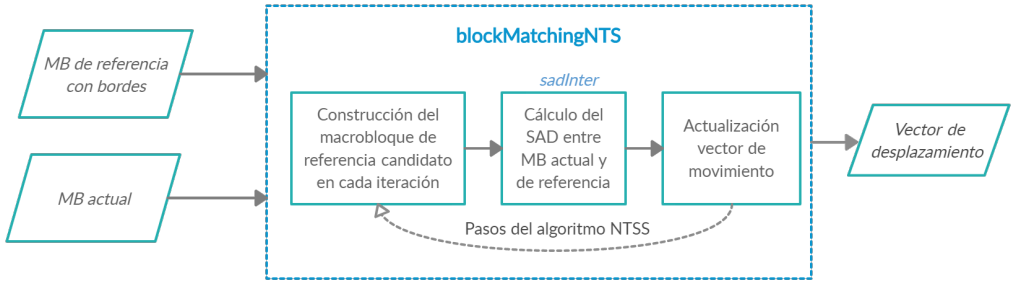

Fig. 4.7: Diagrama del bloque IP diseñado

Así, dentro del algoritmo NTSS se irá construyendo el MB de referencia oportuno a partir del recibido como argumento de la función, dependiendo del vector de movimiento bajo análisis en cada iteración para su comparación con el MB actual bajo estudio.

```
for (i = 0; i < refBlockRows; i++)
{
  for (j = 0; j < refBlockCols; j++)
  {
    refBlock[refBlockCols*i + j] =
refBlockBorders[refBlockBordersCols*(SEARCH_AREA +
candidates[k][0] + i) + SEARCH_AREA + candidates[k][1] + j];
  }
}
```

Listing 4.3: Construcción del macrobloque de referencia

Por último, como se ha podido apreciar en el código expuesto en el desarrollo de esta subsección, es oportuno destacar que se ha decidido sustituir todos los arrays bidimiensionales utilizados para almacenar los distintos *frames* y macrobloques por arrays unidimensionales, con el objetivo de obtener una descripción más *hardware-friendly*, facilitando los accesos a memoria y la anidación de bucles, evitando así posibles errores futuros en la conversión a lenguaje HDL.

No obstante, para intentar ejecutar la mayor parte del diseño dentro del apartado *hardware* y validar que la partición *hardware/software* realizada es correcta, se decidió realizar un nuevo diseño en el que la extensión de bordes se realizara dentro del bloque IP (figura 4.8). De esta forma, se necesitaría recibir el *frame* de referencia completo a la entrada del bloque, así como la posición del macrobloque actual dentro del *frame* mediante las entradas *startCol* y *startRow* para poder crear los macrobloques de referencia en la ubicación correspondiente.

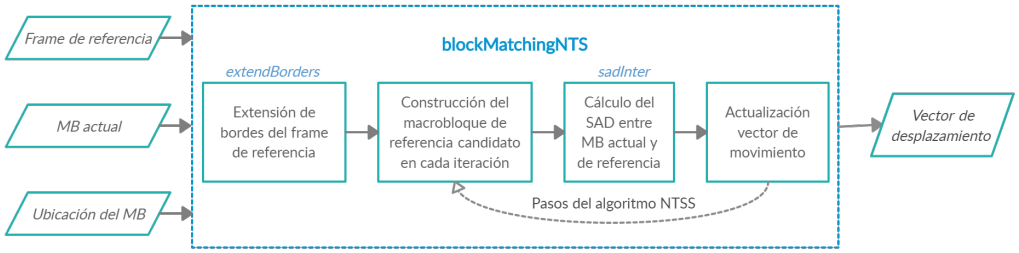

Fig. 4.8: Diagrama del bloque IP modificado

Con esta versión, se realizó la primera síntesis y análisis de los resultados obtenidos, los cuales serán comentados a continuación. De forma anticipada, es preciso mencionar que los resultados obtenidos no fueron satisfactorios y, por ello, se volvió al diseño descrito al inicio de este apartado.

4.2 Síntesis de alto nivel y análisis de resultados

4.2.1 Análisis del diseño con extensión de bordes en el bloque IP

A continuación, nos disponemos a analizar los resultados obtenidos en la síntesis de alto nivel del diseño expuesto en la figura 4.8, algo de vital importancia para obtener una estimación previa de los recursos que se consumirían en la FPGA al ejecutar el código desarrollado, y la latencia que presentará dicha ejecución.

Al realizar esta primera síntesis, se obtuvieron unos resultados bastante alejados de lo esperado. La síntesis de alto nivel realizada resultó en unos valores de latencia del orden de 250.000 ciclos de reloj (figura 4.9) para una secuencia de vídeo con una resolución de 352x288 píxeles, un valor excesivamente elevado que convertiría la estimación de movimiento en un proceso muy lento y costoso computacionalmente, convirtiéndose en el principal cuello de botella del compresor de vídeo. Además, dada la dependencia que tiene este diseño con el tamaño de los *frames*, ya que el bloque IP de este análisis obtiene el *frame* de referencia como entrada, todos estos

valores obtenidos en la síntesis se verían incrementados al analizar secuencias de vídeo de mayor resolución.

Latency		Interval		
min	max	min	max	Type
242708	268650	242708	268650	none

Fig. 4.9: Latencia obtenida en la síntesis de la primera versión

Asimismo, como se puede corroborar en la figura 4.10, pese a que el consumo de LUTs del dispositivo resultaba ser bajo, de un 8 %, la utilización de memoria se elevaba a un 24 %. Este consumo de memoria, como se puede comprobar en la Figura 4.11, se ve principalmente incrementado por la variable *refFrameBorders*, la cual es la que almacena el resultado de la extensión de bordes del *frame* de referencia necesario para la creación de los macrobloques de referencia.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	4	-	-	-
Expression	-	-	0	1867	-
FIFO	-	-	-	-	-
Instance	-	0	478	1323	-
Memory	69	-	0	0	0
Multiplexer	-	-	-	1256	-
Register	-	-	1323	-	-
Total	69	4	1801	4446	0
Available	280	220	106400	53200	0
Utilization (%)	24	1	1	8	0

Fig. 4.10: Recursos consumidos en la síntesis de la primera versión

Memory	Module	BRAM_18K	FF	LUT	URAM	Words	Bits	Banks	W*Bits*Banks
candidates_0_U	blockMatchingNTS_bkb	2	0	0	0	17	32	1	544
candidates_1_U	blockMatchingNTS_bkb	2	0	0	0	17	32	1	544
refBlock_U	blockMatchingNTS_dEe	1	0	0	0	256	8	1	2048
refFrameBorders_U	blockMatchingNTS_eOg	64	0	0	0	113220	8	1	905760
Total	4	69	0	0	0	113510	80	4	908896

Fig. 4.11: Consumo de memoria BRAM en la síntesis de la primera versión

Analizando estos resultados y considerando que la función que alberga el algoritmo de estimación de movimiento no es excesivamente compleja para consumir una cantidad de recursos lógicos tan elevada, se decidió volver al diseño inicial (figura 4.7). Además, la extensión de bordes es una función que carece de sentido realizar dentro del bloque *blockMatchingNTS*, ya que esto conlleva que cada vez que se realice la estimación de movimiento entre dos macrobloques sea necesaria la extensión de bordes de todo el *frame*, traduciéndose en un incremento de latencia totalmente innecesario. En otras palabras, partiendo de la resolución utilizada de 372x288 píxeles y el tamaño de cada MB (16x16 píxeles), cada *frame* consta de 396

macrobloques (22x18), por lo que esta extensión de bordes se ejecutaría 395 veces más de lo necesario al analizar cada *frame*. En cambio, realizando esta extensión de bordes en el momento en el que se lee el *frame* de referencia desde el dominio *software*, este proceso solo es realizado una vez para la estimación de movimiento de cada imagen completa.

Además, volviendo al diseño inicial, se evita tener variables de excesivo tamaño dentro del bloque IP que ocasionaban una alta utilización de la memoria BRAM, como es el caso de la variable *refFrameBorders*, que ya no será necesaria al no realizar la extensión de bordes en *hardware*. Con esto, disponemos de un bloque IP simplificado cuyas únicas entradas son el macrobloque bajo análisis y un macrobloque de referencia. De esta forma, el tamaño de los *frames* y el número de imágenes que conforman el vídeo serán totalmente transparentes al funcionamiento de nuestro bloque IP; en otras palabras, sea cual sea la resolución y la duración del vídeo a codificar, el algoritmo desarrollado se comportará de forma lineal, recibiendo dos macrobloques y generando el vector de movimiento que representa el movimiento sufrido por el MB considerado como actual.

4.2.2 Modelado de interfaces

Una vez se ha desarrollado correctamente nuestro algoritmo en lenguaje de alto nivel, se comienza su adaptación al entorno *hardware*, donde el modelado de interfaces de entrada y salida de nuestro bloque IP supone el primer paso.

Para ello, se hará uso del protocolo *Advanced eXtensible Interface (AXI)*, concretamente AXI4, para las interfaces que conectarán nuestro bloque IP con el resto de bloques que se utilizarán en el diseño de la plataforma. Existen 3 tipos de interfaces AXI4:

- AXI4: centrada en aplicaciones en las que se requiera un alto rendimiento en el mapeo de memoria.
- AXI4-Lite: utilizada para comunicaciones simples, en las que no se necesiten accesos de memoria con un alto *throughput*. Suelen estar orientadas a la implementación de puertos de configuración.
- AXI4-Stream: orientada a transmisiones de datos de alta velocidad (aplicaciones en *streaming* o tipo *dataflow*).

Tal como observa en la Figura 4.12, tanto los dos macrobloques de entrada (*refBlockBorders* y *currentBlock*) pasados como entrada a la función *top* (el bloque NTSS) como el vector de movimiento resultante (*mv*) serán definidos como interfaces *AXI4-*

Stream mediante la directiva *INTERFACE axis*. Gracias a este protocolo, se obtiene una alta velocidad de transmisión con transferencias en modo ráfaga, permitiendo así una comunicación directa y sin restricciones entre nuestro bloque IP y el DMA.

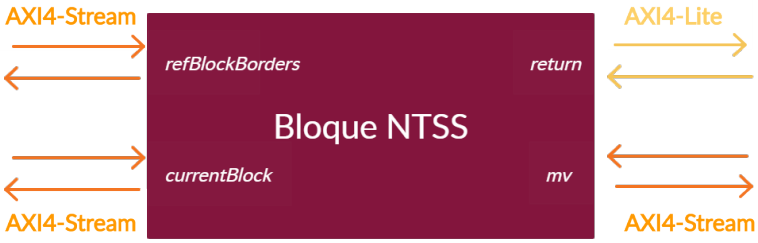

Fig. 4.12: Diagrama de E/S del bloque IP

Para el uso de esta interfaz, los argumentos suelen ser de tipo *ap_axis*, ya que genera todas las señales dentro del estándar *AXI4-Stream*. Sin embargo, para simplificar el proceso, se decidió utilizar el tipo de datos estándar que se había estado empleando hasta el momento (*unsigned char*), lo cual no supondría ningún problema ya que al declarar los parámetros del bloque como interfaces *axis*, se generan automáticamente las señales *TDATA*, *TREADY* y *TVALID*. Por un lado, la señal *TDATA* incorporará los datos que se desean transmitir, es decir, los valores correspondientes a los píxeles que forman los dos macrobloques en el caso de las dos entradas; y el vector de movimiento resultante en el caso de la salida. Por otro lado, las señales *TVALID* y *TREADY* son las encargadas establecer la comunicación correctamente, completando un protocolo de tipo *handshaking* con el DMA. La señal *TVALID* es transmitida por el maestro para indicar que está comenzando una transferencia válida, mientras que la señal *TREADY* es enviada por el esclavo para advertir que puede aceptar una transferencia en el ciclo de reloj actual.

Así, solo quedaría la señal *TLAST* para contar con las señales del protocolo *AXI4-Stream* necesarias en nuestro sistema y poder disponer de una adecuada conexión con el DMA. Para solventar este inconveniente, se añade al diseño una señal *last* de tipo *ap_uint* de 1 bit para la salida del bloque, siendo manejada dentro de la función. Esta señal es transmitida por el maestro a nivel alto para informar del fin de una transmisión de datos. Por ello, en el caso de la salida del bloque (maestro), se pondrá su respectiva señal *last* a nivel alto cuando se termine la escritura del vector de movimiento estimado.

```
#pragma HLS INTERFACE ap_none port=last_o
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE axis register both port=currentBlock
#pragma HLS INTERFACE axis register both port=refBlockBorders
#pragma HLS INTERFACE axis register both port=mv
```

Listing 4.4: Declaración de interfaces

4.2.3 Análisis del diseño sin directivas de optimización

Una vez modeladas las interfaces que describirán las distintas E/S de nuestro bloque IP, es turno de analizar la estimación de resultados obtenidos en la síntesis de alto nivel del diseño considerado en la figura 4.7.

Realizando una comparativa entre las figuras 4.9 y 4.13, resulta muy notable la reducción de latencia que se ha obtenido tras los cambios realizados, pasando de una media de 250.000 ciclos de reloj a unos valores alrededor de los 33.000 ciclos de media.

Latency		Interval		
min	max	min	max	Type
22949	45915	22949	45915	none

Fig. 4.13: Resultados de latencia de la síntesis de alto nivel

En lo respectivo al consumo de recursos lógicos de la FPGA que supone la ejecución del algoritmo de estimación de movimiento, se puede comprobar en la figura 4.14 como se ha reducido ligeramente la utilización de FFs y LUTs respecto a los obtenidos en la sección 4.2.2. No obstante, el aspecto más destacable es la reducción de consumo de BRAMs que se ha logrado tras eliminar la etapa de extensión de bordes del bloque IP. Como ya se había podido comprobar en la Figura 4.11, la mayor parte de la memoria utilizada se debía a una variable en la que se almacenaba un *frame* completo al que se le había realizado la extensión de bordes. Así, sin la existencia de esta variable, se ha obtenido una disminución en el consumo de memoria de un 24 % a un 7 %.

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	1936	-
FIFO	-	-	-	-	-
Instance	0	-	106	258	-
Memory	7	-	0	0	0
Multiplexer	-	-	-	1377	-
Register	-	-	1394	-	-
Total	7	0	1500	3571	0
Available	280	220	106400	53200	0
Utilization (%)	2	0	1	6	0

Fig. 4.14: Consumo de recursos del diseño sin directivas de optimización

No obstante, pese a que se ha reducido significativamente la latencia del diseño, se ha considerado que estos resultados no se encuentran aún en los valores óptimos.

Gracias a contar con un consumo de recursos reducido, se presenta un gran margen para aplicar distintas directivas de optimización que elevarán este consumo en vistas de obtener unos resultados de latencia inferiores.

4.2.4 Análisis del diseño final aplicando directivas de optimización

Para este segundo caso de análisis, se han realizado distintas pruebas aplicando diferentes directivas de optimización como *unroll* y *pipeline* en los diferentes bucles que componen el algoritmo NTSS. Como se ha podido observar en los apartados 4.1.1 y 4.2.3, la función principal presenta una jerarquía de bucles que se repite a lo largo del algoritmo para recorrer y construir los macrobloques de referencia por fila y columna. Por tanto, las pruebas realizadas con las directivas de optimización mencionadas se han llevado a cabo tanto en los bucles interiores como en los exteriores para poder analizar los resultados y así poder obtener una solución equilibrada en términos de latencia y recursos lógicos consumidos.

Tras tantear todas las opciones se ha decidido realizar un *unroll* en los bucles interiores, dado que esta directiva supone la mejor opción en términos de latencia, a expensas de un aumento en el consumo de recursos lógicos directamente proporcional al número de iteraciones del bucle. Aun así, el consumo de LUTs se mantiene por debajo de un 20 %, el límite que se ha considerado para nuestro diseño, teniendo en cuenta que este trabajo se trata únicamente de la etapa de estimación de movimiento que formaría parte de un compresor de vídeo completo que sería implementado en la FPGA. La opción de utilizar esta directiva en los bucles exteriores quedó descartada ya que no mejoraba en gran medida la latencia pero sí incrementaba sustancialmente el consumo de recursos lógicos a valores superiores al 50 %, al ser la condición de ruptura mayor (es decir, se realizan mayor número de iteraciones), que en el caso de los bucles interiores.

Siguiendo este análisis, se optó por optimizar la función *sadInter* que tantas veces es ejecutada durante el algoritmo *New Three Step Search*. En este caso, dicha función, ya explicada en la sección 4.1.1, presenta una jerarquía de dos bucles a los que se ha aplicado *unroll* y *pipeline* al bucle interior y exterior, respectivamente. Con esto, obtenemos una latencia media de 7600 ciclos de reloj sin elevar el consumo de LUTs más allá de un 2 %, valores que dan por completada la optimización de nuestro diseño. En las figuras 4.15 y 4.16 se ilustran los resultados finales obtenidos en la síntesis de alto nivel.

Por último, cabe destacar que no se ha mencionado el consumo de BRAMs ni de FFs obtenidas tras realizar las optimizaciones señaladas durante esta sección debido a

Latency		Interval		
min	max	min	max	Type
4776	10542	4776	10542	none

Fig. 4.15: Latencia final de nuestro diseño

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6617	-
FIFO	-	-	-	-	-
Instance	0	-	312	1511	-
Memory	7	-	0	0	0
Multiplexer	-	-	-	2421	-
Register	-	-	1503	-	-
Total	7	0	1815	10549	0
Available	280	220	106400	53200	0
Utilization (%)	2	0	1	19	0

Fig. 4.16: Consumo de recursos del diseño final

que no han sufrido cambios significativos en el caso de los *Flip-Flops*, o directamente no ha variado su utilización, como es el caso de la memoria dedicada.

4.3 Verificación RTL y encapsulado IP

Para dar por concluido el proceso de modelado del bloque IP, es necesario verificar que la descripción RTL generada se comportará de forma análoga al código C/C++ realizado. Para ello, se emplea la herramienta de cosimulación que nos ofrece Vivado HLS, la cual construye un banco de pruebas a partir del *testbench* realizado en alto nivel, pero en formato *hardware* y ejecuta ambos procesos de forma paralela, verificando que se obtienen los mismos resultados.

Así, tiene lugar la cosimulación de nuestro diseño, donde se eligió *Verilog* como lenguaje *hardware* para la generación de la descripción RTL. Los resultados obtenidos fueron los esperados, sin reportarse ningún tipo de error. En la Figura 4.17 se puede observar como la ejecución de la descripción *hardware* de nuestro diseño requiere de una media de 7000 ciclos de reloj, un valor bastante similar a los obtenidos en la síntesis de alto nivel.

Para verificar el correcto funcionamiento del diseño en varios escenarios, se realizó la cosimulación estableciendo un número de *frames* igual a 5, 10 y 20. Aún así,

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	6071	7028	10559	6072	7029	10560

Fig. 4.17: Resultados de latencia de la cosimulación

tal y como se adelantó en la sección 4.2.2, se obtuvieron los mismos resultados de latencia, ya que el algoritmo se comporta de forma lineal, independientemente del número de *frames* que se procesen. Sin embargo, el tiempo que empleaba Vivado HLS para el proceso de cosimulación se incrementaba en gran medida según se aumentaba el número de *frames*, debido al trasiego de datos para proporcionar al bloque IP las entradas necesarias para realizar su función.

Por último, una vez dados por satisfactorios los resultados obtenidos en la cosimulación, se dispuso a empaquetar y exportar nuestro diseño como un bloque IP para que pueda ser manipulado desde la herramienta IP *Integrator* de Vivado IDE, algo de vital importancia para poder incluir nuestro módulo en la plataforma con los demás bloques necesarios para conformar el sistema final. Para ello, se emplea la opción de encapsulado IP que proporciona Vivado HLS y, de esta forma, ya solo habría que añadir el archivo generado al repositorio IP de Vivado IDE para poder interactuar con él e implementarlo dentro de la plataforma final.

4.4 Conclusiones

En este capítulo se ha descrito la primera parte del diseño de nuestro proyecto, el bloque IP que realiza la estimación de movimiento, comenzando por la descripción de la primera versión en lenguaje C/C++ del algoritmo NTSS y todas las funciones desarrolladas para su correcto funcionamiento.

Tras haber verificado el correcto funcionamiento del algoritmo desarrollado, tuvo lugar una optimización mediante la partición *hardware/software* que permitiría un mejor aprovechamiento de los recursos de los que dispone el dispositivo Zynq, ejecutando parte del desarrollo en la FPGA y otra en uno de los microprocesadores ARM disponibles en el PS. Gracias a ello, se logró obtener un algoritmo de estimación de movimiento de baja complejidad que no depende de la resolución ni duración de la secuencia de vídeo que se disponga a codificar.

Además, aplicando ciertas directivas de optimización a lo largo de los bucles que forman parte del algoritmo, se logró obtener unos valores de latencia bajos sin provocar un exceso en el consumo de recursos lógicos de la FPGA.

Finalmente, generando una descripción *hardware* que se ajusta al diseño implementado y llevando a cabo su proceso de exportación y encapsulado, nuestro bloque IP queda preparado para su integración en la plataforma mediante Vivado IDE.

5 Integración y verificación sobre la plataforma Pynq

Una vez descrito el diseño del bloque IP que realiza la estimación de movimiento, es turno de integrarlo dentro de la plataforma junto con los otros bloques que formarán parte del sistema completo. Se detallará la funcionalidad de cada uno de los módulos integrados, así como la comunicación entre ellos y el desarrollo de sus respectivas configuraciones, tanto en Vivado IDE como su programación en el SDK.

El bloque IP detallado en el capítulo anterior formará parte de un sistema empotrado sobre una plataforma configurable formada por un núcleo procesador ARM y la FPGA. Por una parte, la implementación *hardware* del algoritmo mediante técnicas HLS será ejecutada directamente en la FPGA; y, por otra parte, se desarrollará una aplicación *software*, ejecutada en el *core* ARM, mediante la cual se podrá llevar a cabo la configuración del sistema y el envío de la secuencia de vídeo desde la memoria DDR a la FPGA, y viceversa. De esta forma, realizando una correcta partición de las tareas entre estos dos dominios computacionales, se obtiene una solución de menor complejidad que optimiza los recursos disponibles, en función de la naturaleza interna de las tareas a ejecutar (dependencias de datos, capacidad de paralelizar, etc.).

Además, se recogerán los resultados más relevantes obtenidos durante la verificación completa del diseño, enfocándose principalmente en los tiempos de ejecución y en el consumo de recursos del sistema implementado en FPGA.

5.1 Arquitectura y estructura de la plataforma

La estructura de la plataforma a implementar se basará en la integración de varios bloques IP comunicados entre sí, asegurando que el sistema se encuentre perfectamente controlado y garantizando el correcto intercambio de los datos que circularán por los distintos módulos que conforman el diseño.

Como se puede observar en la Figura 5.1, el *software* empotrado es el encargado de realizar todas las transmisiones, tanto con la memoria DDR como con el bloque IP. El encargado principal de llevar a cabo estas comunicaciones es el bloque *Direct*

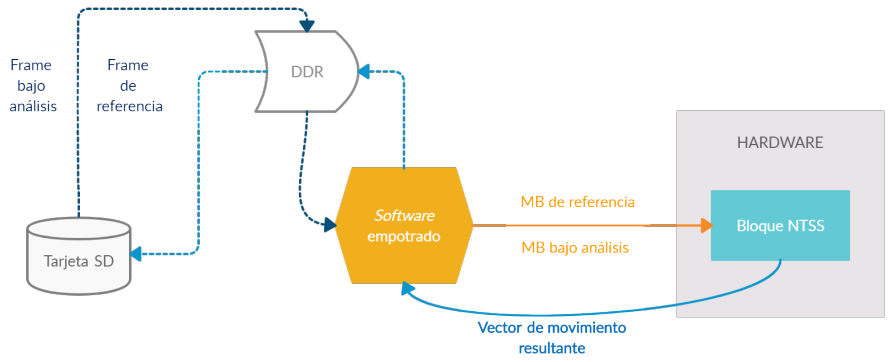

Fig. 5.1: Arquitectura del sistema empotrado

Memory Access (DMA) [39]. El DMA proporciona un alto ancho de banda entre la memoria asignada y el destino de la transmisión, utilizando la interfaz *AXI4 Stream* maestra MM2S (*Memory Mapped to Stream*) para el envío de datos y el esclavo S2MM (*Stream to Memory Mapped*) para la recepción (figura 5.2). Adicionalmente, este bloque IP permite el uso de hasta 16 canales de transmisión entre las dos rutas MM2S y S2MM, al habilitar el modo *Scatter-Gather*, algo necesario en nuestro diseño ya que el DMA estándar solo permite un canal de salida y otro de entrada, cuando en nuestro caso el bloque NTSS con el que se realizarán las comunicaciones precisa de hasta 3 conexiones *AXI4 Stream* (dos entradas y una salida).

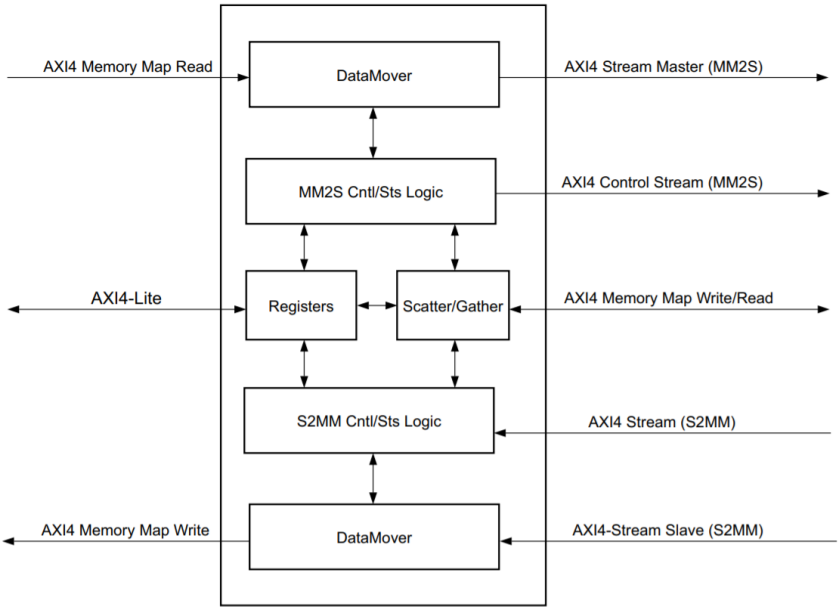

Fig. 5.2: Diagrama del bloque IP AXI DMA

No obstante, la utilización de este modo *Scatter-Gather* eleva considerablemente el tiempo de envío de los datos hacia el bloque IP y la complejidad del desarrollo de la *app software* posterior. Por esta razón, para evitar el uso del modo *Scatter-Gather*, se contemplaron dos posibles estrategias:

- Utilizar 2 bloques DMA. Por un lado, se emplearía un DMA configurado con las interfaces MM2S y S2MM activas para el envío de uno de los macrobloques necesarios por el bloque NTSS y para la recepción del vector de movimiento resultante; y por otro lado, un DMA configurado únicamente con la interfaz MM2S activa para el envío del otro macrobloque necesario para la estimación de movimiento.

- Utilización de un DMA y el bloque *AXI-Stream Switch* [40]. En este caso, tendríamos el bloque *AXI Switch* como intermediario entre el bloque NTSS y el DMA, recibiendo los datos de los distintos macrobloques a través de un único canal de transmisión MM2S por parte del DMA y dirigiéndolos hacia cada una de las entradas de nuestro bloque, en función de la ruta de datos establecida. La recepción del vector de movimiento se haría directamente entre el DMA y el bloque NTSS por la interfaz S2MM.

Finalmente, se decidió llevar a cabo la segunda opción, incluyendo en nuestro diseño un único DMA y un bloque *AXI-Switch*. De esta forma, logramos disponer de un diseño con un área reducida debido al número de recursos requeridos por el *AXI-Switch* para su ejecución, muy inferior al consumo necesario por parte de un DMA. Además, se presentaba como una oportunidad para probar este bloque IP en un diseño dentro del grupo de investigación, donde no se había empleado este bloque en ningún proyecto anterior.

Así pues, el diseño de la plataforma contendrá, como elementos principales, el bloque de procesamiento de la Zynq, un bloque DMA, encargado de realizar el intercambio de información entre la memoria RAM *off-chip* donde se almacenarán los *frames* de la secuencia de vídeo leída de la tarjeta SD, y nuestro bloque IP; y, por último, el bloque *AXI-Stream Switch* que ejercerá de intermediario entre el DMA y el módulo que contiene el algoritmo de estimación de movimiento para el direccionamiento de los datos transmitidos por el DMA hacia cada una de las dos entradas *AXI-Stream* del bloque IP diseñado.

Para llevar a cabo toda esta comunicación entre los distintos módulos de la plataforma se empleará el protocolo AXI. Por un lado, tendremos interfaces AXI4 para la comunicación entre el PS y los periféricos del sistema, asi como para el enlace del DMA con la memoria DDR, a través del *AXI Interconnect*. Por otro lado, todas las comunicaciones que conllevan la transmisión de los paquetes de datos (secuencia de vídeo o vectores de movimiento) se realizarán mediante interfaces *AXI4 Stream*, es decir, las comunicaciones DMA - Switch, Switch - Bloque NTSS y Bloque NTSS - DMA. Por último, se utilizarán interfaces *AXI4 Lite* para poder realizar la inicialización y todo tipo de configuraciones en estos tres últimos bloques mencionados.

5.1.1 Bloques IP utilizados

La plataforma completa estará formada, además de por el bloque IP diseñado para realizar la estimación de movimiento, de otros 10 bloques IP que se encuentran recogidos en el catálogo IP proporcionado por Xilinx. En la figura 5.4 se puede observar el diagrama de bloques de la plataforma, extraída del *IP Integrator* de la herramienta Vivado IDE.

A continuación, se describe la funcionalidad de cada uno de los bloques que componen el sistema:

- *processing_system7_0*: Instancia de la unidad de procesamiento de un SoC de la familia Zynq-7000, encargado de asegurar también las comunicaciones tanto con la lógica programable como con los distintos periféricos externos al dispositivo.

- *axi_dma_0*: El bloque DMA genera una ruta directa a la memoria DDR externa al dispositivo para permitir la realización de las distintas transacciones que se precisen. Por ello, en nuestro sistema el DMA es el encargado de enviar los datos del la memoria DDR hacia el *Switch* que, a su vez, lo retransmitirá hacia el bloque IP diseñado. Cabe destacar que este acceso a memoria se realiza de forma completamente transparente desde el punto de vista del PS, ya que las transacciones realizadas con la memoria se producen sin pasar por la CPU del dispositivo, la cual no es interrumpida.

- *axis_switch_0*: Este módulo proporciona un enrutamiento configurable entre maestros y esclavos que se comunican bajo la interfaz *AXI4 Stream*, pudiéndose realizar mediante dos métodos distintos: un enrutamiento mediante la señal *TDEST* o el enrutamiento mediante un registro de control [40]. Este último será el empleado en nuestro proyecto, configurando la tabla de enrutamiento mediante una interfaz *AXI4-Lite*. En nuestro diseño, este bloque trabajará con un esclavo y dos maestros para realizar la transmisión de los datos provenientes del DMA a cada una de las dos entradas del bloque IP diseñado cuando sea requerido.

- *ps7_0_axi_periph* y *axi_interconnect_0*: Bloques IP encargados de la configuración y establecimiento de la conexión entre las distintas interfaces del bus AXI necesarias para la comunicación entre los distintos bloques IP de la plataforma.

- *rst_ps7_0_100M*: Bloque IP precisado para el manejo de las señales de *reset* de los distintos bloques IP integrados en la plataforma.

- *axis_data_fifo_0-1-2*: Estos tres bloques son empleados para la gestión de las interfaces *AXI4-Stream* a la entrada y salida de nuestro bloque IP diseñado [41]. De esta manera, se logra asegurar que el envío y recepción de los paquetes se realice de forma completamente controlada evitando cuellos de botella debido a diferentes tasas de transmisión y procesamiento.

- *blockMatchingNTS_0*: Bloque IP diseñado siguiendo la metodología de síntesis de alto nivel y encargado de realizar la estimación de movimiento entre los dos macrobloques que se le facilitarán a su entrada, siguiendo el algoritmo NTSS y proporcionando el vector de movimiento que mejor representa el desplazamiento sufrido entre estos los dos MBs en cuestión. La funcionalidad y el diseño de este módulo ha sido previamente explicado en el Capítulo 4. De esta forma, el bloque constará de tres interfaces *AXI4-Stream* para la transmisión de estas 3 señales mencionadas (los dos MBs y el vector de desplazamiento resultante), así como de una interfaz *AXI4-Lite* para la inicialización y configuración del bloque desde la apliación *software* diseñada posteriormente. Por último, como ya se comentó en la sección 4.2.4, se dispondrá de una señal *last* a la salida del bloque para señalar el fin de la transmisión del vector de movimiento. La estructura de este bloque puede observarse en la figura 5.3.

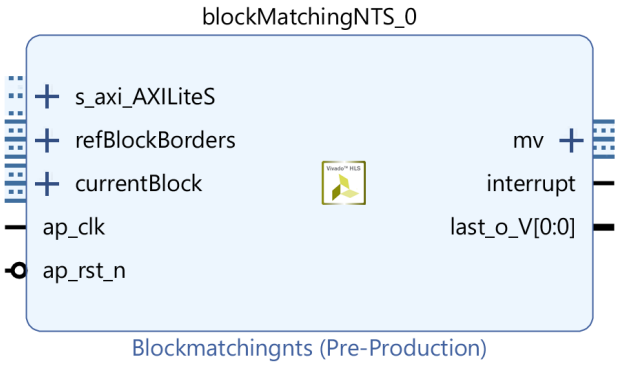

Fig. 5.3: Estructura del bloque IP *blockMatchingNTS_0*

5.1.2 Comunicación y conexionado entre bloques

Como ya se ha ido comentando a lo largo de este capítulo, el protocolo de comunicación utilizado entre los diferentes bloques IP que conforman la plataforma completa se trata de *AXI4*, donde podemos encontrar hasta tres tipos de interfaces empleadas para distintos fines, que se irán definiendo en esta sección: *AXI4*, *AXI4-Stream* y *AXI4-Lite*.

En primer lugar, la conexión entre los módulos integrados en el sistema se realiza a través del bloque *AXI Interconnect*, de forma que permita al PS comunicarse con el resto de bloques de la plataforma que forman parte de la lógica programable

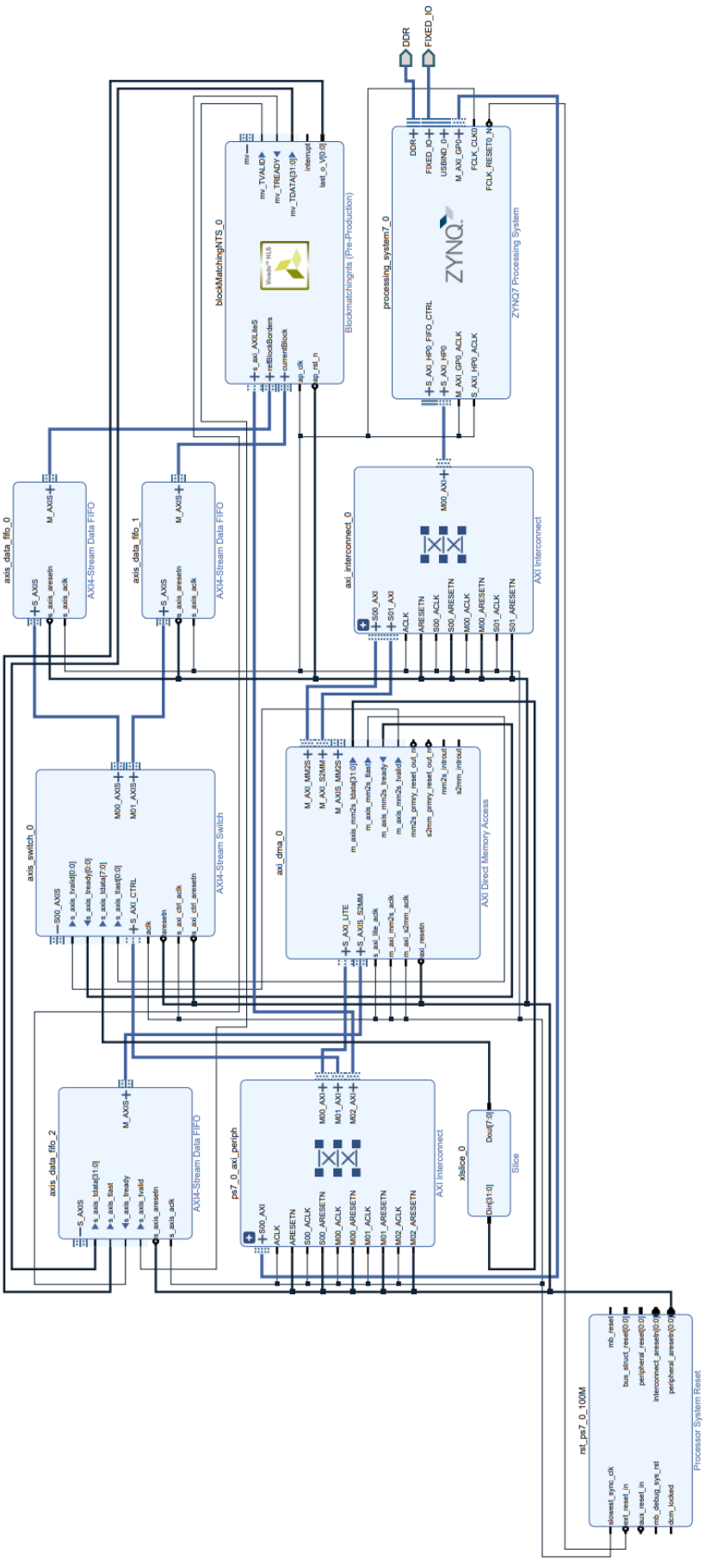

Fig. 5.4: Diagrama de bloques de la plataforma

(es decir, la FPGA). Por un lado, la comunicación entre el DMA y el PS se realiza mediante la interfaz *AXI4*, la cual también puede definirse como *AXI Memory-Mapped* debido a que su modo de trabajo consiste en asignar un rango de direcciones a cada bloque conectado a ese bus para facilitar la realización de transacciones. Es por esto que es considerada como la interfaz idónea para aplicaciones en las que se requiera de un alto rendimiento en el mapeo de memoria.

Por otro lado, la interfaz *AXI4-Lite* será la requerida para realizar las labores de configuración e inicialización de nuestro bloque IP y del *AXI-Stream Switch*, así como para proporcionar al DMA toda la información necesaria para la comunicación con la memoria RAM externa al dispositivo.

Por último, para asegurar una transmisión de paquetes de datos fluida y de alta velocidad entre el DMA, el *switch* y el bloque IP que alberga el algoritmo de estimación de movimiento, se emplearán interfaces *AXI4-Stream*. De esta forma, utilizando esta interfaz, se logra obtener una aceleración en los procesos de lectura y de escritura al poder emplear ráfagas.

5.2 Diseño del software empotrado

Una vez se ha realizado la integración completa de la plataforma con todos los bloques IP necesarios, es turno de la programación e implementación de la aplicación *software* a ejecutar en uno de los procesadores ARM disponibles en el PS, que controlará todo el funcionamiento del sistema.

5.2.1 Funcionalidad general

Tras haber exportado el *hardware* junto con el *bitstream* desde Vivado, ejecutamos el SDK de Xilinx, desde donde crearemos un proyecto asociado a un BSP, el cual contiene todas aquellas librerías necesarias para interactuar de forma transparente tanto con nuestro bloque IP como con el resto de elementos integrados en el sistema. Además, para la lectura y escritura de archivos desde la tarjeta SD externa al dispositivo, se incluye manualmente en el BSP la librería *xilffs* que integra el módulo de sistema de archivos *FatFs* [42].

Inicialmente, se realiza la inicialización y configuración de los distintos bloques que precisan de una programación vía *software*. A continuación, se comienza la etapa de estimación de movimiento. Para ello, se irán dividiendo los *frames* en macrobloques para su envío al bloque IP diseñado y obtener el vector resultante de la estimación de movimiento realizada. Esta acción se realizará continuamente hasta haber obtenido

un vector de desplazamiento para cada uno de los macrobloques que forman cada uno de los *frames* recogidos de la secuencia de vídeo. Por último, se almacenarán todos estos vectores en un archivo de texto, el cual se escribirá en la tarjeta SD para su comparación con lo obtenido en la etapa de HLS y en el *software* de referencia, pudiendo verificar el correcto funcionamiento del sistema.

5.2.2 Inicialización y configuración de los bloques *DMA, Switch y blockMatchingNTS*

Al inicio del desarrollo de la aplicación *software* algunos de los bloques IP implementados en el diseño requieren de una inicialización y una configuración inicial que permitan controlar estos bloques y utilizar sus respectivas funciones generadas en *software* y proporcionadas a través del BSP.

Para ello, se comienza con la búsqueda de la configuración *hardware* del dispositivo mediante la función *(dispositivo)_LookupConfig* existente para cada uno de los tres bloques a programar (*DMA, Switch y blockMatchingNTS*). Cabe destacar que para realizar esta búsqueda es necesario indicar la dirección de memoria base del dispositivo que se desea configurar e inicializar. Esto es posible gracias a la librería *xparameters.h*, donde se pueden encontrar todos los parámetros y direcciones de memoria relacionados con cada uno de los componentes del diseño.

Una vez se obtiene la configuración del bloque en cuestión, es posible inicializarlo mediante la función *(dispositivo)_CfgInitialize*, logrando así incluir la configuración de los datos del dispositivo y garantizar que el *hardware* se encuentra en un estado activo.

Posteriormente, tras realizar la inicialización de los 3 bloques mencionados, se utiliza la función *XAxiDma_HasSg* para comprobar si el DMA está configurado en modo *Scatter-Gather* que, como ya se comentó al inicio de este capítulo, deberá estar deshabilitado ya que utilizaremos un único canal de transmisión. Además, para finalizar con la configuración del DMA, se deshabilitan las interrupciones tanto de salida como de entrada, ya que este bloque IP se utilizará en modo *polling*.

Por último, para completar la configuración del *Switch* se deshabilitan las actualizaciones de registro y todos los puertos de este bloque IP mediante las funciones *XAxisScr_RegUpdateDisable* y *XAxisScr_MiPortDisableAll*.

5.2.3 Desarrollo de la aplicación *software*

Para comenzar con las interacciones con la tarjeta SD, se requiere de un area de trabajo (un objeto del sistema de archivos) que será definido mediante la función *f_mount*. De esta forma, se podrá iniciar la apertura de la secuencia de vídeo previamente preparada en la tarjeta SD y la lectura de los dos primeros *frames*, que serán almacenados temporalmente en la memoria DDR como las variables *refFrame* y *currentFrame*.

```
startByte = (FSIZE_t) (FIRST_FRAME-1)*(bytesY*linesY +
  bytesC*linesC + bytesC*linesC);
Res = f_lseek(&fil, startByte);
Res = f_read(&fil, (void*)&refFrame, frameSize, &NumBytesRead);

for (frameIndex = FIRST_FRAME; frameIndex < LAST_FRAME;
  frameIndex++){
   startByte = (FSIZE_t) frameIndex*(bytesY*linesY + bytesC*linesC
  + bytesC*linesC);
   Res = f_lseek(&fil, startByte);
   Res = f_read(&fil, (void*)&currentFrame, frameSize,
  &NumBytesRead);
```

Listing 5.1: Lectura de los 2 primeros *frames*

Posteriormente, se realiza la extensión de bordes del *frame* de referencia mediante la función *extendBorders* ya explicada en la sección 4.1.1. Con ello, ya dispondremos del *frame* al que se le realizará la estimación de movimiento y del *frame* de referencia requerido para dicha estimación. Así, comienza la construcción del macrobloque de referencia, que es enviado a través del DMA hacia el Switch, mediante la interfaz *AXI Stream* del puerto de salida MM2S, utilizando la función *XAxiDma_SimpleTransfer*.

```
    extendBorders(refFrame, refFrameBorders);
    for (r = 0; r < numBlocksRows; r++) {
      startRow = r*MB_SIZE;
      for(c = 0; c < numBlocksCols; c++){
        startCol = c*MB_SIZE;
        // Contrusccion del macrobloque de referencia con bordes
        for(i = 0; i < refBlockBordersRows; i++) {
          for(j = 0; j < refBlockBordersCols; j++) {
            TxBufferPtr[refBlockBordersCols*i + j] =
  refFrameBorders[refFrameBordersCols*(i + startRow) + j +
  startCol];
        }
      }
```

Listing 5.2: Extensión de bordes del *frame* de referencia y construcción del MB a enviar

En este punto, se habilita la conexión entre el esclavo de entrada del Switch (que recibe los datos del DMA) y el maestro de salida que está conectado a la entrada del bloque NTSS, el cual se encuentra a la espera de los datos del macrobloque de referencia. Con esto, el envío del MB de referencia finaliza y se inicia el mismo proceso para la transmisión del MB actual, habilitando la conexión esclavo-maestro del Switch correspondiente a la entrada del bloque NTSS que espera estos datos. De esta forma, una vez se han enviado los dos macrobloques, nuestro bloque IP genera el vector de movimiento resultante y es transmitido al DMA para su almacenamiento temporal en la memoria RAM externa al dispositivo.

```
        MiIndex = 0;
        SiIndex = 0;
        XAxisScr_MiPortEnable(&AxisSwitch, MiIndex, SiIndex);
        XAxisScr_RegUpdateEnable(&AxisSwitch);

        Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR)
    TxBufferPtr, refBlockBordersSize*sizeof(u32),
    XAXIDMA_DMA_TO_DEVICE);
```

Listing 5.3: Habilitación del *AXI-Switch* y envío del MB de referencia

Todo este proceso es continuamente repetido, construyendo distintos macrobloques de referencia y actuales hasta haber recorrido completamente los dos *frames* bajo análisis, obteniendo así el vector de movimiento que mejor describe el desplazamiento de cada uno de los macrobloques que forman el *frame* bajo estudio. Una vez terminado este proceso, el último *frame* almacenado pasa a ser considerado la imagen de referencia y se realiza la lectura de un nuevo *frame* para el análisis y obtención de sus vectores de desplazamiento.

Por último, cabe destacar que el vector de movimiento resultante de cada iteración se va recogiendo en un archivo de texto, el cual es almacenado en la tarjeta SD para su posterior verificación.

5.3 Validación del sistema

Tras haber realizado la integración de nuestro bloque IP dentro de la plataforma configurable, pasamos a realizar la validación del sistema sobre la Pynq-Z1. Para ello, será necesaria la inclusión de bloques *System ILA* (mencionado en la sección 3.2.3) en nuestro diseño, conectándolos a las entradas y/o salidas de los bloques IP cuyas señales queramos monitorizar y visualizar en el *Hardware Manager* de cara a validar nuestro sistema.

Una vez comprobado el correcto funcionamiento del sistema, se recogerán los resultados temporales obtenidos tras ejecutar nuestro sistema en *hardware* y se compararán con el tiempo obtenido tras ejecutar el sistema completamente en *software*. Todos estos valores se extraerán a través del depurado en Xilinx SDK.

Por otro lado, la información acerca del consumo de recursos de nuestra plataforma al ser mapeada en la FPGA durante la fase de implementación será de vital importancia, mostrando especial interés en el número de *LUTs* y *flip-flops* utilizados, que determinarán el factor de utilización de las *slices* disponibles en el SoC seleccionado como dispositivo final de implementación. Para obtener estos datos, Vivado nos proporciona un informe detallado del consumo de recursos del diseño una vez es implementado sobre la FPGA.

Por último, se comentará el consumo de potencia por parte de cada uno de los elementos que constituyen el dispositivo, al implementar el algoritmo de estimación de movimiento.

5.3.1 Banco de pruebas

Con el propósito de obtener los resultados requeridos en nuestro sistema, es necesario haber configurado y preparado previamente un banco de pruebas con el que poder verificar que el sistema empotrado desarrollado cumpla con las expectativas, realizando correctamente las funciones para las que fue diseñado. Tras este preámbulo, se describe a continuación los cambios realizados y la metodología seguida para llevar a cabo la verificación del sistema, así como los distintos elementos que se han requerido para ello.

Actualización del diseño para la monitorización de señales

Con el objetivo de poder analizar el comportamiento de nuestro diseño durante la transmisión de datos entre los bloques *DMA, Switch y blockMatchingNTS*, se precisa la incorporación de módulos *System ILA* a las entradas y salidas de los bloques previamente mencionados. De esta forma, mediante la herramienta *Hardware Manager* se podrán monitorizar las señales conectadas, siendo en nuestro caso las señales *TVALID, TREADY y TDATA* del protocolo *AXI4 Stream* presente en las I/O de estos bloques. Por lo tanto, los *System ILA* conectados se encontrarían configurados tal y como se ilustra en la figura 5.5, conectando la señal *TVALID y TREADY* a las entradas *probe0* y *probe1* respectivamente, siendo estas de un solo bit; y la señal *TDATA* al *probe2* del bloque, siendo esta de una dimensión de 32 bits para el caso de las I/O del *DMA* y la salida del bloque *blockmatchingNTS*, y de un tamaño de 8 bits para las

entradas de nuestro bloque IP. Estas dimensiones se deben básicamente al tipo de dato utilizado, siendo en el primer caso de tipo *int* (32 bits) y *unsigned char* (8 bits) en el caso de las entradas de nuestro bloque IP.

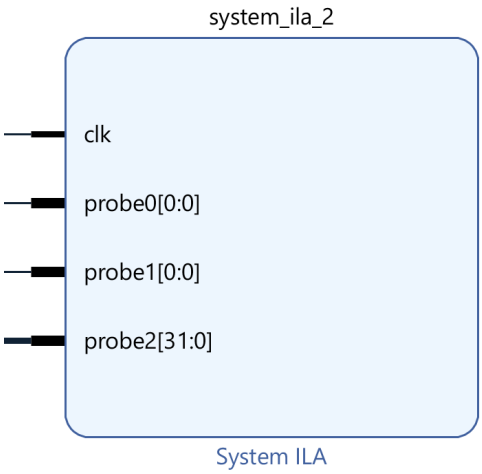

Fig. 5.5: Bloque IP *System ILA*

Preparación del banco de pruebas

Antes de comenzar con la validación, es necesario tener preparada la placa de prototipado con todo lo necesario. En primer lugar, se almacenará la secuencia de vídeo que pasaremos por el proceso de estimación de movimiento dentro de la tarjeta SD que incorporaremos a la placa. Posteriormente, se procede a la comunicación entre la Pynq-Z1 y el ordenador donde habremos desarrollado nuestro proyecto en Vivado IDE y SDK. Esta conexión se realizará a través del puerto micro USB disponible en la placa de prototipado que integra el protocolo JTAG para la programación del dispositivo.

Una vez tenemos nuestro equipo conectado a la placa, podemos comenzar con la validación del sistema, partiendo del modo *debug* de Xilinx SDK, donde realizaremos el reseteo del sistema y la programación del SoC Zynq de forma previa al comienzo del depurado, tal y como puede observarse en la figura 5.6. Además, arrancaremos la herramienta *Hardware Manager* en Vivado IDE, de forma que podamos visualizar las formas de onda de nuestras señales paralelamente al transcurso del depurado en el SDK y así verificar que el sistema desempeña sus funciones correctamente.

5.3.2 Proceso de validación

La validación de nuestro sistema, como ya se ha introducido, parte del depurado de nuestro proyecto en SDK y del uso del *Hardware Manager* en Vivado. De esta manera,

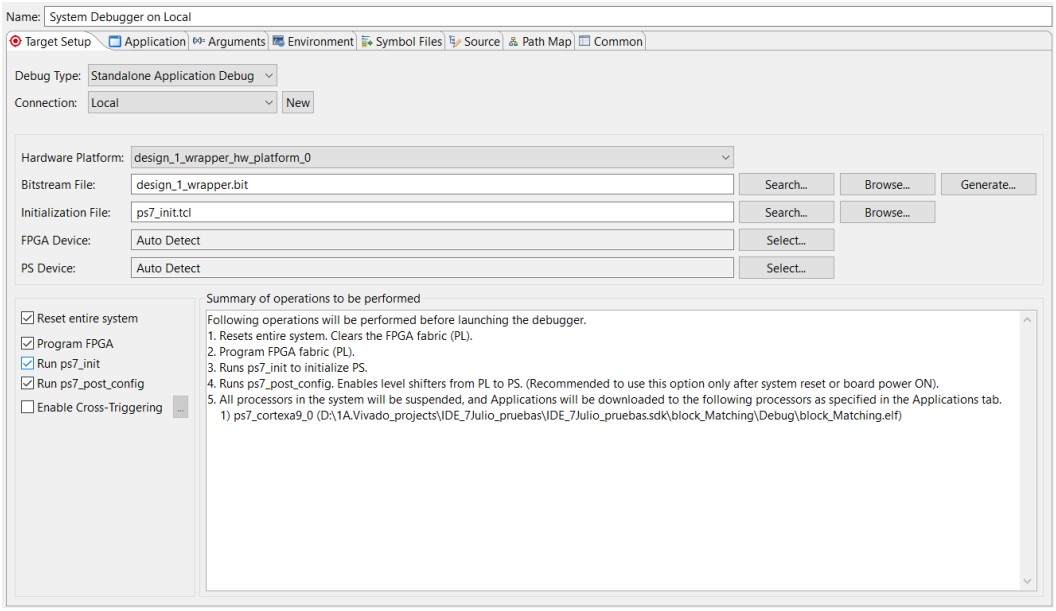

Fig. 5.6: Configuración del modo *debug* en Xilinx SDK

se ha logrado visualizar paso a paso el comportamiento de las señales de interés al ejecutar la aplicación *software* desarrollada, enfatizando aquellas líneas de código que producirán cambios en dichas señales, principalmente en el envío y recepción de los datos. Gracias a ello, se ha facilitado enormemente el descubrimiento y solución de pequeños errores que ocasionaban la obtención de resultados no esperados. A continuación, se describe el proceso de validación llevado a cabo, cuyos resultados mostrados han sido recogidos una vez todos los errores han sido corregidos.

Para comenzar, desde el *Hardware Manager* se especifican las condiciones de *trigger* que deseamos establecer para obtener la forma de onda de la señal en el instante en el que se cumplen dichas condiciones. Es decir, se señalan los valores que deseamos visualizar de la señal en cuestión, de forma que en el instante en que se ejecute una línea de código desde el modo *debug* del SDK que modifique dicha señal y se obtenga dicho valor, se extrae la forma de onda de la señal en ese intervalo. En nuestro caso, se establecerá que se produzca el *trigger* en el momento en el que se produzca cualquier variación en las señales de los 5 ILAs incluidos en el diseño.

Así, una vez depuradas las líneas de inicialización y configuración de los bloques, donde se resetean cada una de sus señales, las primeras variaciones se producen al realizar la transmisión del primer macrobloque de referencia del DMA a nuestro bloque IP mediante la función *XAxiDma_SimpleTransfer*. En la figura 5.7 se pueden observar las señales *TVALID, TREADY y TDATA* de salida del DMA (o de entrada al *Switch*), donde podemos comprobar como la señal TREADY se encuentra a 1 (desde la inicialización del bloque) y como la señal TVALID pasa a nivel alto al inicio de la transmisión, comenzando así el envío de los datos mediante la señal TDATA; y

pasa a nivel bajo al finalizar la transmisión. Se puede observar en la imagen cómo la transmisión se extiende efectivamente durante un total 1024 ns, siendo este el tamaño completo del macrobloque de referencia (32x32 píxeles), ya que se envía un dato por ciclo de reloj.

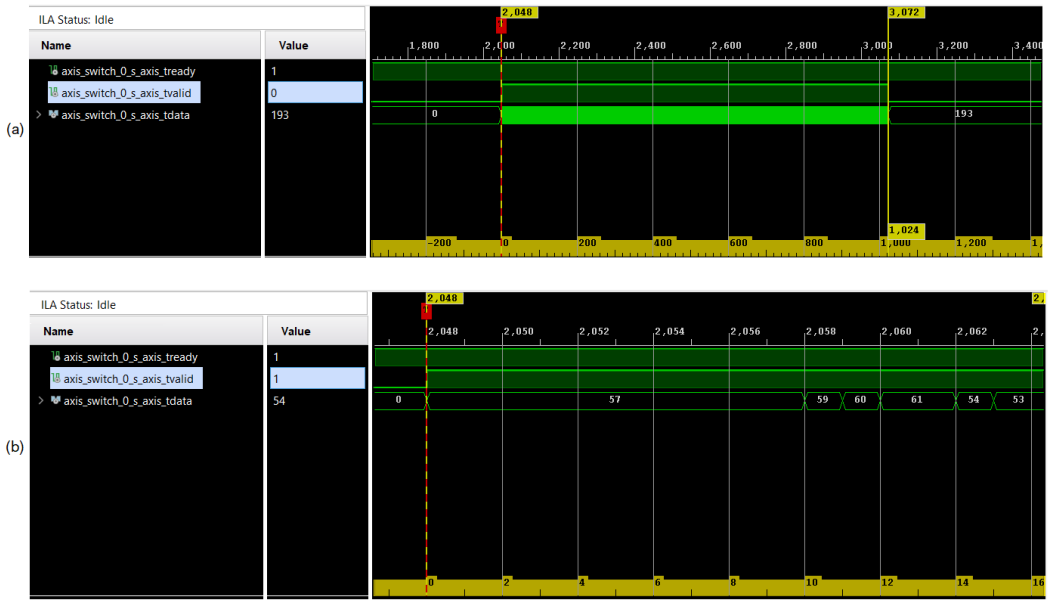

Fig. 5.7: Forma de onda de la salida del DMA durante la transmisión del primer macrobloque: (a) Transmisión completa (b) Zoom de la señal

De la misma forma, en la transmisión del macrobloque considerado como actual, la salida del DMA adquiere los valores de los 256 píxeles que componen este macrobloque (16x16). En este punto, también se ha podido comprobar el correcto enrutamiento de los datos a través del bloque *AXI-Stream Switch* hasta nuestro bloque IP, mediante la visualización de los ILAs situados a las entradas de este bloque. En la figura 5.8 se encuentra ilustrada la forma de onda de la entrada correspondiente al macrobloque actual dentro del bloque *blockMatchingNTS*, donde las señales adquieren los mismos valores que a la salida del DMA, razón por la cual se verifica la correcta transmisión de los macrobloques.

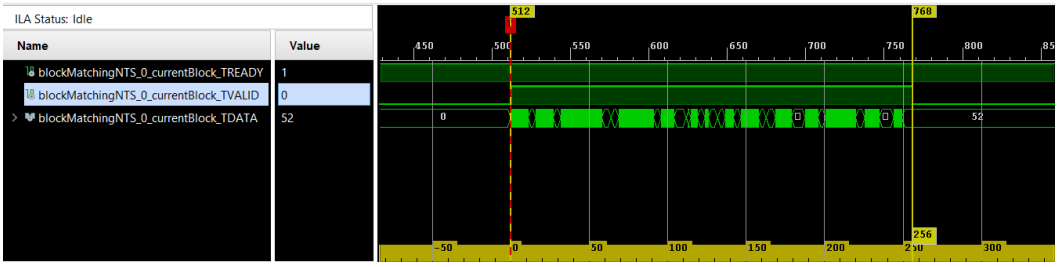

Fig. 5.8: Forma de onda a la entrada *currentBlock* del bloque *blockMatchingNTS* durante la transmisión del macrobloque actual

Desde que se reciben los dos macrobloques en el bloque IP *blockMatchingNTS*, se realiza la estimación de movimiento, obteniéndose así el vector de movimiento que

mejor describe el desplazamiento sufrido por el macrobloque de referencia respecto al macrobloque actual. Esto se ha podido demostrar gracias a los valores recogidos de la forma de onda a la salida de nuestro bloque IP y a la entrada del DMA, los cuales, efectivamente, son idénticos.

Por último, todos los vectores de movimiento recibidos por el DMA van siendo almacenados en un archivo de texto, de manera similar a la etapa HLS, con el objetivo de comprobar que la estimación de movimiento se ha realizado correctamente mediante su comparación con el archivo generado con el *software* de referencia.

5.3.3 Resultados obtenidos

En primer lugar, hemos fijado la frecuencia del reloj de nuestro sistema a 100 MHz para la lógica programable, obteniendo un *slack*(*WNS* en la figura 5.9) positivo que nos indica cuánto se puede disminuir el período de reloj del sistema para lograr una frecuencia más alta de funcionamiento; es decir, cuanto más cercano a 0 se encuentre este valor, más próximo estará nuestro sistema a trabajar a la máxima frecuencia posible.

Fig. 5.9: *Slack* obtenido con un reloj de 100MHz

Aumentando la frecuencia de reloj del PL a 110 MHz (realmente 111.11 MHz debido a que el reloj es generado por un PLL y no se puede fijar de forma exacta) el *slack* pasa a ser de -0.014 ns (figura 5.10), lo que indica que nos encontramos por encima de la frecuencia máxima a la que el sistema puede trabajar. Dado que el margen negativo del *slack* es bastante cercano a 0, intentamos reducir la ruta crítica aplicando estrategias de optimización de *retiming* ofrecidas por Vivado; sin embargo, los resultados no sufrieron cambio alguno por lo que, teniendo en cuenta que no es posible variar la frecuencia de reloj del PL entre 100MHz y 111.11 MHz (lo que equivale a un período de 10 y 9 ns, respectivamente) podemos concluir que la frecuencia máxima a la que puede trabajar nuestro sistema es de 100MHz.

A fin de obtener una comparativa que demuestre la ventaja del sistema *hardware* frente a uno plenamente *software*, la latencia de nuestro algoritmo de estimación de movimiento será calculada tanto en *hardware* como en *software*. Por un lado, para el cálculo de la latencia *software*, se empleará la librería *xtime_l.h* proporcionada por Xilinx, mediante la cual se pueden aplicar marcas temporales al comienzo y fin de

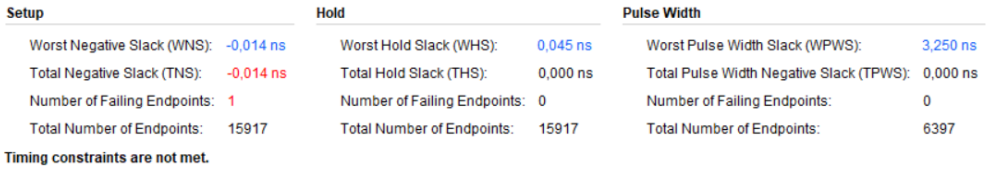

Fig. 5.10: *Slack* obtenido con un reloj de 111MHz

un bloque de código determinado, obteniendo el número de ciclos de reloj que han sido necesarios para la ejecución del mismo.

Así, tras colocar las marcas temporales al inicio y al final del algoritmo de estimación de movimiento en el diseño completamente *software*, se ha obtenido un valor de 184885 ciclos de reloj. Sin embargo, se debe tener en cuenta que la librería utilizada devuelve la mitad de ciclos empleados y, por tanto, se debe multiplicar este valor por 2. Finalmente, para obtener la latencia del diseño es necesario dividir por la frecuencia del microprocesador empotrado empleado, el cual trabaja a una frecuencia de 666.66 MHz. Aplicando estas puntualizaciones, la latencia obtenida en *software* tiene un valor de:

$$Latencia_{software} = 2xN_{ciclos}/freq_{microprocesador} = 2x184885/666{,}66x10^6 = 554{,}66\mu s \tag{5.1}$$

Por otro lado, la latencia del diseño en *hardware* se ha obtenido mediante la herramienta *Hardware Manager*. Tal y como se puede observar en la figura 5.11, fijando las marcas de tiempo entre el inicio y el final de la recepción de los dos macrobloques de entrada al algoritmo se obtuvo un tiempo de ejecución 1281 ciclos de reloj, empleando un ciclo por dato de cada macrobloque enviado.

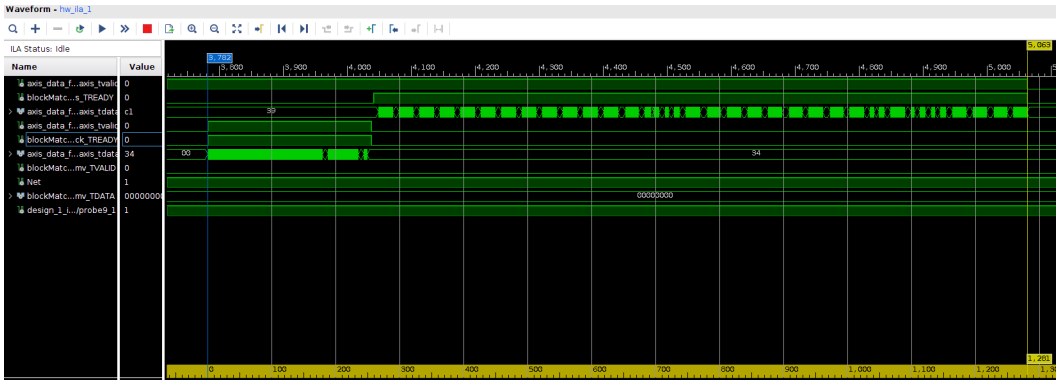

Fig. 5.11: Forma de onda de las dos entradas del bloque IP

Luego, en la figura 5.12 se puede comprobar, al igual que con los macrobloques de entrada, que la obtención del vector de movimiento resultante se realiza en un ciclo de reloj por cada dato, obteniéndose por tanto en 2 ciclos de reloj.

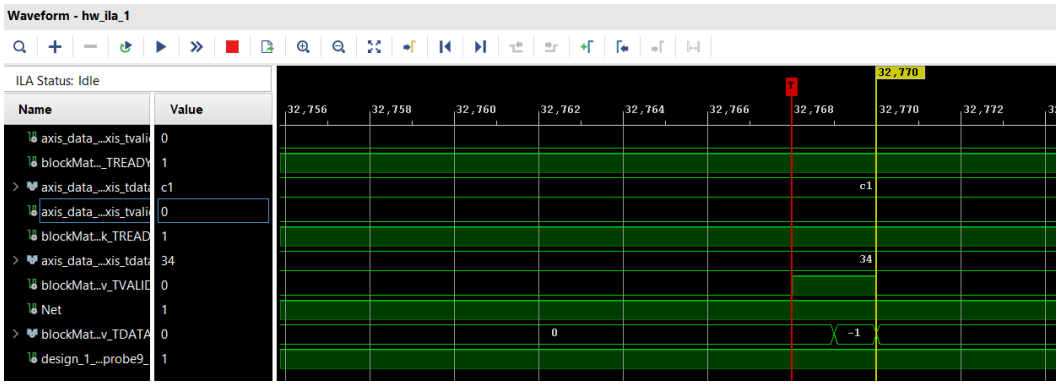

Fig. 5.12: Forma de onda de la salida del bloque IP, el vector de movimiento

Con esto, solo quedaría obtener el tiempo requerido por el bloque NTSS para ejecutar el algoritmo de estimación de movimiento. En la figura 5.13 se encuentra ilustrado el tiempo completo de todo el proceso del bloque NTSS, desde la recepción del primer dato hasta el envío del vector de movimiento resultante. Tal como se ilustra en la esquina inferior derecha, el proceso completo de estimación de movimiento requiere un total de 6898 ciclos de reloj en *hardware*.

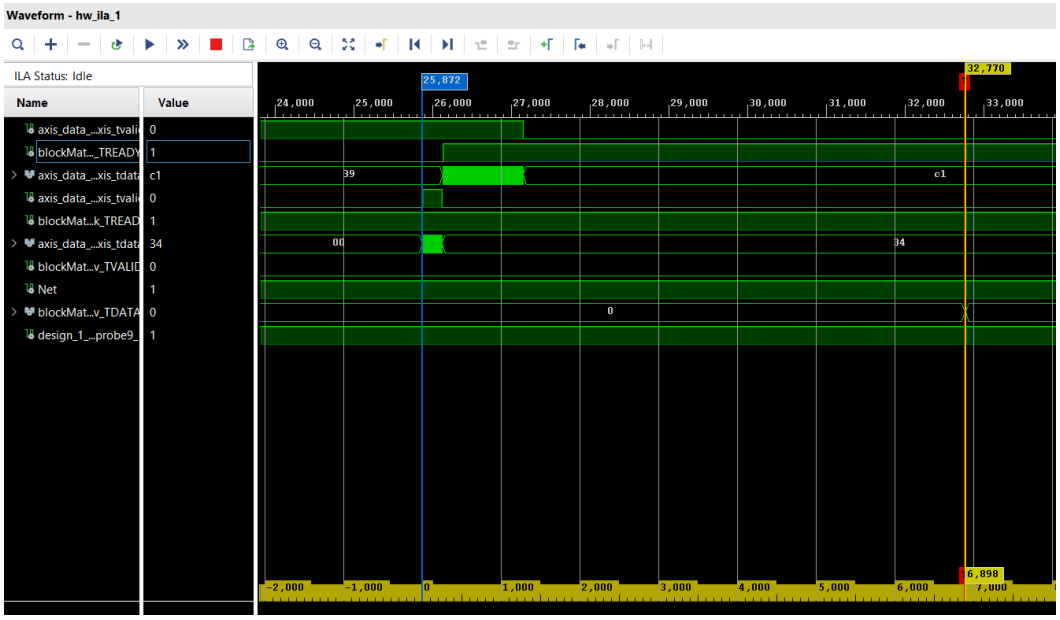

Fig. 5.13: Tiempo de ejecución de la estimación de movimiento, desde la recepción del primer macrobloque hasta el envío del vector de movimiento

En este caso, teniendo en cuenta que la frecuencia de trabajo considerada será 100MHz, la latencia obtenida en nuestro diseño *hardware* sería:

$$Latencia_{hardware} = 6898/100x10^6 = 68{,}98\mu s \tag{5.2}$$

Dividiendo a continuación los resultados obtenidos en las ecuaciones 5.1 y 5.2 entre sí, obtenemos el *speed-up* o factor de aceleración de nuestro sistema:

$$Speedup = 554{,}66\mu s/68{,}98\mu s = \mathbf{8.04} \tag{5.3}$$

En vista de los resultados temporales obtenidos, podemos concluir que el diseño *hardware* del algoritmo de estimación de movimiento NTSS supone un incremento de las prestaciones de $\times 8$ superior respecto al diseño *software*.

Continuando con la validación del sistema, resulta necesario analizar también el consumo de recursos del dispositivo por parte de nuestro diseño, lo cual se puede observar en la tabla 5.1.

Tab. 5.1: Consumo de recursos del bloque IP y la plataforma

Recurso	Total en la FPGA	Utilizado por la plataforma	Utilizado por el bloque IP
Slices	13300	2182 (16.41 %)	852 (6.41 %)
Slice LUTs	53200	5621 (10.57 %)	2621 (4.93 %)
Slice flip-flops	106400	5492 (5.16 %)	1442 (1.36 %)
LUTs como lógica	53200	5415 (10.18 %)	2621 (4.93 %)
LUTs como memoria distribuida	17400	206 (1.18 %)	0 (0 %)
Multiplexores F7	26600	3 (0.01 %)	3 (0.01 %)
BRAM	140	18 (5.16 %)	3.5 (1.36 %)

En este sentido, resulta de gran importancia la estimación del factor de utilización de las *slices*. Cada *slice* se compone de 4 LUTs y 8 *flip-flops* en la tecnología empleada, por lo que el factor de utilización ideal de las LUTs resulta ser 4 y el valor ideal del factor de utilización de *flip-flops* es considerado 8. Los factores de utilización obtenidos en nuestro diseño son los siguientes:

$$FU_{LUTs} = N_{LUTs}/N_{slices} = 2621/852 = 3{,}08 \tag{5.4}$$

$$FU_{flip-flops} = N_{flip-flops}/N_{slices} = 1442/852 = 1{,}69 \tag{5.5}$$

Como podemos ver en los resultados obtenidos, el factor de utilización de las LUTs se encuentra cercano al valor ideal ya que, al implementar el diseño, gran parte de la

lógica empleada requiere de un tratamiento combinacional. Por otro lado, el factor de utilización de *flip-flops* se encuentra alejado del valor ideal, algo que se puede ver en la escasa necesidad de empleo de registros dentro del bloque IP en comparación con la cantidad de operaciones realizadas mediante LUTs.

Por último, resulta de gran interés el análisis de la potencia consumida por nuestro diseño, para así comprender en mayor medida las prestaciones del sistema empotrado desarrollado. La potencia total consumida por nuestra plataforma, tal y como se puede apreciar en la figura 5.14, es la suma de la potencia dinámica y la potencia estática. Esta última depende exclusivamente de la tecnología FPGA utilizada, ya que hace referencia a la potencia consumida por el dispositivo al estar operativo, de forma independiente al diseño implementado. En cambio, la potencia dinámica hace referencia a la potencia consumida por las distintas partes de la plataforma al implementar y entrar en funcionamiento el sistema desarrollado, donde se aprecia claramente como el bloque de procesamiento consume la mayor parte de la potencia (1.256 W).

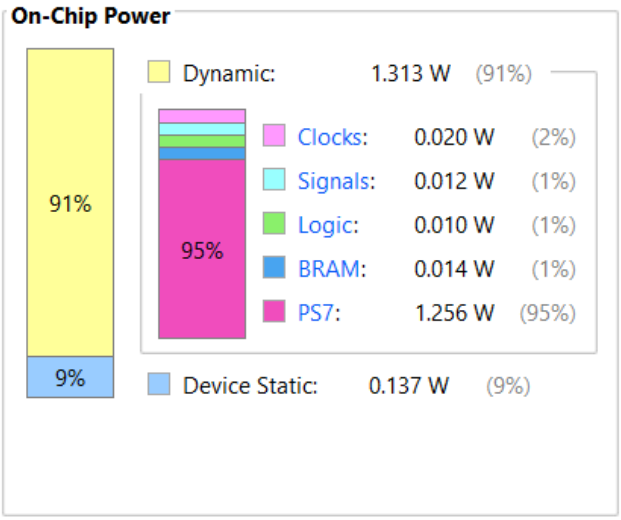

Fig. 5.14: Potencia consumida por el dispositivo

5.4 Conclusiones

En este capítulo se describe cómo se encuentra estructurada la plataforma configurable sobre la que se integra el bloque IP diseñado con nuestro algoritmo de estimación de movimiento, siendo de vital importancia la separación *hardware-software* dentro del sistema empotrado, por una parte, la FPGA, donde se realiza la estimación de movimiento mediante el algoritmo NTSS; y, por otra parte, el procesador ARM en el que se ejecutará la aplicación *software*.

Adicionalmente, como parte de esta descripción de la estructura del sistema, se detalla la funcionalidad de cada uno de los bloques que forman parte del diseño y que aseguran la correcta comunicación entre los distintos elementos del sistema. Además, hay que destacar la importancia de la aplicación *software* dentro del sistema empotrado, donde se realizan las inicializaciones y configuraciones de los bloques principales del diseño, además de llevarse a cabo la preparación y transmisión de los datos necesarios en nuestro bloque IP para poder obtener los resultados de la estimación de movimiento.

Por último, con el bloque IP correctamente integrado en la plataforma, tiene lugar el proceso de validación del diseño, realizando la preparación del banco de pruebas con el que podremos comprobar el funcionamiento de nuestro sistema empotrado. De esta forma, con el diseño validado, se exponen los resultados obtenidos, tanto en términos de consumo de potencia y área de utilización de la Pynq, como temporales mediante la ejecución de nuestro algoritmo de estimación de movimiento en *hardware* a través del sistema empotrado desarrollado y en *software*, a fin de evaluar la eficiencia de nuestro diseño, en términos de latencia de ejecución.

6 Conclusiones y trabajos futuros

A lo largo del presente documento, se ha expuesto de manera detallada todas las etapas llevadas a cabo para el diseño de un sistema empotrado encargado de realizar la estimación de movimiento de una secuencia de vídeo a través de una plataforma configurable. Este capítulo contiene las conclusiones del proyecto y las posibles líneas de investigación que se pueden seguir en un futuro a raíz del sistema desarrollado.

6.1 Conclusiones del proyecto

Al inicio de este proyecto se realizó un análisis de los principales algoritmos de estimación de movimiento existentes para su implementación en aplicaciones en satélite, optando finalmente por el algoritmo *New Three Step Search*, una técnica basada en *Block Matching*. Esta decisión fue tomada debido a que en aplicaciones espaciales no resulta necesario que cada píxel de una secuencia de vídeo cuente con un vector de movimiento propio, ya que la mayor parte de los píxeles seguirán un comportamiento similar (movimiento global de la escena) ocasionado por el desplazamiento del propio satélite en su órbita.

Para la implementación del algoritmo NTSS, se ha desarrollado un bloque IP en lenguaje de alto nivel (C/C++) siguiendo la metodología HLS mediante el uso del entorno de diseño *Vivado Design Suite*, partiendo del diseño del módulo funcional en lenguaje de alto nivel y llegando hasta la programación del dispositivo. Para este bloque IP, resulta de vital importancia que el consumo de recursos se encuentre en valores mínimos, teniendo en cuenta que este trabajo se trata únicamente de la etapa de estimación de movimiento que formaría parte de un compresor de vídeo completo que sería implementado en la FPGA. Así, tras realizar la síntesis de alto nivel y habiendo aplicado distintas directivas de optimización sobre el diseño integrado en *hardware*, se obtuvo un consumo estimado de un 2 % de la memoria dedicada disponible, un 1 % de los FFs y un 19 % de LUTs; todo ello con una latencia media de 7028 ciclos de reloj. Fijando la frecuencia máxima de reloj a 100 MHz, esto supone una latencia media de 70.28 μs.

Además, para el diseño de este bloque IP, se decidió realizar una partición *hardware/software*, logrando un mejor aprovechamiento de los recursos de los que dispone

el dispositivo Zynq, ejecutando parte del desarrollo en la FPGA y otra parte en uno de los microprocesadores ARM disponibles en el PS. Entonces, por un lado, la implementación *hardware* del algoritmo se realiza sobre la FPGA y, por otro lado, en el *core* ARM se desarrolla la aplicación *software* encargada de realizar las inicializaciones y configuraciones de los bloques principales del diseño, además de preparar y transmitir los datos necesarios en nuestro bloque IP para poder obtener los resultados de la estimación de movimiento.

El sistema final desarrollado está constituido, entre otros módulos, por un bloque *AXI-Stream Switch*, el cual se sitúa entre el DMA y el bloque NTSS para llevar los datos de la única salida del DMA hasta cada una de las dos entradas que presenta el bloque IP diseñado. Pese a la posibilidad de utilizar dos bloques DMA para realizar la comunicación con el bloque NTSS de forma directa, la utilización del *AXI Switch* se estimó como una mejor opción debido al escaso número de recursos que requería frente al empleo de dos bloques DMA.

Luego, tuvo lugar la fase de validación del sistema sobre una Pynq-Z1, una placa de prototipado que posee un SoC de la familia Zynq-7000 de Xilinx. En ella, se ha podido verificar el correcto funcionamiento del algoritmo y del sistema completo, habiendo comparado los vectores de movimiento resultantes de los primeros 5 *frames* de una secuencia de vídeo con los vectores de movimiento obtenidos de un *software* de referencia desarrollado dentro del grupo de investigación. Además, a fin de evaluar la eficiencia de nuestro diseño, se llevó a cabo la ejecución de nuestro algoritmo de estimación de movimiento en *hardware* a través del sistema empotrado desarrollado y de la versión puramente *software* del algoritmo. Durante este proceso, se obtuvo una latencia del diseño en *hardware* de 68.98 μs, muy similar a la obtenida en la estimación realizada en la síntesis de alto nivel (70.28 μs), frente a los 554.66 μs que conllevó la ejecución del algoritmo en *software*. Con ello, se ha podido demostrar que la implementación *hardware* realizada obtiene una latencia de hasta 8 veces menor respecto al modelo *software* o, lo que es lo mismo, un *speed-up* de aproximadamente $\times 8$.

Adicionalmente, se comprobaron los resultados obtenidos de área de utilización y potencia consumida por la plataforma. Por un lado, el sistema consume únicamente un 16.41 % de los *slices* disponibles en la FPGA y un 5.16 % de la memoria dedicada. Por otro lado, se obtuvo unos valores muy bajos de potencia total consumida por nuestra plataforma, donde casi el 96 % de la potencia era consumida por el bloque de procesamiento (1.256 W). Por tanto, una vez analizados estos resultados, se ha podido concluir que el sistema empotrado desarrollado resulta ser una solución bastante eficiente en términos de prestaciones temporales, potencia consumida y consumo de recursos.

Por último, cabe destacar que todas las pruebas realizadas han sido utilizando vídeos de una resolución de 352x288. No se ha podido llegar a resoluciones de HD o superiores debido a las limitaciones en el entorno de Vivado con las dimensiones de los *arrays* en los que se almacenan cada uno de los *frames* que forman la secuencia de vídeo. Sin embargo, gracias a la optimización *hardware/software* realizada durante la etapa de diseño del bloque IP (sección 4.1.3), el tamaño de los *frames* es completamente transparente para nuestro bloque IP, ya que este recibirá siempre dos macrobloques de 16x16 píxeles y generará el vector de movimiento que define el desplazamiento sufrido entre los dos MB en cuestión, de forma completamente independiente a la resolución de la imagen. Por ello, en cualquier cambio que se precise para lograr adaptar el diseño a resoluciones superiores, no se considerará necesario modificar el bloque IP que realiza la estimación de movimiento.

6.2 Trabajos futuros

Finalmente, se contemplan diversas líneas de investigación a seguir en trabajos futuros, con el fin de lograr mejoras y optimizaciones del trabajo descrito en el presente TFG.

En primer lugar, se plantea el desarrollo de un nuevo sistema de validación en el que se sustituya el bloque *AXI-Stream Switch* que lleva a cabo el direccionamiento de los datos entre el DMA y las dos entradas del bloque NTSS y se implemente la otra estrategia considerada para llevar a cabo esta función: el uso de dos bloques DMA. Con ello, se puede prever un aumento sustancial de los recursos lógicos consumidos por el sistema (actualmente se dispone de cierto margen), pero podremos evaluar si los resultados temporales del sistema final son mejores a los obtenidos y respecto a su ejecución completa en *software*, ya que se podrían recibir los dos macrobloques de entrada al bloque NTSS al mismo tiempo.

Luego, con respecto al algoritmo de estimación de movimiento empleado, resultaría de gran interés implementar el *Diamond Search* dentro del diseño como sustitución al algoritmo *New Three Step Search*, ya que ambos presentan una complejidad muy similar. Con ello, se compararían los resultados temporales y de *throughput* obtenidos, así como los recursos requeridos por el sistema para su correcto funcionamiento, logrando de esta forma la certeza sobre qué algoritmo se adapta mejor a una aplicación con tantas restricciones como es la industria espacial.

Además, independientemente del estimador de movimiento utilizado, se plantea la posibilidad de optimizar el *set-up* de validación, reemplazando el método de obtención de las imágenes. Actualmente, este proceso se realiza mediante la lectura desde

una tarjeta SD incluida en la placa de prototipado. Esta gestión podría realizarse a través de una interfaz de alta velocidad, como Ethernet o PCIe, disminuyendo en gran medida el tiempo empleado para la lectura de los *frames* de la secuencia de vídeo.

Por otro lado, teniendo en cuenta que el estimador de movimiento es tan solo una de las etapas dentro de la codificación de vídeo, se presenta como opción la integración del resto de módulos necesarios para contar con el codificador de vídeo completo dentro de la plataforma. De esta forma, dispondríamos de todo el proceso de codificación de vídeo desarrollado sobre una FPGA instalada a bordo del satélite. Sin embargo, para llevar a cabo este proyecto en un caso real, se precisa de una FPGA calificada para trabajar en misiones especiales. Por ello, se propone la implementación del diseño sobre una FPGA de esta índole, como la *Kintex UltraScale KCU060* [43], o su equivalente comercial *KCU040*, la cual ha sido adquirida recientemente por el grupo de investigación.

Por último, se estudiará la posibilidad modificar el diseño para soportar formatos de vídeo de mayor resolución, HD o, incluso, 4k. Como ya se ha comentado anteriormente, el bloque IP no se debería ver afectado por los cambios que se realicen en este aspecto, ya que su funcionamiento es independiente del tamaño del *frame*.

7 Presupuesto

En este capítulo se expondrá el presupuesto relativo a la realización de este Trabajo Fin de Grado, donde ha sido necesaria la utilización de diversos recursos, tanto humanos como materiales.

7.1 Recursos humanos

Esta sección incluye los honorarios que recibe el ingeniero a cargo del desarrollo del proyecto, según las horas trabajadas. Para el cálculo se ha tenido en cuenta que el proyecto se desarrolla en la Universidad de Las Palmas de Gran Canaria; por lo tanto, la tarifa que se aplica es la correspondiente al personal técnico (graduado) según la tabla de clasificación y retribución del personal contratado con cargo a proyectos, convenios y contratos correspondiente al BOULPGC del 3 de junio de 2019 [44]. Al mismo tiempo, también se ha tenido en cuenta la duración del proyecto (4 meses) y el tiempo empleado diariamente (4 horas). El resultado final se encuentra en la Tabla 7.1.

Tab. 7.1: Trabajo tarifado por tiempo empleado

Personal	Coste total mensual	Tiempo	Total
Ingeniero técnico	711.90 €	4 meses	2847.60 €

El coste final del trabajo tarifado por tiempo empleado es de DOS MIL OCHOCIENTOS CUARENTA Y SIETE EUROS Y SESENTA CÉNTIMOS.

7.2 Recursos materiales

Para el cálculo del coste relativo a los recursos materiales empleados, tanto *hardware* como *software*, se tiene en cuenta un periodo de amortización de 3 años de carácter lineal. El cálculo de la cuota de amortización anual se obtendrá mediante la expresión 7.1:

$$C = \frac{V_{ad} - V_{res}}{N} \tag{7.1}$$

donde *C* hace referencia a la cuota de amortización anual; *V_ad* es el valor de adquisición; *V_res* equivale al valor residual, el cuál será considerado 0; y *N* el número de años considerados para la amortización del producto.

7.2.1 Recursos *software*

Teniendo en cuenta que el *software* empleado para el desarrollo del trabajo, desde la etapa de diseño del bloque HLS hasta la validación del sistema completo, ha sido *Vivado Desaign Suite HL WebPACK Edition*, cuya licencia es gratuita para estudiantes (aplicando ciertas restricciones en términos de líneas de código y dispositivos a usar para la implementación), y puesto que Overleaf ha sido el editor de texto utilizado para la redacción del documento (empleando LaTeX), también gratuita, los costes *software* dentro del presupuesto suponen un total de cero euros.

7.2.2 Recursos *hardware*

Dado que el trabajo se ha elaborado en un periodo inferior a 3 años (que es el estipulado para la amortización), se realiza una amortización equiparable al periodo de duración del trabajo (4 meses). Los resultados finales se observan en la Tabla 7.2:

Tab. 7.2: Precios y costes de los recursos *hardware*

Descrip.	Unid.	Val. de adq.	Tpo. de uso	C. anual	C. final
Pynq-Z1	1	232.19 €	4 meses	58.04 €	19.35 €
Ordenador	1	775 €	4 meses	258.33 €	86.11 €
Total					**105.46 €**

El coste final para el material *hardware* es de CIENTO CINCO EUROS Y CUARENTA Y SEIS CÉNTIMOS.

7.3 Redacción del documento

El coste de la redacción del documento se calcula a través de la ecuación 7.2, donde *P* es el presupuesto y C_n el coeficiente de ponderación del presupuesto. Para este caso, C_n vale uno debido a que el coste total del proyecto no supera los 30,050.00 €.

$$R = 0{,}07 \times P \times C_n \tag{7.2}$$

El presupuesto se calcula como la suma del coste de las amortizaciones y el trabajo tarifado por tiempo empleado. El resultado de esta operación se observa en la Tabla 7.3.

Tab. 7.3: Suma de las amortizaciones y el tarifado por tiempo empleado

Concepto	Coste asociado
Tarifado por tiempo empleado	2847.60 €
Amortización *software*	0 €
Amortización *hardware*	105.46 €
Total	**2953.06 €**

Finalmente, tras hallar el precio del presupuesto, se aplica la ecuación 7.3 y se obtiene lo siguiente:

$$R = 0{,}07 \times 2953{,}06 \times 1 = 206{,}71 \tag{7.3}$$

El coste asociado a la redacción del documento es de DOSCIENTOS SEIS EUROS Y SETENTA Y UN CÉNTIMOS.

7.4 Derechos de visado del COITT

El coste del visado dependerá del presupuesto del proyecto; es decir, el mencionado previamente añadiendo el coste de la redacción del documento y el coeficiente reductor en función del presupuesto, que al igual que antes, al no alcanzar los umbrales, será unitario. Por tanto, según lo establecido por el COITT en [45], el coste del visado para el presente proyecto será:

$$R = 0{,}05 \times (2953{,}06 + 206{,}71) \times 1 = 157{,}99 \tag{7.4}$$

El coste final de los derechos de visado del COITT es de CIENTO CINCUENTA Y SIETE EUROS Y NOVENTA Y NUEVE CÉNTIMOS.

7.5 Presupuesto final del proyecto

La Tabla 7.4 muestra, con todos los apartados desglosados, el coste total del proyecto:

Tab. 7.4: Coste final del proyecto

Concepto	**Coste asociado**
Tarifado por tiempo empleado	2847.60 €
Recursos *software*	0 €
Recursos *hardware*	105.46 €
Redaccción del documento	206.71 €
Visado COITT	157.99 €
Subtotal	3317.76 €
IGIC (7 %)	232,24 €
Total	**3550.00 €**

De esta forma, el presupuesto total del trabajo asciende a TRES MIL QUINIENTOS CINCUENTA EUROS.

Bibliografía

[1] Cristóbal Nieto-Peroy y M. Reza Emami. „CubeSat Mission: From Design to Operation“. En: *Applied Sciences* 9.15 (2019), pág. 3110 (vid. pág. 1).

[2] K. Khurshid, R. Mahmood y Q. ul Islam. „A survey of camera modules for CubeSats - Design of imaging payload of ICUBE-1“. En: *2013 6th International Conference on Recent Advances in Space Technologies (RAST)*. 2013, págs. 875-879 (vid. pág. 1).

[3] R. Mahmood, K. Khurshid y Q. u. Islam. „Institute of Space Technology CubeSat: ICUBE-1 subsystem analysis and design“. En: *2011 Aerospace Conference*. 2011, págs. 1-11 (vid. pág. 2).

[4] James B. Bull Jaime Esper Thomas P. Flatley. *Small Rocket/Spacecraft Technology (SMART) Platform*. 2011. URL: https://earth.esa.int/web/eoportal/satellite-missions/s/smart (vid. pág. 3).

[5] Alex Knapp. *Google Is Selling Its Satellite Business Terra Bella To Satellite Startup Planet*. 2017. URL: https://www.forbes.com/sites/alexknapp/2017/02/07/google-is-selling-its-satellite-business-terra-bella-to-satellite-startup-planet/#500071962311 (vid. pág. 3).

[6] Christopher Beam. *Soon, satellites will be able to watch you everywhere all the time*. 2019. URL: https://www.technologyreview.com/2019/06/26/102931/satellites-threaten-privacy/ (vid. pág. 3).

[7] Debra Werner. *British startup Earth-i shares first color video from its VividX2 satellite, a prototype for imagery constellation*. 2018. URL: https://spacenews.com/earth-i-unveils-first-color-video-captured-by-its-vividx2-satellite/ (vid. pág. 3).

[8] Mohamed Abomhara, Othman Khalifa, Oula Zakaria y col. „Video Compression Techniques: An Overview“. En: *Journal of Applied Sciences* 10 (dic. de 2010) (vid. pág. 3).

[9] Yun Q. Shi y Huifang Sun. *Image and video compression for multimedia engineering: fundamentals, algorithms and standards*. 1.ª ed. CRC Press LLC, 2000 (vid. págs. 3, 17).

[10] M. Wirthlin. „High-Reliability FPGA-Based Systems: Space, High-Energy Physics, and Beyond“. En: *Proceedings of the IEEE* 103.3 (2015), págs. 379-389 (vid. pág. 5).

[11] R. L. Davidson y C. P. Bridges. „Error Resilient GPU Accelerated Image Processing for Space Applications“. En: *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), págs. 1990-2003 (vid. pág. 5).

[12]Edward J. Wyrwas. „Body of Knowledge for Graphics Processing Units (GPUs)“. En: 2018 (vid. pág. 5).

[13]S Habinc. „Suitability of reprogrammable FPGAs in space applications“. En: *Gaisler Research* (2002), págs. 1-44 (vid. pág. 5).

[14]Z. Li, J. Li, Y. Zhao, C. Rong y J. Ma. „A SoC Design and Implementation of H.264 Video Encoding System Based on FPGA“. En: *2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics*. Vol. 2. 2014, págs. 321-324 (vid. pág. 5).

[15]M. Kthiri, H. Loukil, I. Werda y col. „Hardware implementation of fast block matching algorithm in FPGA for H.264/AVC“. En: *2009 6th International Multi-Conference on Systems, Signals and Devices*. 2009, págs. 1-4 (vid. pág. 5).

[16]M. Kthiri, P. Kadionik, H. Lévi y col. „An FPGA implementation of motion estimation algorithm for H.264/AVC“. En: *2010 5th International Symposium On I/V Communications and Mobile Network*. 2010, págs. 1-4 (vid. pág. 5).

[17]J. Cong, B. Liu, S. Neuendorffer y col. „High-Level Synthesis for FPGAs: From Prototyping to Deployment“. En: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), págs. 473-491 (vid. pág. 6).

[18]Yanhu Shan, Yongfeng Ren, Guoyong Zhen y Kaiqun Wang. „An Enhanced Run-Length Encoding Compression Method for Telemetry Data“. En: *Metrology and Measurement Systems* 24 (sep. de 2017) (vid. pág. 9).

[19]B. Alias, A. Mehra y P. Harsha. „Hardware implementation and testing of effective DPCM image compression technique using multiple-LUT“. En: *2014 International Conference on Advances in Electronics Computers and Communications*. 2014, págs. 1-4 (vid. pág. 10).

[20]Douglas A. Kerr. „Chrominance Subsampling in Digital Images“. En: *The Pumpkin* 1 (ene. de 2009) (vid. pág. 10).

[21]S. Rimac-Drlje, O. Nemcic y M. Vranjes. „Scalable Video Coding extension of the H.264/AVC standard“. En: *2008 50th International Symposium ELMAR*. Vol. 1. 2008, págs. 9-12 (vid. pág. 13).

[22]X. Jiang, T. Song, T. Shimamoto y L. Wang. „AMVP prediction algorithm for adaptive parallel improvement of HEVC“. En: *2014 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2014, págs. 511-514 (vid. pág. 14).

[23]K. Imamura, M. Takabayashi y H. Hashimoto. „Comparison of motion vector detection based on gradient method for video sequences including large motions“. En: *6th International Conference on Signal Processing, 2002.* Vol. 1. 2002, 800-803 vol.1 (vid. pág. 17).

[24]A. N. Netravali y J. D. Robbins. „Motion-compensated television coding: Part I“. En: *The Bell System Technical Journal* 58.3 (1979), págs. 631-670 (vid. pág. 17).

[25]Shan Zhu y Kai-Kuang Ma. „A new diamond search algorithm for fast block matching motion estimation“. En: *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat.* Vol. 1. 1997, 292-296 vol.1 (vid. págs. 19, 20).

[26]Reoxiang Li, Bing Zeng y M. L. Liou. „A new three-step search algorithm for block motion estimation“. En: *IEEE Transactions on Circuits and Systems for Video Technology* 4.4 (1994), págs. 438-442 (vid. pág. 19).

[27]*Zynq-7000 SoC Data Sheet: Overview*. DS190. Xilinx Inc. 2018 (vid. págs. 23, 24, 26).

[28]*PYNQ-Z1 Board Reference Manual*. DIGILENT. 1300 Henley Court, Pullman, WA99163, 2017 (vid. págs. 24, 26, 27).

[29]*Vivado Design Suite User Guide Design Flows Overview*. Xilinx Inc. 2020 (vid. pág. 28).

[30]*Vivado Design Suite User Guide Designing with IP*. Xilinx Inc. 2020 (vid. pág. 29).

[31]Philippe Coussy, Daniel Gajski, Michael Meredith y Andres Takach. „An Introduction to High-Level Synthesis“. En: *Design Test of Computers, IEEE* 26 (sep. de 2009), págs. 8 -17 (vid. pág. 29).

[32]*Vivado Design Suite User Guide High-Level Synthesis*. Xilinx Inc. 2020 (vid. págs. 30, 31).

[33]*Vivado Design Suite User Guide: Using the Vivado IDE*. Xilinx Inc. 2018 (vid. pág. 31).

[34]*Vivado Design Suite User Guide Implementation*. Xilinx Inc. 2019 (vid. pág. 32).

[35]*Vivado Design Suite User Guide Programming and Debugging*. Xilinx Inc. 2020 (vid. pág. 32).

[36]*Integrated Logic Analyzer v6.1 LogiCORE IP Product Guide*. Xilinx Inc. 2016 (vid. pág. 32).

[37]*Xilinx Vitis Unified Software Platform User Guide*. Xilinx Inc. 2020 (vid. pág. 34).

[38]T. Szydzik, G. M. Callico y A. Nunez. „Efficient FPGA implementation of a high-quality super-resolution algorithm with real-time performance“. En: *IEEE Transactions on Consumer Electronics* 57.2 (2011), págs. 664-672 (vid. pág. 45).

[39]*AXI DMA v7.1, LogiCORE IP Product Guide*. Xilinx Inc. 2019 (vid. pág. 58).

[40]*AXI4-Stream Infrastructure IP Suite v3.0, LogiCORE IP Product Guide*. Xilinx Inc. 2018 (vid. págs. 59, 60).

[41]*AXI4-Stream FIFO v4.2, LogiCORE IP Product Guide*. Xilinx Inc. 2019 (vid. pág. 61).

[42]Chan. *FatFs - Generic FAT Filesystem Module*. 2019. URL: `http://elm-chan.org/` (vid. pág. 63).

[43]*UltraScale Architecture and Product Data Sheet: Overview*. DS890. Xilinx Inc. 2020 (vid. pág. 80).

[44]ULPGC. *Boletín Oficial de la Universidad de Las Palmas de Gran Canaria*. Jun. de 2019 (vid. pág. 81).

[45]COITT. *Derechos de visado 2018 COITT*. 2018 (vid. pág. 83).