



ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



PROYECTO DE TFG

**Implementación de la etapa de predicción del estándar de
compresión de imágenes hiperespectrales CCSDS 123.0-B-2 en
FPGA**

Titulación: Grado en Ingeniería en Tecnologías de la
Telecomunicación

Mención: Sistemas Electrónicos

Autor/a: Carlos Vega García

Tutores/as: Roberto Sarmiento Rodríguez
Yubal Barrios Alfaro

Fecha: Noviembre de 2020

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



PROYECTO DE TFG

Implementación de la etapa de predicción del estándar de
compresión de imágenes hiperespectrales CCSDS 123.0-B-2 en
FPGA

HOJA DE EVALUACIÓN

Calificación: _____
Presidente

Fdo.:
Vocal

Secretario/a

Fdo.:
Fecha: Noviembre de 2020

Fdo.:

AGRADECIMIENTOS

Me gustaría agradecer a todos los que han permitido que se realice este trabajo. Por ello, en primer lugar, me gustaría mencionar a mis padres y a toda mi familia. Sin su apoyo desde que era pequeño no estaría hoy escribiendo las últimas líneas de este trabajo.

También quiero agradecer a mis tutores, Yubal Barrios Alfaro y Roberto Sarmiento Rodríguez. Me han brindado la oportunidad de realizar este trabajo en un tema de actualidad y me han introducido dentro del sector aeroespacial, del que nunca me hubiese esperado participar. Indudablemente, agradezco su apoyo y esfuerzo durante la realización de este proyecto.

Por último, agradecer el apoyo que he recibido por los docentes que me han impartido clase en todos mis años de estudio. Los conocimientos que me han transmitido son la base sobre la que se ha fundamentado este proyecto, y la pasión de muchos por su trabajo, resulta una inspiración.

RESUMEN

En este trabajo se realiza la implementación en FPGA del predictor de imágenes hiperespectrales con pérdidas definido en el estándar CCSDS 123.0-B-2. Para lograr este objetivo, se ha optado por seguir un flujo de trabajo basado en el entorno de Matlab y Simulink, ya que ofrecía claras ventajas para la comprensión inicial del algoritmo y para reducir el tiempo de diseño.

Para facilitar la verificación, inicialmente se ha implementado la versión sin pérdidas del predictor, definida en el estándar CCSDS 123.0-B-1. Posteriormente, se han incluido sobre el diseño las mejoras que incorpora el nuevo predictor en su versión B-2. De esta forma, se ha podido validar y comparar el sistema diseñado con un trabajo realizado previamente dentro de la División de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA) que implementaba el CCSDS 123 en su versión sin pérdidas.

Como primer paso se desarrollaron y verificaron ambas versiones del predictor en el entorno de Matlab. El objetivo ha sido pasar de una descripción matemática, partiendo del documento del estándar, a un algoritmo funcional con un alto nivel de abstracción.

A continuación, se realizó la implementación sobre el entorno de Simulink, nuevamente para ambas versiones. Dentro de Simulink se ha realizado el diseño implementado en la FPGA. Mediante el paquete HDL Coder, se ha sintetizado el diseño en VHDL y se ha generado como bloque IP para ser implementado empleando el entorno de Vivado.

Para validar la implementación, se ha diseñado un banco de pruebas sobre una plataforma configurable de Xilinx de la familia ZYNQ. En este entorno se realizó la integración del predictor añadiendo los elementos necesarios para proporcionar al IP una imagen hiperespectral de prueba y almacenar los resultados generados (residuos) en una tarjeta SD.

A partir de los datos obtenidos, se ha comprobado el flujo de datos generado en la FPGA con los simulados inicialmente y con los de referencia. Los resultados obtenidos demuestran la funcionalidad del sistema diseñado y su validez frente al estándar. Finalmente, se comparan los resultados y rendimiento obtenido bajo este flujo de trabajo, con las implementaciones previas del estándar sin pérdidas, realizadas por el grupo DSI.

ABSTRACT

In this work, the lossy hyperspectral image predictor defined at the standard CCSDS 123.0-B-2 has been implemented on an FPGA. A design flow based on the Matlab and Simulink environments was followed to accomplish this goal. This offers an advantage in the initial comprehension of the algorithm, reducing at the same time the design process.

By choosing this design flow, it has been necessary to implement the lossless version of the predictor stage, defined in the previous CCSDS 123.0-B-1 standard. Then, the improvements incorporated by the new predictor were added to the design. By this way, we have been able to verify and compare the system designed with a previous implementation of the lossless predictor, developed by the DSI (División de Sistemas Integrados) of the IUMA (Instituto Universitario de Microelectrónica Aplicada), which is part of the ULPGC (Universidad de las Palmas de Gran Canaria).

Initially, both versions of the predictor were developed and tested in Matlab. The objective was to get a functional algorithm from the mathematical definition, taking the standard as starting point, although the result was at a high level of abstraction.

Next, we changed the environment to Simulink, developing again both versions of the predictor. The Simulink implementations are the ones that are going to be implemented on the FPGA. This is achieved by the HDL Coder package, which synthesizes the design to VHDL and creates an IP block to be implemented by using the Vivado environment.

The implementation has been validated using a benchmark run on an FPGA from the Zynq family provided by Xilinx. In this environment, the predictor was integrated with the system needed to feed the IP with a hyperspectral image and save the residuals generated in an SD card.

The data output obtained from the FPGA has been compared against the data obtained from the simulations and the golden reference. In this way, the functionality of the generated IP has been validated. Finally, we compared the results and performance obtained following this design flow, with the ones achieved by a previous implementation of the lossless standard developed by the DSI group.

Tabla de Contenidos

1.	Introducción.....	1
1.1.	Antecedentes	1
1.2.	Objetivos	7
1.3.	Estructura del documento	8
2.	Algoritmos de compresión de imágenes hiperespectrales	9
2.1.	Revisión del estado del arte	9
2.1.1.	CCSDS	9
2.1.2.	Fundamentos de un compresor de imágenes	10
2.2.	Algoritmos de compresión de imágenes existentes	12
2.2.1.	JPEG-2000	12
2.2.2.	JPEG-XR	14
2.2.3.	CCSDS 122.0-B-2	14
2.2.4.	CCSDS 122.1-B-2	16
2.3.	CCSDS 123.0-B-1 (<i>loss/less</i>).....	18
2.3.1.	Suma local (Local Sum).....	19
2.3.2.	Modos de predicción:.....	20
2.3.3.	Número de bandas de predicción	20
2.3.4.	Vector de diferencias locales.....	20
2.3.5.	Vector de pesos.....	21
2.3.6.	Cálculo de la predicción.....	22
2.3.7.	Actualización del vector de pesos	22
2.3.8.	Mapeo del residuo predicho	23
2.3.9.	Parámetros Implicados.....	24
2.4.	CCSDS 123.0-B-2 (near-lossless)	25
3.	Flujo de diseño empleado	31
3.1.	Matlab	32
3.2.	Simulink.....	33
3.3.	HDL Coder	35
3.4.	Vivado	39
4.	Diseño del predictor en Matlab-Simulink	45
4.1.	Introducción.....	45
4.2.	Modelado en Matlab.....	45

4.2.1. Entorno de pruebas	50
4.2.2. Verificación de la Implementación en Matlab	52
4.3. Diseño del diagrama de bloques en Simulink	55
4.3.1. Predictor Lossless	57
4.3.2. Lossy Compressor.....	68
4.3.3. Inicialización y configuración del predictor.....	74
4.4. Análisis resultados HDL-Coder	76
5. Validación del sistema y análisis de resultados.....	79
5.1. Descripción general del set-up.....	79
5.2. Diagrama de bloques del set-up en Vivado.....	80
5.3. Aplicación software en Vivado SDK.....	81
5.4. Metodología de verificación.....	84
5.5. Resultados obtenidos.....	88
5.6. Utilización de recursos	89
5.6.1. Predictor Lossless	89
5.6.2. Predictor Lossy	92
5.7. Throughput.....	93
5.8. Comparativa con otras implementaciones	95
6. Conclusiones y líneas futuras.....	101
6.1. Trabajos futuros	102
7. Presupuesto	105
7.1. Recursos humanos	105
7.2. Recursos hardware	106
7.3. Recursos software	106
7.4. Material fungible.....	107
7.5. Presupuesto total del proyecto.....	107
BIBLIOGRAFÍA.....	109

Índice de figuras

FIGURA 1: ESPECTRO DE REFLECTANCIA PARA DIFERENTES SUPERFICIES DE LA TIERRA A UNA ALTA RESOLUCIÓN ESPECTRAL.	2
FIGURA 2: ESQUEMA COMPRESIÓN IMÁGENES HIPERESPECTRALES EN EL ESPACIO.	4
FIGURA 3: ESQUEMA PREDICTOR DESCRITO EN EL ESTÁNDAR CCSDS123.0-B-1.	4
FIGURA 4: ESQUEMA GENERAL PREDICTOR DEFINIDO EN EL ESTÁNDAR CCSDS 123.0-B-2.....	5
FIGURA 5: ÁREAS DESARROLLO CCSDS [19].....	9
FIGURA 6: ESQUEMA GENÉRICO COMPRESOR LOSSY.	11
FIGURA 7: ESQUEMA GENÉRICO COMPRESOR LOSSLESS BASADO EN TRANSFORMADA	11
FIGURA 8: ESQUEMA GENERAL COMPRESOR JPEG-2000. [28].....	12
FIGURA 9: ESTRUCTURA ORGANIZACIÓN COEFICIENTES CODIFICADOR JPEG-2000.	13
FIGURA 10: FLUJO DE DATOS DENTRO DEL CODIFICADOR ENTRÓPICO DEL JPEG-2000.	13
FIGURA 11: ESQUEMA GENERAL COMPRESOR JPEG-XR.....	14
FIGURA 12: ESQUEMA GENERAL COMPRESOR CCSDS 122.0-B-2.	15
FIGURA 13: ESQUEMA BLOQUES SOBRE LA IMAGEN TRANSFORMADA MEDIANTE WAVELETS.	15
FIGURA 14: ESTRUCTURA DE UNA SEGMENTO CODIFICADO.	16
FIGURA 15: ESQUEMA GENERAL COMPRESOR CCSDS 122.1-B-2.	17
FIGURA 16: ESQUEMA GENERAL COMPRESOR [10].	18
FIGURA 17: ESQUEMA GENERAL PREDICTOR LOSSLESS.	19
FIGURA 18: MODOS CÁLCULO LOCAL SUM CCSDS 123.0-B-1 [10]	19
FIGURA 19: CÁLCULO DIRECCIONAL LOCAL DIFFERENCES CCSDS 123.0-B-1 [10].....	20
FIGURA 20: ESQUEMA FUNCIONAMIENTO CCSDS 123.0-B-2 [7].....	25
FIGURA 21: ENTORNO DE PRUEBAS SIMULINK DEL BLOQUE IP.	33
FIGURA 22: INTERFAZ WORKFLOW ADVISOR.	35
FIGURA 23: CONFIGURACIÓN INTERFACES E/S WORKFLOW ADVISOR.	36
FIGURA 24: PANTALLA FINAL WORKFLOW ADVISOR.....	37
FIGURA 25: CAPTURA ENTORNO VIVADO.....	38
FIGURA 26: CAPTURA SETTINGS VIVADO.....	39
FIGURA 27: CAPTURA ENTORNO VIVADO.....	40
FIGURA 28: DIAGRAMA DE BLOQUES VIVADO.	41
FIGURA 29: EJEMPLO VISTA IMPLEMENTACIÓN VIVADO.	42
FIGURA 30: EJEMPLO ENTORNO VITIS.	42
FIGURA 31: CAPTURA DE LAS HERRAMIENTAS DE DEPURACIÓN DE VIVADO	43
FIGURA 32: ESQUEMA FUNCIONAMIENTO IMPLEMENTACIÓN MATLAB CCSD123.0-B-1.....	47
FIGURA 33: ESQUEMA FUNCIONAMIENTO IMPLEMENTACIÓN MATLAB CCSD123.0-B-2.....	48
FIGURA 34: REPRESENTACIÓN ETAPAS PREDICCIÓN CCSDS 123.0-B-1 PARA LA IMAGEN AVIRIS EN BANDA 4... 49	49
FIGURA 35: REPRESENTACIÓN ETAPAS PREDICCIÓN CCSDS 123.0-B-2 PARA LA IMAGEN AVIRIS EN BANDA 4... 49	49
FIGURA 36: REPRESENTACIÓN DEL PROCESO DE VERIFICACIÓN PARA LA IMAGEN AVIRIS, PARA EL PREDICTOR BASADO EN CCSDS 123.0-B-1.	53
FIGURA 37: REPRESENTACIÓN DEL PROCESO DE VALIDACIÓN PARA LA IMAGEN AVIRIS, PARA PREDICTOR BASADO EN CCSDS 123.0-B-2.	54
FIGURA 38: ENTORNO DE PRUEBAS BLOQUE PREDICTOR.	55
FIGURA 39: DIAGRAMA PREDICTOR_BLOCK_IP.....	56
FIGURA 40: CAPTURA CICLO COMUNICACIONES SIMULINK.....	57
FIGURA 41: DIAGRAMA IMPLEMENTACIÓN SIMULINK PREDICTOR LOSSLESS.	59
FIGURA 42: IMPLEMENTACIÓN SIMULINK BLOQUE PREDICTOR_HIGH_RES.....	60
FIGURA 43: CÓMPUTO ERRORES SIMULINK.	61
FIGURA 44: IMPLEMENTACIÓN SIMULINK BLOQUE ACTUALIZACIÓN DE DIFERENCIAS.....	62
FIGURA 45: DIAGRAMA IMPLEMENTACIÓN SIMULINK PREDICTOR LOSSLESS.	70

FIGURA 46: IMPLEMENTACIÓN SIMULINK BLOQUE CUANTIFICADOR.	71
FIGURA 47: IMPLEMENTACIÓN SIMULINK SAMPLE REPRESENTATIVE.	72
FIGURA 48: IMPLEMENTACIÓN SIMULINK BLOQUE QUANTIZED_BIN_CENTER.	72
FIGURA 49: IMPLEMENTACIÓN SIMULINK BLOQUE DOUBLE_RESOLUTION_SAMPLE_REPRESENTATIVE.	73
FIGURA 50: IMPLEMENTACIÓN SIMULINK BLOQUE FINAL_SAMPLE_REPRESENTATIVE.	73
FIGURA 51: IMPLEMENTACIÓN EN SIMULINK DEL BLOQUE CUANTIFICADOR	76
FIGURA 52: KIT DE DESARROLLO PYNQ.	80
FIGURA 53: DISEÑO DE BLOQUES SISTEMA EN VIVADO.	80
FIGURA 54: IMAGEN INSTRUMENTACIÓN AVIRIS-NG [44].	86
FIGURA 55: IMAGEN DE REFERENCIA AVIRIS (SE REPRESENTA ÚNICAMENTE LA BANDA 4).	86
FIGURA 56: CAPTURA SALIDAS PREDICTOR EN VISTA COMBINADA.	88
FIGURA 57: RECURSOS CONSUMIDOS IMPLEMENTACIÓN LOSSLESS.	90
FIGURA 58: ANÁLISIS CONSUMO POTENCIA IMPLEMENTACIÓN LOSSLESS.	90
FIGURA 59: ANÁLISIS TEMPORAL LOSSLESS.	91
FIGURA 60: CAPTURA CICLO PREDICCIÓN BLOQUE IP.	91
FIGURA 61: RECURSOS CONSUMIDOS IMPLEMENTACIÓN LOSSY.	92
FIGURA 62: ANÁLISIS CONSUMO POTENCIA IMPLEMENTACIÓN LOSSY.	92
FIGURA 63: ANÁLISIS TEMPORAL LOSSY	93
FIGURA 64: ESQUEMA IMPLEMENTACIÓN CCSDS 123.0-B-1 SHYLOC [46]	96
FIGURA 65: GRÁFICO COMPARACIÓN RESULTADOS CON OTRAS IMPLEMENTACIONES DEL ESTÁNDAR CCSDS 123.0-B-1	98

Índice de tablas

TABLA 1: PARÁMETROS CONFIGURACIÓN CCSDS 123.0-B-1.	24
TABLA 2: PARÁMETROS CONFIGURACIÓN CCSDS 1230-B-2	29
TABLA 3: IMÁGENES PARA VERIFICACIÓN EN MATLAB	52
TABLA 4: CONFIGURACIONES VALIDACIÓN PREDICTOR LOSSY	54
TABLA 5: E/S PREDICTOR_BLOQUE_IP.	55
TABLA 6:E/S PREDICTOR BLOCK.	56
TABLA 7: VARIABLES DE CONFIGURACIÓN BLOQUE IP Y SUS VALORES DE REFERENCIA	75
TABLA 8:PARÁMETROS CONFIGURACIÓN TEST	87
TABLA 9: USO DE RECURSOS DE IMPLEMENTACIONES EN VHDL PREVIAS DEL CCSDS123.0-B-1 [46]	95
TABLA 10: USO DE RECURSOS BLOQUE IP LOSSLESS DESARROLLADO SOBRE ZYNQ-7020.	95
TABLA 11: COMPARACIÓN USO DE RECURSOS IMPLEMENTACIONES HLS	97
TABLA 12: COSTES EN RECURSOS HUMANOS DEL PROYECTO	105
TABLA 13: COSTE RECURSOS HARDWARE	106
TABLA 14: COSTE RECURSO SOFTWARE	106
TABLA 15: COSTE TOTAL DEL PROYECTO	107

ACRÓNIMOS

Acrónimo	Definición
ASIC	Application-Specific Integrated Circuit
AVIRIS	Airborne Visible/Infrared Imaging Spectrometer
AXI	Advanced eXtensible Interface
BPC	fractional Bit-Plane Coding
BPE	Bit Plane Encoder
BRAM	Block RAM
CCSDS	Consultative Committee for Space Data Systems
CHIME	Copernicus Hyperspectral Imaging Mission for the Environment
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DSP	Digital Signal Processor
EBCOT	Embedded Block Coding with Optimized Truncation
ESA	European Space Agency
FIFO	First In First Out
FPGA	Field-Programmable Gate Arrays
GPP	General Purpose Processor
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HDR	High Dynamic Range

HLS	High-Level Synthesis
ILA	Integrated Logic Analyzer
LUT	Lookup Table
PCT	Photo Core Transform
POT	Photo Overlap Transform
PS	Processing System
RAM	Random Access Memory
RGB	Red Green Blue
RTL	Register-Transfer Level
SEE	Single Event Effects
SNR	Signal to Noise Ratio
SOC	System On Chip
SWIR	Short-wave infrared
TID	Total Ionizing Doce
TLM	Transaction Level Model
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
WNS	Worst Negative Slack

1. Introducción

1.1. Antecedentes

Las cámaras hiperespectrales se diferencian de las tradicionales RGB (Red, Green, Blue) en que son capaces de captar centenares de longitudes de onda dentro y fuera del espectro visible. Desde los años 70 existen multitud de cámaras multiespectrales integradas en misiones espaciales que producen imágenes con unas pocas bandas de frecuencia, pero con ellas ha sido posible obtener mucha información que no somos capaces de ver a simple vista. Por tanto, si hemos sido capaces de esto con unas pocas bandas, ¿no sería mejor con cientos o incluso miles de bandas a analizar? [1]. Esta capacidad la aportan las imágenes hiperespectrales, que proporcionan centenares de bandas con resoluciones elevadas (menores a 5 nm).

En estos momentos, la *European Space Agency* (ESA), dentro del programa Copernicus [2], está desarrollando la sexta generación de satélites Sentinel [3] para la observación de la tierra. En los satélites que actualmente orbitan alrededor de la Tierra se encuentran cámaras multiespectrales, pero se espera que para próximas generaciones se dé el salto al uso de cámaras hiperespectrales, capaces de obtener mucha más información que las anteriores.

Entre las posibles ampliaciones de los satélites Sentinel se encuentra la misión CHIME (*Copernicus Hyperspectral Imaging Mission for the Environment*) [4]. Su objetivo principal es: *“Proveer de observaciones hiperespectrales rutinarias a través del programa Copernicus en apoyo de las políticas europeas para la gestión de los recursos naturales, bienes y beneficios. Esta única espectrografía basada en el rango SWIR (Short-wave infrared) permitirá soportar nuevos servicios para seguridad alimenticia, agricultura y materias primas. Esto incluye agricultura sostenible y gestión de la biodiversidad, caracterización de las propiedades del terreno, prácticas de minería sostenible y preservación del medio ambiente”* [5].

Todos estos servicios que se plantean son posibles gracias a las distintas propiedades de absorción de los distintos compuestos a bandas de radiación específicas. En la Figura 1 vemos el espectro de reflectancia (también conocido como firma espectral) de las siguientes superficies de la Tierra: vegetación, tierra, caolinita (tipo de arcilla) y agua; mientras que se marcan las bandas específicas captadas por las cámaras multiespectrales del satélite Sentinel 2. Se aprecia que el perfil espectrográfico contiene grandes variaciones y que, según el terreno, se obtienen perfiles completamente distintos. Además, se marcan las bandas específicas en las que se pueden reconocer las variaciones de distintos compuestos del terreno o vegetación. El potencial de las cámaras hiperespectrales en este ámbito, proviene de añadir un mayor número de bandas de rango estrecho (continuas) con una alta SNR (*Signal-to-Noise Ratio*) frente a los sistemas multiespectrales convencionales como el del Sentinel 2.

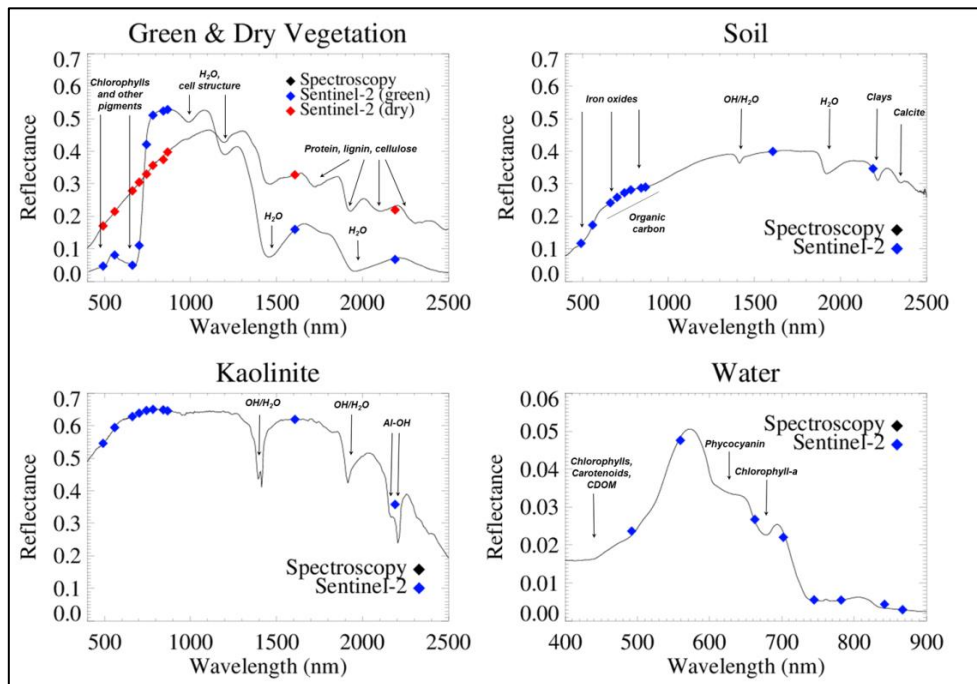


Figura 1: Espectro de reflectancia para diferentes superficies de la Tierra a una alta resolución espectral.

Ante este camino se presentan una nueva cantidad de retos, debido a que la tecnología hiperespectral es relativamente nueva a bordo de satélites y, por tanto, como siempre que algo nuevo se propone en el espacio, aparecen dificultades que en tierra no se plantean. Algunos ejemplos de estas pueden ser: las limitaciones en capacidad computacional del *hardware*, el consumo de potencia, el área ocupada o los efectos de la radiación sobre los dispositivos electrónicos.

Todo satélite que se envíe al espacio se encuentra dividido en dos secciones principales: plataforma y carga (*payload*). La plataforma consiste en 5 subsistemas encargados del soporte de la carga: subsistema estructural, subsistema de telemetría, subsistema de guiado y control, subsistema de generación eléctrica y distribución, subsistema de control térmico y subsistema de control de altura y velocidad. Esto establece las bases necesarias para la carga, como pueden ser las cámaras hiperespectrales [6].

Dentro del satélite, todos los componentes electrónicos deben trabajar correctamente, ya que cualquier fallo no podrá ser reparado, incurriendo en un incorrecto funcionamiento y pudiendo suponer el fin de la misión. Los componentes electrónicos necesitan tener un estándar de muy alta calidad y ser tolerantes a los efectos de la radiación. Las formas en las que la radiación puede afectar sobre los dispositivos electrónicos se divide en dos grupos: TID (*Total Ionizing Dose*) y SEEs (*Single Event Effects*) [7].

- TID es un efecto acumulativo de degradación, que ocurre durante el tiempo de vida de la misión. Se expresa en términos de tasa de error aleatoria.
- SEE se produce por partículas altamente energéticas que atraviesan los circuitos, inyectando energía sobre estos. Su efecto puede resultar en errores leves, como cambios en el estado de la memoria, o en errores graves, que producen daños permanentes en los equipos.

Estos errores provocados por la radiación suponen un desafío, ya que los componentes requieren de estrategias de diseño específicas para evitarlos. Antes de ser aptos para uso espacial, los componentes deben superar tests de altas y bajas dosis de radiación. Además, debe comprobarse que son resistentes a temperaturas de uso entre -55°C hasta 125°C. Como última etapa, es esencial que carezcan de los componentes prohibidos para su uso espacial, como estaño, zinc o cadmio, ya que sufren problemas de degradación [6].

Otra limitación muy importante en los satélites es el ancho de banda de transmisión con las estaciones terrenas. Este es un recurso limitado y ante la gran cantidad de datos que generaran las nuevas instrumentaciones se hace necesario comprimir la información, para hacer el menor uso posible de este recurso. Es en este sentido cuando la compresión de la información generada a bordo del satélite adquiere relevancia.

En la compresión de cualquier tipo de información podemos utilizar dos enfoques: compresiones sin pérdidas (*lossless*) o compresión con pérdidas (*lossy*). Una compresión *lossless* se caracteriza por eliminar la información redundante en los datos, pero manteniendo el 100% de su calidad, lo que la hace especialmente interesante para aplicaciones científicas, como detección de anomalías o clasificación, donde es relevante preservar la mayor cantidad de información posible. Esta técnica alcanza ratios de compresión moderados (normalmente entre 2 y 4). Una técnica *lossy* no solo elimina información redundante, sino que produce una pérdida, de mayor o menor medida, a cambio de una reducción del tamaño de la información, alcanzando ratios de compresión más altos que las técnicas *lossless*. [8]

En este contexto, el CCSDS (*Consultative Committee for Space Data Systems*)¹ ha definido varios estándares sobre la compresión de datos (incluyendo imágenes hiperespectrales) en instrumentos espaciales, tanto *lossless* como *lossy*. Este organismo fue fundado en 1982 por las mayores agencias espaciales del mundo, con el objetivo de desarrollar estándares para las comunicaciones y sistemas de datos para las misiones espaciales. Actualmente, es una referencia dentro del campo aeroespacial y más de 900 misiones han usado sus propuestas, de ahí el interés en el trabajo que realizan [9].

¹ <https://public.ccsds.org/default.aspx>

Este trabajo se centra en los estándares CCSDS 123.0-B-1 (*Standard for Lossless Multispectral & Hyperspectral Image Compression*) [10] y CCSDS 123.0-B-2 (*Low-Complexity Lossless and Near-Lossless Multispectral and Hyperspectral Image Compression*) [7]. Dichos estándares de compresión están especialmente pensados para imágenes en tres dimensiones, concretamente para imágenes multi e hiperespectrales. Enfocan su campo de uso en futuras misiones espaciales (Figura 2), donde este tipo de sensores se irán integrando con mayor asiduidad para tareas de observación y monitorización.

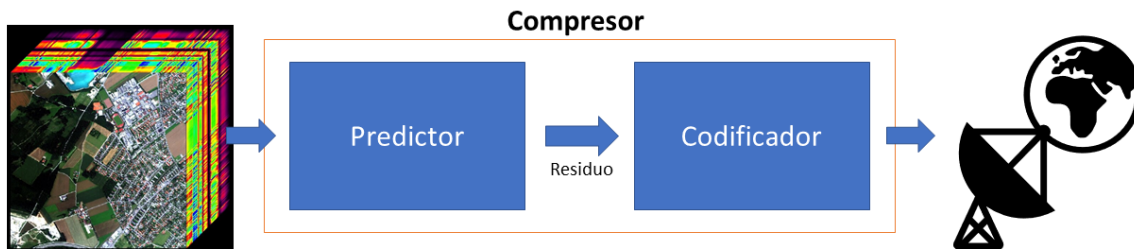


Figura 2: Esquema compresión imágenes hiperespectrales en el espacio.

El estándar CCSDS 123.0-B-1 surge en 2012 como un estándar de compresión *lossless* de imágenes en tres dimensiones. El estándar divide el compresor en dos etapas: predictor y codificador (Figura 2). En este trabajo, se analiza únicamente la etapa de predicción de ambas versiones del estándar. En la Figura 3 se encuentra el esquema de bloques general del predictor.

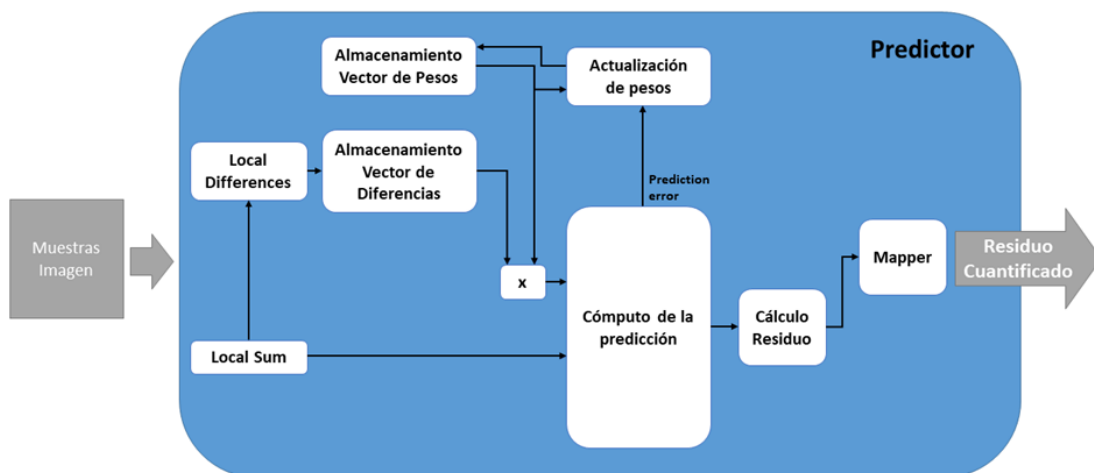


Figura 3: Esquema predictor descrito en el estándar CCSDS123.0-B-1.

La versión CCSDS 123.0-B-2 se publica en 2019 como una extensión del estándar original. En este se recogen profundos cambios a la estructura del predictor para permitir compresiones *near-lossless*, sin perder la compatibilidad con el modo *lossless*. Se han descrito dos nuevos bloques, *Quantizer* y *Sample Representative*, que integran las etapas de introducción de pérdidas y reconstrucción de la muestra predicha, para corregir el error introducido por el cuantificador (Figura 4). Si lo

comparamos con la versión previa, toda la etapa de predicción (Figura 3) se ha integrado dentro del bloque de cómputo de predicción, que permanece con menores variaciones.

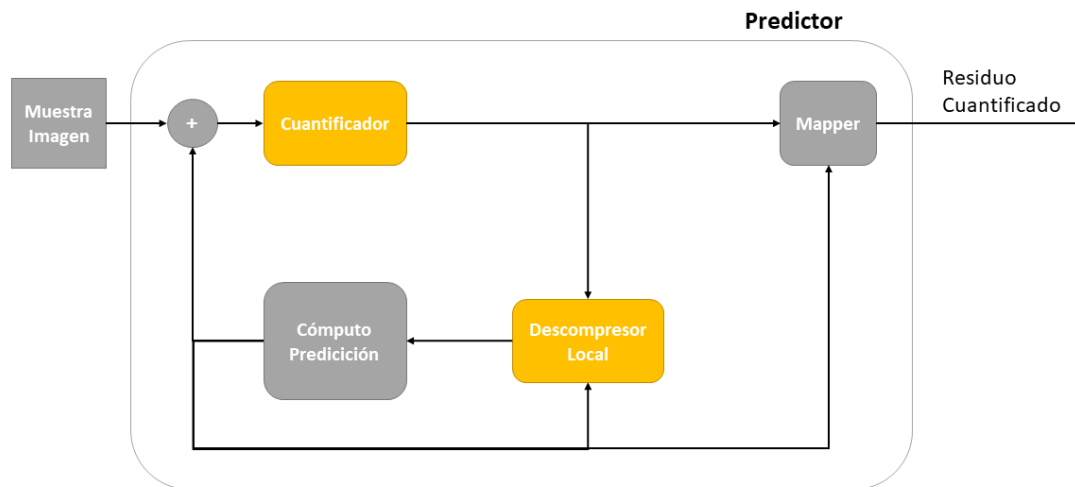


Figura 4: Esquema general predictor definido en el estándar CCSDS 123.0-B-2.

Por otro lado, las FPGAs (*Field-Programmable Gate Arrays*) son definidas por la empresa Xilinx como: “Dispositivos semiconductores basados en una matriz de bloques lógicos configurables conectadas vía interconexiones programables. Las FPGAs pueden ser reprogramadas para una aplicación o funcionalidad deseada tras su fabricación” [11].

Las FPGAs nos aportan múltiples ventajas frente a los GPPs (*General Purpose Processors*) como:

- Una mayor eficiencia energética.
- Gran capacidad de paralelismo en las tareas, que se traduce en un mayor rendimiento para cierta funcionalidad en la que no exista dependencia de datos.
- Flexibilidad para incluir nueva funcionalidad o para modificar la ya existente.

Todas estas ventajas llevan asociadas un mayor costo y tiempo de desarrollo que el desarrollo del software para los GPPs². Dentro del entorno espacial esta tecnología se encuentra en una curva de rápido aumento, por lo que resulta de gran interés de cara a un futuro cercano.

Dentro de la Universidad de Las Palmas de Gran Canaria (ULPGC), la división de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA) colabora en varios proyectos con la ESA y actualmente se encuentran en un nuevo proyecto implementando la nueva versión del estándar sobre lenguaje C, pero enfocado en su futura síntesis sobre lógica programable siguiendo técnicas basadas en *High-Level Synthesis* (HLS).

² En el sector aeroespacial la diferencia es menor, ya que el software tiene altos requisitos de fiabilidad.

Dentro del trabajo previo del grupo se encuentra el proyecto SHyLoC [12], en el que se ha realizado una implementación en VHDL (*VHSIC Hardware Description Language*) de la versión sin pérdidas del estándar (CCSDS 123.0-B-1). Dentro del proyecto se incluye la implementación en ASICs (*Application-Specific Integrated Circuit*) y FPGAs tolerantes a alta radiación, como la Xilinx Virtex5 FX130[13] o la Microsemi RTG4[14] o la Europea NanoXplore Brave.

1.2. Objetivos

Este proyecto se centra en la implementación en FPGA de la nueva funcionalidad de la etapa de predicción del estándar CCSDS 123.0-B-2, mediante un flujo de diseño basado en Matlab-Simulink.

Se realizarán dos IPs distintos, basados en los estándares CCSDS 123.0-B-1 y CCSDS 123.0-B-2, respectivamente. Primero se ha desarrollado el basado en el estándar *lossless*, del cual existen referencias de su implementación y resultados previos en el estado del arte [15]. Posteriormente, se desarrolla el IP basado CCSDS 123.0-B-2, como extensión del anterior, para lo cual se añaden los bloques *Quantizer* y *Sample Representative*, que permiten realizar compresiones con pérdidas.

La descripción final se ha realizado en lenguaje VHDL tecnológicamente independiente para permitir su implementación en diferentes tecnologías FPGA. En concreto este proyecto se validará usando FPGAs de Xilinx, aunque puede ser implementado en otras como por ejemplo la familia BRAVE de NanoXplore [16] u otros dispositivos resistentes a la radiación de distintos fabricantes.

A partir de dicho objetivo general, se citan los siguientes objetivos específicos:

- **Estudio en profundidad de los algoritmos CCSDS 123.0-B-1 y CCSDS 123.0-B-2 para la compresión con y sin pérdidas de imágenes hiperespectrales.**
- **Implementación del estándar CCSDS 123.0-B-1 para compresión *lossless* en VHDL.** Para esta implementación se utilizarán los desarrollos previamente realizados por el Grupo.
- **Implementación de las novedades aportadas por el estándar CCSDS 123.0-B-2 para compresión *near-lossless* en VHDL.** Este segundo IP, extenderá las funcionalidades del CCSDS123.0-B-1 con aquellas recogidas en el nuevo estándar (CCSDS123.0-B-2) para la compresión con pérdidas.
- **Verificación de las implementaciones realizadas mediante simulación.** Se debe simular el funcionamiento del predictor ante un flujo de datos de entrada continuo, proveniente de la instrumentación. Para poder realizarse la simulación, se debe desarrollar el software necesario para leer imágenes hiperespectrales de distintos tamaños y formatos (BSQ, BIL o BIP).
- **Validación completa del sistema sobre FPGA.** El objetivo final es obtener una implementación válida del estándar CCSDS123.0-B-2 para FPGA. El sistema desarrollado debe ser validado en un set-up de pruebas físico que demuestre su correcto funcionamiento y proporcione resultados sobre su rendimiento en términos temporales, de consumo de recursos (área) y de potencia.

1.3. Estructura del documento

En este primer capítulo se comentan los antecedentes que han derivado en la realización de este trabajo; también se expresa su propósito y sus objetivos concretos.

En el capítulo 2 se describe el algoritmo de compresión CCSDS 123.0-B-1 y se describen las características que añade la nueva versión (CCSDS 123.0-B-2). En este capítulo también se explica la función del comité CCSDS y donde se encuentra el estándar, dentro de su área de trabajo.

El capítulo 3 muestra el flujo de diseño empleado y las estrategias a utilizar en las herramientas implicadas.

A partir del capítulo 4 se pasa a describir la implementación que se ha realizado. En este cuarto capítulo se describe la implementación dentro del entorno de MathWorks, primero en Matlab [7] y posteriormente en Simulink [17]. Finalmente se realiza un breve análisis del código VHDL autogenerated.

El capítulo 5 explica cómo ha sido el set-up utilizado para las pruebas y se comentan los resultados obtenidos. Se describe también la integración realizada de los dos bloques IP generados en Vivado para el sistema de pruebas. Finalmente, se analizan los recursos utilizados sobre la FPGA y cómo se comparan sobre otras implementaciones previas.

En el capítulo 6 se citan las conclusiones alcanzadas tras el trabajo y se plantean los futuros trabajos a realizar sobre el proyecto.

2. Algoritmos de compresión de imágenes hiperespectrales

2.1. Revisión del estado del arte

2.1.1. CCSDS

En los antecedentes del proyecto se mencionó que el CCSDS ha definido el nuevo estándar de compresión de imágenes hiperespectrales CCSDS 123.0-B-2. Este estándar se incluye dentro del trabajo de estandarización realizado por el comité, en el ámbito de transmisión de datos en el espacio.

El comité CCSDS fue constituido en 1982 por las mayores agencias espaciales para proveer de un fórum de discusión común de los problemas en el desarrollo de sistemas de datos para el espacio [18]. Actualmente, esta agencia es el referente en su ámbito, contando con 11 agencias miembros, 28 agencias observadoras y más de 140 asociados industriales.

En la Figura 5 se encuentran las distintas áreas de las comunicaciones aeroespaciales que recoge el CCSDS en sus trabajos. El caso del estándar CCSDS 123.0 se encuentra dentro del área de Servicios de enlace espacial (*Space Link Services*), junto con otros 52 documentos. Dentro de este conjunto también se definen estándares para compresión de datos, protocolos por paquetes en el espacio, etc.

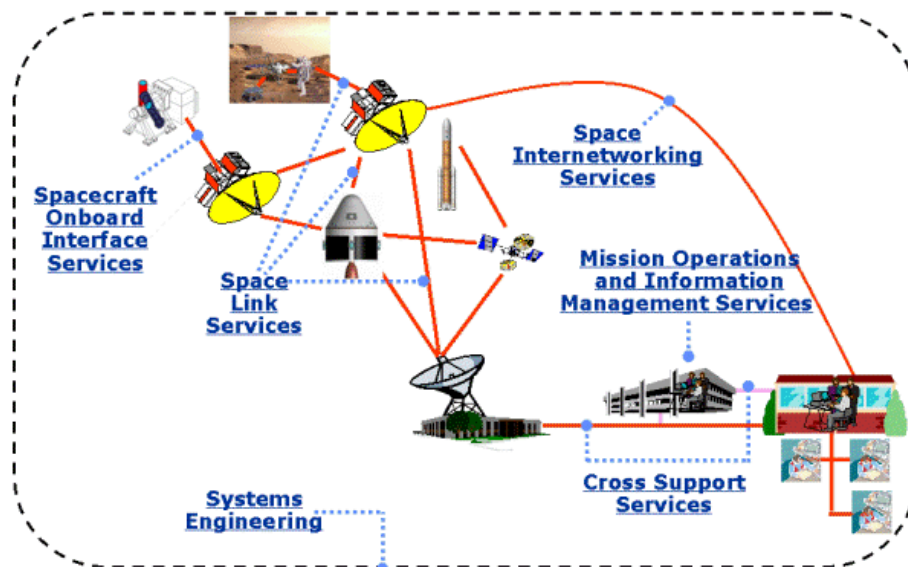


Figura 5: Áreas desarrollo CCSDS [19]

A su vez, la documentación generada se organiza por código de color según el tipo de información que proporcionan [20]:

- Azul: hace referencia a los estándares recomendados. Deben definir interfaces, capacidades técnicas o protocolos de forma específica.
- Magenta: las prácticas recomendadas. Se generan mediante la deliberación de la comunidad CCSDS y ofrecen una estrategia para obtener el mejor resultado en el ámbito que tratan.

- Verdes: documentos informativos. Su propósito es aportar información general de relevancia en el momento de su publicación.
- Naranjas: Experimentales. Documentos que reflejan líneas de trabajo aún en desarrollo y que pueden indicar cómo serán los estándares futuros.
- Amarillos: Registros. Información concreta sobre una reunión, prueba o trabajo.
- Gris: Histórico. Documento oficial emitido que ha sido actualizado por una versión posterior.

El compresor, que desarrolla este TFG, se refleja como un documento de tipo azul (un estándar oficial). Su primera definición es de febrero de 2019, aunque ha recibido una última actualización técnica en agosto de 2020. La versión anterior del estándar (CCSDS 123.0-B-1) fue definida en mayo 2012 y se encuentra recogida por el comité como documento histórico desde la aparición de la nueva versión.

2.1.2. Fundamentos de un compresor de imágenes

El propósito de un compresor de imágenes es reducir la información redundante mediante un método que permita su reconstrucción. En el proceso se debe de reducir el tamaño en bytes del fichero, pero sin suponer una pérdida de calidad hasta un punto indeseable [21].

Existen tres tipos fundamentales de redundancia en una imagen digital:

- Redundancia psico-visual: Información que el ojo no es capaz de captar o a la que muestra una baja sensibilidad.
- Redundancia entre píxeles: Redundancia por dependencias estadísticas entre píxeles, especialmente entre los adyacentes o vecinos.
- Redundancia en la codificación: Provocada al codificar cada valor con el mismo peso, independientemente de su repetibilidad en la imagen.

Los compresores tratan de minimizar estas redundancias mediante distintos métodos, pero a nivel general se dividen en dos grupos:

- Con pérdidas: El proceso de compresión es irreversible. La imagen reconstruida carece de información comparada con la original.
- Sin pérdidas: El proceso de compresión es reversible; por tanto, la imagen reconstruida contiene exactamente la misma información que la original.

Los algoritmos de compresión dentro de estos grupos suelen componerse por las mismas etapas, así que a continuación vamos a analizarlas. Para el caso de los compresores con pérdidas existen tres etapas, que luego se realizan en el descompresor de forma inversa (Figura 6). En la primera etapa de transformación se eliminan las redundancias entre píxeles. El cuantificador se encarga de eliminar las

redundancias visuales y de empaquetar la información con la menor cantidad de bits posibles. Finalmente, el codificador entrópico consigue una reducción extra eliminando la redundancia en la codificación [22].

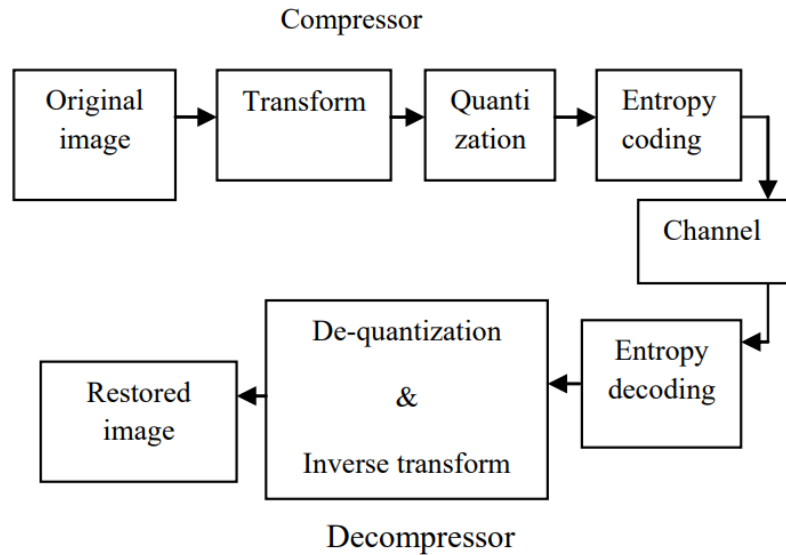


Figura 6: Esquema genérico compresor lossy.

Los compresores sin pérdidas son similares, pero se diferencian principalmente en que carecen de la etapa de cuantificación (Figura 7). Estos típicamente se basan en algoritmos de dos etapas.

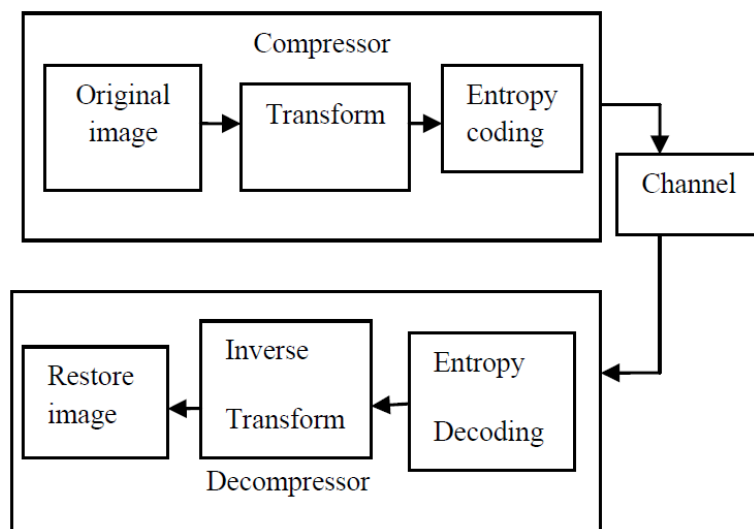


Figura 7: Esquema genérico compresor lossless basado en transformada

2.2. Algoritmos de compresión de imágenes existentes

A continuación, vamos a analizar el estado del arte en compresión de imágenes multispectrales/hiperestrales utilizados en la actualidad. Nos centraremos en aquellos que se han estandarizado y principalmente con aplicaciones espaciales en mente.

En primer lugar, expondremos algunos de los estándares de compresión de imágenes en dos dimensiones, ya que son la base para algunos de los estándares que se mencionaran posteriormente. En concreto haremos referencia al JPEG 2000 [23], que se basa en la transformada Wavelet. Continuaremos con el compresor JPEG-XR [24], el más reciente y que se basa en la transformada PCT (*Photo Core Transform*).

A continuación, analizaremos cómo se aplican los estándares de compresión de imágenes basados en transformada al sector espacial. El compresor CCSDS 122.0-B-2 [25], es el actual referente, basándose en la misma tecnología del JPEG-2000 pero adaptada a las necesidades específicas del sector. Este estándar se ha ampliado, en su versión CCSDS 122.1-B-2 [26], para su uso en imágenes de tres dimensiones.

2.2.1. JPEG-2000

JPEG 2000 es un compresor de imágenes basado en la transformada Wavelet. A nivel general, el modelo es el que vemos en la Figura 8. Tenemos una primera etapa de preprocesamiento, en el que la imagen se subdivide en secciones rectangulares y se convierte la codificación del color. A continuación, se realiza la transformada Wavelet, que nos permite descomponer la imagen en sus componentes de frecuencia horizontales y verticales para cada área [27]. Posteriormente, se introduce en el compresor, que se encarga de cuantificar de forma variable las componentes, para seguidamente eliminar la redundancia de codificación empleando un codificador entrópico. Finalmente, se encuentran los bloques de control de flujo y ordenación de los datos, que se encargan de equilibrar el flujo de datos saliente para que sea constante y de dar formato a estos [28].

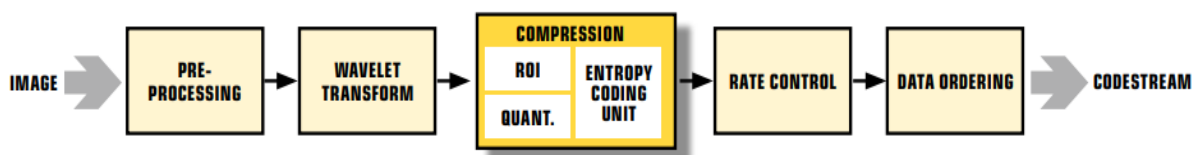


Figura 8: Esquema General Compresor JPEG-2000. [28]

Vamos a analizar a continuación cómo es el funcionamiento del codificador entrópico dentro del bloque compresor. Este es el bloque más complejo dentro del estándar y supone el 70% del tiempo de procesamiento para la compresión de una imagen.

Tras cuantificar los coeficientes de la transformada Wavelet, estos se subdividen en bloques rectangulares, denominados *code-blocks*. A su vez, estos bloques se subdividen en P planos de bits, que pasan a ser codificados del más significativo al menos significativo, siendo la estructura de datos resultante la de la Figura 9.

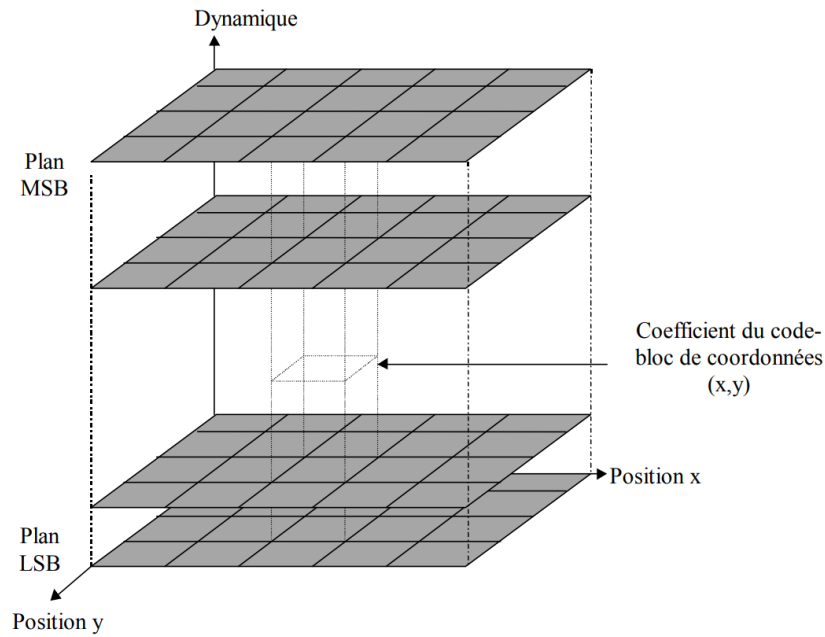


Figura 9: Estructura Organización Coeficientes Codificador JPEG-2000.

Cada plano de bits se codifica primero por un BPC (*fractional Bit-Plane Coding*), un mecanismo que genera datos intermedios en forma de contexto y un valor binario de decisión para cada posición. Para JPEG 2000, se ha elegido para la etapa BPC el algoritmo EBCOT (*Embedded Block Coding with Optimized Truncation*), que permite ajustar su salida a partir de unos coeficientes de entrada que se reciben junto los *code-blocks*. Los datos generados en este algoritmo son enviados a un codificador aritmético, denominado *MQ-coder*, que los codifica en un único flujo binario (Figura 10) [29].

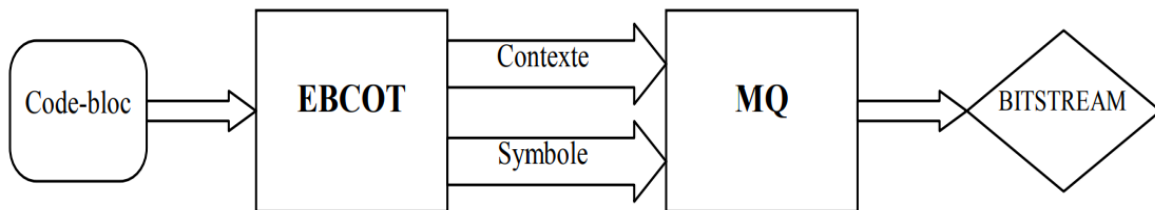


Figura 10: Flujo de datos dentro del codificador entrópico del JPEG-2000.

2.2.2. JPEG-XR

El estándar JPEG-XR surge como una mejora del estándar JPEG original de 1992. El JPEG se basa en la transformación de la imagen mediante la DCT (*Discrete Cosine Transform*), que tiene ligeras pérdidas debido a los errores de redondeo. El nuevo estándar utiliza una transformada basada en enteros denominada PCT, que evita el problema de la anterior al trabajar con coeficientes enteros. Frente al estándar JPEG-2000, busca subsanar dos de sus debilidades: menor uso de recursos computacionales y soporte para imágenes HDR (High Dynamic Range) [24].

Su funcionamiento se divide en 5 etapas principales (Figura 11).

- Transformación al espacio de color: Transforma la imagen de entrada al espacio YCbCr, en el que Y define la luminancia y Cb-Cr los componentes cromáticos.
- Transformada PCT/POT: Se divide la imagen en macrobloques y por sucesivas aplicaciones de las transformadas PCT y POT (*Photo Overlap Transform*) se obtienen los coeficientes de los macrobloques. La transformada POT se aplica en combinación con la PCT, ya que corrige los defectos que la primera introduce en las regiones límite de los bloques [30].
- Cuantificación: Se cuantifican los coeficientes generados con una resolución variable, permitiendo que se puedan realizar compresiones *lossless*.
- Codificador Entrópico: Elimina la redundancia en la codificación.
- Formateo del flujo: Se da formato al *bitstream* generado para su posterior reconstrucción.

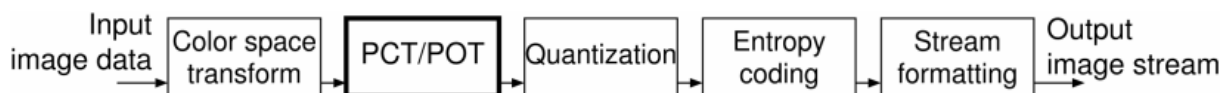


Figura 11: Esquema general compresor JPEG-XR.

2.2.3. CCSDS 122.0-B-2

La última versión del estándar fue publicada en septiembre de 2017 por el CCSDS. La técnica de compresión descrita permite realizar compresiones con y sin pérdidas. Al igual que el estándar JPEG-2000, se basa en la transformada Wavelet, pero difiere de esta en ciertos aspectos:

1. Se enfoca específicamente en instrumentos de gran flujo de datos a bordo de satélites.
2. Se ha compensado entre rendimiento en la compresión y la complejidad del sistema.
3. Una menor complejidad permite una mayor velocidad y su implementación en equipos de menor potencia.
4. Tiene un limitado conjunto de opciones.

El compresor viene definido por dos partes, resumidas en la Figura 12: un módulo DWT y un codificador entrópico de plano de bits.

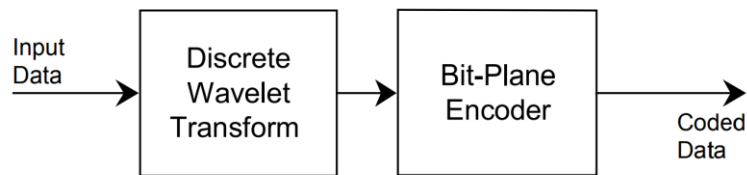


Figura 12: Esquema general compresor CCSDS 122.0-B-2.

El primer módulo utiliza tres niveles, de dos dimensiones, de la Transformada Discreta de Wavelet. La transformación se puede realizar por coeficientes *float* o por la aproximación de la transformada con número enteros. Para realizar compresión sin pérdidas se escoge la transformada con enteros, pero la de tipo *float* ofrece mejores resultados con pérdidas.

Los coeficientes obtenidos para cada plano tienen que ser convertidos a enteros, antes de pasar al codificador. El tamaño de palabra máximo con el que se representan viene definido por: la elección del tipo de transformada, los bits R de la imagen de entrada y si los píxeles son con signo o sin signo. En el caso de que la transformada sea de tipo *float*, se redondean los coeficientes para convertirlos a tipo entero. En caso de que sea de tipo entero, los coeficientes son multiplicados por el peso asociado a cada sub-banda.

El codificador de planos de bits, definido como BPE (*Bit Plane Encoder*), procesa los coeficientes generados en grupos de 64 coeficientes, definidos como bloques (Figura 13). Cada bloque corresponde con un área específica de la imagen original.

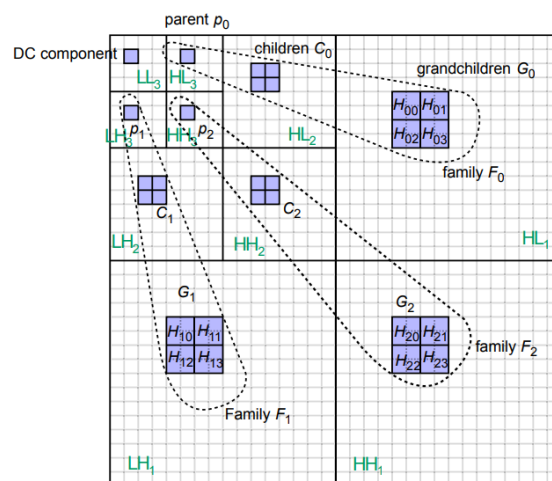


Figura 13: Esquema bloques sobre la imagen transformada mediante Wavelets.

En cada bloque hay un coeficiente de tipo DC (de continua) mientras que el resto son de tipo AC (de frecuencia). A su vez, los AC son divididos en tres familias y posteriormente en padres, hijos y nietos.

Para su procesamiento, los bloques se agrupan en segmentos de un número S de bloques. Y dentro de los segmentos en *gaggles*, que son conjuntos de 16 bloques.

A continuación, se codifican los datos de cada segmento. Para ello, primero se codifican los datos de los coeficientes DC de cada bloque, y posteriormente los coeficientes se transmiten por etapas. Primero, se calcula la profundidad de bits necesaria para transmitir el mayor coeficiente del segmento y posteriormente se envían en etapas, del bit más significativo al menos significativo (Figura 14).

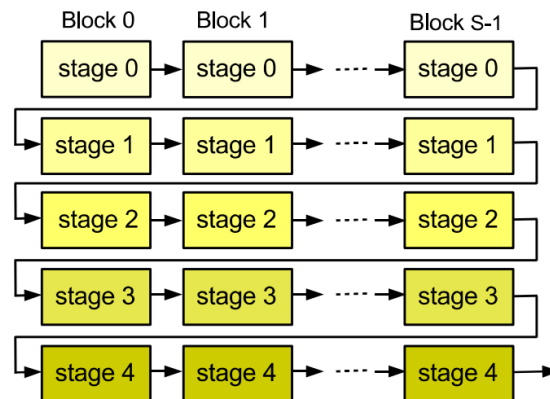


Figura 14: Estructura de una segmento codificado.

La compensación entre la calidad reconstruida y el volumen de los datos comprimidos para cada segmento se especifica mediante:

- *SegByteLimit*: Define el tamaño de *bytes* máximo que puede ocupar un segmento.
- Límites de calidad: Limitan la cantidad de información que puede ser codificada de cada coeficiente.

Los datos de cada segmento son codificados hasta que llegamos a uno de estos límites.

2.2.4. CCSDS 122.1-B-2

Este estándar se publica de forma conjunta con el estándar CCSDS 122.0-B-2 en septiembre de 2017. Extiende los conceptos analizados en el apartado 2.2.3 para su aplicación sobre imágenes 3D.

El compresor se divide en dos partes principales (Figura 15): una transformación espectral y un conjunto de codificadores 2D.

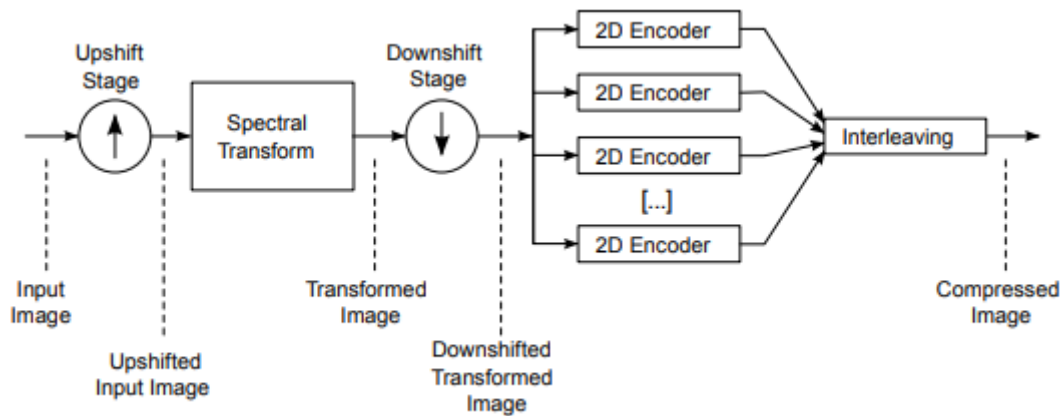


Figura 15: Esquema general compresor CCSDS 122.1-B-2.

El propósito de la transformación espectral es explotar las similitudes entre las distintas bandas de la imagen (redundancia espectral), creando una representación que pueda ser comprimida de forma más eficiente por los codificadores bidimensionales. Dentro del estándar se recogen múltiples transformadas para esta etapa, que podrán ser elegidas según la naturaleza de los datos de entradas y del costo computacional asociado.

Los codificadores 2D comprimen de forma independiente cada banda de la imagen transformada. Estos son equivalentes a los descritos previamente en el estándar CCSDS 122.0-B-2. Cada codificador 2D puede ser reutilizado múltiples veces en una compresión para codificar varias bandas de la imagen transformada, pero aumentando el número de codificadores se permite escalar el nivel de paralelismo de la compresión.

Finalmente, también se incluyen los bloques *upshift* y *downshift*, que permiten adaptar la profundidad de bits para la etapa de transformación espectral.

2.3. CCSDS 123.0-B-1 (*lossless*)

El propósito del estándar CCSDS 123.0-B-1, tal y como se define en el documento, es establecer un algoritmo de compresión de datos aplicado a imágenes tridimensionales provenientes de los sensores de imágenes multi e hiperespectrales. Como principales beneficios asociados a este estándar y la compresión en general se nombran:

- Reducción del ancho de banda del canal de transmisión.
- Reducción de los requerimientos de *buffering* y almacenamiento.
- Reducción de la transmisión de datos a una temporización marcada.

Dentro del estándar se describen dos etapas principales: *predictor* y *encoder*. En este trabajo solo trataremos la etapa de predicción; pero el codificador, pese a tener uno descrito, puede intercambiarse por otros definidos también por el CCSDS. El esquema general del compresor se aprecia en la Figura 16.

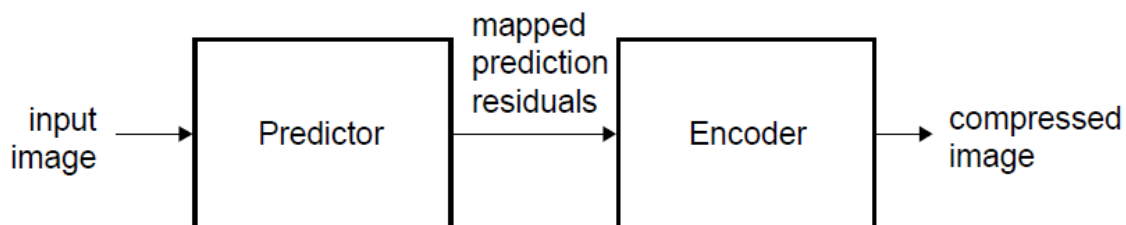


Figura 16: Esquema general compresor [10].

El predictor especificado puede ser implementado para trabajar la imagen en un único ciclo. Su funcionamiento se basa en la generación de un residuo $\{\delta_{z,y,x}\}$, partiendo de la diferencia entre la muestra $\{S_{z,y,x}\}$, de la imagen entrante y la muestra predicha $\{\hat{S}_{z,y,x}\}$.

El cálculo de la muestra predicha $\{\hat{S}_{z,y,x}\}$ proviene de dos flujos de datos:

- Tenemos un flujo directo que proviene de la suma local (*local sum*) y que se alimenta de las muestras entrantes de la imagen.
- Tenemos un flujo de realimentación. Este se implementa por medio del vector de diferencias y de pesos, y sus valores provienen de los errores producidos en las predicciones previas.

Estos dos flujos de datos se unen dentro de la función de cálculo de la predicción y generan un valor esperado, que se compara con la muestra entrante.

Tras este análisis general, pasamos a analizar todos los elementos que intervienen en dicha etapa de predicción. El esquema general se encuentra en la Figura 17.

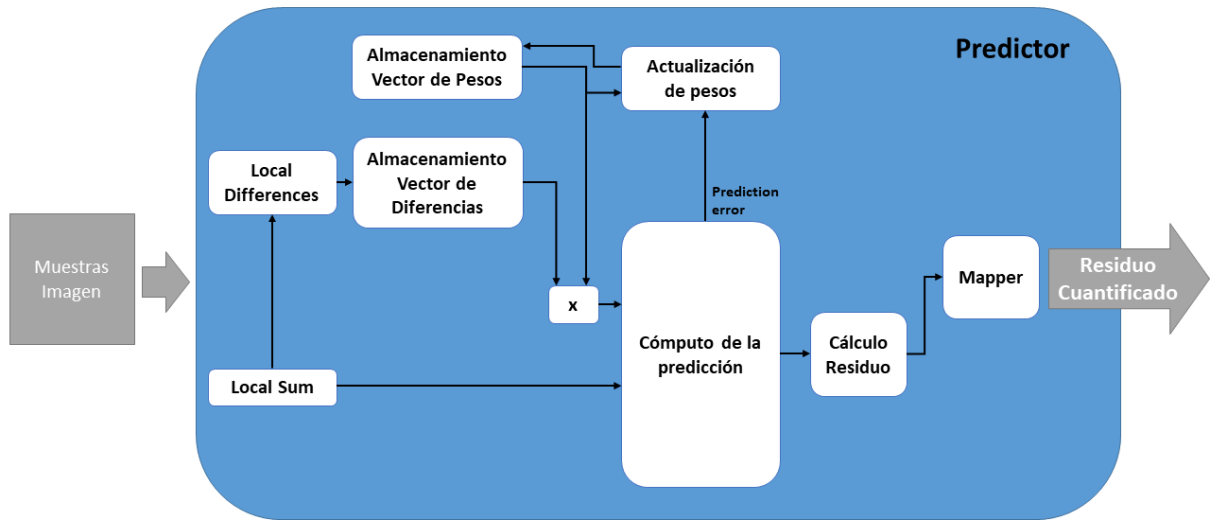


Figura 17: Esquema general predictor lossless.

2.3.1. Suma local (Local Sum)

La *local sum* $\{\sigma_{z,y,x}\}$ es una suma compensada de las muestras adyacentes a $\{s_{z,y,x}\}$, que se encuentran en la misma banda z . Dentro del estándar se plantean dos modos de cálculo: *neighbour-oriented* y *column-oriented*. Sus estrategias de cálculo se ven en la Figura 18, aunque tienen excepciones cuando la posición es $x=0$, $y=0$ o $x = N_x-1$.

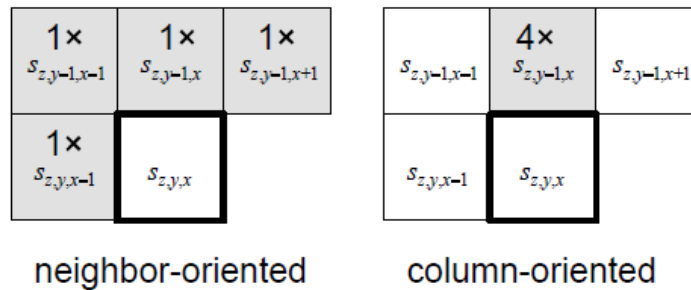


Figura 18: Modos cálculo local sum CCSDS 123.0-B-1 [10]

Por tanto, la ecuación para el cálculo mediante *neighbor-oriented* es:

$$\sigma_{z,y,x} = \begin{cases} s_{z,y,x-1} + s_{z,y-1,x-1} + s_{z,y-1,x} + s_{z,y-1,x+1}, & y > 0, 0 < x < N_x - 1 \\ 4s_{z,y,x-1}, & y = 0, x > 0 \\ 2(s_{z,y-1,x} + s_{z,y-1,x+1}), & y > 0, x = 0 \\ s_{z,y,x-1} + s_{z,y-1,x-1} + 2s_{z,y-1,x}, & y > 0, x = N_x - 1, \end{cases} \quad (1)$$

Y mediante el modo *column-oriented* es:

$$\sigma_{z,y,x} = \begin{cases} 4s_{z,y-1,x}, & y > 0 \\ 4s_{z,y,x-1}, & y = 0, x > 0. \end{cases} \quad (2)$$

2.3.2. Modos de predicción:

Dentro del estándar se recogen dos modos de predicción: completo y reducido. En el documento trataremos el reducido, ya que es el implementado en este TFG.

Su principal diferencia radica en que añadimos en los cálculos las tres *directional local differences*: $d_{z,y,x}^N$, $d_{z,y,x}^W$, $d_{z,y,x}^{NW}$, si seleccionamos el modo completo de predicción. En caso contrario (esto es, bajo el modo reducido), su valor equivale a la diferencia entre la local sum $\{\sigma_{z,y,x}\}$ y 4 veces la muestra señalada, como se puede ver en la Figura 19.

NW $s_{z,y-1,x-1}$	N $s_{z,y-1,x}$	$s_{z,y-1,x+1}$
W $s_{z,y,x-1}$	central $s_{z,y,x}$	

Figura 19: Cálculo Directional Local Differences CCSDS 123.0-B-1 [10].

El valor calculado pasa posteriormente a integrarse en las tres primeras posiciones del vector de diferencias y se añaden tres posiciones al vector de pesos para equilibrarlo.

2.3.3. Número de bandas de predicción

Se basa en el valor P determinado por el usuario, que puede ser un número entero entre 0 y 15. Representa el número de bandas previas que afectan sobre el cálculo de la muestra actual. Además, este valor afecta al tamaño de los vectores de pesos y diferencias, cuyo tamaño viene definido por C_z . Este valor se obtiene de la siguiente ecuación:

$$C_z = \begin{cases} P_z^*, & \text{reduced prediction mode} \\ P_z^* + 3, & \text{full prediction mode.} \end{cases} \quad (3)$$

2.3.4. Vector de diferencias locales

El vector de diferencias locales $U_z(t)$ es un vector de longitud C_z que es usado en el cálculo de la muestra predicha. En el modo reducido su definición se encuentra en la ecuación 4, a excepción de cuando estamos en la primera banda, en cuyo caso sus componentes deben ser iguales a 0.

$$\mathbf{U}_z(t) = \begin{bmatrix} d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P_z^*}(t) \end{bmatrix}. \quad (4)$$

Como se aprecia, se compone por C_z diferencias previas para la posición actual. Para ser más concretos, los componentes $d_z(t)$ hacen referencia al valor *de las diferencias centrales*. Esta diferencia representa el error generado entre la *local sum* y la muestra entrante. Su definición es la siguiente:

$$d_{z,y,x} = 4s_{z,y,x} - \sigma_{z,y,x}. \quad (5)$$

2.3.5. Vector de pesos

El vector de pesos $\mathbf{W}_z(t)$ es, al igual que el vector de diferencias, un vector de C_z posiciones. En combinación con el vector de diferencias, es usado para el cálculo de la muestra predicha. La definición $\mathbf{W}_z(t)$ para la predicción reducida es esta:

$$\mathbf{W}_z(t) = \begin{bmatrix} \omega_z^{(1)}(t) \\ \omega_z^{(2)}(t) \\ \vdots \\ \omega_z^{(P_z^*)}(t) \end{bmatrix} \quad (6)$$

En esta, cada valor $\omega_z^i(t)$ corresponde con los componentes del vector de pesos. Estos deben ser inicializados debidamente y son actualizados tras cada predicción, partiendo del error obtenido.

Su proceso de inicialización por defecto se define en la fórmula 7, pero el estándar también recoge la posibilidad de alterar este proceso. Para el inicio customizado se debe partir de un vector Λ_z , que puede ser generado externamente, y definir el parámetro *weight initialization resolution* (Q). La función 8 define cómo realizar el cálculo del valor inicial del vector partiendo de estos.

$$\omega_z^{(1)}(1) = \frac{7}{8}2^\Omega, \quad \omega_z^{(i)}(1) = \left\lfloor \frac{1}{8}\omega_z^{(i-1)}(1) \right\rfloor, \quad i = 2, 3, \dots, P_z^*. \quad (7)$$

$$\mathbf{W}_z(1) = 2^{\Omega+3-Q} \Lambda_z + \left[2^{\Omega+2-Q} - 1 \right] \mathbf{1} \quad (8)$$

2.3.6. Cálculo de la predicción

El cálculo de la muestra predicha se realiza partiendo de la *local sum*, el vector de diferencias y el de pesos.

En la versión *lossless*, la predicción se realiza en dos etapas: predicción a doble resolución y reducción a escala final. El estándar con pérdidas divide la predicción a doble resolución en dos etapas, pasando a ser 3 en total: predicción de alta resolución, reducción a doble resolución y reducción a escala final. El funcionamiento de ambos es equivalente y por ello se ha hecho uso únicamente de la segunda definición, que se explicará en la siguiente versión del estándar.

Analizando la definición descrita en la versión original, primero se debe calcular la *predicted central local difference*. Esta corresponde con el producto de matrices entre el vector de pesos y el vector de diferencias (Ecuación 9). En caso de que $z = 0$, el resultado de esta operación debe ser 0.

$$\hat{d}_z(t) = \mathbf{W}_z^T(t) \mathbf{U}_z(t) \quad (9)$$

A continuación, se debe realizar el cálculo de la muestra predicha escalada ($\tilde{s}_z(t)$), la cual es un entero con el doble de resolución de la muestra de entrada. Su cálculo es el siguiente:

$$\tilde{s}_z(t) = \begin{cases} \text{clip} \left(\left\lfloor \frac{\text{mod}_R^* \left[\hat{d}_z(t) + 2^\Omega (\sigma_z(t) - 4s_{\text{mid}}) \right]}{2^{\Omega+1}} \right\rfloor + 2s_{\text{mid}} + 1, \{2s_{\text{min}}, 2s_{\text{max}} + 1\} \right), & t > 0 \\ 2s_{z-1}(t), & t = 0, P > 0, z > 0 \\ 2s_{\text{mid}}, & t = 0 \text{ and } (P = 0 \text{ or } z = 0). \end{cases} \quad (10)$$

Finalmente, este valor debe ser reducido al tamaño de la muestra original. Para ello, se divide entre 2 y se trunca el residuo de la operación (Ecuación 11).

$$\hat{s}_z(t) = \left\lfloor \frac{\tilde{s}_z(t)}{2} \right\rfloor. \quad (11)$$

2.3.7. Actualización del vector de pesos

Tras realizarse el cálculo de la muestra predicha, se deben recalcular los errores y actualizar el vector de pesos. Estos se actualizan a partir del *scaled prediction error* (Ecuación 12), que básicamente compara la predicción de doble resolución con la muestra entrante.

$$e_z(t) = 2s_z(t) - \tilde{s}_z(t). \quad (12)$$

A continuación, es necesario calcular el exponente de escalado de la actualización de pesos (*weight update scaling exponent*), que es un exponente $\rho(t)$ que permite adaptar la velocidad de actualización de los pesos a los cambios en la imagen:

$$\rho(t) = \text{clip} \left(v_{\min} + \left\lfloor \frac{t - N_x}{t_{\text{inc}}} \right\rfloor, \{v_{\min}, v_{\max}\} \right) + D - \Omega, \quad (13)$$

El control del valor lo realizaremos por medio de las siguientes variables:

- v_{\min} , que establece el valor mínimo del exponente.
- v_{\max} , que establece el valor máximo del exponente.
- t_{inc} , que es el factor de cambio en el intervalo; debe ser una potencia de 2 entre 2^4 y 2^{11} .

Finalmente, se actualizan los componentes del vector de pesos. Su valor se limita entre ω_{\max} y ω_{\min} .

$$\mathbf{W}_z(t+1) = \text{clip} \left(\mathbf{W}_z(t) + \left\lfloor \frac{1}{2} (\text{sgn}^+ [e_z(t)] \cdot 2^{-\rho(t)} \cdot \mathbf{U}_z(t) + \mathbf{1}) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (14)$$

2.3.8. Mapeo del residuo predicho

Esta función define el proceso para mapear el residuo como un valor sin signo. Este puede ser representado por un entero sin signo (*unsigned*) de D bits. Siendo D el rango dinámico. La función que lo define es la siguiente:

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t), & |\Delta_z(t)| > \theta_z(t) \\ 2|\Delta_z(t)|, & 0 \leq (-1)^{\tilde{s}_z(t)} \Delta_z(t) \leq \theta_z(t) \\ 2|\Delta_z(t)| - 1, & \text{otherwise} \end{cases} \quad (15)$$

Como se aprecia en la ecuación, hay una dependencia con los siguientes valores:

- $\Delta_z(t)$, que indica el valor del residuo de predicción.
- $\theta_z(t)$, que indica la diferencia mínima entre el valor predicho y uno de los máximos del rango dinámico, siguiendo la Ecuación 16:

$$\theta_z(t) = \min \{ \hat{s}_z(t) - s_{\min}, s_{\max} - \hat{s}_z(t) \}. \quad (16)$$

2.3.9. Parámetros Implicados.

En este punto, ya se han visto todas las etapas implicadas en el predictor del estándar CCSDS 123.0-B-1. A continuación, se incluye Tabla 1 con todos los parámetros de configuración del predictor, sus significados y rangos de valores.

Tabla 1: Parámetros configuración CCSDS 123.0-B-1.

Parámetro	Descripción	Rango Valores
N_x	Tamaño Dimensión X	X
N_y	Tamaño Dimensión Y	Y
N_z	Tamaño Dimensión Z	Z
S_{max}	Valor máximo muestra	$2^{16} - 1$
S_{mid}	Valor medio muestra	$2^{(16-1)}$
S_{min}	Valor mínimo muestra	0
D	Rango dinámico en bits	16
P_z	Número de bandas previas en predicción	$0 \leq P_z \leq 15$
Ω	Resolución pesos	$4 \leq \Omega \leq 19$
R	Tamaño Registro	$\max\{32, D + \Omega + 2\} \leq R \leq 64$
v_{max}	Límite superior del weight update scaling exponent	$v_{min} \leq v_{max} \leq 9$
v_{min}	Límite inferior del weight update scaling exponent	$-6 \leq v_{min} \leq v_{max}$
t_{inc}	Factor que regula la velocidad de adaptación de los pesos a la imagen.	$2^4 \leq t_{inc} \leq 2^{11}$
ω_{max}	Límite superior componentes del vector de pesos	$2^{\Omega+2} - 1$
ω_{min}	Límite inferior componentes del vector de pesos	$-2^{\Omega+2}$

2.4. CCSDS 123.0-B-2 (near-lossless)

Esta nueva versión del documento busca extender el estándar CCSDS 123.0-B-1 para añadirle métodos de compresión con pérdidas o casi sin pérdidas (*near-lossless*). Con el término *near-lossless* se hace referencia a la capacidad de realizar la compresión en modo que el máximo error introducido en la imagen reconstruida (es decir, las pérdidas) pueda ser especificado por el usuario.

Entre ambas versiones el cambio sustancial se encuentra en el bucle de cuantificación. El predictor se mantiene con diferencias menores frente a la versión previa, pero el residuo generado pasa a cuantificarse con un cuantificador uniforme. Esto supone que el residuo pasa a contener un error y al carecer en el descompresor de la muestra original, tampoco podemos hacer uso de ella durante el proceso de predicción en la compresión. Por este motivo, se introduce un descompresor local (*sample representative*), que regenera el valor original partiendo del residuo cuantificado y de las estadísticas mantenidas por el propio predictor. Los valores regenerados son los que se utilizan para realimentar el sistema, tal como se ve en la Figura 20.

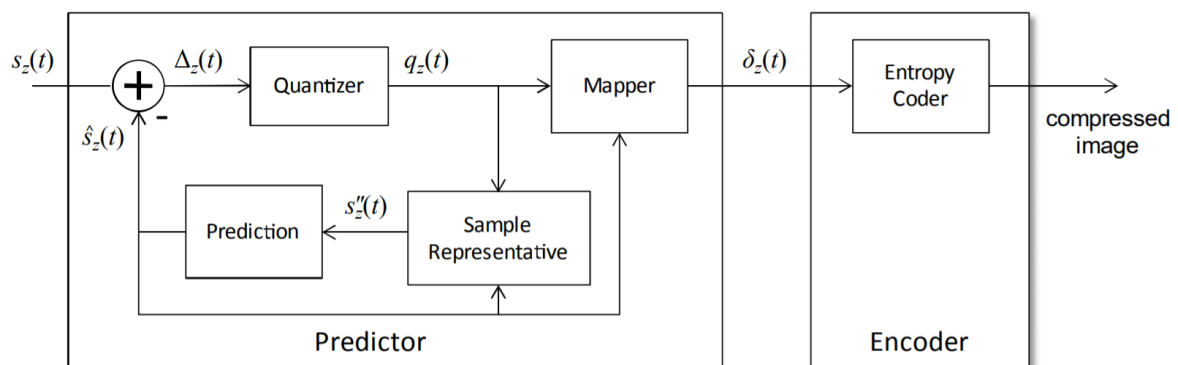


Figura 20: Esquema funcionamiento CCSDS 123.0-B-2 [7].

Dentro de esta visión general se aprecia que el nuevo estándar aporta novedades importantes en cuanto a su potencial, mientras que mantiene la compatibilidad con el algoritmo de compresión definido en el estándar previo. Se sigue permitiendo configurar la compresión para que se realice sin pérdidas, de forma equivalente a la versión anterior; para ello, basta con definir el error introducido a 0.

A continuación, se va a describir, en profundidad, los nuevos bloques introducidos. En primer lugar, trataremos el cuantificador.

El bloque *quantizer* (cuantificador) integra la etapa de pérdidas. En este, la diferencia entre la muestra entrante y la predicha no se calcula de forma exacta, sino que se aproxima con un cuantificador uniforme. Con esta estrategia, el máximo error introducido queda definido por la variable $m_z(t)$. Esta variable puede ser definida de tres formas, según como se introduzca el tipo de error:

- Error absoluto: Donde $m_z(t) = a_z$, siendo a_z un entero en el rango de $0 \leq a_z \leq 2^{D_a} - 1$.
- Error relativo: Donde se especifica el valor r_z que indica el error relativo. Se aplica mediante la ecuación 17. El valor de r_z debe encontrarse en el rango de $0 \leq a_z \leq 2^{D_r} - 1$, siendo D_r un valor especificado pro el usuario (*relative error limit bit depth*):

$$m_z(t) = \left\lfloor \frac{r_z |\hat{s}_z(t)|}{2^D} \right\rfloor \quad (17)$$

- Error combinado: Incluye los dos anteriores y utiliza en cada caso el error mínimo de estos.

$$m_z(t) = \min \left(a_z, \left\lfloor \frac{r_z |\hat{s}_z(t)|}{2^D} \right\rfloor \right) \quad (18)$$

En cuanto al cálculo del índice de cuantificación ($q_z(t)$), se especifica en la ecuación 19. En esta se produce un escalón de cuantificación fijo, de tamaño $2m_z(t) + 1$.

$$q_z(t) = \begin{cases} \Delta_z(0), & t = 0 \\ \text{sgn}(\Delta_z(t)) \left\lfloor \frac{|\Delta_z(t)| + m_z(t)}{2m_z(t) + 1} \right\rfloor, & t > 0 \end{cases}, \quad (19)$$

El otro bloque que se agrega es el *sample representative*. Su labor es obtener una muestra equivalente a la original, pero con el error asociado, ya que el descompresor carecerá de los valores exactos. Es esencia, se encarga de realimentar al predictor con los valores calculados para cada muestra, donde en la versión anterior se realimentaba directamente con la muestra predicha. La definición de su comportamiento se realiza por las tres ecuaciones siguientes:

$$s_z''(t) = \begin{cases} s_z(0), & t = 0 \\ \left\lfloor \frac{\tilde{s}_z''(t) + 1}{2} \right\rfloor, & t > 0 \end{cases} \quad (20)$$

$$\tilde{s}_z''(t) = \left\lfloor \frac{4(2^\Theta - \phi_z) \cdot (s_z'(t) \cdot 2^\Omega - \text{sgn}(q_z(t)) \cdot m_z(t) \cdot \psi_z \cdot 2^{\Omega-\Theta}) + \phi_z \cdot \tilde{s}_z(t) - \phi_z \cdot 2^{\Omega+1}}{2^{\Omega+\Theta+1}} \right\rfloor, \quad (21)$$

$$s_z'(t) = \text{clip}(\hat{s}_z(t) + q_z(t)(2m_z(t) + 1), \{s_{\min}, s_{\max}\}) \quad (22)$$

Se comienza el cálculo con la ecuación 20 (*quantized bin center*). El valor $S'_z(t)$ refleja la suma del residuo reescalado y la muestra predicha. Seguimos calculando el *double resolution sample representative* (Ecuación 21), que regenera la muestra original con resolución doble. Su cálculo depende de las variables $S'_z(t)$, $q_z(t)$ (residuo cuantificado) y de $\tilde{S}_z(t)$ (muestra predicha de alta resolución). También se introducen los siguientes parámetros de configuración:

- Θ : S.R. *Resolution*: Debe ser un entero en el rango de $0 \leq \Theta \leq 4$.
- ϕ_z : S.R. *Damping*: Debe ser un entero en el rango de $0 \leq \phi_z \leq 2^\Theta - 1$.
- Ψ_z : S.R. *Offset*: Debe ser un entero en el rango de $0 \leq \Psi_z \leq 2^\Theta - 1$.

Finalmente, tras aplicar la función se obtiene el valor final de la *sample representative* $S''_z(t)$, que tiene la misma resolución que la muestra original (Ecuación 22).

Como se comentó previamente, en esta versión del estándar se realizan algunos cambios dentro de otros bloques del compresor. En el predictor, pasan a definirse tres etapas: *high-resolution predicted sample*, *double-resolution predicted sample* y *predicted sample*. Estas operan de forma equivalente, pero la muestra predicha escalada ($\tilde{S}_z(t)$) se ha subdividido en dos bloques (ecuaciones 23 y 24):

$$\tilde{s}_z(t) = \text{clip} \left(\text{mod}_R^* \left[\hat{d}_z(t) + 2^\Omega (\sigma_z(t) - 4s_{\text{mid}}) \right] + 2^{\Omega+2} s_{\text{mid}} + 2^{\Omega+1}, \left\{ 2^{\Omega+2} s_{\text{min}}, 2^{\Omega+2} s_{\text{max}} + 2^{\Omega+1} \right\} \right), \quad (23)$$

$$\tilde{s}_z(t) = \begin{cases} \left\lfloor \frac{\tilde{s}_z(t)}{2^{\Omega+1}} \right\rfloor, & t > 0 \\ 2s_{z-1}(t), & t = 0, P > 0, z > 0 \\ 2s_{\text{mid}}, & t = 0 \text{ and } (P = 0 \text{ or } z = 0) \end{cases}. \quad (24)$$

A su vez, se han realizado cambios dentro del bloque *mapper*. En este se ha redefinido la ecuación $\theta_z(t)$ con el objetivo de adaptarse al error de cuantificación $m_z(t)$. La nueva definición es la que se encuentra en la siguiente ecuación:

$$\theta_z(t) = \begin{cases} \min \{ \hat{s}_z(0) - s_{\text{min}}, s_{\text{max}} - \hat{s}_z(0) \}, & t = 0 \\ \min \left\{ \left\lfloor \frac{\hat{s}_z(t) - s_{\text{min}} + m_z(t)}{2m_z(t) + 1} \right\rfloor, \left\lfloor \frac{s_{\text{max}} - \hat{s}_z(t) + m_z(t)}{2m_z(t) + 1} \right\rfloor \right\}, & t > 0 \end{cases}. \quad (25)$$

Por último, se han añadido nuevos métodos para el cálculo de la *Local Sum*. Se mantienen las descripciones *neighbour oriented* y *column oriented*, pero estos agregan la posibilidad de ser *wide* o *narrow*. Las ecuaciones que describen estos nuevos algoritmos son:

Wide Neighbor-oriented

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y,x-1} + s''_{z,y-1,x-1} + s''_{z,y-1,x} + s''_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 4s''_{z,y,x-1}, & y = 0, x > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}), & y > 0, x = 0 \\ s''_{z,y,x-1} + s''_{z,y-1,x-1} + 2s''_{z,y-1,x}, & y > 0, x = N_X - 1 \end{cases} ; \quad (26)$$

Narrow Neighbor-oriented

$$\sigma_{z,y,x} = \begin{cases} s''_{z,y-1,x-1} + 2s''_{z,y-1,x} + s''_{z,y-1,x+1}, & y > 0, 0 < x < N_X - 1 \\ 4s''_{z-1,y,x-1}, & y = 0, x > 0, z > 0 \\ 2(s''_{z,y-1,x} + s''_{z,y-1,x+1}), & y > 0, x = 0 \\ 2(s''_{z,y-1,x-1} + s''_{z,y-1,x}), & y > 0, x = N_X - 1 \\ 4s_{mid}, & y = 0, x > 0, z = 0 \end{cases} ; \quad (27)$$

Wide Column-oriented

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x}, & y > 0 \\ 4s''_{z,y,x-1}, & y = 0, x > 0 \end{cases} ; \quad (28)$$

Narrow Column-oriented

$$\sigma_{z,y,x} = \begin{cases} 4s''_{z,y-1,x}, & y > 0 \\ 4s''_{z-1,y,x-1}, & y = 0, x > 0, z > 0, \\ 4s_{mid}, & y = 0, x > 0, z = 0 \end{cases} ; \quad (29)$$

Como último análisis de esta nueva versión, se incluye a continuación la Tabla 2. En esta se exponen todos los parámetros configurables por el usuario dentro de la versión CCSDS 123.0-B-2 y cuáles son sus efectos sobre el funcionamiento del compresor.

Tabla 2: Parámetros configuración CCSDS 1230-B-2

Parámetro	Descripción	Rango Valores
N_x	Tamaño Dimensión X	X
N_y	Tamaño Dimensión Y	Y
N_z	Tamaño Dimensión Z	Z
S_{max}	Valor máximo muestra	$2^{16} - 1$
S_{mid}	Valor medio muestra	$2^{(16-1)}$
S_{min}	Valor mínimo muestra	0
D	Rango dinámico en bits	16
P_z	Número de bandas previas en predicción	$0 \leq P_z \leq 15$
Ω	Resolución pesos	$4 \leq \Omega \leq 19$
R	Tamaño Registro	$\max\{32, D + \Omega + 2\} \leq R \leq 64$
v_{max}	Límite superior del weight update scaling exponent	$v_{min} \leq v_{max} \leq 9$
v_{min}	Límite inferior del weight update scaling exponent	$-6 \leq v_{min} \leq v_{max}$
t_{inc}	Factor que regula la velocidad de adaptación de los pesos a la imagen.	$2^4 \leq t_{inc} \leq 2^{11}$
ω_{max}	Límite superior componentes del vector de pesos	$2^{\Omega+2} - 1$
ω_{min}	Límite inferior componentes del vector de pesos	$-2^{\Omega+2}$
Novedades CCSDS 123.0-B-2		
a_z	Error Absoluto	$0 \leq a_z \leq 2^{D_a} - 1$
r_z	Error Relativo	$0 \leq a_z \leq 2^{D_r} - 1$
D_a	Límite número bits error absoluto	$1 \leq D_a \leq \min\{D - 1, 16\}$
D_r	Límite número bits error relativo	$1 \leq D_r \leq \min\{D - 1, 16\}$
θ	Resolución sample Representative	$0 \leq \theta \leq 4$
ϕ_z	Damping sample representative	$0 \leq \phi_z \leq 2^\theta - 1$
Ψ_z	Offset sample representative	$0 \leq \Psi_z \leq 2^\theta - 1$
ζ_z^*	Offset actualización de pesos	$-6 \leq \zeta_z \leq 5$

3. Flujo de diseño empleado

Para desarrollar sistemas en FPGA, la estrategia tradicional es partir de un algoritmo implementado en C, u otro lenguaje de alto nivel, el cual es usado como referencia y realizar su descripción en System C, Verilog o VHDL. Sin embargo, este flujo de trabajo no ha sido el empleado para el desarrollo de este proyecto.

El flujo de trabajo se ha basado en la *suite* de desarrollo ofrecida por Matlab que, con sus diferentes paquetes y aplicaciones, nos permite exportar los sistemas funcionales sobre Simulink a una FPGA.

Esta estrategia nos ofrece una aproximación más intuitiva para la implementación de algoritmos a nivel de sistemas. Permite centrarse inicialmente en el correcto funcionamiento del algoritmo y su comprensión, sin tener un conocimiento minucioso del *hardware* final. Al comenzar el modelado desde alto nivel (en este caso, desde Matlab), se relegan el establecimiento de las estrategias necesarias para la implementación *hardware* a etapas posteriores. Matlab, además, tiene la ventaja que permite analizar y representar gráficamente cada variable utilizada. Por tanto, empezamos en un nivel alto de abstracción del que posteriormente se bajará progresivamente hasta llegar a la implementación *hardware*.

El paquete utilizado para realizar la síntesis se llama HDL Coder [31]. Desde su interfaz *Workflow Advisor*, nos permite generar los siguientes proyectos para las FPGAs del fabricante Xilinx:

1. Proyecto VHDL completo, con el código generado y la configuración ya realizada para el entorno de Vivado.
2. Bloque IP, listo para incluir dentro de cualquier diseño. Es destacable que permite autogenerar interfaces estandarizadas AXI (Advanced eXtensible Interface): AXI4, AXI4-Lite, AXI4-Stream, etc [32].
3. Bloque IP con un sistema operativo (SO). Con las placas compatibles de la familia ZYNQ permite, desde su entorno, cargar el IP con un SO específico al dispositivo final. Posteriormente, ya se puede alimentar nuestro sistema con las señales que tengamos en nuestro *workspace* de Matlab y recoger la respuesta en tiempo real.

Para los objetivos de este proyecto, se ha seleccionado la segunda opción, la generación de un bloque IP para el entorno de VIVADO. Finalmente, este bloque se ha importado en un proyecto de síntesis, donde se ha conectado como parte integrante de un diseño basado en la estructura ZYNQ. Este último proyecto es el que se ha sintetizado y con el que se ha programado nuestro kit de desarrollo para su test.

El Grupo en el cual se realiza este TFG (la división DSI del IUMA) no ha tenido experiencias previas con el uso del entorno de Matlab-Simulink, por lo que se consideró muy interesante tener tanto la versión B1 como la B2 en este entorno para futuras aplicaciones y ampliaciones. De esta forma, primero se realizó una implementación del estándar CCSDS 123.0-B-1, del cual ya se tiene disponible una descripción en alto nivel (lenguaje C) y otra en VHDL (realizado mediante financiación de la ESA del proyecto SHyLoC y actualmente disponible en el *IP-Portfolio* de la ESA).

Y en segundo lugar, se realizó una implementación que extiende las funcionalidades para adaptarse al estándar CCSDS 123.0-B-2. Por tanto, tras este flujo de trabajo obtenemos dos bloques IP funcionales basados en los estándares CCSDS 123.0-B-1 y CCSDS 123.0-B-2, respectivamente.

En este capítulo se dará una introducción a las herramientas utilizadas durante el flujo de diseño y cuáles son sus particularidades para nuestro caso de uso. Se verán en detalle los procedimientos utilizados y las estrategias aprendidas durante el proceso de desarrollo.

3.1. Matlab

MATLAB (*MATRIX LABORATORY*) es una herramienta de cómputo numérico que ofrece un entorno propio de diseño con un lenguaje de programación propio, denominado lenguaje *m*. El entorno de desarrollo está centrado en la manipulación de matrices y diseño de algoritmos matemáticos. Con el paso de los años, se ha visto ampliado sucesivamente con paquetes que permiten desarrollar algoritmos para distintas funciones y plataformas, lo que la hace ideal para comenzar a desarrollar cualquier flujo de trabajo [33].

Sus principales ventajas, de cara a nuestros objetivos, son:

1. Despreocupación de los requisitos para la implementación final. Por defecto, todas las variables son de tipo *double* con coma flotante, haciendo que no tengamos que preocuparnos de qué tipo de valor estamos almacenando.
2. Visualización rápida. Podemos representar gráficamente cualquier variable que usemos y guardarla para futuras depuraciones.
3. Sencillez del entorno. Se plantea como un entorno intuitivo y fácil de usar, se ha simplificado todo para que pueda ser usado por diseñadores de distintas disciplinas.

Se ha partido de las descripciones del estándar CCSDS 123.0-B-1 y CCSDS 123.0-B-2. Los documentos reflejan la operatoria y algorítmica necesarias para realizar la etapa de compresión, pero al realizar la implementación hay que tener otras consideraciones.

Se comenzó implementando la versión *lossless* del estándar, del cual ya existía una versión completa sobre C. Esta versión en C se ha tomado como referencia (*Golden Reference*) para verificar todo el desarrollo en Matlab, empezando por la verificación de los distintos módulos hasta verificar el sistema completo. Para ello, se ha comparado la salida del predictor (es decir, el residuo mapeado) de la descripción en Matlab y la descripción en C tomada como *Golden Reference*. Como prueba de interoperabilidad, se introdujo el residuo generado en Matlab en el codificador entrópico disponible en la versión en C del compresor para posteriormente descomprimirse; la imagen debe ser exactamente igual a la original si el proceso ha sido correcto.

Al trabajar con dos lenguajes distintos, resulta esencial comprobar la perfecta equivalencia de las operaciones realizadas, es decir, que muestran los mismos resultados exactos ante la misma entrada. Esto es esencial para MATLAB, ya que la estrategia al redondear o la indexación de matrices es completamente distinta a C.

Cuando finalmente se obtuvo una equivalencia del 100% entre los resultados del predictor en Matlab y C, se continuó añadiendo las novedades de la versión con pérdidas. En este caso no se dispuso de código C como referencia, pero se trabajó con una versión ejecutable del predictor *lossy*, que se usó para comprobar los resultados generados.

El proceso seguido fue equivalente al sin pérdidas, pero partiendo de este y añadiendo los nuevos bloques *Sample Representative* y *Quantizer*, además de modificaciones en otros módulos. Tras comprobar de nuevo la completa equivalencia de los resultados con la referencia, se dio el salto a su implementación Simulink.

3.2. Simulink

Simulink es una herramienta que se encuentra en el entorno de la suite Matlab, pero con un enfoque radicalmente distinto. Su objetivo es el diseño de sistemas físicos mediante bloques configurables y, por tanto, ofrece una forma completamente distinta de trabajar [34]. Dentro de este entorno pasan a ser esenciales los tipos de datos, temporizaciones, acceso a memoria, etc.

Antes de comenzar a implementar todo el algoritmo, descrito previamente en Matlab, tenemos que analizar cómo enviar los datos a nuestro sistema y leerlos posteriormente. Para nuestro caso (Figura 21), nos encontramos con un flujo de datos entrante a partir de una imagen, por lo que haremos uso del bloque “*Triggered Signal From Workspace*”, que enviará una muestra por pulso de la imagen cargada que se encuentra en el *workspace* de Matlab. El residuo de salida es almacenado en el vector *residual_out*.

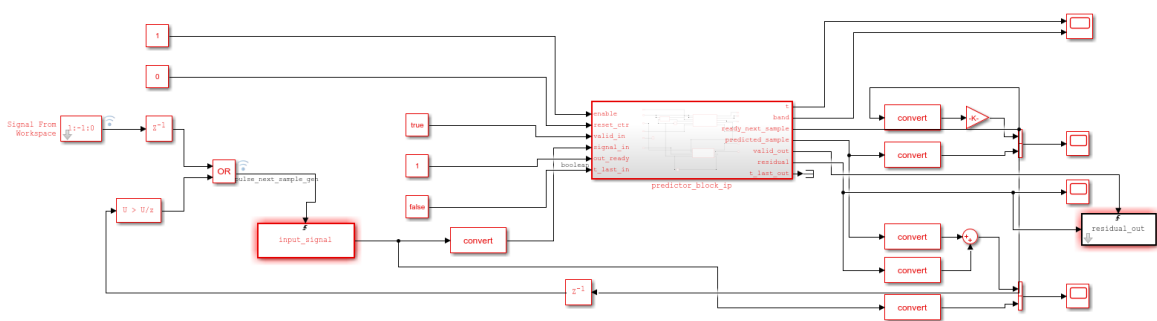


Figura 21: Entorno de pruebas Simulink del bloque IP.

Con la interfaz de entrada y salida preparada, podemos comenzar la implementación de nuestro algoritmo. Para ello, lo ideal es que trabajemos todo dentro de un *subsystem*, dejando la interfaz exterior sin alterar.

Para pasar el código desde Matlab, podemos utilizar dos estrategias:

1. Utilizar los bloques de tipo *Matlab Function* [35], que te permiten copiar código *m* y generan las interfaces de entrada y salida.
2. Hacer las funciones con los bloques específicos de Simulink.

Ambas alternativas son válidas, y pueden ser igual de eficientes. Cuando trabajamos con algoritmos complejos que tengan múltiples condiciones o bucles tiende a ser más cómodo el realizarlo mediante código Matlab, pero hay que tener en cuenta que no todo es implementable de la misma forma que en la etapa anterior de diseño. Dentro de Simulink, como implementamos un sistema físico, cobra una especial importancia el analizar cómo tratamos los tipos de datos.

Si decidimos realizar la función mediante bloques, la principal ventaja es que vemos exactamente de forma gráfica el diseño creado y cualquier configuración que se realice será generable.

El sistema que estamos diseñando se encuentra enfocado en su síntesis como HDL y, por tanto, debemos tener unas consideraciones añadidas en nuestro flujo de trabajo, además de las habituales de Simulink [36]. Las más destacables son las siguientes:

- No demos hacer uso de variables *double* o *single*, ya que no son compatibles (en principio) con la generación de VHDL. Esto puede suponer un desafío, porque son los formatos por defecto de MATLAB, lo que significa que deberemos especificar en cada punto el tipo de dato de salida que deseamos.
- No podemos hacer uso de bucles de fin variable; el número de ciclos de este debe encontrarse definido al inicio de la ejecución.
- No todas las funciones matemáticas permiten la generación en HDL, hay que revisar cada caso. Por ejemplo, la función *pow* (x^y) no permite que el exponente sea una variable, tiene que ser constante.
- No podemos hacer completo uso de las funciones de gestión de la memoria RAM (*Random Access Memory*) desde el código Matlab. Solo se puede llamar a la memoria una vez y nunca dentro de una condición o bucle.

Si realizamos todo correctamente obtendremos nuestro bloque, pero en este momento pasa a ser esencial un proceso de depuración. Hay que comprobar que su funcionamiento es completamente equivalente al ya testado en Matlab y, a su vez, tratar de optimizar el uso de recursos.

Cuando se ha terminado el proceso de depuración, deberemos hacer uso de la interfaz *Workflow Advisor*, del paquete HDL Coder. Esta nos guiará por los pasos necesarios para generar la síntesis de nuestro sistema a VHDL. A continuación la analizaremos en detalle.

3.3. HDL Coder

El proceso de exportación se realiza dentro de la interfaz *Workflow Advisor*, parte del paquete HDL Coder (Figura 22). Esta herramienta nos guía por el proceso de síntesis del diseño en Simulink hacia *Hardware Description Language*. La interfaz es la siguiente:

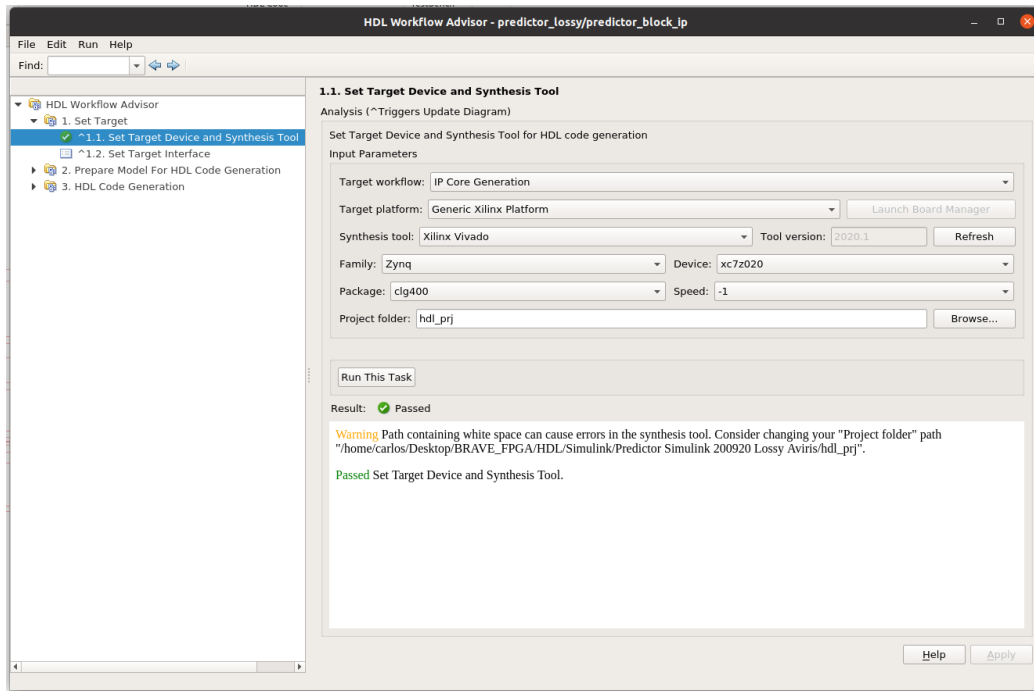


Figura 22: interfaz Workflow Advisor.

Al comenzar el proceso, deberemos especificar cuál es nuestro objetivo y dispositivo para el que queremos generar el diseño. En nuestro caso, el objetivo es un bloque IP para la plataforma Xilinx y el dispositivo es el SoC (System On Chip) XC7Z020 de la familia Zynq [37]. Ejecutamos esta pantalla y continuamos con la definición de la interfaz (Figura 23).

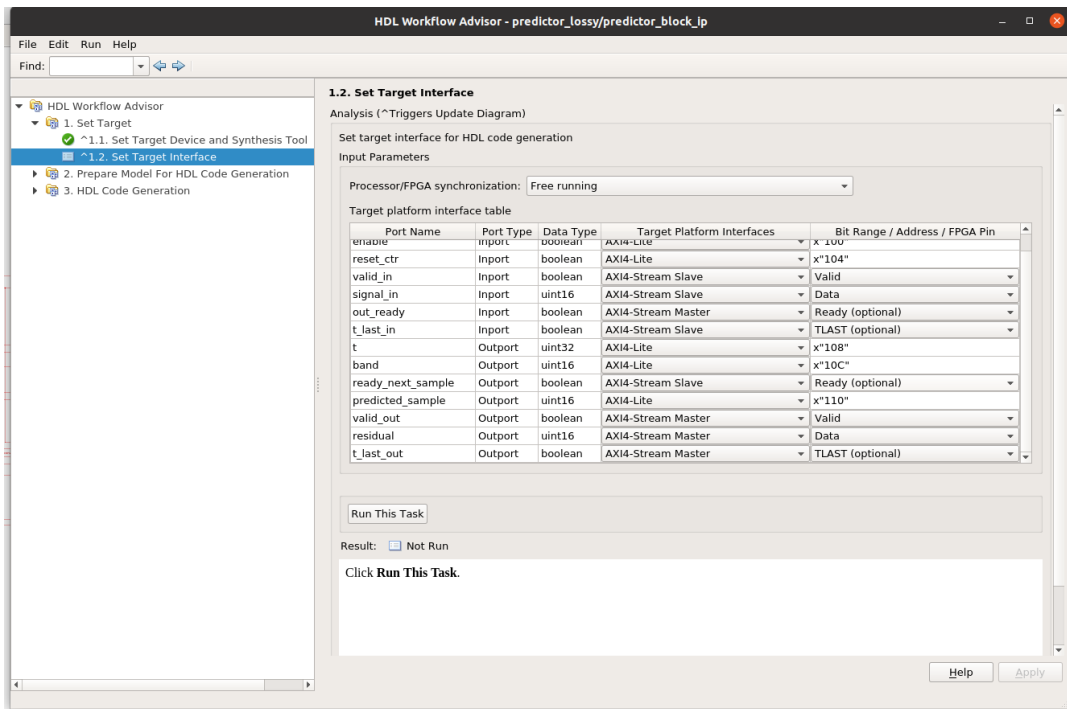


Figura 23: configuración interfaces E/S Workflow Advisor.

Tendremos una tabla con las entradas y salidas del bloque que hemos diseñado y deberemos especificar qué tipo de interfaz queremos asignarle, entre las siguientes:

- AXI4 Lite
- AXI4 Stream Master
- AXI4 Stream Slave
- AXI4-Stream Video Master
- AXI4-Stream Video Slave
- AXI4 Master Read
- AXI4 Master Write

Estas interfaces están preparadas para la comunicación con otros bloques utilizados dentro de los sistemas Xilinx. Por ello, es importante conocer cuál es el dispositivo en el que vamos a implementar el diseño. La configuración estándar es la conexión AXI4 Lite, que dentro de la gama Zynq, permite comunicarse directamente con el microprocesador ARM del sistema para labores de inicialización y configuración de los distintos módulos que conforman el sistema.

Para nuestro caso, se ha utilizado la interfaz AXI4 Lite para pasar los parámetros de configuración en tiempo de compilación y las muestras se transmiten por medio de la interfaz AXI4-Stream. Se ha elegido la interfaz AXI4-Stream para el flujo de datos porque nos permite aumentar la velocidad de transmisión y descargar de carga computacional al microprocesador. Esto ocurre ya que el bloque DMA (*Direct Memory Access*), incluido en el sistema, es capaz de gestionar un flujo de datos entrante interactuando directamente con la memoria RAM *off-chip*, sin hacer uso de la CPU (Central Processing Unit) en el proceso.

Tras definir las interfaces, continuaremos comprobando la compatibilidad del sistema para la síntesis. Será necesario que se realicen las siguientes comprobaciones:

1. *Check Global Settings*
2. *Check Algebraic Loops*
3. *Check Block Compatibility*
4. *Check Sample Times*

Si el sistema no presenta ningún error que haya que subsanar, se continúa con la etapa final de síntesis y generación del bloque. En esta podremos, en primer lugar, gestionar las opciones de generación, informes, estilo del código y de optimización. Cuando estén configuradas acorde a nuestras necesidades, avanzaremos a la pantalla *Generate RTL Code and IP Core* (Figura 24) para obtener la síntesis funcional del bloque.

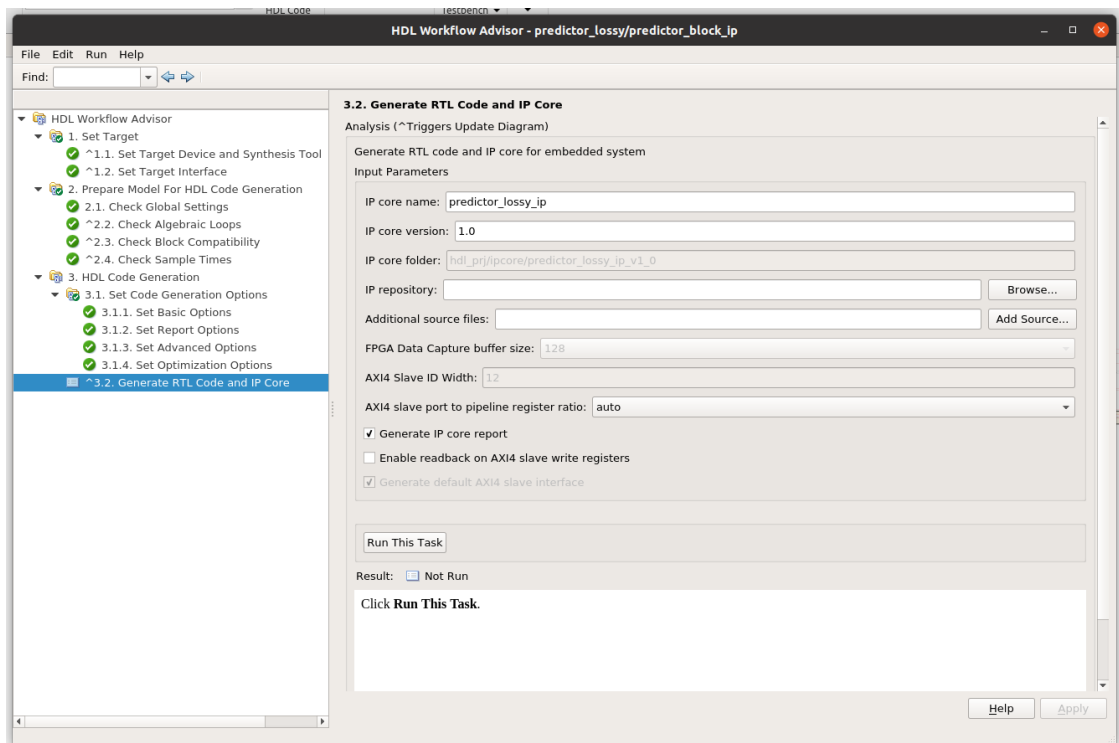


Figura 24: Pantalla final Workflow Advisor.

Al terminar el proceso de síntesis, se generará automáticamente un informe del proceso y el HDL producido. El bloque generado se encontrará dentro de la carpeta del proyecto Simulink, por defecto en la ubicación `/hdl_prj/ip_core`. Para cambiar de ubicación el bloque generado deberemos mover la carpeta, en dicha ubicación, con el nombre igual al asignado el bloque IP.

El proceso se continúa dentro de la interfaz de Vivado. Primero tenemos que generar o abrir un proyecto, obteniendo la interfaz de usuario que vemos en la Figura 25.

Ahora deberemos incluir el bloque IP que hemos generado dentro del repositorio de bloques IP. Para ello haremos *click* en *Settings* y posteriormente en la opción *Repository*, dentro de la sección IP, tal como se ve en la Figura 26. A continuación, pulsaremos el botón “+” y en el formulario buscaremos la ubicación en la que guardamos el bloque IP.

Finalmente, ya el bloque IP se encontrará dentro de la lista de todos los existentes. Para agregarlos solo será necesario pulsar el botón + dentro de nuestro diagrama de bloques, y buscarlos por el nombre que le dimos en *Workflow Advisor*. Con esto ya deberíamos tenerlo listo para integrarlo en nuestro diseño *hardware*.

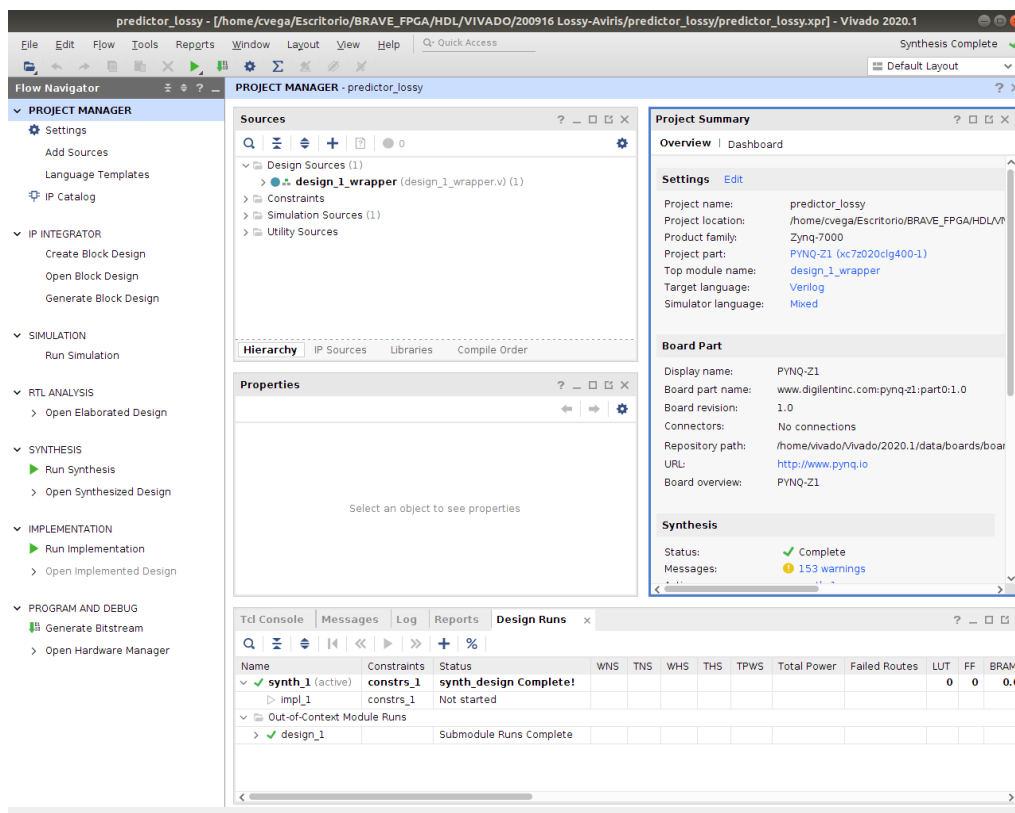


Figura 25: Captura entorno Vivado.

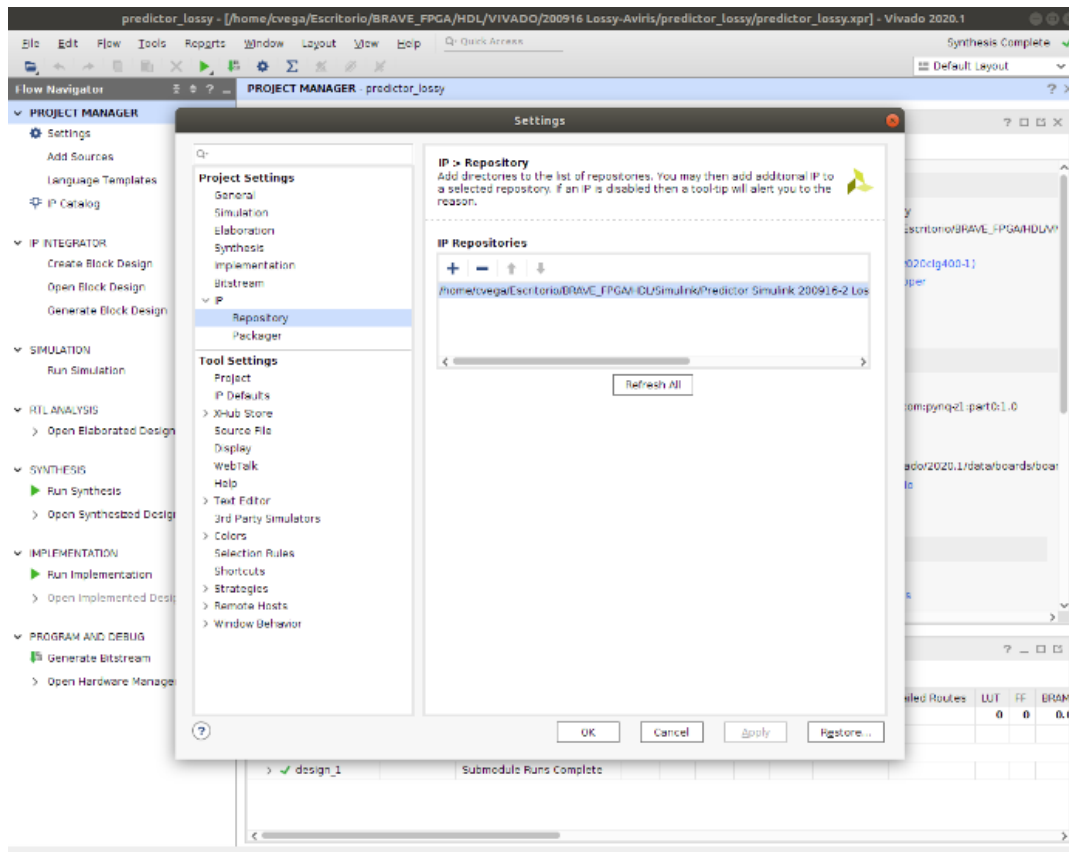


Figura 26: Captura settings Vivado.

3.4. Vivado

Xilinx Vivado Suite es un entorno de herramientas de Xilinx para el desarrollo de sistemas embebidos sobre sus FPGAs (Figura 27). El entorno nos permite partir de un modelo descrito en un lenguaje de alto nivel y realizar los diseños basándonos en distintas metodologías, por ejemplo, RTL (*Register-Transfer Level*) o TLM (*Transaction Level Model*). El propósito de esta herramienta es reducir los tiempos de diseño y, por tanto, agilizar la implementación de sistemas con uso de *hardware* programable. Proporciona un entorno en el que se pueden optimizar todos los parámetros claves para adaptar el resultado a las necesidades específicas del producto y también gestionar las interfaces E/S del *hardware*.

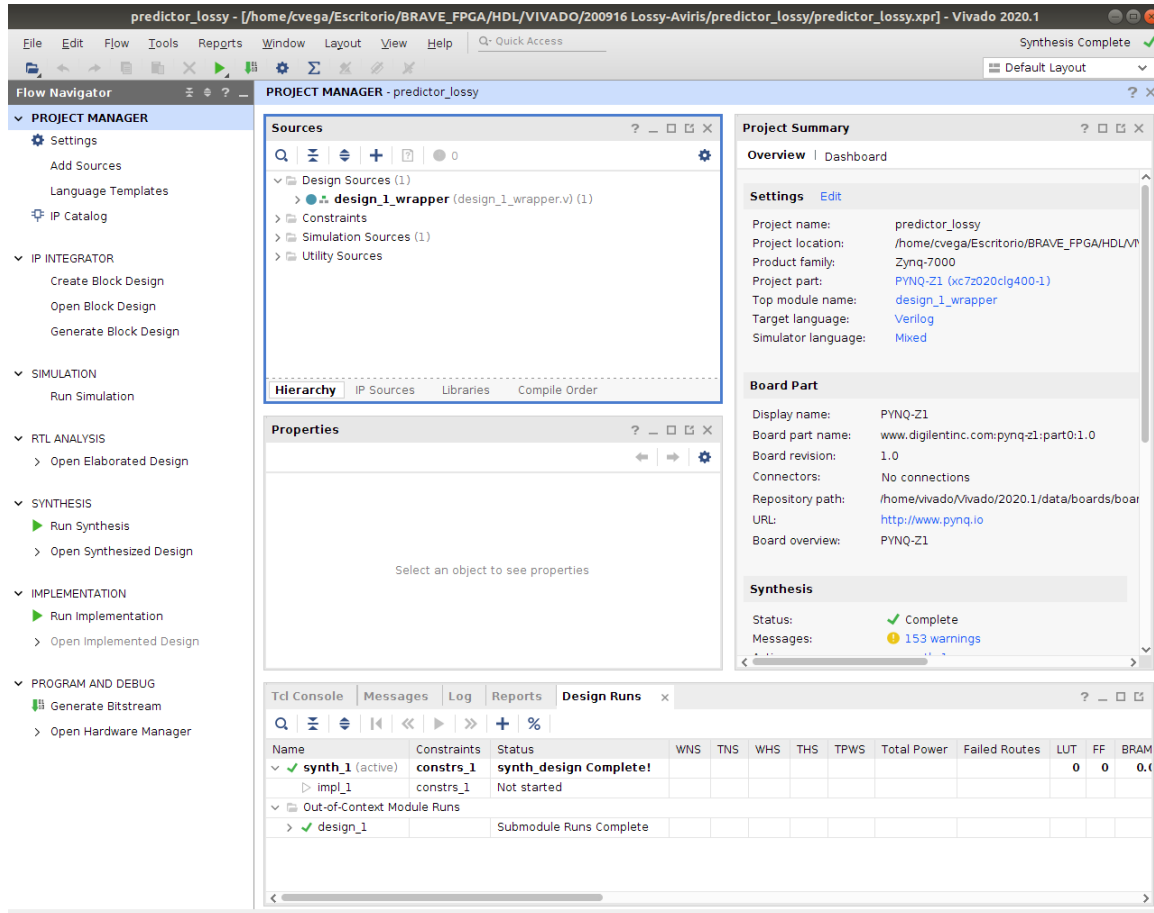


Figura 27: Captura entorno Vivado.

En nuestro caso, el flujo de trabajo utilizado ha comenzado dentro de la etapa de diseño de bloques. Esta es una plataforma alternativa que ofrece Xilinx y que permite la simplificación del diseño en bloques IP, que pueden ser del catálogo de Xilinx o generados por el usuario, que se interconectan mediante arquitecturas de comunicación estandarizadas.

Para nuestro flujo, debemos comenzar añadiendo al repositorio de Xilinx la ubicación de nuestro bloque IP generado previamente en Simulink. Al agregarla, ya nos aparecerá dentro del entorno de *IP design*. A continuación, se deberá de conectar mediante las interfaces ya definidas con el sistema en el que se desee usar. En la Figura 28, vemos la estructura utilizada en nuestro caso para la realización de las pruebas.

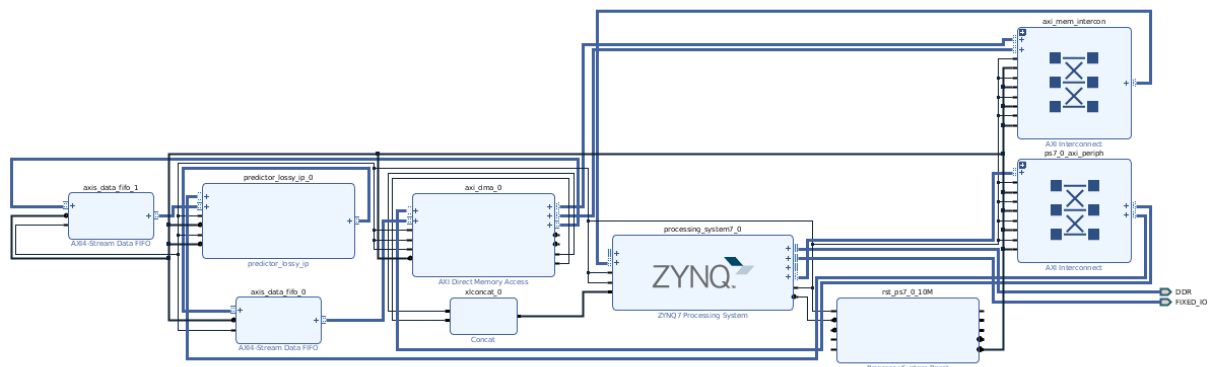


Figura 28: Diagrama de bloques Vivado.

Cuando el sistema se encuentre listo y validado pasaremos a las etapas de síntesis lógica e implementación.

La síntesis lógica se encarga de transformar la descripción en HDL del sistema a un nivel lógico, lo que supone que pasamos a tener una descripción a nivel de puertas, que para el caso de Xilinx hace uso de las primitivas básicas disponibles. También se genera en esta fase un análisis temporal, con el objetivo de comprobar el correcto cumplimiento de las restricciones temporales definidas.

En la etapa de implementación se asignan y distribuyen los diseños, generados en la síntesis, entre el *hardware* de la FPGA. La etapa se compone a su vez por las fases de mapeado, asignación y *routing*. Al finalizar el proceso, se realiza un nuevo análisis temporal para comprobar que las restricciones se siguen cumpliendo. Además, a partir de ahora podemos ver el esquema de cómo nuestro diseño se implementa sobre el *hardware* seleccionado (Figura 29).

A partir de este momento ya está todo listo para cargarlo en la FPGA, solo falta la generación del *bitstream*. Es la última etapa y la plataforma nos permite exportar la definición del *hardware* para que, en caso de que usemos un microcontrolador, se pueda continuar programando desde la herramienta Vitis. Para las pruebas haremos uso de esta opción, ya que haremos uso del ARM integrado en la serie ZYNQ.

Vitis es el entorno de programación de Vivado Suite para microcontroladores de la empresa. Está basado en el entorno de Eclipse y permite programar en C y C++ [38].

Dentro de este SDK podemos crear una aplicación a partir de la definición del *hardware* que hemos exportado en Vivado. Esto creará una solución, como la que vemos en la Figura 30, desde la que se comenzará el desarrollo del *software* dentro de la carpeta *src*.

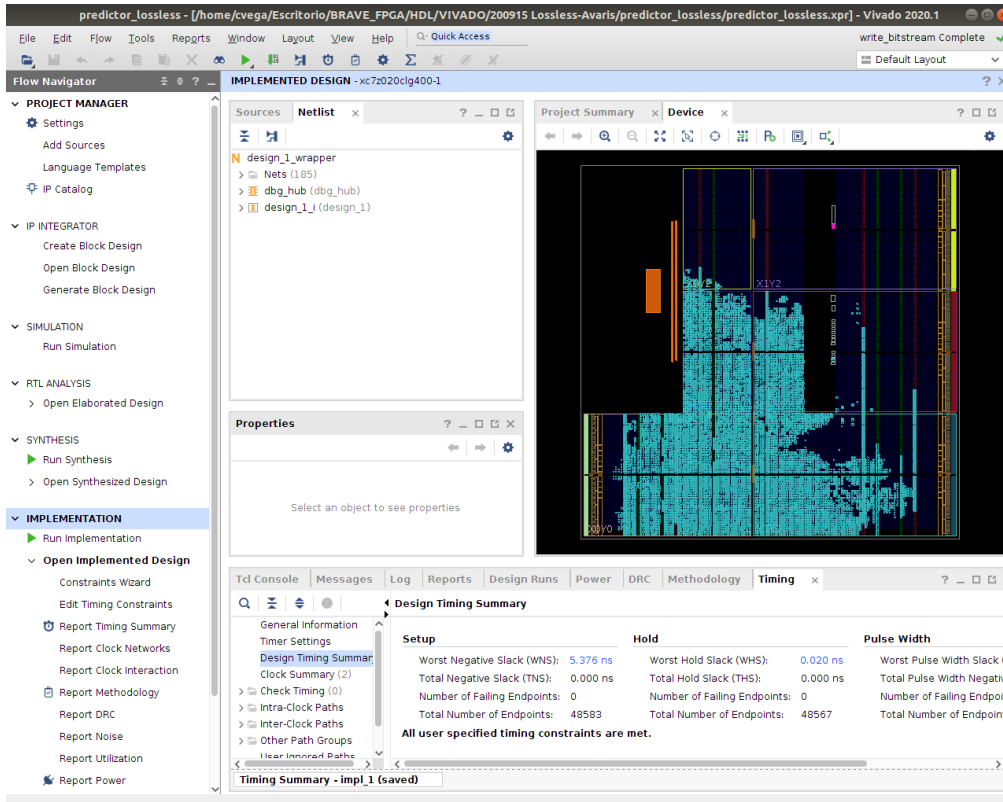


Figura 29: Ejemplo vista implementación Vivado.

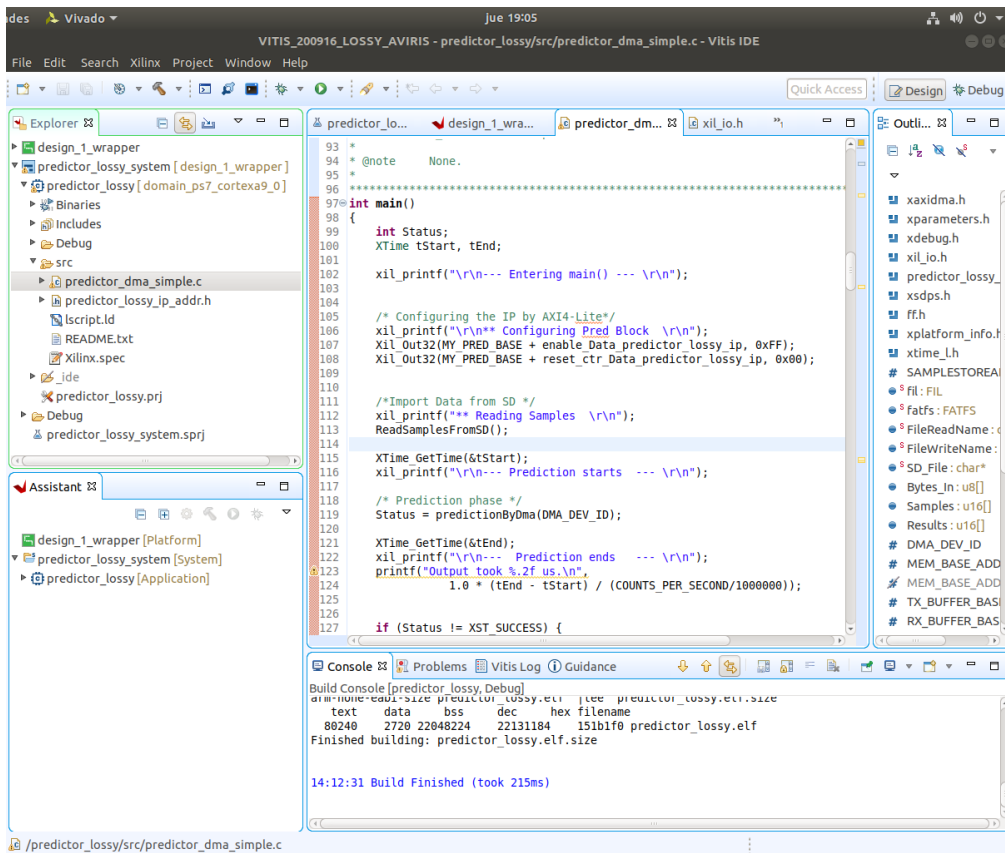


Figura 30: Ejemplo entorno Vitis.

Para depurar o ejecutar el *software* en el *hardware*, haremos *click* en la función *debug* o ejecución y, automáticamente, se cargará en la FPGA el diseño generado en Vivado. Al terminar el proceso de carga, el sistema comenzará la ejecución del programa.

Si hay que modificar el IP deberemos regenerar todo el proceso desde Simulink, en el que introduciremos los cambios. Luego repetiremos el proceso de síntesis y exportación que acabamos de comentar. También podremos hacer uso de las herramientas de depuración de Vivado (Figura 31), las cuales nos permiten ver las comunicaciones por los buses. Para ello, deberemos introducir un ILA (*Integrated Logic Analyzer*) [38] en el diseño de bloques y conectarlo con las líneas que se deseen depurar.

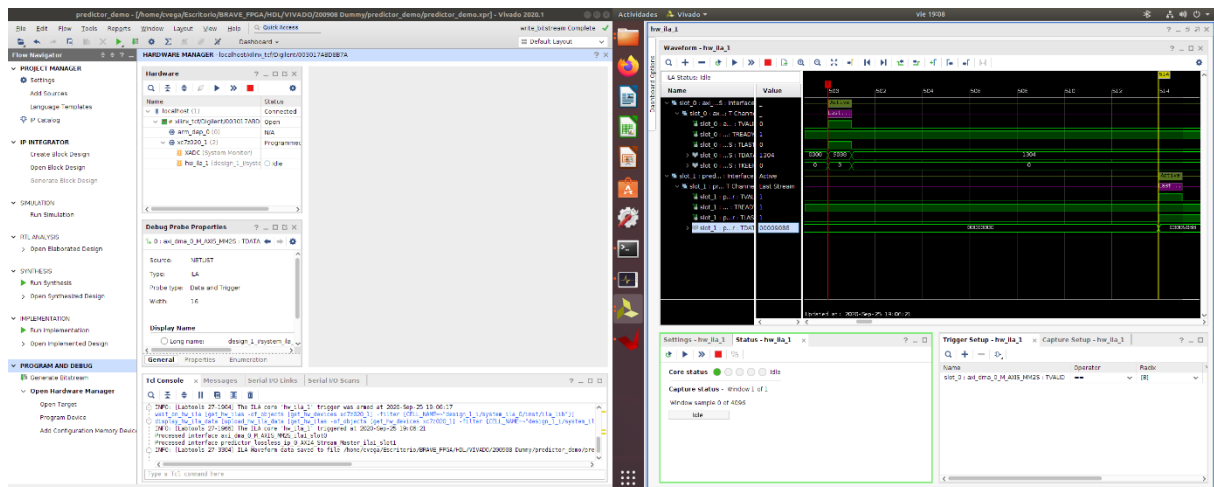


Figura 31: Captura de las herramientas de depuración de Vivado

4. Diseño del predictor en Matlab-Simulink

4.1. Introducción

En este capítulo se describirán las implementaciones del estándar realizadas en el entorno de Matlab y Simulink. Estas se han realizado por separado debido a las características de cada entorno, aunque se han reutilizado fragmentos de código descritos en Matlab sobre Simulink.

En cada entorno, a su vez, se han realizado dos implementaciones independientes. Primero, se ha diseñado según la descripción del estándar CCSDS 123.0-B-1, que es la versión sin pérdidas (*lossless*). Posteriormente, se ha dado el salto CCSDS 123.0-B-2, implementando sobre la anterior versión las principales novedades del nuevo estándar.

4.2. Modelado en Matlab

El modelado dentro de Matlab se basa en un bucle principal, que recorre todas las bandas de nuestra imagen y dentro otro segundo bucle que recorre las muestras de esta en el dominio espacial. Dentro de este último bucle se llama a distintas funciones que realizan las etapas del predictor ya descritas.

Si dividimos ambas versiones (*lossless* y *lossy*) en tres etapas, tenemos una primera etapa de preparación antes de la predicción. En esta calculamos la posición x e y (las coordenadas espaciales), a partir del índice t , y actualizamos el vector de diferencias (vector U) para la posición actual, partiendo de los valores previos.

A continuación, se encuentra la etapa principal, la de predicción. En esta se calcula la *local sum* y se realizan los cálculos de predicción de la muestra para la posición actual. Los cálculos de predicción se subdividen en:

- *Prediction_high_res*: donde se calcula la muestra actual partiendo de la *local sum*, el vector de pesos y el vector diferencias. Produce el valor de alta resolución, 64 bits por muestra.
- *Double_res*: se reduce la resolución de la muestra anterior a 32 bits y se añaden excepciones para cuando nos encontramos en la primera posición de cada banda.
- *Predicted_sample*: reduce a la resolución final de 16 bits por muestra, que es igual a la resolución por píxel de la imagen de entrada.

Cuando se finaliza este cálculo, se carga la muestra real y pasamos a los cálculos post-predicción. En esta etapa se realizan las siguientes funciones:

- Calculamos el residuo, que es la diferencia entre la muestra predicha y la real.
- Mapeamos el residuo como un valor *unsigned*.
- Calculamos los errores de predicción.
- Actualizamos el vector de pesos.

Además, cuando estamos dentro de la versión *lossy* debemos tener en cuenta que hay que realizar las funciones de cuantificación y reconstrucción de la muestra.

Todas las tareas que se realizan dentro de la predicción no se han programado directamente en el bucle, sino que se han subdividido en múltiples funciones para facilitar la verificación del sistema. Solo para la versión *lossless* del predictor trabajamos con las siguientes funciones añadidas, que corresponden con los nombres dados a las distintas partes descritas en el estándar.

- *Local_sum*
- *Prediction_high_res*
- *Double_res*
- *Predicted_sample*
- *Prediction_res*
- *Mapper*
- *Central_differnce*
- *Init_weight*
- *update_vector_weight*
- *Update_vector_difference*

A continuación, vemos en la Figura 32 el esquema de funcionamiento de la versión *lossless*. Tal como se comentó anteriormente, tenemos dos bucles encadenados que recorren todas las muestras de la imagen entrante. En el diagrama hay dos tipos de líneas, azules y verdes. Las líneas verdes representan el flujo de los datos dentro del programa, mientras que las azules indican el hilo de ejecución que se realiza.

Al analizar los flujos de datos se aprecia que la función de predicción se alimenta de la *local sum* y de los vectores U (vector de diferencias) y W (Vector de pesos). A su vez, el valor de la nueva muestra no se incluye en el flujo hasta la etapa post-predicción, junto con la muestra predicha en la función de cálculo del residuo. La salida del sistema es el valor generado por el *mapper*, que se guarda en el vector *residualMappedTot*.

El diagrama de la versión con pérdidas se encuentra en la Figura 33. Su diferencia frente a la *lossless* radica en la etapa post-predicción, donde se añaden los bloques *quantizer* y *sample representative*.

Estas dos funciones se introducen entre el cálculo del residuo y el cálculo de los errores de predicción. Otro cambio que se realiza es la entrada del *mapper*, que pasa a convertir los datos generados por el cuantificador. En esta versión, los errores de predicción pasan a calcularse con la salida del *sample representative*, siendo dependiente de este la realimentación del compresor.

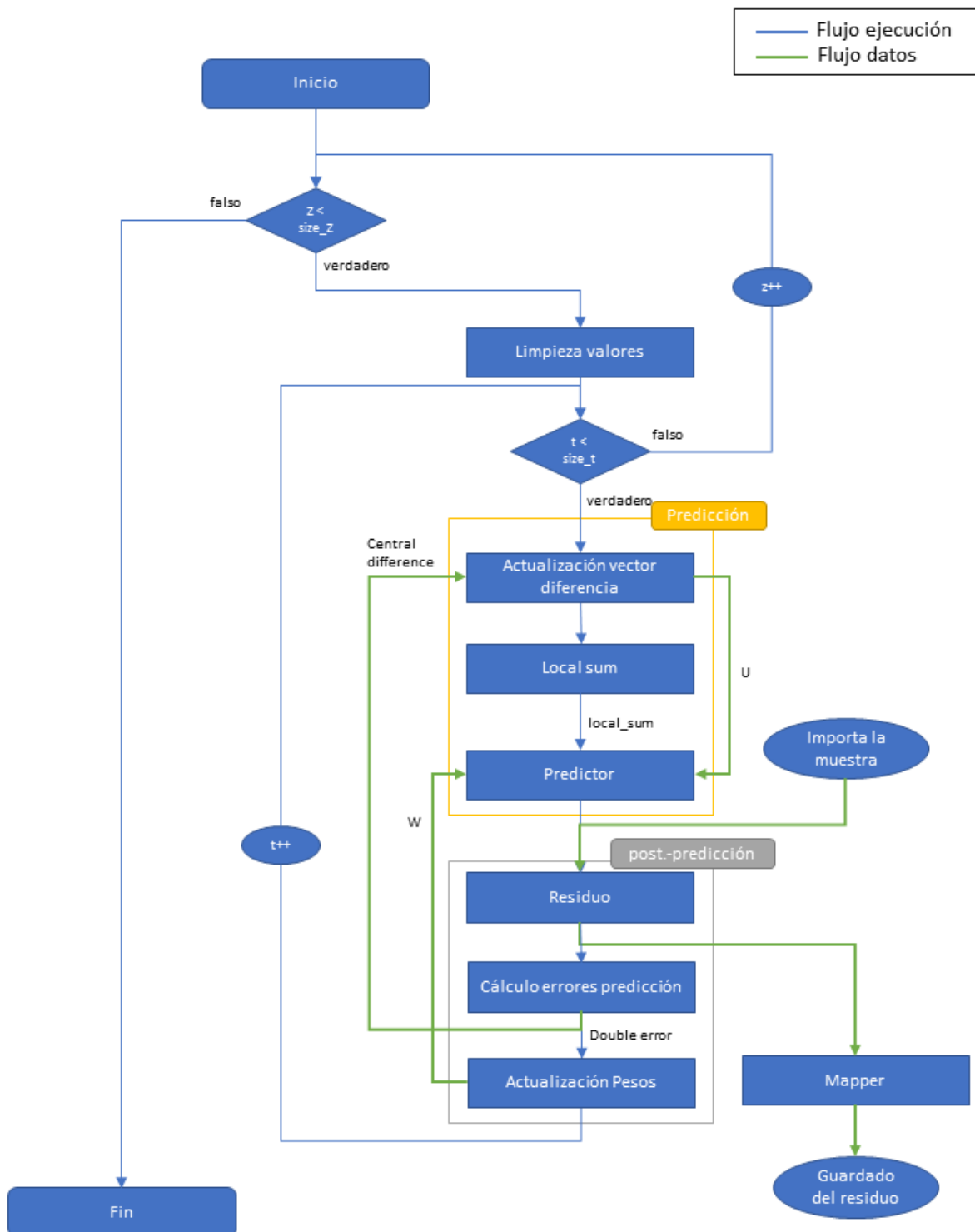


Figura 32: Esquema funcionamiento implementación Matlab CCSD123.0-B-1.

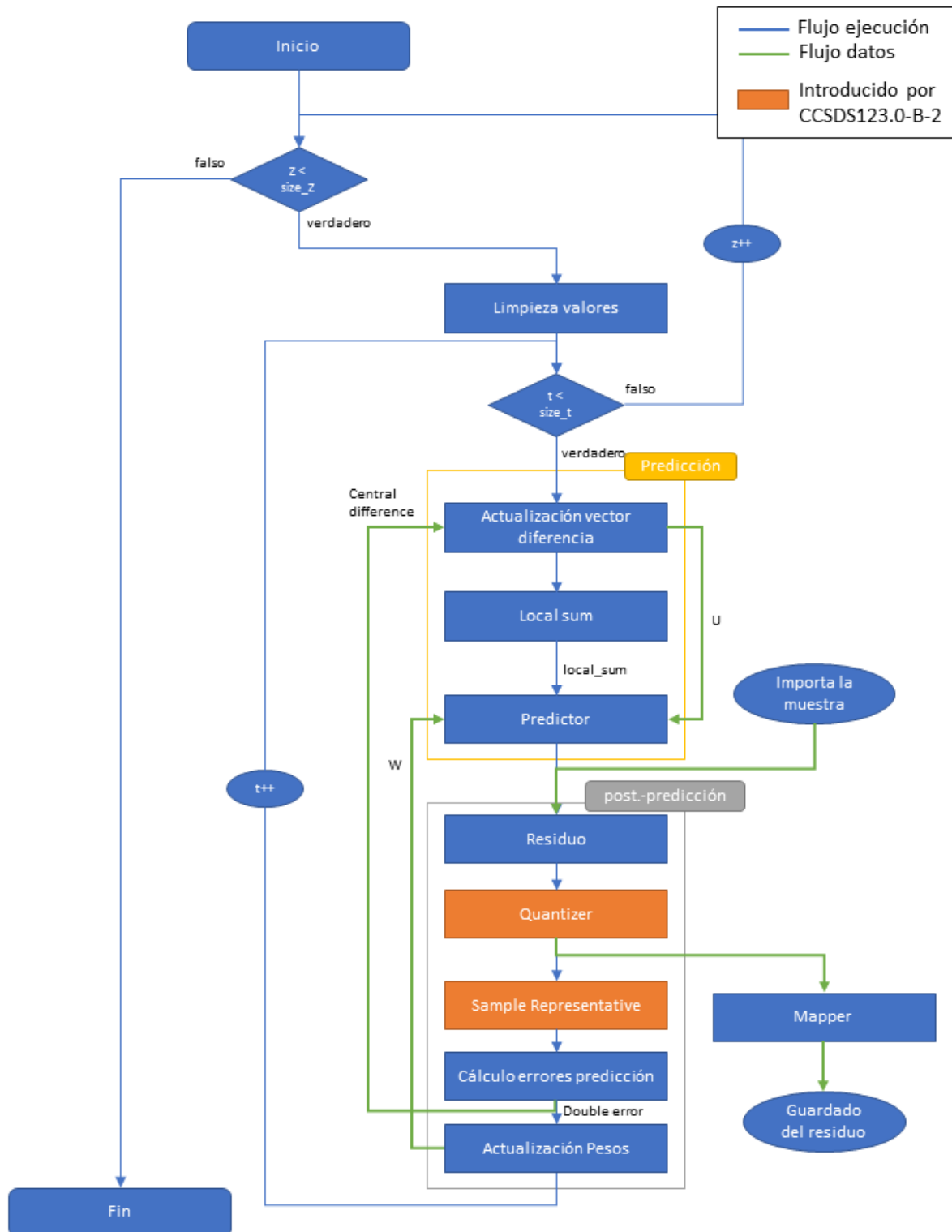


Figura 33: Esquema funcionamiento implementación Matlab CCSD123.0-B-2.

Uno de los puntos fuertes de la implementación descrita en Matlab es la representación visual, la que ayudará al entendimiento del funcionamiento del algoritmo. En la Figura 34 se muestra el proceso compresión de la banda 4 de la imagen de referencia procedente de AVIRIS (*Airborne Visible/Infrared Imaging Spectrometer*) para el algoritmo basado en el CCSDS 123.0-B-1. A la izquierda se encuentra la imagen original, mientras que a continuación se presenta el resultado de la etapa de predicción. Para la imagen posterior se calcula la diferencia entre los valores predichos y los originales, obteniendo el residuo predicción. Finalmente, esta se mapea para convertir sus valores en *unsigned* y se obtiene la última de las imágenes, cuyos datos son los que pasan a ser codificados.

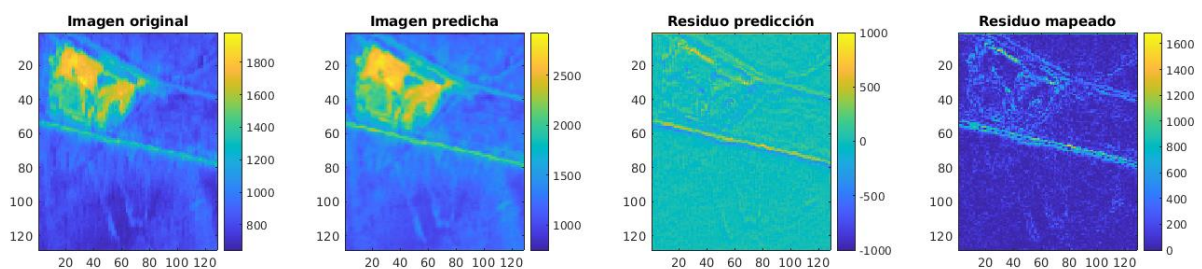


Figura 34: Representación etapas predicción CCSDS 123.0-B-1 para la imagen Aviris (banda 4).

En la Figura 35 se representa el mismo proceso, pero en el predictor *lossy* (basado en el estándar CCSDS 123.0-B-2). Para este caso, vemos unos resultados muy similares a los previos, pero se ha introducido un elevado error de cuantificación para que las diferencias sean visibles. Las tres primeras imágenes son equivalentes a las del compresor sin pérdidas (Figura 34) pero si nos fijamos en el residuo mapeado (última imagen), si se aprecian diferencias.

Vemos que, pese a tener una escala visual de rango más reducido, la cantidad de detalle transmitido es notoriamente inferior al generados en versión *lossless*. Este efecto se produce por la pérdida de información generada en el cuantificador, que para este caso es de 8 bits.

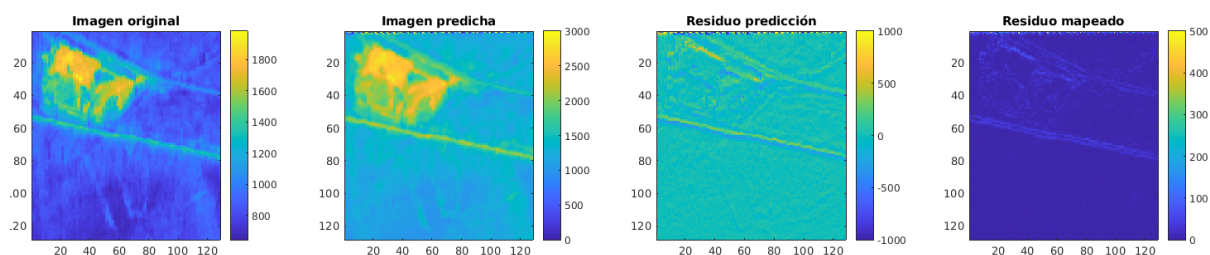


Figura 35: Representación etapas predicción CCSDS 123.0-B-2 para la imagen Aviris (banda 4).

Destacar que las bandas presentadas no se obtienen en tiempo de ejecución, ya que cada punto de la imagen se calcula de forma secuencial. Este sería el resultado de almacenar los flujos de datos de la etapa de predicción y representarlos posteriormente.

Finalmente, de la implementación en Matlab no se va a realizar un mayor análisis sobre cómo se ha realizado la implementación del código a bajo nivel. Sí se realizará a continuación del diseño en Simulink, ya que es el diseño que finalmente se implementará en la FPGA.

4.2.1. Entorno de pruebas

Previamente se ha descrito el funcionamiento del predictor, pero para hacerlo posible es necesario cargar en el entorno la imagen y los parámetros de configuración. Para facilitar esta labor, se desarrolló un *script* que importa la imagen, carga los parámetros en memoria, ejecuta el predictor y finalmente almacena los resultados en un fichero.

Para probar nuestro sistema basta con ejecutar el siguiente *script*:

Código 1: Script ejecución predictor Matlab.

```
%% Inicialización
%Cargamos la imagen
filename = '/home/cvega/Escritorio/BRAVE_FPGA/ShyLoc/Images/aviris_cropped_h128_w128_n22';
nr = 128;
nc = 128;
nb = 224;
endian = 'ieee-be';
format = 'uint16=>uint16';
pixelsOrder = 'BIL';

Img = bi_import_img(filename,nc,nr,nb,1);
Img_lineal = bi_import(filename,nc,nr,nb,1);

%Definimos los parámetros de la imagen
params_aviris;

%% Predicción
predictor_new;

%% Export
% Exportamos a la carpeta del encoder para finalizar la compresión

file_name = '/home/cvega/Escritorio/BRAVE_FPGA/ShyLoc/Images/residualsArtifacts.cmp';
bi_export(file_name, residualMappedTot, nc, nr, nb, 1);
```

Como se aprecia, primero se la carga la imagen en memoria con las funciones:

- *Bi_import_img*, que carga la muestra como un array de 3 dimensiones (Banda, x, y). Esta nos permite visualizar la imagen gráficamente.
- *Bi_import*, que carga también la imagen, pero como un array de 2 dimensiones (Banda y muestra). El resultado (*Img_lineal*) es el que posteriormente se utiliza dentro del predictor.

Hay que destacar que estas funciones son relativamente complejas; leen la imagen desde el fichero en memoria y se encargan de descomprimirla y ordenar los datos según su formato.

A continuación, se cargan los parámetros de la imagen. Estos parámetros coinciden en nomenclatura con los reflejados en el estándar y permiten optimizar el compresor según el tipo de imagen de entrada que dispongamos, con el objetivo de maximizar la ratio de compresión. En el siguiente fragmento (Código 2) se encuentra este fichero configurado para la imagen procedente de AVIRIS.

Código 2: Script configuración parámetros imagen AVIRIS para Matlab.

```
1 %Parameters of the predictor
2 params.wr = 19;      % Weight Resolution (OMEGA) 19
3 params.R = 48;
4 params.Pz = 3;
5 params.Cz = params.Pz; %Reduced mode
6
7 %Parameters of the Image
8 params.D = 16; %Dinamic Range
9 params.Smax = 2^(params.D)-1;
10 params.Smin = 0;
11 params.Smid = 2^(params.D-1);
12
13 %Dimensions of the image
14 params.Nx = 128;
15 params.Ny = 128;
16 params.Nz = 224;
17 params.Nt = params.Nx * (params.Ny);
18
19 %Parameters of the quantizer
20 params.a = 1; %Losless (=0)
21 params.m = params.a; %Absolute error
22
23 %Parameters of the Sample representative
24 params.srres = 2; % Sample Representative Resolution (THETA) 2
25 params.sroff = 1; % Sample Representative offset (PSI) 1
26 params.srdam = 1; % Sample Representative damping (FI) 1
27
28 %Parameters of Weight Update
29 params.vmin = 0; % >= -6
30 params.vmax = 4; % <= 9
31 params.tinc = 256; %Entre 2^4 y 2^11
32 params.wexpoff = 0; %Inter-band weight exponent offset REVISAR (Puede ser un
vector para cada Pz)
33 params.wcmax = 2^(params.wr + 2)-1;
34 params.wcmin = -2^(params.wr + 2);
```

En este momento, todo lo necesario se encuentra cargado en el *workspace* y, por tanto, se puede iniciar el predictor. Este se ejecuta externamente en el *script* explicado previamente y funciona cargando todos sus datos de las variables en memoria. Al terminar el residuo se almacena en la variable *residualMappedTot*.

Finalmente, con la función *bi_export* se almacena el residuo generado en una ubicación dada, con el formato BI. Este fichero permitirá comparar los resultados con los generados por los *softwares* de referencia y, a su vez, introducir el resultado en el *encoder* para terminar el proceso completo de compresión de la imagen.

4.2.2. Verificación de la Implementación en Matlab

Para verificar el correcto modelado de cada algoritmo, se han realizado dos procedimientos distintos, debido a la diferente disponibilidad de trabajos previos según la versión del estándar. Las imágenes con las que se han validado ambas versiones del predictor se muestran en la Tabla 3.

Tabla 3: Imágenes para verificación en Matlab

Imagen	Dimensión X	Dimensión y	Número de bandas	Formato Datos	Codificación Imagen
Artifacts	8	7	17	unsigned 16 bits	bil
Input 512	512	512	256	signed 16 bits	bip
Aviris (reducida)	128	128	224	unsigned 16 bits	bil

Para el algoritmo basado en el CCSDS 123.0-B-1, se ha realizado un proceso de verificación del compresor completo: predicción en Matlab y su posterior codificación y descompresión con el código C de SHyLoC. Para tener una referencia, también se ha realizado este mismo proceso con el predictor y codificador de referencia (software en lenguaje C). La verificación se ha realizado sobre las tres imágenes de la Tabla 3.

El codificador no se encuentra dentro del *scope* de este proyecto; por tanto, se ha separado esta etapa del resto del compresor realizado en C bajo el proyecto SHyLoC. Al separar el codificador, podemos alimentar a éste con el residuo generado del algoritmo en Matlab, cerrando el proceso global de compresión.

Para verificar el sistema se han tomado dos medidas:

1. Se ha obtenido la SNR de los residuos generados por ambos predictores. La validación es correcta si la equivalencia es exacta.
2. Se obtiene la SNR de la imagen reconstruida, tras ser codificada y descomprimida, frente a la imagen original. Si la equivalencia con la imagen original es perfecta, estamos ante un compresor *lossless*, reafirmando la validez del algoritmo desarrollado.

En la Figura 36 se muestra visualmente la representación del proceso de validación para la banda 4 de la imagen AVIRIS. En esta vemos la imagen original en la parte superior, mientras que a continuación se muestra el resultado del predictor de Matlab (centro-izquierda) y a la derecha el equivalente obtenido del software de SHyLoC. La última imagen de la izquierda muestra el resultado de descomprimir el residuo de la predicción de Matlab, por medio del descompresor de referencia (el procedimiento explicado previamente). A su derecha, la imagen descomprimida mediante el flujo de referencia.

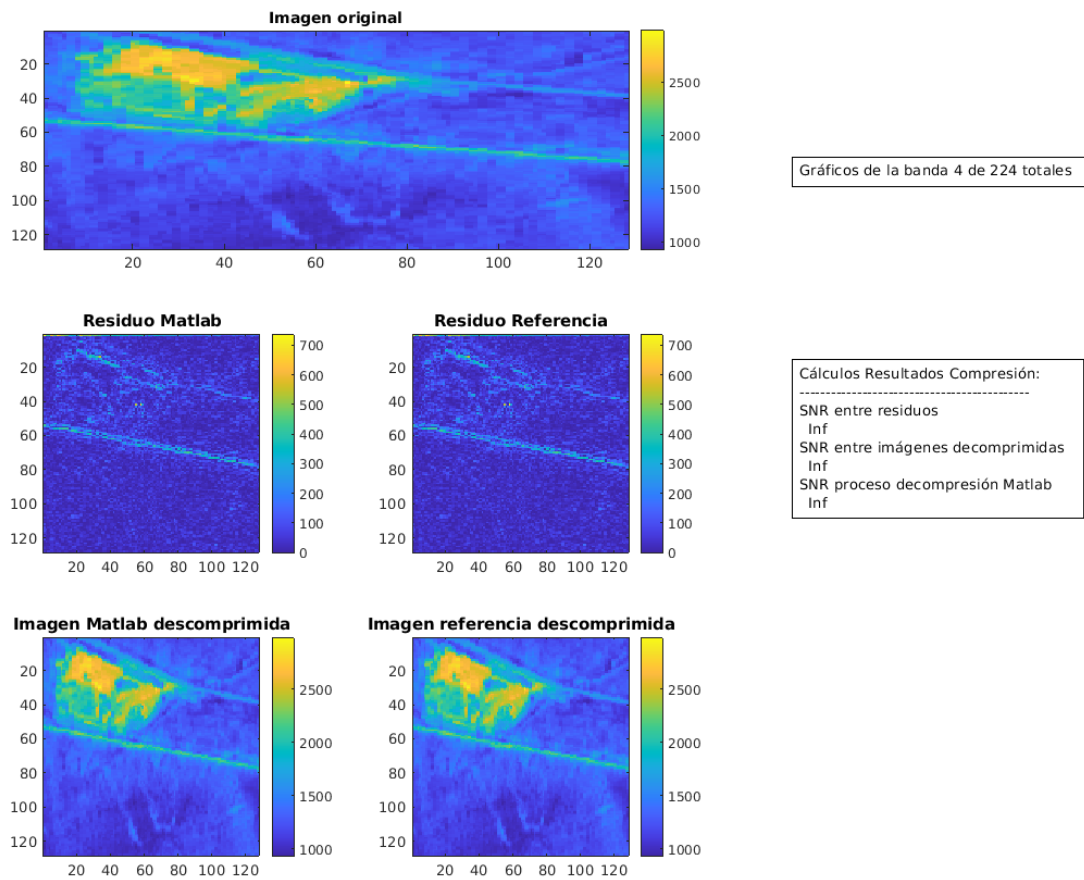


Figura 36: Representación del proceso de verificación para la imagen AVIRIS, para el predictor basado en CCSDS 123.0-B-1.

Si analizamos los resultados, vemos una perfecta equivalencia visual con el proceso realizado con el predictor de referencia. Esta impresión se ratifica con los resultados de la comparación mediante la SNR, que muestran una perfecta equivalencia entre los residuos, las imágenes descomprimidas y entre la descomprimida y original (Figura 36). Estos resultados se repitieron para todas las imágenes de prueba, lo que validó la funcionalidad del predictor sin pérdidas.

Para verificar el compresor con pérdidas (basado en el estándar CCSDS 123.0-B-2), se ha tenido que tomar otro enfoque, ya que no se disponía de una etapa codificadora externa ni de un descompresor. La validación se ha realizado por la primera de las medidas vistas para el predictor *lossless*, que no es otra que la correspondencia entre los residuos obtenidos por el predictor en Matlab y los de un compresor de referencia. Para este caso, se ha tenido acceso al ejecutable del compresor, aún en desarrollo, dentro de la división DSI del IUMA.

En la Figura 37: Representación del proceso de validación para la imagen AVIRIS, para predictor basado en CCSDS 123.0-B-2. se muestra el resultado equivalente al visto en la Figura 36: Representación del proceso de verificación para la imagen

AVIRIS, para el predictor basado en CCSDS 123.0-B-1.. Empleando la misma imagen que para el caso anterior, AVIRIS en su banda 4, pero con compresión *near-lossless* y error constante de $m = 1$. A la izquierda se aprecia el residuo mapeado obtenido en Matlab y a la derecha el obtenido del compresor de referencia. Se ha calculado el error entre ellos y se ha obtenido una plena concordancia ($SNR = Inf$).

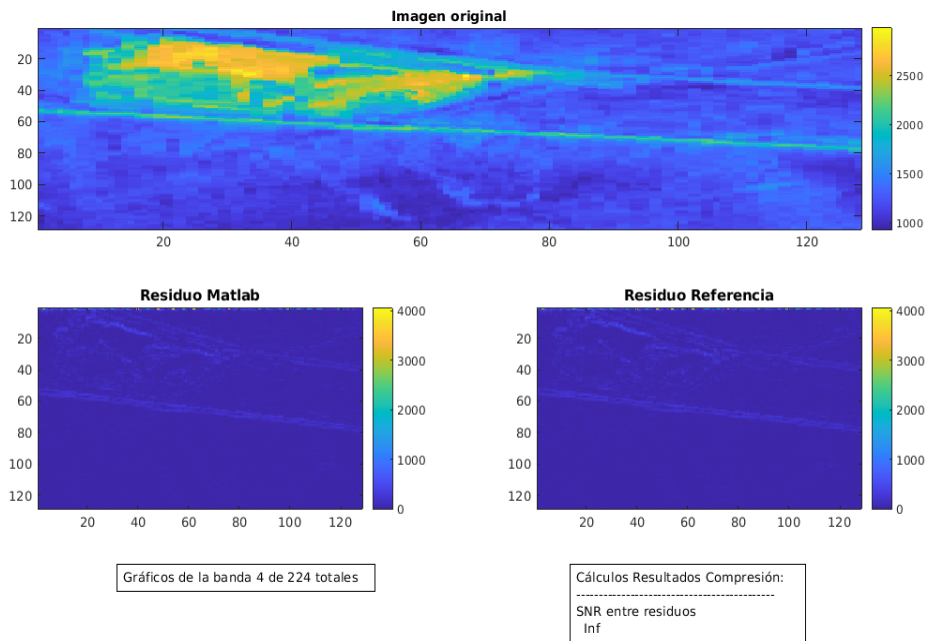


Figura 37: Representación del proceso de validación para la imagen AVIRIS, para predictor basado en CCSDS 123.0-B-2.

Finalmente, este proceso de verificación se realizó sobre las tres imágenes de referencia con las configuraciones de la Tabla 4. Se obtuvieron resultados iguales a los vistos, de perfecta equivalencia, para todos los casos a analizados, por lo que se comprueba la validez de la implementación *lossy* frente al estándar CCSDS 123.0-B-2.

Tabla 4: Configuraciones validación predictor *lossy*

Configuraciones validación predictor basado en CCSDS 123.0-B-2								
Imagen	Artifacts	Artifacts	Artifacts	Input 512	Input 512	Aviris	Aviris	Aviris
Error Cuantificación	1	1	4	1	1	1	3	8
sr_res	0	2	2	2	0	2	2	2
sr_off	0	1	1	1	0	1	1	1
sr_dam	1	1	1	1	1	1	1	1

4.3. Diseño del diagrama de bloques en Simulink

Para desarrollar la implementación física sobre la FPGA, tenemos que dar el salto al entorno de Simulink. Desde este se ha realizado el diseño final de ambas versiones del predictor y la gestión de interfaces externas. Las interfaces son comunes para ambas versiones, por lo que se desarrollaron para el IP basado en el CCSDS 123.0-B-1 y se mantuvieron para la extensión al CCSDS 123.0-B-2. Esto último permite que sean intercambiables en tiempo de diseño.

En la Figura 38 vemos el entorno de prueba sobre el que se ha desarrollado el bloque. A la izquierda tenemos todas las entradas del bloque y a la derecha las salidas, que podemos visualizarlas en los *Scope*. La señal de entrada con la que excitamos al sistema proviene del bloque *Triggered Signal From Workspace*, que introduce serialmente los valores del vector *input_signal*, sincronizados con el flanco de bajada de la línea *pulse_next_sample_gen*. El residuo del predictor es guardado por el bloque *Triggered To Workspace*, que hace la labor inversa al anterior; para cada flanco de la señal *valid_out*, guarda una nueva muestra en la variable *residual_out*.

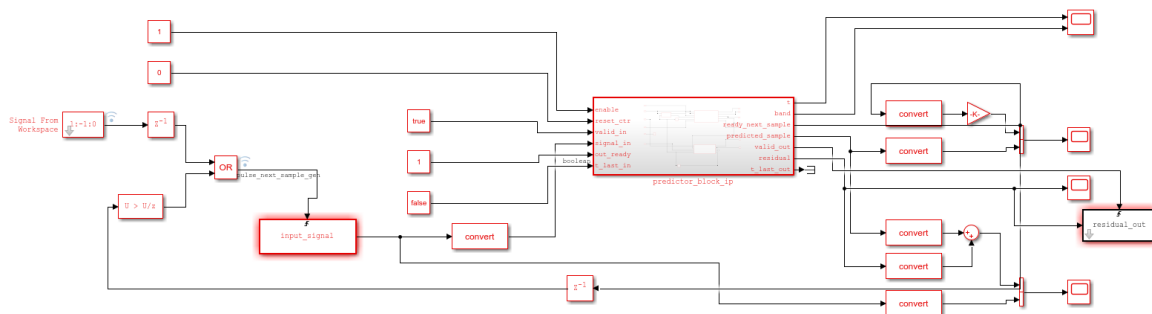


Figura 38: Entorno de pruebas bloque predictor.

En el centro de este diagrama se encuentra el bloque *predictor_block_ip*, el predictor en sí y el bloque que se ha sintetizado. Sus entradas y salidas corresponden con las del bloque final, resumidas en la Tabla 5.

Tabla 5: E/S predictor_bloque_ip.

Puerto	Sentido	Formato
enable	Entrada	boolean
reset	Entrada	boolean
valid_in	Entrada	boolean
signal_in	Entrada	Unsigned int 16 bits
out_ready	Entrada	boolean
t_last_in	Entrada	boolean
t	Salida	Unsigned int 32 bits
band	Salida	Unsigned int 16 bits
ready_next_sample	Salida	boolean
predicted_sample	Salida	Unsigned int 16 bits

valid_out	Salida	boolean
residual	Salida	Unsigned int 16 bits
t_last_out	Salida	boolean

En la Figura 39 vemos el interior del bloque, en el que se encuentran tres elementos:

- *Predictor_block*: este bloque implementa el predictor. Encontraremos diferencias en su comportamiento según estemos en la versión lossless o lossy.
- *Counter*: se encarga de generar los valores de posición *t* y *z* partiendo de una señal pulsada. Se configura con los valores de la imagen que vamos a introducir al sistema.
- *Axi_stream_control*: este bloque controla las comunicaciones AXI4-Stream que llegan al bloque y gestiona el trasiego de datos. Actúa como interfaz entre las líneas externas, con interfaces AXI estandarizadas, y las del bloque predictor que se encuentra dentro. A su vez, genera los pulsos que inician el proceso de predicción de la muestra actual.

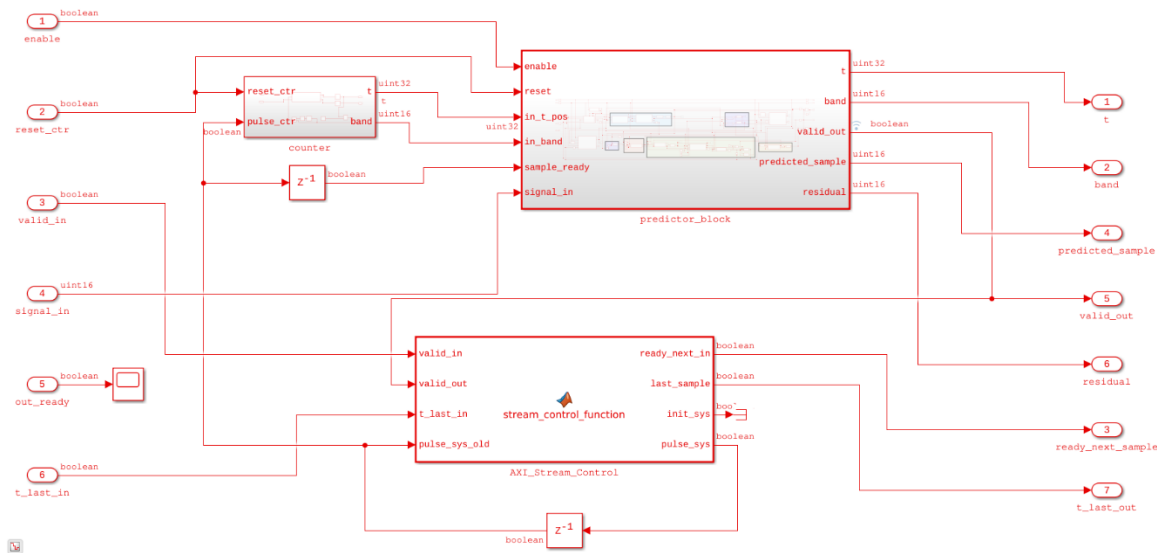


Figura 39: Diagrama predictor_block_ip.

Ahora que ya hemos visto las interfaces exteriores del predictor, vamos a comenzar a analizar cómo se ha implementado internamente el predictor. En primer lugar, vamos a tratar su interfaz exterior; en la Tabla 6 podemos ver cada señal y su formato.

Tabla 6: E/S predictor block.

Puerto	Sentido	Formato
enable	Entrada	boolean
reset	Entrada	boolean
In_t_pos	Entrada	Unsigned int 32 bits
In_band	Entrada	Unsigned int 16 bits
Sample_ready	Entrada	boolean

Signal_in	Entrada	Unsigned int 16 bits
t	Salida	Unsigned int 32 bits
band	Salida	Unsigned int 16 bits
Valid_out	Salida	boolean
predicted_sample	Salida	Unsigned int 16 bits
residual	Salida	Unsigned int 16 bits

Si lo analizamos, vemos que el protocolo implementado es muy sencillo, apreciándose en la Figura 40 su funcionamiento. Primero, se indica la posición que nos encontramos por medio de: *in_t_pos* (línea azul) e *in_band*. Al mismo tiempo, la muestra se introduce por el puerto *signal_in*. Cuando están listas, se lanza un pulso en la señal *sample_ready* (señal amarilla) y comienza la predicción. Al finalizar, la salida *valid_out* (señal naranja) propagará un pulso, que marcará el momento de lectura de las salidas. Tras leer los datos, se repite el proceso con la muestra siguiente.

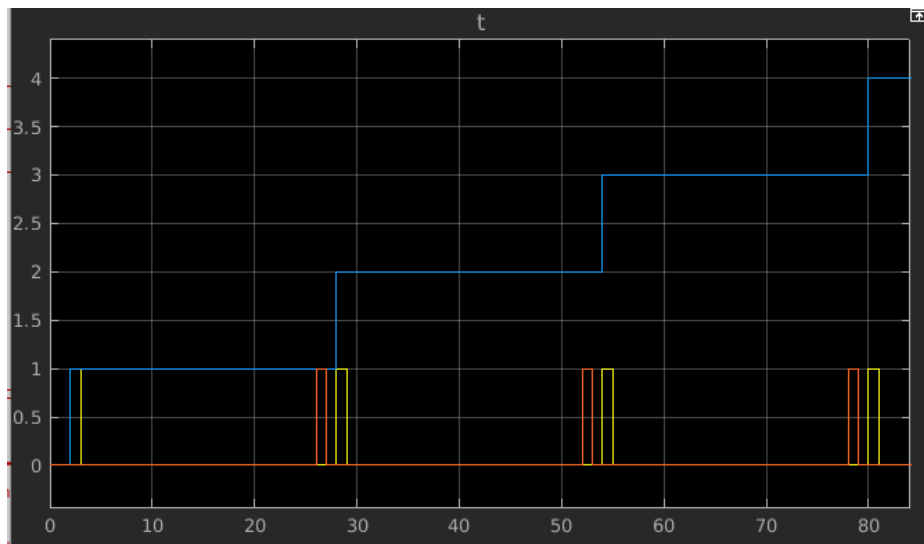


Figura 40: Captura ciclo comunicaciones Simulink

A continuación, pasamos a describir cómo se ha realizado la implementación del predictor *lossless*. Posteriormente se explicará la versión *lossy* y cómo ha sido su extensión desde el primero.

4.3.1. Predictor Lossless

En la Figura 41 se puede ver el diagrama general de la implementación realizada. Pasaremos analizar los bloques y su implementación punto por punto, pero cabe destacar que todo el diseño se basa en la primera versión generada sobre MATLAB, aunque para su implementación sobre un sistema físico son necesarios una gran cantidad de cambios.

En el sistema, la señal entrante llega al bloque *local_sum*, el cual la almacena en memoria y propaga por el sistema. A su vez, dentro del *local_sum*, almacenamos tres vectores con los valores pasados:

- *Actual_row*, almacena las muestras de la fila que estamos calculando.
- *Prev_row*, almacena las muestras de la fila previa, ya calculada.
- *Prev_band_fst_row*, almacena las muestras de la primera fila de la banda anterior.

Estos tres vectores tienen la longitud del ancho de la imagen (*size_x*) y son los valores mínimos que debemos almacenar para poder realizar la predicción del tipo *Narrow Neighbour*. Con esta estrategia se reducen, frente a la implementación de Matlab en la que almacenábamos todas las muestras previas, los requisitos del almacenamiento; pasamos de 3.670.016 muestras almacenadas a solo 384 muestras en memoria (datos para la configuración de la imagen AVIRIS). A continuación, se puede ver la captura de cómo se ha realizado dicha implementación:

Código 3: Implementación Local Sum.

```

if(valid_in)

    localsum = uint32(0);

    %Cálculo de la Local Sum
    if(z== 1 && y== 1 && x > 1)
        localsum = 4 * uint32(sMid);
    elseif(y>1 && x == 1)
        localsum = 2 * (uint32(prev_row(x)) + uint32(prev_row(x+1)));
    elseif(y>1 && x == size_x)
        localsum = 2 * (uint32(prev_row(x)) + uint32(prev_row(x-1)));
    elseif(z > 1 && y== 1 && x > 1)
        localsum = 4 * uint32( prev_band_fst_row(x-1));
    elseif(y > 1 && x > 1 && x < size_x)
        localsum = uint32(prev_row(x-1)) + 2*uint32(prev_row(x)) + uint32(prev_row(x+1));
    else
        localsum = 4 * uint32(sMid); %Valor no necesario, no se encuentra definido en el estándar
    end

    last_local_sum = localsum;

    %Almacena la muestra actual en memoria
    actual_row(x) = cast(newSample,'uint16');

    %Si estamos en la primera fila de la banda, almacenamos estos
    %valores en prev_band_fst_row.
    if(y == 1 && x == size_x)
        prev_band_fst_row(:) = actual_row(:);
    end
end

```

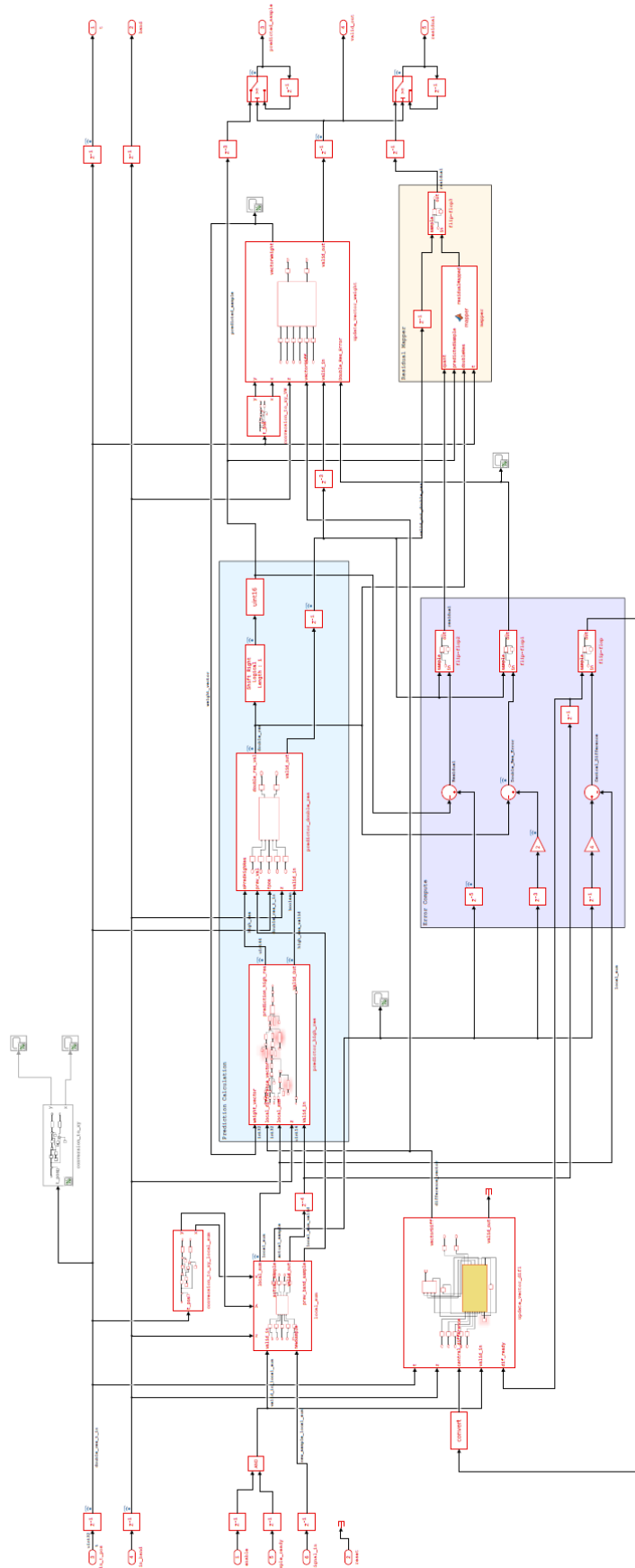


Figura 41: Diagrama implementación Simulink predictor Lossless.

Continuando con el flujo de la fase de predicción, se encuentra el bloque *predictor_high_res*. Este ha sido implementado mediante los bloques propios de Simulink, tal como se ve en la Figura 42, en vez de con código Matlab. La operación implementada ha sido la siguiente:

$$\tilde{s}_z(t) = \text{clip} \left(\text{mod}_R^* \left[\hat{d}_z(t) + 2^{\Omega} (\sigma_z(t) - 4s_{\text{mid}}) \right] + 2^{\Omega+2} s_{\text{mid}} + 2^{\Omega+1} \cdot \{ 2^{\Omega+2} s_{\text{min}}, 2^{\Omega+2} s_{\text{max}} + 2^{\Omega+1} \} \right) \quad (30)$$

Siendo el vector \hat{d}_z el producto vectorial entre el vector de pesos (W) y el vector diferencias (U).

Por tanto, las entradas que tenemos al bloque son el vector de diferencias (*local_difference_vector*), el vector de pesos (*weight_vector*), la banda actual, la *local_sum* y, como en todos los bloques, un *valid_in*. El resultado de las operaciones del bloque es el valor *prediction_high_res* que se representa como una señal *unsigned* de 64 bits.

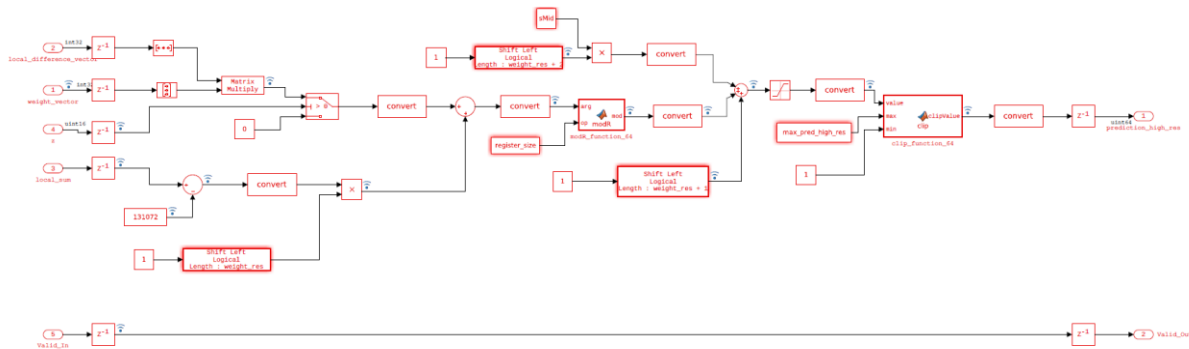


Figura 42: Implementación Simulink bloque predictor_high_res.

La señal *prediction_high_res* entra directamente en el bloque predictor *double_res*. Este se encarga de reducir la resolución de la muestra a *unsigned* de 32 bits y añade las siguientes excepciones al cálculo:

- Si $t=1$ y $Z > 1$. La salida *double_res* es 2 veces el valor de la primera muestra de la banda anterior. Esta la almacena la *local_sum* y la envía por la línea *prev_val*.
- Si $t=1$ Y $Z = 0$. La salida *double_res* es 2 veces el valor *S_{mid}* (valor medio del rango dinámico).

En el posterior fragmento Código 4 se puede apreciar la implementación realizada. Un detalle para destacar es que para que la función *floor* opere correctamente es necesario que trabajemos con decimales; si no, al realizar la división el resultado tiende a redondear previamente el valor, por lo que se ha realizado la división en varias etapas.

Código 4: Implementación predictor double resolution

```

if(valid_in)
  if(tpos > 1)
    %Divide predicted high res entre 2 elevado a weight_res + 1;
    div = 2^(uint64(weight_res)+ 1);
    sPredHighRes_fi = fi(sPredHighRes, 0, 66,2);
    test= sPredHighRes_fi / div;

    %Reduce la resolución
    double_res_val = uint32(floor(test));

  elseif(tpos == 1 && (z > 1 || P > 0))
    double_res_val = 2*uint32(prev_val);

  else
    double_res_val = uint32(2 * sMid);

  end
  last_double_res = double_res_val;
  valid_out = true;
else
  double_res_val = last_double_res;
  valid_out = false;
end
end

```

La señal *double_res* posteriormente se divide entre dos y se reduce a un *unsigned* de 16 bits, terminando en este punto la etapa de predicción.

A continuación, se calculan los errores de predicción con los que posteriormente se realimentará al sistema, por medio del vector de pesos y el vector de diferencias. Tal y como se define en el estándar [39], contamos con tres errores distintos:

- *Residual*: $actual_sample - predicted_sample$.
- *Double Resolution Error*: $2 * actual_sample - double_res_val$.
- *Central Difference*: $4 * actual_sample - local_sum$.

En la Figura 43 se aprecia cómo se han calculado estos errores y posteriormente se almacenan en un *flip-flop* para facilitar su lectura.

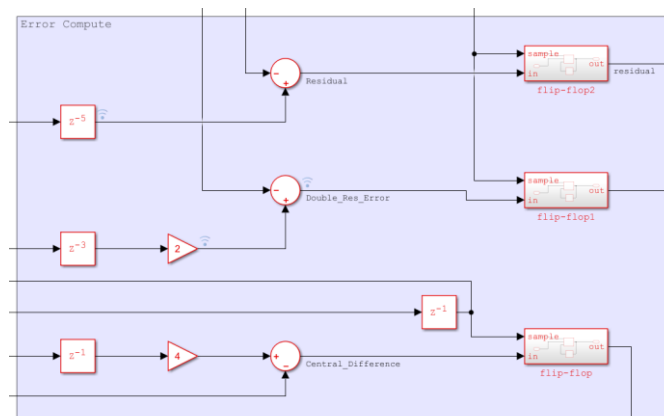


Figura 43: Cómputo errores Simulink.

Uno de los bloques más complejos en su implementación física y correcto funcionamiento ha sido el de actualización del vector de pesos (*update_vector_dif*). Este se ha diseñado para que haga uso de los *block RAMs* de la FPGA, pero debido a la limitaciones de Matlab para la síntesis HDL, ha sido diseñado de una forma algo especial [40]. El funcionamiento del bloque es sencillo. Almacena las *central differences* de las muestras pasadas y con ellas genera el vector de diferencias. El vector de diferencias es la concatenación en un vector de las *Pz* diferencias en las bandas previas para la posición t , en la que nos encontramos dentro de la imagen, tal como viene reflejado a continuación:

$$\mathbf{U}_z(t) = \begin{bmatrix} d_{z-1}(t) \\ d_{z-2}(t) \\ \vdots \\ d_{z-P_z}(t) \end{bmatrix}. \quad (31)$$

Esto es cierto teniendo en cuenta que estamos trabajando únicamente con la versión reducida del compresor; en caso contrario, también incluiríamos tres posiciones extra con las *Directional Local Differences* en el vector \mathbf{U} .

En la Figura 44 se muestra la implementación del bloque. Se aprecia claramente que trabajamos con dos bloques distintos: una *Dual Port RAM* y un bloque con el código Matlab. Para gestionar la memoria tenemos que trabajar con las direcciones de lectura y escritura, las cuales determinan la necesidad de recursos con su tamaño, además de con el bus de escritura y lectura, que para nuestro caso será del tipo *signed* de 19 bits.

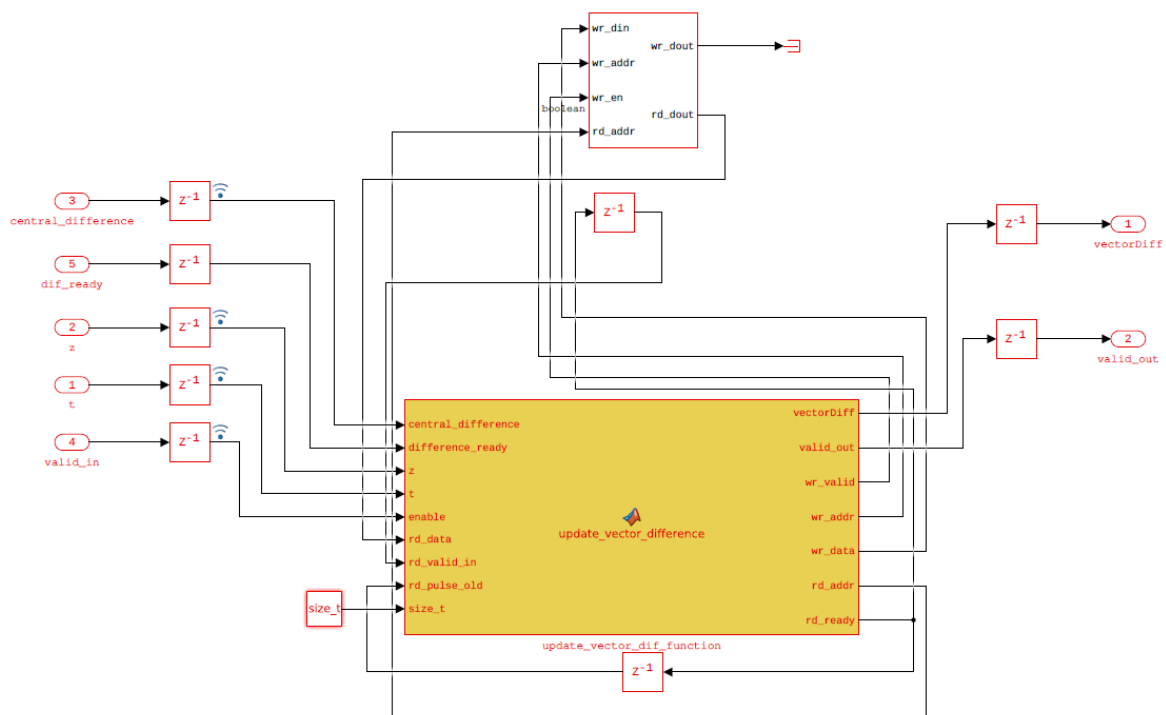


Figura 44: Implementación Simulink bloque actualización de diferencias.

Si analizamos el código dentro del bloque Matlab que se encuentra a continuación, se aprecia que dentro solo almacenamos el valor actual del vector U. Por tanto, todos los valores que almacenemos tendrán que recogerse de la memoria RAM. La memoria dedicada de la FPGA es relativamente elevada pero no infinita, por lo que para optimizar su uso solo guardaremos las últimas Pz bandas analizadas y la actual. Para el caso de la imagen de prueba AVIRIS supone pasar de necesitar almacenar 69,7Mbits a necesitar solo 1,25 Mbits, un 1,8% del consumo de recursos original.

Para hacer uso de esta estructura, se deben realizar varias tareas por separado:

- Leer las diferencias de las bandas previas y generar el vector U.
- Guardar en RAM la nueva diferencia tras ser calculada.
- Desplazar los valores en memoria para solo almacenar las Pz últimas.

Con este fin, se ha decidido que el bloque puede estar en tres estados:

1. De reposo, en el que espera una nueva entrada.

Código 5: Estado reposo bloque actualización vector diferencias

```
%Cálculo del vector de diferencias
if(enable)

    if(z > 1)
        pos = uint32(1);
        vectorDiff = int32(zeros(3,1));

        %Activa el flag de lectura de la RAM
        reading_data = true;

        %Comienza la lectura del módulo RAM
        adress = t + (pos) * local_size_t;
        rd_addr = fi(adress,0,17,0);
        rd_pulse = true;

    else

        differences_vector = int32(zeros(Cz,1));
        vectorDiff = differences_vector;
    end

    valid_out = true;

end
```

2. De lectura, en el que tras recibir un *valid_in* comienza la lectura de memoria de las anteriores diferencias y activa el *flag* del proceso de lectura. En el proceso de lectura (*Reading_data*) se lee en cada ciclo una muestra de la memoria y se genera progresivamente el vector U.

Código 6: Estado lectura bloque actualización vector diferencias

```

%Se cargan las diferencias previas hasta llenar el vector U.
if(reading_data)
    valid_out = false;
    if(rd_valid_in)
        nBands = uint16(0);
        %Comprobamos que haya suficientes bandas para llenar el vector
        if(z>Pz)
            nBands = uint16(Pz);
        else
            nBands = uint16(z-1);
        end

        differences_vector(pos) = rd_data;

        if(pos >= nBands)
            reading_data = false;
            vectorDiff = differences_vector;
            pos = uint32(1);
        else
            pos = pos + 1;
            adressrd = t + (pos)* local_size_t ;
            rd_addr = fi(adressrd,0,17,0);
            rd_pulse = true;
        end
    end
else
    valid_out = true;
end

```

3. El estado de desplazamiento se activa al llegar una nueva diferencia y tras guardar la última muestra, activa el *flag* del proceso de desplazamiento. En el proceso de desplazamiento se lee en cada ciclo una diferencia de la posición *t* actual y se reescribe en la banda anterior.

Código 7: Estado desplazamiento datos bloque actualización vector diferencias

```

if(difference_ready)

    %Escribimos el nuevo valor en memoria
    wr_data = central_difference;
    wr_addr = fi(t,0,17,0);
    wr_valid = true;

    %Iniciamos el desplazamiento de los valores anteriores de banda
    pos = uint32(Pz);
    shift_data = true;

    %Comenzamos leyendo la penúltima posición y escribiendo desde la última
    adress = t + (pos-1) * local_size_t;
    rd_addr = fi(adress,0,17,0);
    rd_pulse = true;

else
    if(shift_data == false)
        wr_valid = false;
    end
end

```

```

%Desplaza de banda los valores de la posición previa.
if(shift_data)
    if(rd_valid_in)

        wr_data = rd_data;
        adresswr = t + (pos) * local_size_t;
        wr_addr = fi(adresswr,0,17,0) ;
        wr_valid = true;

        if(pos <= 1)
            shift_data = false;
        else
            pos = pos - 1;
            adressrd = t + (pos-1) * local_size_t ;
            rd_addr = fi(adressrd,0,17,0);
            rd_pulse = true;
        end
    end
end
end

```

Aún falta el bloque de actualización del vector de pesos (*update_vector_weight*), que se encarga de realimentar el sistema junto con el vector de diferencias. Dentro del bloque se recogen dos funciones definidas en el estándar de forma independiente: inicialización del vector y su actualización.

La inicialización se define en el estándar por medio de la siguiente función:

$$\omega_i^{(1)}(1) = \frac{7}{8} 2^\Omega, \quad \omega_i^{(i)}(1) = \left\lfloor \frac{1}{8} \omega_i^{(i-1)}(1) \right\rfloor, i = 2, 3, \dots, P_i^* \quad (32)$$

La inicialización del vector se realiza cuando nos encontramos en la primera muestra de cada banda y es dependiente de la banda en la que nos encontremos. El estándar también refleja la posibilidad de inicializar el vector W de una forma customizada, pero no ha sido implementada en este caso.

En cuanto a la implementación, se asigna el valor para la primera banda y luego, sucesivamente, se divide el valor anterior entre 8. El bucle se mantiene de tamaño fijo, debido a que el paquete HDL Coder solo permite generar bucles de tamaño predefinido y no alterables en tiempo de ejecución.

Código 8: Inicialización vector de pesos

```

if(x ==1 && y == 1)

    nBands = uint16(0);

    %Definimos el número de valores a escribir
    if(z > Pz)
        nBands = uint16(Pz);
    else
        nBands = uint16(z-1);
    end

    %Definimos el vector W
    vector_weight = int32(zeros(Cz,1));

    %Calculamos el vector para cada posición
    vector_weight(1) = (7 * 2^int32(weight_res)) / 8;
    for i=uint16(2:3)
        if (i<=nBands)
            vector_weight(i) = vector_weight(i-1) / int32(8);
        end
    end

    vectorWeight = vector_weight;
    valid_out = true;

elseif(z>1)

```

La segunda función es la actualización del vector de pesos. Dentro del estándar se define por medio de una serie de ecuaciones. Primero es necesario calcular el exponente de escalado de la actualización de pesos (*weight update scaling exponent*), que es un exponente que permite adaptar a qué velocidad se adaptan los pesos a los cambios en la imagen:

$$\rho(t) = \text{clip} \left(v_{\min} + \left\lfloor \frac{t - N_X}{t_{\text{inc}}} \right\rfloor, \{v_{\min}, v_{\max}\} \right) + D - \Omega, \quad (33)$$

El control del valor lo realizaremos por medio de las variables:

- v_{\min} (v_{\min} en el código) que establece el valor mínimo del exponente.
- v_{\max} (v_{\max} en el código) que establece el valor máximo del exponente.
- t_{inc} (t_{inc} en el código) que es el factor de cambio en el intervalo; debe ser una potencia de 2 entre 2^4 y 2^{11} .

Y, tras calcular el exponente, se calcula el nuevo peso para cada posición del vector por la siguiente ecuación:

$$\mathbf{W}_z(t+1) = \text{clip} \left(\mathbf{W}_z(t) + \left\lfloor \frac{1}{2} (\text{sgn}^+ [e_z(t)] \cdot 2^{-\rho(t)} \cdot \mathbf{U}_z(t) + \mathbf{1}) \right\rfloor, \{\omega_{\min}, \omega_{\max}\} \right) \quad (34)$$

De cara a su implementación, se ha realizado el cálculo por partes. Como se aprecia en el Código 9, la operación $2^{-\rho(t)}$ se ha realizado al principio y por separado debido a que HDL Coder no permite trabajar con exponentes variables. Para superar esa limitación, se ha realizado la operación mediante el producto, o división sucesiva de la base de la potencia. Cuando se ha calculado este valor, ya se realiza la operación completa con cada una de las posiciones del vector de pesos (líneas 101 a 105).

Código 9: Actualización vector de pesos.

```
76         temp_1 = int32(-(fp_weightUpdScalingExp + fp_weight_exp_off) );
77         temp_2 = fi(1, 0, 32, 4);
78
79         if(temp_1 >=0)
80             %Calcula el exponente si es positivo
81             for i = uint16(1:5)
82                 if(i <= temp_1)
83                     temp_2 = fi(temp_2 * fi(2, 0, 8, 0), 0, 32, 4);
84                 end
85             end
86         else
87             temp_1 = -temp_1;
88             %Calcula el exponente si es negativo
89             for i = uint16(1:5)
90                 if(i <= temp_1)
91                     temp_2 = fi(temp_2 / fi(2, 0, 8, 0), 0, 32, 4);
92                 end
93             end
94         end
95
96         wc_max = int32(2^(double(weight_res) + 2)-1);
97         wc_min = int32(-2^(double(weight_res) + 2));
98
99         %Actualiza el vector de pesos W
100        for i=uint16(1:3)
101            if (i<=nBands)
102                temp = fi( 0.5, 1, 8, 4)* (fp_signError * temp_2* \
fp_differences(i) + fi( 1, 1, 8, 0));
103                new_vector_weight(i) = fp_vector_weight(i) + floor(temp);
104                new_vector_weight(i) = clip_int32(new_vector_weight(i), \
wc_max,wc_min);
105            end
106        end
107
108        vectorWeight = new_vector_weight;
109        vector_weight = new_vector_weight;
```

En este punto, solo queda tratar el bloque de salida, el *mapper*. Su función es realizar la conversión del residuo, de tipo entero con signo y 16 bits, a sin signo de 16 bits. Dentro del estándar viene definido por las siguientes funciones:

$$\delta_z(t) = \begin{cases} |\Delta_z(t)| + \theta_z(t), & |\Delta_z(t)| > \theta_z(t) \\ 2|\Delta_z(t)|, & 0 \leq (-1)^{\hat{s}_z(t)} \Delta_z(t) \leq \theta_z(t) \\ 2|\Delta_z(t)| - 1, & \text{otherwise} \end{cases} \quad (35)$$

$$\theta_z(t) = \min\{\hat{s}_z(t) - s_{\min}, s_{\max} - \hat{s}_z(t)\}. \quad (36)$$

La salida, que será la salida del sistema, es $\delta_z(t)$ pero para su cálculo necesitamos del residuo $\Delta_z(t)$, que es una entrada del bloque. Para la implementación se han calculado θ tal cual se define en el estándar, pero para implementar $(-1)^{\hat{s}_z(t)}$, simplemente se ha calculado si la *double predicted sample* es par o impar y con ello se aporta el signo a la operación. De esta forma, no es solo más eficiente, sino que además es realizable en hardware. Como hemos mencionado anteriormente en HDL no se puede hacer uso de un exponente variable.

Esta es la implementación del *mapper* para la versión *lossless*, ya que para la versión con pérdidas tiene algunas diferencias:

Código 10: Implementación *lossless mapper*

```
%Distancia de la muestra predicha al máximo o mínimo
omega = min(uint32(predictedSample) - sMin, sMax - uint32(predictedSample));

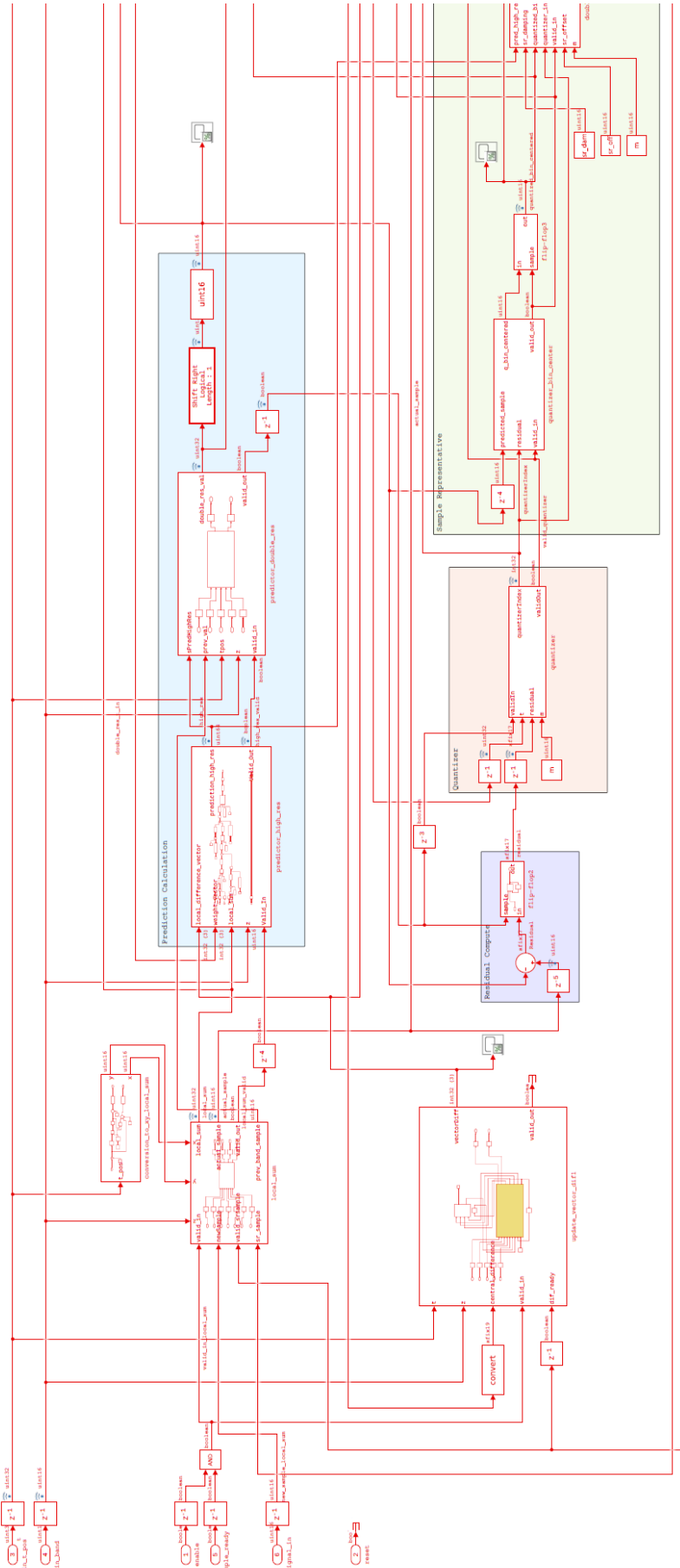
resto = uint8(mod(doubleRes, 2));
if resto == 0
    sign = int8(1);
else
    sign = int8(-1);
end

temp = int32(sign) * int32(quant);

%Asignamos el valor según los casos definidos en el estándar.
if(abs(quant) > omega)
    residualMapped = uint16(abs(quant) + int32(omega));
elseif (0 <= temp && temp <= omega)
    residualMapped = uint16(2 * abs(quant));
else
    residualMapped = uint16((2 * abs(quant)) - 1);
end
```

4.3.2. Lossy Compressor

Se ha analizado la implementación del compresor *lossless* y, tal como se ha comentado, esta es la base del compresor con pérdidas. El estándar CCSDS 123.0-B-2 basa su compresor en el definido en la versión previa, pero añade los bloques *Sample Representative* y *Quantizer*, además de algunas pequeñas variaciones. Esto también se aprecia en el esquema del sistema que se puede ver en la Figura 45.



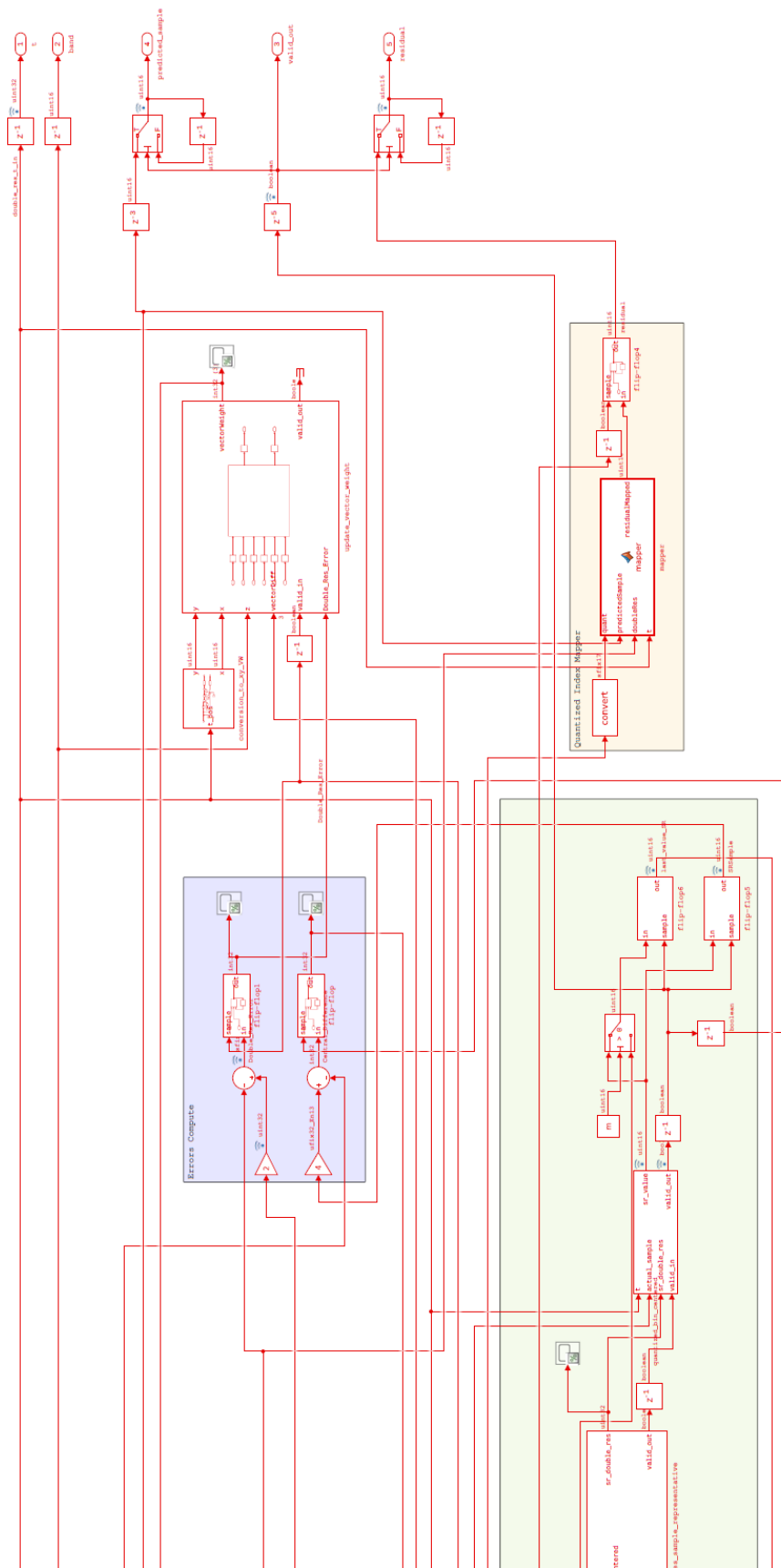


Figura 45: Diagrama implementación Simulink predictor Lossless.

En este esquema, el bloque *quantizer* coincide con la sección coloreada de rojo y la verde con el *sample representative*. En el resto del sistema mantenemos una configuración equivalente, pero destacan las siguientes diferencias globales:

- El bloque *local sum* añade una nueva entrada para la señal reconstruida (*SR_sample*), ya que es el valor que almacenará para generar sus próximos cálculos.
- Se retrasa el cálculo de los errores *double_res_error* y *central_difference*, ya que ahora pasan a compararse con el resultado del *sample representative* en vez de con la muestra entrante.
- El *mapper* recibe modificaciones en la forma de calcular θ , la nueva ecuación es:

$$\theta_z(t) = \begin{cases} \min\{\hat{s}_z(0) - s_{\min}, s_{\max} - \hat{s}_z(0)\} & t = 0 \\ \min\left\{\left\lfloor \frac{\hat{s}_z(t) - s_{\min} + m_z(t)}{2m_z(t) + 1} \right\rfloor, \left\lfloor \frac{s_{\max} - \hat{s}_z(t) + m_z(t)}{2m_z(t) + 1} \right\rfloor\right\}, & t > 0 \end{cases} \quad (37)$$

- La entrada del *mapper* es el residuo cuantificado $q_z(t)$.

A continuación, entramos a describir los nuevos bloques de la versión con pérdidas y comenzamos por el primero en el flujo de ejecución: el *quantizer*. En la Figura 46 se puede ver la implementación realizada.

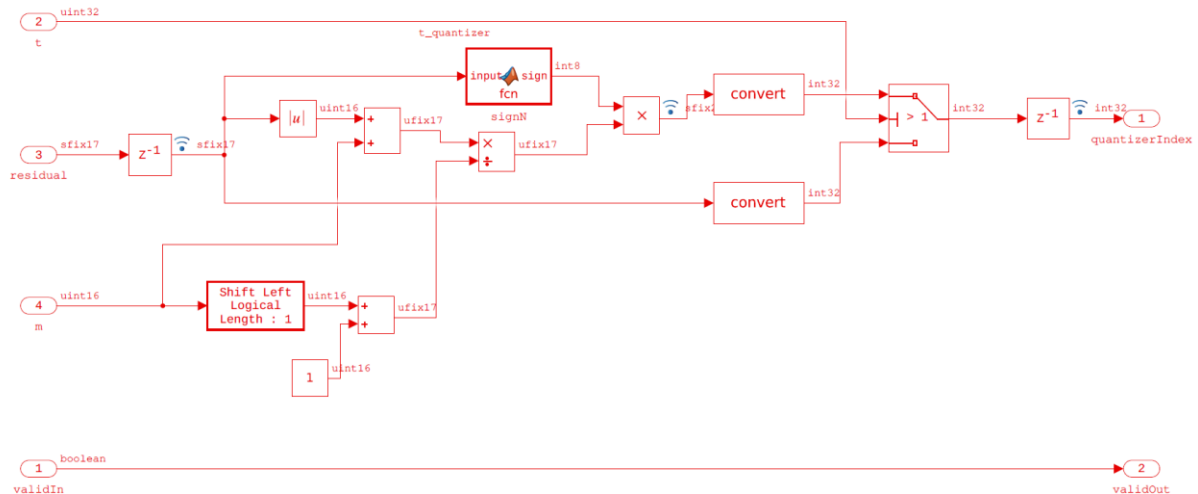


Figura 46: Implementación Simulink bloque cuantificador.

Destaca que para este bloque la implementación se ha realizado únicamente mediante bloques Matlab, pero es debido a que la dificultad lógica de este hacía más conveniente esta estrategia. Si nos fijamos en la ecuación que lo define:

$$q_z(t) = \begin{cases} \Delta_z(0), & t = 0 \\ \text{sgn}(\Delta_z(t)) \left\lfloor \frac{|\Delta_z(t)| + m_z(t)}{2m_z(t) + 1} \right\rfloor, & t > 0 \end{cases}, \quad (38)$$

Se aprecia que el factor de reducción tan solo depende del parámetro $m_z(t)$ que, a su vez, depende solo de a_z , ya que trabajamos únicamente con error absoluto.

El *sample representative* se divide en tres bloques, que coinciden con las tres ecuaciones que definen su funcionamiento dentro del estándar: *quantizer bin center*, *double-resolution sample representative* y *sample representative* (Figura 47: Implementación Simulink *sample representative*).

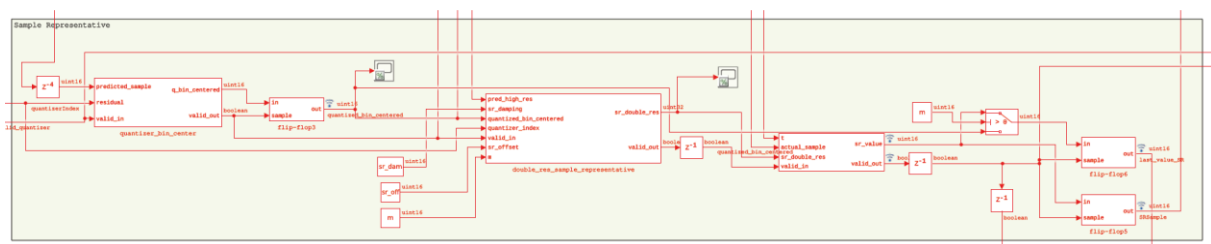


Figura 47: Implementación Simulink *sample representative*.

El primer bloque es el *quantizer bin center*, cuya labor es reescalar el residuo a la resolución original y sumárselo a la muestra predicha. A nivel matemático su implementación es la siguiente:

$$s'_z(t) = \text{clip}(\hat{s}_z(t) + q_z(t)(2m_z(t) + 1), \{s_{\min}, s_{\max}\}) \quad (39)$$

De cara a su implementación, esta es muy sencilla (Figura 48), pero se le ha añadido una condición. En caso de que el error de cuantificación sea nulo, el residuo no es multiplicado, evitando así posibles errores inesperados.

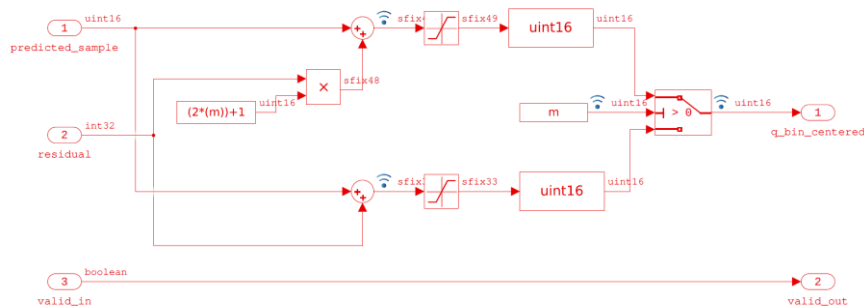


Figura 48: Implementación Simulink bloque *quantized_bin_center*.

El siguiente bloque es *el double resolution sample representative*. Este es el más complejo de todos, ya que trata de reconstruir la muestra original teniendo en cuenta todos los factores descritos anteriormente. Su ecuación es la siguiente:

$$\tilde{s}_z''(t) = \left\lfloor \frac{4(2^\theta - \phi_z) \cdot (s_z'(t) \cdot 2^\Omega - \text{sgn}(q_z(t)) \cdot m_z(t) \cdot \psi_z \cdot 2^{\Omega-\theta}) + \phi_z \cdot \tilde{s}_z(t) - \phi_z \cdot 2^{\Omega+1}}{2^{\Omega+\theta+1}} \right\rfloor, \quad (40)$$

De cara a su implementación, se ha tratado de realizarla de la forma más sencilla posible y por eso se ha optado por la estrategia de bloques de Simulink (Figura 49). La salida del bloque es de tipo *unsigned* de 32 bits y es la entrada del *final_sample_representative*.

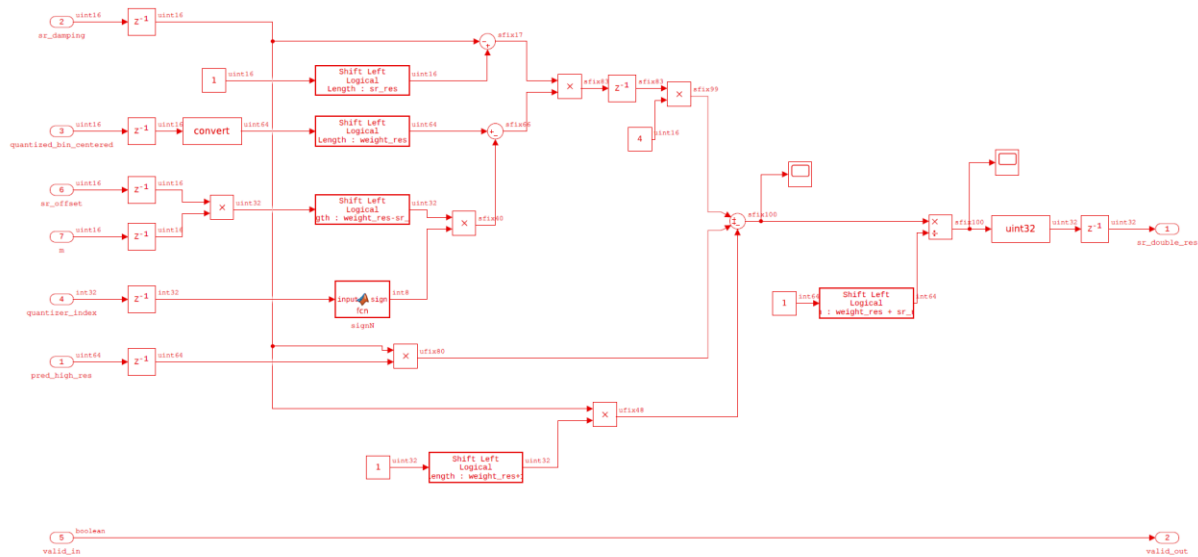


Figura 49: Implementación Simulink bloque *double_resolution_sample_representative*.

Este último bloque (Figura 50) únicamente se encarga de reducir la resolución de la muestra a 16 bits y añade una condición: cuando nos encontramos en la primera muestra de la banda, la salida del *sample representative* será la muestra real de entrada al sistema. La salida (*sr_value*) debería coincidir, con un mínimo margen de error, con la muestra de entrada al predictor.

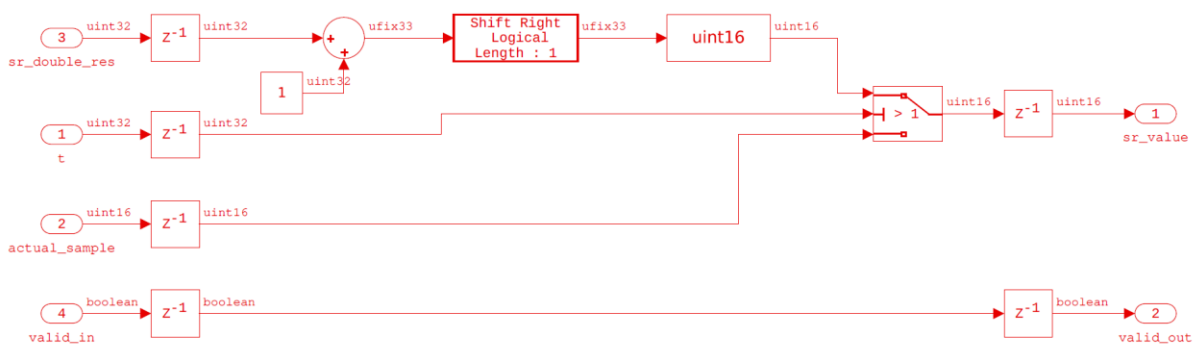


Figura 50: Implementación Simulink bloque *final_sample_representative*.

Con esto queda descrito al completo el proceso de modelado del predictor para su versión CCSDS123.0-B-1 y CCSDS123.0-B-2. A continuación, tan solo nos queda comentar cómo se realiza la configuración del entorno y de los parámetros del predictor.

4.3.3. Inicialización y configuración del predictor

Dentro del estándar se definen una gran cantidad de parámetros que permiten ajustar la etapa de predicción según nuestros objetivos. De forma equivalente a como se explicó para Matlab, se ha creado un *script* de inicialización que carga todos los parámetros necesarios en el *workspace*.

El predictor diseñado hace uso de una gran cantidad de parámetros estáticos que se deben de configurar previamente a su inicialización. Estos parámetros definen en gran medida el comportamiento del bloque y son integrados como constantes dentro del bloque IP generado.

La imagen debe ser cargada en memoria en primer lugar y debe incluirse como un vector de tipo *uint16* (denominación para una variable *unsigned* de 16 bits) que debe llamarse *input_signal*. Además, deben cargarse en memoria todos los parámetros que se encuentran en la Tabla 7.

Tabla 7: Variables de configuración bloque IP y sus valores de referencia

Variable	Denominación estándar	Formato	Valores Referencia
size_x	N_x	uint16	X
size_y	N_y	uint16	Y
size_z	N_z	uint16	Z
size_t		uint32	$N_x * N_y$
sMax	S_{max}	uint32	$2^{16} - 1$
sMid	S_{mid}	uint32	$2^{(16-1)}$
sMin	S_{min}	uint32	0
dynamic_range	D	uint16	16
Cz	C_z	uint16	3
Pz	P_z	uint16	Pz
weight_res	Ω	uint16	19
register_size	R	uint16	48
v_max	v_{max}	uint16	4
v_min	v_{min}	uint16	0
t_inc	t_{inc}	uint16	256
weight_exp_off	$\zeta_z *$	uint16	0
wc_max	ω_{max}	int32	$2^{\Omega+2} - 1$
wc_min	ω_{min}	int32	$-2^{\Omega+2}$
m	$m_z(t)$	uint16	1
sr_res	θ	uint16	2
sr_off	ϕ_z	uint16	1
sr_dam	Ψ_z	uint16	1
input_signal	$S_{z,y,x}$	uint16	Img Linealizada

4.4. Análisis resultados HDL-Coder

Tras la verificación de los bloques diseñados en Simulink mediante simulación, se inició el proceso de síntesis a HDL. Este proceso se realiza con la herramienta HDL Coder, descrita en el apartado 3.3. HDL Coder ofrece múltiples opciones de exportación, pero para este proyecto se ha usado la exportación del diseño como bloque IP. El bloque IP que generado se describe internamente en VHDL, manteniendo una estructura equivalente a la del diseño inicial. A continuación, haremos un breve análisis sobre el formato del código generado y su facilidad para una posterior edición.

Para la edición resulta esencial tener un código legible y comprensible. En el caso de HDL Coder, este permite incluir varias opciones de estilo en el código que mejoran la legibilidad. Las funciones más interesantes que ofrece son:

- Trazabilidad de las funciones. Dentro del informe que se genera, se permite ver el código fuente y se incluyen, dentro de este informe, enlaces a las funciones o líneas de código que cada fragmento de código define.
- Continuidad en el nombre de las variables. Las variables generadas mantienen el nombre de las variables o líneas que representan.
- Inclusión de los comentarios. En las funciones Matlab generadas, se incluyen los comentarios del código original. Así, se permite seguir la funcionalidad que se está implementando en los distintos puntos del código.

El siguiente fragmento de código ha sido generado por HDL Coder. Este código implementa la funcionalidad del bloque cuantificador, que se observa en la Figura 51. En el Código 11 se aprecia claramente que los nombres de las variables concuerdan con los del bloque en Simulink, generando un código sencillo y legible. En el caso de este fragmento, se aprecia un código comparable con el que podría haber sido descrito directamente en VHDL.

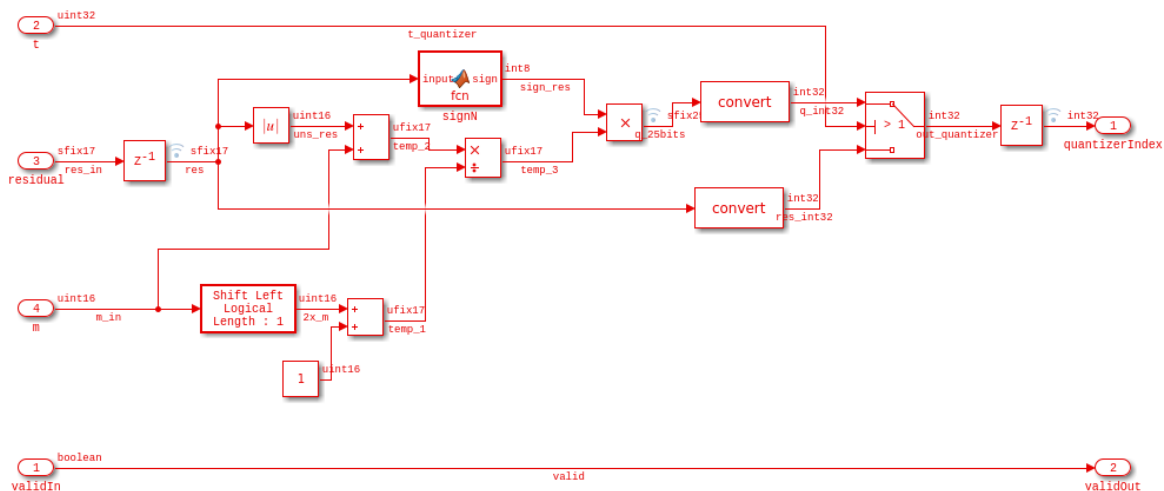


Figura 51: Implementación en Simulink del bloque cuantificador

Código 11: Muestra código VHDL autogenerado

```
m_unsigned <= unsigned(m);
alpha2x_m <= m_unsigned sll 1;
res_int32 <= resize(res, 32);
sign_signed <= signed(sign);
temp_2 <= resize(uns_res, 17) + resize(m_unsigned, 17);
Constant_out1 <= (0 => '1', OTHERS => '0');
temp_1 <= resize(alpha2x_m, 17) + resize(Constant_out1, 17);

Divide_output : PROCESS (temp_1, temp_2)
| VARIABLE div_temp : unsigned(17 DOWNT0 0);
BEGIN
| div_temp := (OTHERS => '0');
| IF temp_1 = to_unsigned(16#00000#, 17) THEN
|   temp_3 <= (OTHERS => '1');
| ELSE
|   div_temp := resize(temp_2, 18) / temp_1;
|   IF div_temp(17) /= '0' THEN
|     temp_3 <= (OTHERS => '1');
|   ELSE
|     temp_3 <= div_temp(16 DOWNT0 0);
|   END IF;
| END IF;
END PROCESS Divide_output;

Product_cast <= signed(resize(temp_3, 18));
Product_mul_temp <= sign_signed * Product_cast;
q_25bits <= Product_mul_temp(24 DOWNT0 0);

q_int32 <= resize(q_25bits, 32);

out_quantizer <= res_int32 WHEN switch_compare_1 = '0' ELSE
| q_int32;
```


5. Validación del sistema y análisis de resultados

En esta sección del documento se analiza cómo ha sido el proceso de validación de los bloques IP generados sobre un dispositivo *hardware* y cuáles han sido los resultados obtenidos.

Comenzaremos analizando el set-up que será utilizado para las pruebas. A continuación, se mostrará la integración de los bloques IP para el set-up de pruebas, dentro del entorno de Vivado. Proseguiremos con la metodología utilizada en el proceso de validación. Finalmente, cerramos el capítulo con la evaluación los datos obtenidos sobre la FPGA. Se comprobará su validez respecto al *software* de referencia, se analizará el uso de recursos lógicos y finalmente se comparará frente a otras implementaciones previas del estándar CCSDS 123.0-B-1.

5.1. Descripción general del set-up

Durante el desarrollo de este proyecto, el código desarrollado ha sido verificado al finalizar cada etapa, con el fin de comprobar su equivalencia con la *golden reference*. Pese a ello, sigue siendo esencial realizar una comprobación del sistema en *hardware*. De estas pruebas no solo se extrae la compatibilidad de los bloques, sino líneas de mejora futuras sobre el rendimiento y sus puntos críticos.

Las pruebas se han realizado en el laboratorio. El *set-up* de test se compone de:

- Kit de desarrollo PYNQ-Z1.
- Ordenador de sobremesa con el entorno de Vivado.
- Tarjeta microSD.

El kit de desarrollo PYNQ Z1 se basa en el mencionado SoC XC7Z020[41]. Dentro de su lógica programable, basada en el diseño de una ARTIX-7, se implementa el bloque predictor junto con el bloque DMA para el acceso a memoria. El *software* se ejecutará dentro de uno de los núcleos ARM-A9 que dispone.

La imagen de prueba se encontrará almacenada en una tarjeta microSD. Esta se introduce en su respectivo conector sobre la PYNQ. En la tarjeta también se almacena el residuo generado al finalizar la prueba.

El ordenador se conecta con el *hardware* mediante micro USB (Universal Serial Bus). El ordenador gestiona tres tareas dentro del banco de pruebas. En primer lugar, desde este se lanza la ejecución de la aplicación, en concreto, desde el entorno de Vitis. Además, durante el proceso de depuración podemos visualizar las comunicaciones por los buses internos a través del ILA, integrado en la FPGA. Finalmente, se leen y visualizan las comunicaciones seriales enviadas por la aplicación, las cuales indican el estado de la predicción.

Se ha elegido el kit PYNQ Z1 (Figura 52) para completar este *set-up* por su reducido costo y disponibilidad dentro de la división DSI. También se ha tenido en consideración el hecho de constar con una CPU integrada, lo que facilita enormemente el tiempo de desarrollo del entorno de pruebas. Destacar que, si el funcionamiento en esta plataforma es correcto, sería posible migrarlo a cualquier otra FPGA, incluyendo aquellas tolerantes a la radiación y empleadas en la industria aeroespacial.

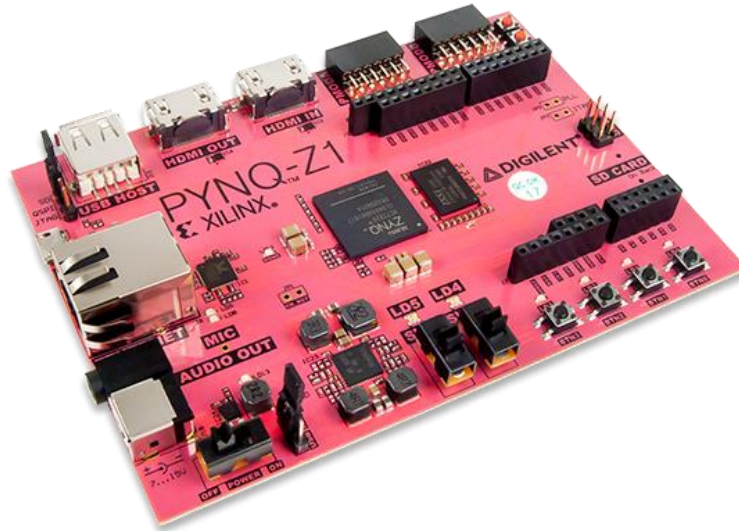


Figura 52: Kit de desarrollo PYNQ.

5.2. Diagrama de bloques del set-up en Vivado

El diseño de bloques final (Figura 53) es relativamente sencillo. El bloque principal es el ZYNQ7 *Processing System*, que recoge las funcionalidades de cómputo del microprocesador, gestión del acceso a periféricos y generación de la señal de reloj para el sistema, entre otras.

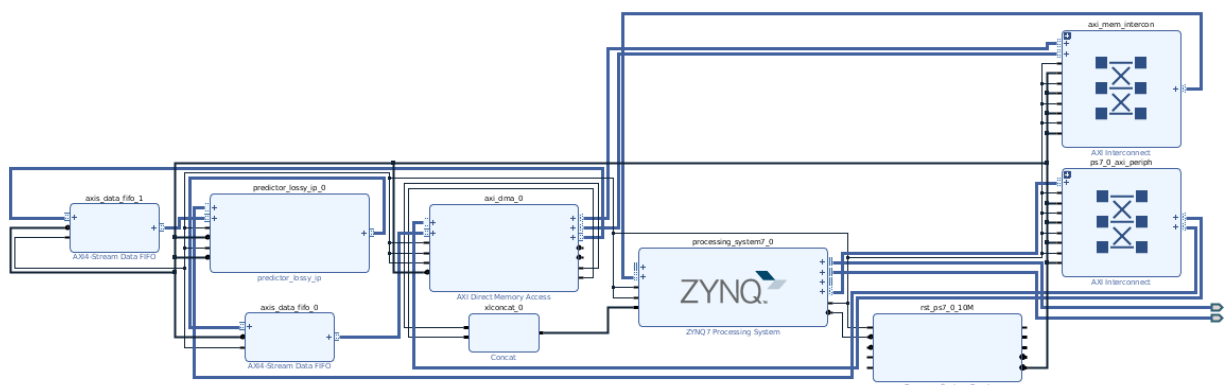


Figura 53: Diseño de bloques sistema en Vivado.

Con el fin de descargar al microcontrolador de una gran cantidad de cómputo, el bloque predictor se encuentra conectado, por medio de dos FIFOs (*First In First Out*), con un bloque AXI DMA. Por este enlace llega el flujo de datos de la imagen entrante y se recoge el residuo de la predicción.

El DMA permite enviar y recibir los datos directamente desde la memoria RAM. Para ello, se conecta a esta por una conexión AXI High Performance y se comunica con la CPU por medio de una conexión AXI4 Lite. Con esta estrategia tan solo es necesario configurar el DMA y enviar la dirección de memoria de la que leer y en la que escribir los datos; desde este momento, el DMA se encarga de realizar el *streaming* de datos.

Al otro lado, el predictor se comunica con el DMA por un flujo AXI4 Stream. Tenemos dos puertos: uno de entrada, para las muestras de la imagen, y otro de salida, para el resultado de la predicción. Las demás comunicaciones con el bloque IP se realizarán por el enlace AXI4 Lite con el PS (*Processing System*). Por medio de este enlace se realizarán las funciones de configuración y se podrán leer algunas variables internas del bloque.

5.3. Aplicación software en Vivado SDK

La implementación de este proyecto se ha realizado sobre el SoC ZYNQ XC7Z020 [41]. Al contar este con un microprocesador, vamos a hacer uso de él para conectarnos con el bloque IP, cargar la imagen desde una tarjeta microSD y controlar el flujo de datos.

El *software* se ha diseñado sin hacer uso de un sistema operativo (*bare-metal*). Las comunicaciones de configuración con el bloque predictor se realizan por AXI4-Lite. Este protocolo se basa en un direccionado en memoria, de tal forma que, a partir de la definición generada en HDL Coder, tenemos definidas las direcciones de cada variable en el registro. Además, para realizar el *software* se ha configurado la definición del Board Support Package (BSP) [42]. El BSP incluye una definición de los módulos implementados en el *hardware* y de las librerías necesarias para su gestión. Para gestionar las comunicaciones con la tarjeta SD se le han añadido las librerías *xilffs* y *xilrsa*.

El flujo de ejecución se encuentra dentro de la función *main()* (Código 12), donde se llaman a las funciones que realizan las tareas específicas. El sistema comienza configurando el bloque predictor mediante el bus AXI4 Lite. A continuación, se carga en memoria la imagen desde la tarjeta SD; el número de muestras de la misma se debe configurar en la contante *SAMPLESTOREAD*. Seguimos guardando la hora del sistema en memoria, para calcular la duración de la predicción posteriormente, y se llama a la función *predictionByDma()*. Esta función gestiona el flujo de comunicación entre el bloque y el DMA durante el proceso de predicción. Al terminar la predicción, los resultados se encuentran almacenados en la memoria RAM, dentro del vector *Results*. Por tanto, tan solo queda calcular el tiempo empleado en la predicción y guardar el residuo en un fichero dentro la SD. El residuo se guarda en el fichero por medio de la función *WriteSamplesToSD()*.

Código 12: Main aplicación micro ZYNQ

```

97 int main()
98 {
99     int Status;
100    XTime tStart, tEnd;
101
102    xil_printf("\r\n--- Entering main() --- \r\n");
103
104
105    /* Configuring the IP by AXI4-Lite*/
106    xil_printf("\r\n** Configuring Pred Block \r\n");
107    Xil_Out32(MY_PRED_BASE + enable_Data_predictor_lossy_ip, 0xFF);
108    Xil_Out32(MY_PRED_BASE + reset_ctr_Data_predictor_lossy_ip, 0x00);
109
110
111    /*Import Data from SD */
112    xil_printf("*** Reading Samples \r\n");
113    ReadSamplesFromSD();
114
115    XTime_GetTime(&tStart);
116    xil_printf("\r\n--- Prediction starts --- \r\n");
117
118    /* Prediction phase */
119    Status = predictionByDma(DMA_DEV_ID);
120
121    XTime_GetTime(&tEnd);
122    xil_printf("\r\n--- Prediction ends --- \r\n");
123    printf("Output took %.2f us.\n",
124          1.0 * (tEnd - tStart) / (COUNTS_PER_SECOND/1000000));
125
126
127    if (Status != XST_SUCCESS) {
128        xil_printf("Some error happend during prediction\r\n");
129        return XST_FAILURE;
130    }
131
132    xil_printf("\r\n** Saving Results \r\n");
133
134    Status = WriteSamplesToSD();
135    if (Status != XST_SUCCESS) {
136        xil_printf("Error Saving samples\r\n");
137        return XST_FAILURE;
138    }
139
140    xil_printf("\r\n--- Exiting main() --- \r\n");
141
142    return XST_SUCCESS;
143

```

El proceso de comunicación durante la predicción se realiza dentro de la función *predictionByDma* (Código 13). La función requiere como parámetro el valor *DeviceId*, que se refiere a la dirección en el bus del bloque DMA.

El funcionamiento de esta función es sencillo. Al inicio, configura el DMA para trabajar en modo *simple* e inhabilita las interrupciones, ya que se trabaja en modo *polling*. A continuación, se pasa al bucle de transmisión. En este bucle la comunicación ocurre en 4 etapas:

1. Se lee de memoria la muestra *n* de la imagen y se carga en el *buffer* de transmisión.
2. Se comunica al DMA de que se va a realizar una transmisión hacia el bloque y de que se va a recibir una desde este. Se indica que ambas comunicaciones son de 16 bits.
3. Se espera a que ambas transmisiones terminen.
4. Se guarda en el vector *Results* el dato leído del *buffer* de recepción y se reinicia el bucle.

El bucle se repetirá para cada una de las muestras de la imagen. Si durante el proceso no se produce ningún error, la función devolverá la constante XST_SUCCESS.

Código 13: Función de comunicaciones con el bloque IP

```
162 int predictionByDma(u16 DeviceId)
163 {
164     XAxiDma_Config *CfgPtr;
165     int Status;
166     int Elements = SAMPLESTOREAD;
167     unsigned int Index;
168
169     u16 *TxBufferPtr;
170     u16 *RxBufferPtr;
171
172     TxBufferPtr = (u16 *)TX_BUFFER_BASE ;
173     RxBufferPtr = (u16 *)RX_BUFFER_BASE;
174
175     /* Initialize the XAxiDma device.
176     */
177     CfgPtr = XAxiDma_LookupConfig(DeviceId);
178     if (!CfgPtr) {
179         xil_printf("No config found for %d\r\n", DeviceId);
180         return XST_FAILURE;
181     }
182
183     Status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
184     if (Status != XST_SUCCESS) {
185         xil_printf("Initialization failed %d\r\n", Status);
186         return XST_FAILURE;
187     }
188
189     if(XAxiDma_HasSg(&AxiDma)){
190         xil_printf("Device configured as SG mode \r\n");
191         return XST_FAILURE;
192     }
193
194     /* Disable interrupts, we use polling mode
195     */
196     XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
197                        XAXIDMA_DEVICE_TO_DMA);
198     XAxiDma_IntrDisable(&AxiDma, XAXIDMA_IRQ_ALL_MASK,
199                        XAXIDMA_DMA_TO_DEVICE);
200
201
202     for(Index = 0; Index < Elements; Index ++ ) {
203
204         //Write the new data to send into the buffer
205         TxBufferPtr[0] = (u16)Samples[Index];
206         //xil_printf("Data Written %d: %d\r\n", Index, (unsigned int)TxBufferPtr[0]);
207
208         /* Flush the buffers before the DMA transfer, in case the Data Cache
209         * is enabled
210         */
211         Xil_DCacheFlushRange((UINTPTR)TxBufferPtr, MAX_PKT_LEN*2);
212         Xil_DCacheFlushRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN*2);
213
214     }
```

```

215     Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) RxBufferPtr,
216                                     MAX_PKT_LEN*2, XAXIDMA_DEVICE_TO_DMA);
217
218     if (Status != XST_SUCCESS) {
219         return XST_FAILURE;
220     }
221
222     Status = XAxiDma_SimpleTransfer(&AxiDma,(UINTPTR) TxBufferPtr,
223                                     MAX_PKT_LEN*2, XAXIDMA_DMA_TO_DEVICE);
224
225     if (Status != XST_SUCCESS) {
226         return XST_FAILURE;
227     }
228
229
230     while ((XAxiDma_Busy(&AxiDma,XAXIDMA_DEVICE_TO_DMA)) ||
231           (XAxiDma_Busy(&AxiDma,XAXIDMA_DMA_TO_DEVICE))) {
232         // Wait
233     }
234
235     Xil_DCacheInvalidateRange((UINTPTR)RxBufferPtr, MAX_PKT_LEN*2);
236
237     //Prints serial the incoming value
238     //xil_printf("Data Read: %d\r\n", (unsigned int)RxBufferPtr[0]);
239
240     //Saves the incoming value into the array
241     Results[Index] = (unsigned int)RxBufferPtr[0];
242
243     if (Status != XST_SUCCESS) {
244         return XST_FAILURE;
245     }
246
247 }
248
249 /* Test finishes successfully
250 */
251 return XST_SUCCESS;
252 }

```

5.4. Metodología de verificación

Para nuestro caso, resulta esencial comprobar una compatibilidad perfecta con la implementación original realizada dentro del grupo de investigación. El objetivo es que el bloque generado sea intercambiable con los ya realizados, además de que se pueda decodificar con el *software* ya existente.

Por tanto, ¿cómo se va a realizar la verificación de que el resultado es satisfactorio con esta premisa? En primer lugar, se van a comprobar por separado las versiones *lossless* y *lossy* implementadas. La comprobación se realizará comparando el residuo generado, bajo las mismas condiciones, por:

- El software C de referencia.
- El código Matlab implementado.
- El *set-up* físico con la tarjeta PYNQ Z1.

Las pruebas se realizarán dentro del entorno de MATLAB por el script en Código 14. Para ello, primero deberán cargarse los residuos, que están almacenados en

memoria. Una vez dentro se calcularán las SNR entre todas las salidas obtenidas. El test se dará por válido si para los tres casos obtenemos una SNR infinita, en otras palabras, que existe una concordancia perfecta entre los residuos.

Código 14: Script verificación residuos.

```
1 %%SCRIPT TEST RESIDUOS
2 % En este script se comprueban los residuos generados
3
4 %Cargamos el residuo de la imagen de referencia
5 file_name_ref = './Reference/residuals.cmp';
6
7 %Cargamos el residuo del cálculo de Matlab
8 file_name_matlab = './Matlab/residualsAviris.cmp';
9
10 %Cargamos el residuo importado de la FPGA
11 file_name_fpga = './FPGA/RESULT.BIN';
12
13 %Introducimos los parámetros de la imagen
14 D = 16;
15 size_y = 128;
16 size_x = 128;
17 size_z = 224;
18 endian = 'ieee-be';
19 format = 'uint16=>uint16';
20 pixelsOrder = 'BIP';
21
22 %Importamos los residuos
23 res_reference = import_bi_res(file_name_ref,size_x,size_y,size_z);
24 res_matlab = import_bi_res(file_name_matlab,size_x,size_y,size_z);
25
26 n_elements = size_x * size_y * size_z;
27 res_fpga = import_raw_res(file_name_fpga,n_elements);
28
29 %Calculamos la SNR de los residuos entre sí
30 snr_ref_mat = metrics_SNR(res_reference,res_matlab);
31 snr_ref_fpga = metrics_SNR(res_reference,res_fpga);
32 snr_mat_fpga = metrics_SNR(res_matlab,res_fpga);
33
34 if(snr_ref_mat == inf && snr_ref_fpga == inf && snr_mat_fpga == inf)
35     disp("Los residuos coinciden.");
36     disp("SNR = " + snr_ref_fpga);
37
38 else
39     disp("ERROR! Los residuos no coinciden. SNR = " + snr_ref_fpga);
40 end
41
```

La imagen utilizada en la etapa final de validación sobre FPGA, proviene de una imagen capturada por los instrumentos de la NASA embarcados en el proyecto AVIRIS (*Airborne Visible/Infrared Imaging Spectrometer*) [43]. Los sensores (Figura 54) se han diseñado para realizar las capturas desde aviones a una altura de 20 km. Las imágenes producidas tienen una resolución de 677 píxeles, 512 líneas y 224 bandas [44].

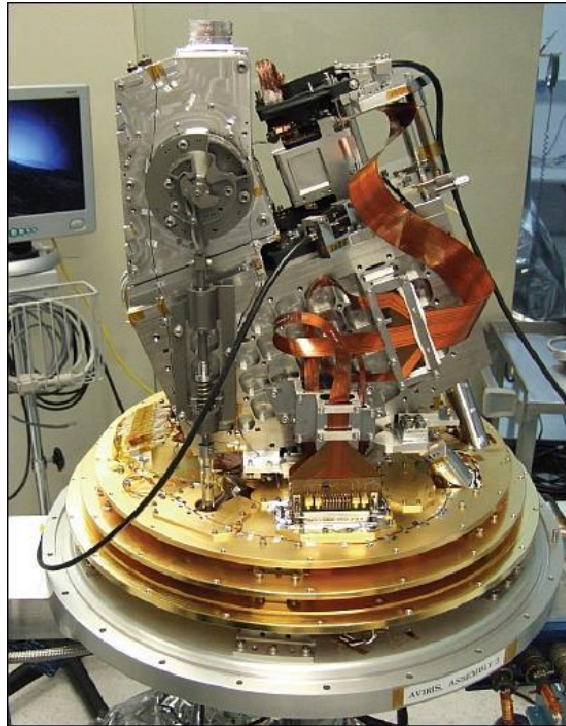


Figura 54: Imagen instrumentación AVIRIS-NG [44].

Para nuestra prueba se ha reducido la resolución de la imagen a 128 líneas y 128 columnas, manteniendo las 224 bandas. Se ha tomado la decisión debido a las especificaciones de la FPGA empleada, la cual carece de suficiente memoria dedicada para almacenar los vectores de test necesarios para mayores resoluciones de imagen. A continuación, se muestra en la Figura 55 la primera banda de la imagen ya recortada.

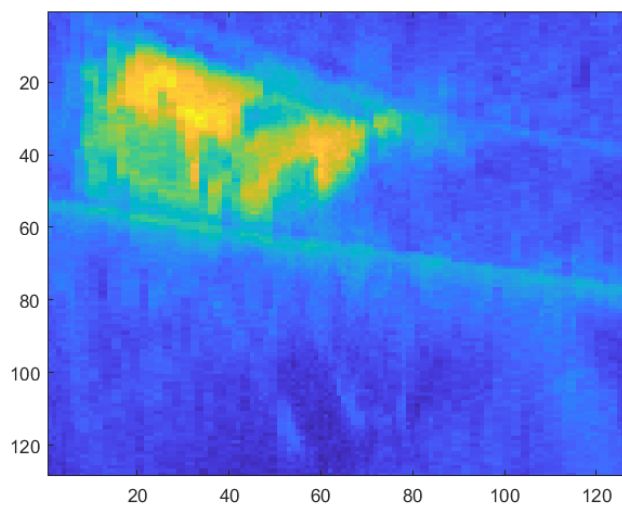


Figura 55: Imagen de referencia Aviris (se representa únicamente la banda 4).

Las pruebas se realizarán con las configuraciones recomendadas para ambos casos, *lossless* y *lossy*. Los parámetros específicos configurados se aprecian en la Tabla 8.

Tabla 8: Parámetros configuración test

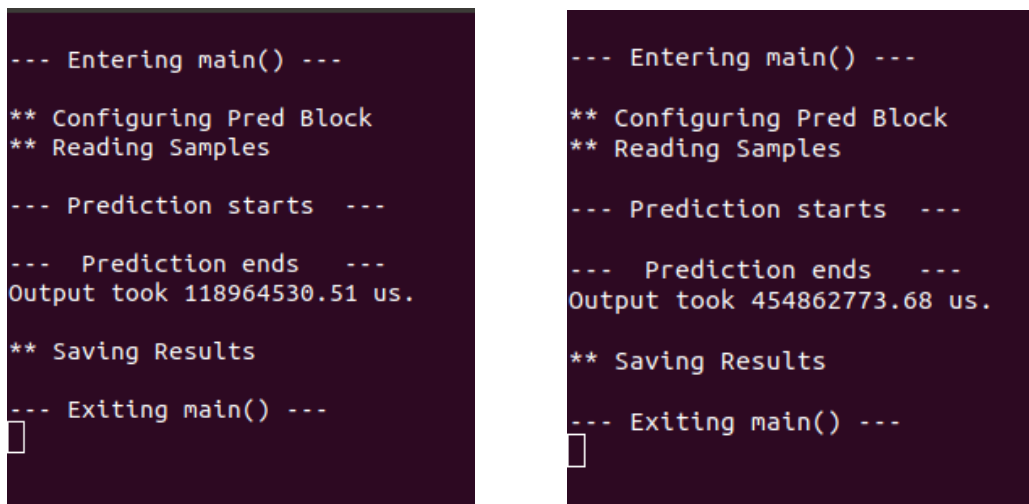
Variable	Denominación estándar	Formato	Valor
size_x	N_x	uint16	128
size_y	N_y	uint16	128
size_z	N_z	uint16	224
size_t		uint32	16384
sMax	S_{max}	uint32	65535
sMid	S_{mid}	uint32	32768
sMin	S_{min}	uint32	0
dynamic_range	D	uint16	16
Cz	C_z	uint16	3
Pz	P_z	uint16	3
weight_res	Ω	uint16	19
register_size	R	uint16	48
v_max	v_{max}	uint16	4
v_min	v_{min}	uint16	0
t_inc	t_{inc}	uint16	256
weight_exp_off	$\zeta_z *$	uint16	0
wc_max	ω_{max}	int32	2097151
wc_min	ω_{min}	int32	-2097152
m	$m_z(t)$	uint16	1
sr_res	θ	uint16	2
sr_off	ϕ_z	uint16	1
sr_dam	Ψ_z	uint16	1

5.5. Resultados obtenidos

Tras realizar las pruebas sobre la imagen de AVIRIS (128x128x224), se pueden extraer los siguientes datos:

En primer lugar, los resultados obtenidos en la implementación validan el comportamiento de los IPs frente a sus respectivos estándares. Se han comprobado los residuos generados por el método explicado previamente y en ambas pruebas hemos obtenido una concordancia completa entre los residuos generados por la referencia, el código en Matlab y la implementación en FPGA.

Si entramos a comparar el tiempo de ejecución de ambos casos, veremos que, como era de esperar, la predicción con pérdidas requiere un mayor tiempo de ejecución. En la Figura 56 podemos ver la salida generada por ambas aplicaciones; a la izquierda se muestra la salida de la prueba en *lossless* y a la derecha en *lossy*. En los datos de salida se representa el tiempo total empleado únicamente en la fase de predicción, que incluye el tiempo empleado por el bloque y por las comunicaciones a través del DMA y el PS. Para la prueba *lossless* se obtiene un tiempo de predicción de 119 s, mientras que para la versión *lossy* aumenta a 454,9 s, es decir 7 minutos y 34,8 segundos.



```
--- Entering main() ---
** Configuring Pred Block
** Reading Samples
--- Prediction starts ---
--- Prediction ends ---
Output took 118964530.51 us.
** Saving Results
--- Exiting main() ---

--- Entering main() ---
** Configuring Pred Block
** Reading Samples
--- Prediction starts ---
--- Prediction ends ---
Output took 454862773.68 us.
** Saving Results
--- Exiting main() ---
```

Figura 56: Captura salidas predictor en vista combinada.

La variación del tiempo de ejecución se debe, en gran medida, a que el reloj no ha podido configurarse para la misma frecuencia en ambas pruebas. Además, las dependencias que introduce el predictor *lossy* con su lazo de realimentación limitan significativamente la ruta crítica. Entraremos a analizarlo a continuación, junto con el resto de los parámetros de utilización.

5.6. Utilización de recursos

Los recursos usados han variado entre la implementación con pérdidas y sin pérdidas. Esto está dentro de lo esperable, ya que en el estándar *lossy* se incluyen etapas adicionales dentro la predicción. A continuación, pasaremos a analizar cada caso y luego se compararán las diferencias. Para ello, se tratarán los siguientes apartados:

- Uso de recursos físicos: LookUp Tables (LUT), Block RAMs (BRAM), Digital Signal Processors (DSP), etc.
- Consumo de Potencia
- Velocidad de Reloj y latencia

5.6.1. Predictor Lossless

El predictor *lossless* ha sido el primero en implementarse y el que marcó la viabilidad de la implementación. En la Figura 57 se aprecia que, como había sido comentado, el factor limitante para la implementación ha sido el uso de BRAMs. Con la imagen de referencias AVIRIS, vemos que se ha hecho uso de un 57% de la memoria presente en la lógica configurable de la Zynq. Sobre el resto de los parámetros, destacar que el uso de LUTs no ha sido muy elevado, un 24%, mientras que en la primera versión implementada superaba el 100% para una imagen de esta resolución.

En esta primera versión ocurría que no se estaba utilizando los BRAMs como memoria, sino que HDL Coder por defecto asignaba los registros. Dentro del código Matlab un bloque RAM se encuentra modelado como un objeto del sistema. Para actuar sobre éste, debe excitarse mediante la función *step*. Esta función pasa al objeto la dirección de escritura, los datos para escribir y la dirección de lectura; y este objeto devuelve los datos leídos de la posición de memoria indicada. La limitación es que la función *step* solo puede utilizarse bajo las siguientes condiciones:

- Sólo puede llamar una única vez a un System Object.
- Una función *step* no puede llamarse dentro de una función condicional o bucle.
- Una función *step* no puede llamarse dentro de una declaración condicional que contenga una operación de indexación matricial.

Estas limitaciones suponen que solo se puede realizar un único acceso a un bloque de memoria dentro de un script y a su vez, al no poder utilizarse el acceso con condicionales, reduce las posibilidades de uso a unos casos muy concretos.

Para solucionar esta situación se pasó a realizar un proceso de optimización global, el cual fue descrito dentro de la descripción de bloques del apartado 4.3.1 (Diseño en Simulink del predictor *lossless*). Para el bloque *local_sum*, que inicialmente almacenaba todas las muestras recibidas de la imagen, se redujo a un 0,01% sus necesidades de almacenamiento.

En el bloque de actualización del vector de pesos se realizó un proceso de optimización equivalente, que redujo al 1,8% la cantidad de datos en memoria. Pero aún la optimización no era suficiente y hacía falta hacer uso correcto de los BRAMs frente a los registros. Se optó por hacer un rediseño al completo del bloque,

modelando la BRAM como un bloque externo al código principal (Figura 44: Implementación Simulink bloque actualización de diferencias.), al que se accede ciclo a ciclo según la operación realizada.

Tras todo el proceso de optimización, los recursos usados, mostrados en la Figura 57: Recursos consumidos implementación lossless 50, se consideran satisfactorios, siendo el principal factor limitante la cantidad de BRAMs presentes en el SOC utilizado. Este consumo de memoria también es dependiente del tamaño de la imagen empleada (concretamente, del número de columnas y de bandas).

Resource	Utilization	Available	Utilization %
LUT	12661	53200	23.80
LUTRAM	255	17400	1.47
FF	11282	106400	10.60
BRAM	79.50	140	56.79
DSP	51	220	23.18

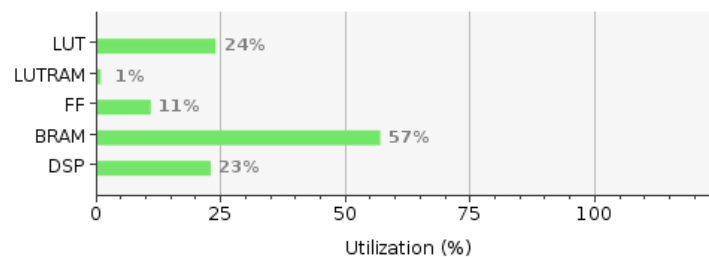


Figura 57: Recursos consumidos implementación lossless.

Si nos centramos en los resultados en cuanto al consumo de potencia (Figura 58: Análisis consumo potencia implementación lossless), se aprecia que el consumo realizado por el bloque predictor es mínimo en comparación con el total. El 95% lo consume la CPU con sus periféricos, mientras que la suma de los demás elementos es inferior al 5%. Por tanto, el consumo del bloque es inferior a 22 mW, que es el consumo dinámico total de los elementos de la lógica programable.

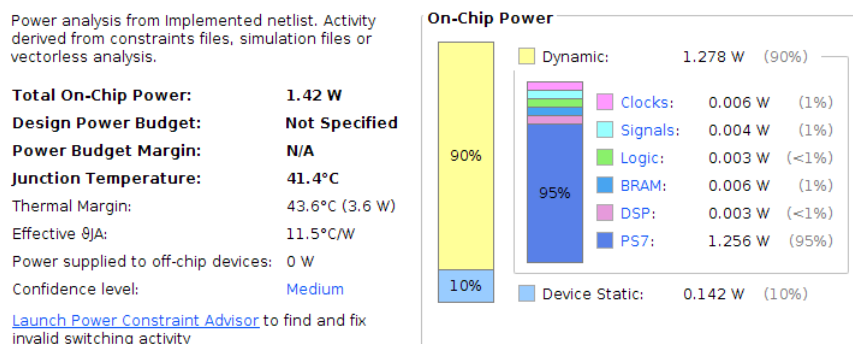


Figura 58: Análisis consumo potencia implementación lossless.

Por último, trataremos la temporización del bloque. La velocidad de reloj de los periféricos se ha establecido a 10 Mhz (Figura 59: Análisis temporal lossless.). Se ha escogido este valor debido a que es valor máximo que podemos configurar el PLL (*Phase-locked loop*) del reloj antes de violar las restricciones temporales del diseño.

El WNS (*Worse Negative Slack*), de 5,376 ns, nos indica cómo de cerca estamos del límite máximo de la frecuencia del diseño; este valor tiene un resultado ideal de 0 ns si trabajamos a la máxima frecuencia posible del sistema, obteniendo valores negativos en caso de violar la temporización de las rutas críticas del diseño.

Tras analizar el valor del WNS en Vivado, se encuentra que está limitado por la ruta crítica de nuestro diseño. Esta se encuentra dentro del bloque predictor IP, cuya implementación proviene directamente de la síntesis realizada por HDL Coder.

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 5.376 ns	Worst Hold Slack (WHS): 0.020 ns	Worst Pulse Width Slack (WPWS): 15.250 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 48583	Total Number of Endpoints: 48567	Total Number of Endpoints: 19584

All user specified timing constraints are met.

Figura 59: Análisis temporal lossless.

La frecuencia máxima afecta directamente al tiempo que tomará el predictor en cada muestra. Para tomar esta medida se ha hecho uso de un ILA, que se ha conectado a las líneas de entrada y salida del predictor. En la Figura 60 se ve el periodo empleado por el bloque predictor en realizar los cálculos para una muestra. En la línea azul se marca la llegada del dato al IP y en el amarillo la obtención del resultado. En total, transcurren 159 ciclos de reloj, que teniendo en cuenta que el ILA realiza la lectura a 50 MHz, supone un tiempo de predicción de 3,18 μ s.

Este dato refleja ciertas carencias en la implementación ya que, si realizamos el cálculo a partir del tiempo total empleado en la predicción, sabemos que se ha tardado 118 964 530 μ s para predecir 36 700 016 muestras. Si dividimos estos datos, obtenemos que se ha tardado una media de 32 μ s por muestra bajo la configuración actual. Por tanto, solo un 10% del tiempo se está empleando realmente en la predicción, mientras que el resto se está perdiendo en las comunicaciones, gestión del DMA y almacenamiento en memoria por el módulo PS.

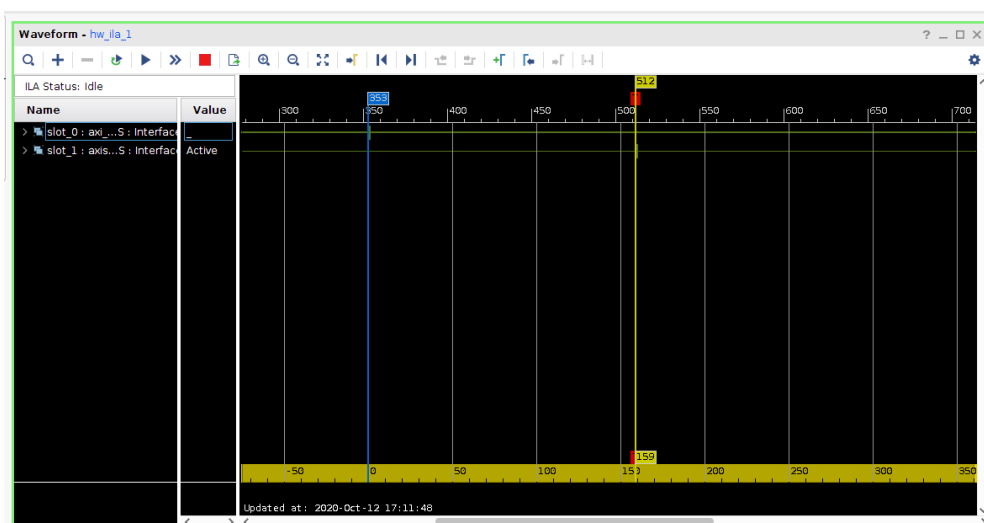


Figura 60: Captura ciclo predicción bloque IP.

5.6.2. Predictor Lossy

Ahora, que ya se han tratado los resultados de la versión sin pérdidas, se va a analizar la versión *lossy* y a su vez se realizará la comparación con los resultados previos.

En cuanto al uso de recursos, mostrado en la Figura 61, vemos que las BRAMs no tienen diferencia, mientras que el uso de LUTs y DSPs aumentan en un 66% y un 35%, respectivamente. Este cambio se produce porque hemos añadido los bloques *sample representative* y *quantizer*, que no se encontraban en la versión anterior del estándar. En estos nuevos módulos se realizan costosas operaciones aritméticas no presentes en la versión *lossless*, incluyendo tanto multiplicaciones (que suponen un mayor consumo de DSPs) como divisiones (no optimizadas en el flujo de diseño empleado, incrementando el uso de LUTs).

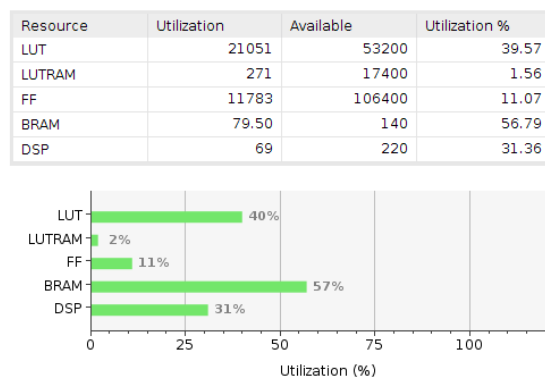


Figura 61: Recursos consumidos implementación *lossy*.

En cuanto a la potencia, podemos decir que los datos son idénticos a los de la versión sin pérdidas (Figura 62). Acapara el 95% del consumo energético el PS, mientras que el 5% restante es utilizado por la FPGA.

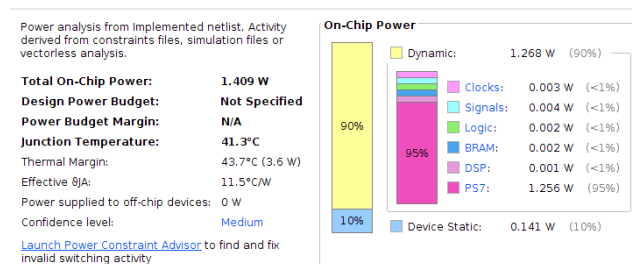


Figura 62: Análisis consumo potencia implementación *lossy*.

En cuanto a los resultados temporales, la situación es bastante distinta. Para cumplir con los requisitos de temporización, se ha reducido la frecuencia del reloj a 3 MHz (Figura 63: Análisis temporal *lossy*).

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 1.128 ns	Worst Hold Slack (WHS): 0.029 ns	Worst Pulse Width Slack (WPWS): 15.250 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 42157	Total Number of Endpoints: 42141	Total Number of Endpoints: 15196

All user specified timing constraints are met.

Figura 63: Análisis temporal lossy

Pese a este valor de frecuencia, se aprecia que el WNS es 1,128 ns, dando un escaso margen de mejora. Tras este resultado, se ha realizado un análisis del *timing* en el entorno de Vivado y nos muestra que las limitaciones provienen de la ruta crítica del bloque IP predictor. En concreto, se ha encontrado la fuente del problema en la función MODR.

En esta función no se está incluyendo ningún tipo de registros y, por tanto, se está generando una ruta demasiado larga para su apropiada implementación. El motivo proviene de la síntesis de Simulink a VHDL, pero al ser un hecho puntual para esta función, necesitará de una posterior revisión. El error probablemente se deba a problemas de configuración de HDL Coder sobre la función específica. No obstante, es conocido que al añadir más bloques en el lazo de realimentación del predictor (bloque cuantificador y *sample representative*) los tiempos de procesamiento en el CCDS 123.0-B-2 son bastante mayores que en el B1. Este es uno de los retos más importantes de la implementación de este nuevo estándar que tiene dificultades conocidas para procesar las imágenes a ritmo de una muestra por ciclo, cosa que si es posible en la versión B1 [45].

Respecto al análisis mediante ILA, no ha podido realizarse para este IP debido a la baja frecuencia del bus, pero podemos extraer alguna información a partir de los datos de ejecución. Sabemos que la predicción ha durado 454 862 773 μs . Si repetimos el cálculo previo, extraemos que el tiempo de predicción medio por muestra es de 123,9 μs . Este valor es 3,9 veces mayor que para la versión *lossless*, pero corresponde con lo esperado dentro de la reducción de frecuencia. Asumiendo que nos enfrentamos al mismo fenómeno con las comunicaciones, podemos estimar que el tiempo de predicción real del bloque IP será aproximadamente de 12,4 μs .

5.7. Throughput

Partiendo de los datos analizados previamente de temporización, podemos extraer el *throughput* del sistema implementado. En nuestro caso, el *throughput* lo vamos a medir como el flujo de datos generado por el predictor por unidad de tiempo. La fórmula es al siguiente:

$$\text{Throughput} = \frac{\text{bits muestra}}{\text{latencia muestra}} \quad (41)$$

Para el predictor sin pérdidas sabemos que tenemos una latencia de 32us, mientras que para la versión *lossy* es de 123,9 μs . A su vez, el residuo generado es de 16 bits por muestra. Por tanto, el *throughput* generado es:

$$\textit{Throughput}_{\textit{lossless}} = \frac{16}{32 * 10^{-6}} = 500 \textit{ kbps} \quad (42)$$

$$\textit{Throughput}_{\textit{lossy}} = \frac{16}{123 * 10^{-6}} = 130 \textit{ kbps} \quad (43)$$

Con este dato obtenemos el tamaño del canal necesario para poder transmitir el flujo de datos generado. Otro dato más interesante en el ámbito del procesamiento de imágenes es el número de muestras por segundo máximo que podría gestionar el bloque IP actual si excluimos los tiempos empleados en las comunicaciones, DMA y PS.

$$\frac{\textit{Sample}}{s} = \frac{1}{\textit{lantencia muestra}} \quad (44)$$

Si empleamos los datos obtenidos previamente, sabemos que para la versión *lossless* del predictor obtenemos un tiempo de predicción de $3,18 \mu s$; por tanto, seríamos capaces de obtener 314 KSamples/s. Para la versión *lossy* se ha estimado un tiempo de predicción de $12,4 \mu s$, por lo que podríamos obtener hasta 81 KSamples/s.

Esta diferencia entre ambos valores se debe a la diferencia en frecuencia entre los bloques, además de la nueva ruta crítica en la realimentación del predictor, resultado de la inclusión de los nuevos módulos. Estos datos variarán notoriamente cuando se realice una optimización μs temporal del sistema que permita aumentar las frecuencias de funcionamiento.

5.8. Comparativa con otras implementaciones

A continuación, vamos a comparar los resultados obtenidos con el flujo de trabajo utilizado frente a otras alternativas. Nos centraremos en los resultados que ofrece una descripción RTL en VHDL, pero también analizaremos otras implementaciones del CCSDS 123.0-B-1 basadas HLS.

Partiremos de la actualización que se ha realizado sobre el trabajo inicial de referencia, denominado SHyLoC 2.0 [46]. En este documento se incluye una tabla comparativa del IP generado frente a otras soluciones existentes en el estado del arte (Tabla 9), pero nos interesa la última entrada de la tabla; SHyLoC 2.0 implementado sobre un SOC de la familia ZYNQ. En concreto, la implementación se realiza sobre el SOC XC7Z035 que incluye una PL basada en la Kintex-7, la cual es superior en número de recursos disponibles a la usada en el SOC de este TFG, que es una Artix-7 [37]. Pese a esta diferencia, tienen una estructura similar y será el punto de referencia para el análisis.

Tabla 9: Uso de recursos de implementaciones en VHDL previas del CCSDS123.0-B-1 [46]

Implementation	Order	P	D	Encoder	Device	LUTs	FFs	DSPs	BRAMs	freq. (MHz)	Throughput (MSamples/s)
HyLoC, Santos et al. [24]	BSQ	3	16	Sample	Virtex-5 XQR5VFX130	2342	1535	1	0	134	11.3
Keymeulen et al. [27]	BIP	3	13	Golomb-Rice	Virtex-5 SX50T	12697	1586	3	8	40	40
Bascones et al. [28]	BIP	0-15	16	Sample	Virtex-7 XC7VX690T	-	-	-	-	50	47.6
Bascones et al. (per core) [29]	BIP	0-15	16	Sample	Virtex-7 XC7VX690T	6931	-	18	88	-	.*
Tsigkanos et al. [30]	BIP	3	16	Sample	Virtex-5 FX130T	9462	9990	6	83	213	213
Pereira et al. [31]	BIP	3	16	No	Zynq-7020	2244	630	3	0	142.9	20.4
Fjeldtvedt et al. [32]	BIP	0-15	16	Sample	Zynq-7020	3012	2528	6	84	147	147
Orlandic et al. (per core) [33]	BIP	0-15	16	Sample	Zynq-7035	3747	2887	9	33	150	150
Rodríguez et al. [34]	BSQ	3	16	Sample	Zynq-7100	51070	26830	16	256	100	67.04
SHyLoC 1.0 [14]	All	0-15	16	Sample/Block	Virtex-5 XQR5VFX130	5815	3658	10	74	113.9	113.3
This work	All	0-15	16	Sample/Block	Virtex-5 XQR5VFX130	4809	2736	10	74	138.3	138.3
This work	All	0-15	16	Sample/Block	Zynq-7035	4619	2765	8	74	151.1	151.1

*It is not possible to infer a per-core throughput from [29]. The authors provide a value of 219.4 MSamples/s for a parallel implementation with 7 cores

Ahora deberíamos compararlo con los recursos utilizados por el bloque IP que hemos generado. En la Tabla 10 se ve el desglose de recursos para cada bloque y destacado en azul se encuentran los recursos asociados al bloque IP que se ha generado en este proyecto.

Tabla 10: Uso de recursos bloque IP lossless desarrollado sobre ZYNQ-7020.

Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Slice (13300)	LUT as Logic (53200)	LUT as Memory (17400)	Block RAM Tile (140)	DSPs (220)
design_1_wrapper	12661	11282	1084	512	4401	12406	255	79.5	51
design_1_i (design_1)	12661	11282	1084	512	4401	12406	255	79.5	51
xlconcat_0 (design_1)	0	0	0	0	0	0	0	0	0
rst_ps7_0_100M (des)	17	33	0	0	11	16	1	0	0
ps7_0_axi_periph (de)	505	675	0	0	215	444	61	0	0
processing_system7_	0	0	0	0	0	0	0	0	0
predictor_lossless_ip	9955	7647	1084	512	3329	9909	46	76	51
U0 (design_1_pred)	9871	7647	1084	512	3320	9825	46	76	51
u_predictor_loss	2	2	0	0	2	2	0	0	0
u_predictor_loss	9695	7372	1084	512	3245	9677	18	76	51
u_predictor_loss	63	92	0	0	46	63	0	0	0
u_predictor_loss	56	91	0	0	23	42	14	0	0
u_predictor_loss	56	90	0	0	23	42	14	0	0
axis_data_fifo_1 (des)	44	57	0	0	20	44	0	1	0
axis_data_fifo_0 (des)	44	57	0	0	21	44	0	0.5	0
axi_mem_intercon (de)	884	1045	0	0	324	829	55	0	0
axi_dma_0 (design_1)	1214	1768	0	0	519	1122	92	2	0

Si pasamos a analizar estos datos, veremos que el uso de recursos es menor en el bloque SHyLoC. En concreto, el uso de LUTs es un 53% menor (9625 frente a 4619), un 63% menor en el caso de los registros (7372 frente a 2765) y un 84% menor de uso de DSPs (51 frente a 8). En cuanto al uso de BRAMs, es muy similar, con una diferencia del 3% (76 frente a 74).

Esta diferencia es muy importante, pero permiten extraer algunas conclusiones relevantes. Respecto al uso de memoria, este se encuentra cerca del límite de optimización que permite el estándar, por lo que obtenemos un número de BRAMs comparable a la implementación en VHDL.

Las variaciones del resto de parámetros se deben a las diferencias en la estructura del sistema implementado, a la falta de optimización y a las ineficiencias de la síntesis mediante HDL Coder. En cuanto a las diferencias de estructura, en el esquema de SHyLoC (Figura 64) se aprecia cómo obtiene los valores de las posiciones adyacentes para la *Local Sum* mediante FIFOs, mientras que el IP desarrollado en este IP utiliza vectores de almacenamiento para este propósito. El resultado funcional es equivalente, pero la estructura FIFO hace un menor uso de recursos y permite mejorar los resultados temporales del predictor.

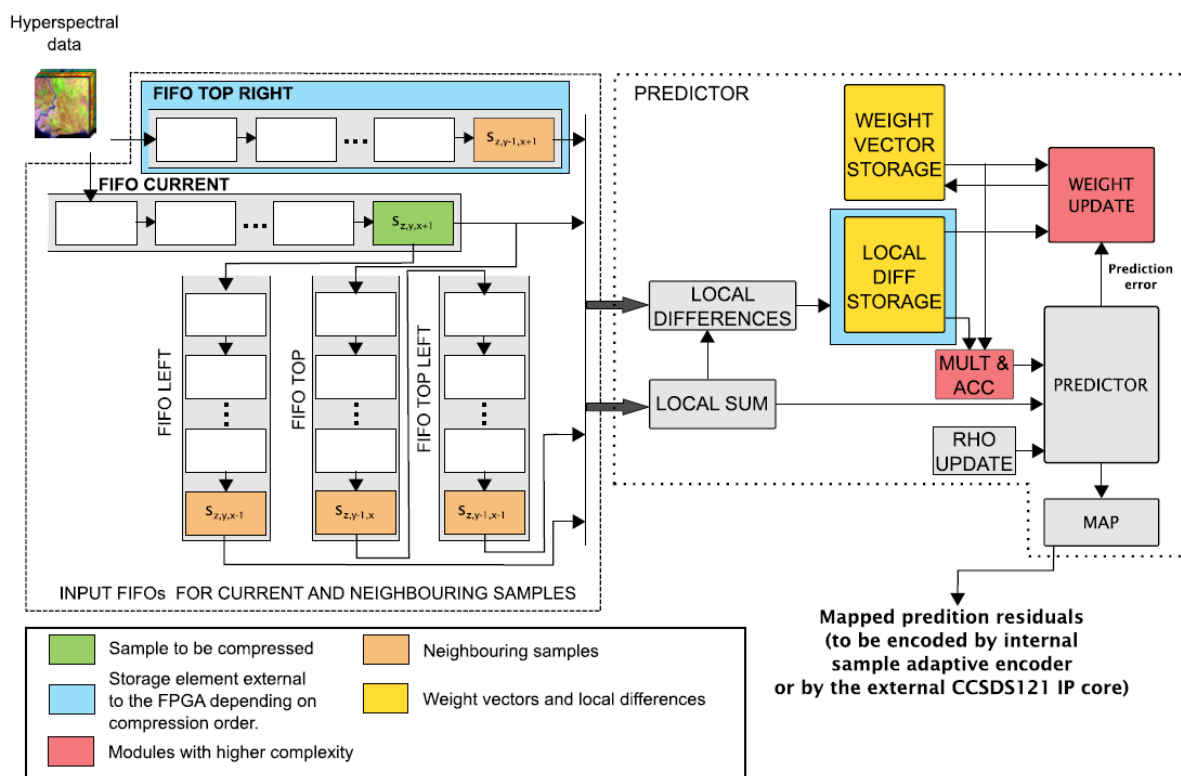


Figura 64: Esquema implementación CCSDS 123.0-B-1 SHyLoC [46]

Dentro de la variación apreciada, destaca que la implementación de SHyLoC 2.0 obtiene una frecuencia de funcionamiento de 151,1 MHz frente a los 10MHz obtenidos para nuestro bloque. Esta diferencia viene dada por el escaso trabajo de optimización temporal dentro del entorno de Simulink, siendo necesario un posterior trabajo de mejora. Cuando se trabaja con implementaciones HDL cobra una vital importancia el

concepto de *Critical Path*, que hace referencia al camino de mayor retardo en el diseño. Este es un factor limitante, ya que el mayor retardo define la frecuencia máxima para el circuito. Por tanto, una sola ruta que se encuentre mal optimizada en Simulink puede llevar al bajo rendimiento en frecuencia que se está experimentando.

Lo analizado previamente hace referencia únicamente a un flujo de trabajo basado en una descripción RTL en VHDL. A continuación, vamos a realizar una comparación frente a resultados aportados mediante HLS (*High-Level Synthesis*) en otras implementaciones previas. En concreto, los datos provienen de una comparación realizada por el DSI [47] entre SHyLoC 1.0, desarrollado en VHDL, y los resultados obtenidos de realizar la síntesis desde C mediante Vivado HLS y CatapultC [48]. Estas dos últimas herramientas realizan el proceso de sintetizar un modelo descrito en un lenguaje de alto nivel, típicamente C/C++ o SystemC, para obtener de forma automática (aunque guiada por el diseñador) una descripción RTL equivalente funcionalmente.

Los resultados mostrados en [47] se han realizado sobre un SOC de la familia ZYNQ, por lo que son comparables con los obtenidos este proyecto. El uso de recursos para cada diseño de la comparación se muestra en la Tabla 11.

Tabla 11: Comparación uso de recursos implementaciones HLS

Impl.	LUTs	FFs	RAM blocks	DSPs	Max. freq.
Vivado HLS	6144 (11.55%)	2365 (4.45%)	35.5 (25.36%)	18 (8.18%)	74.44 MHz
CatapultC	13007 (24.45%)	3313 (6.23%)	39.5 (28.21%)	11 (5.00%)	51.61 MHz
SHyLoC [8] (RTL)	7504 (14.11%)	2631 (4.95%)	70.5 (50.36%)	9 (4.09%)	70.25 MHz

Si analizamos los resultados, vemos que entre las dos herramientas de HLS existe una diferencia a favor de Vivado HLS. Esta proviene de que Vivado HLS es una herramienta específica de Xilinx para sus FPGAs, proveyendo de un mayor nivel de optimización.

Por el contrario, CatapultC es una herramienta de Mentor que genera descripciones RTL para múltiples *hardware*. Esta funcionalidad ofrece la posibilidad de trabajar con productos de distintos fabricantes. Para los propósitos de este trabajo compararemos los resultados de nuestro IP con los de CatapultC, ya que al igual que HDL Coder sus resultados son portables a varias arquitecturas y fabricantes.

Frente a los resultados del predictor *loss/less* desarrollado, presentados en la Tabla 10, observaremos algunas diferencias respecto a la comparación realizada para SHyLoC 2.0. Respecto al uso de recursos, se aprecia una mayor utilización de LUTs en CatapultC frente a la del IP desarrollado (13007 frente a 9625); mientras que apreciamos un menor uso del resto de recursos en la implementación generada por CatapultC. Las discrepancias entre el consumo de LUTs y de BRAMs entre ambas

implementaciones puede ser debida a que las herramientas empleadas no equilibran el consumo de ambos recursos de la misma forma a la hora de mapear las memorias.

La frecuencia de funcionamiento es otro de los aspectos claves. Si lo comparamos con nuestro IP, la frecuencia de funcionamiento que obtenemos continúa siendo muy inferior (51,61 MHz frente a 10 MHz). A partir de los datos de la Tabla 11 también se aprecie que CatapultC provoca una pérdida del 26% en la frecuencia máxima frente a una implementación RTL pura. Se considera que este sería el resultado que debería alcanzarse con un posterior trabajo de optimización sobre este proyecto y nos ofrece una referencia de la pérdida de rendimiento para las soluciones HLS respecto a seguir el flujo de diseño tradicional desde RTL.

Cerrando esta comparativa, se aprecia que la implementación realizada en este proyecto tiene un rendimiento muy inferior al generado por el IP SHyLoC 2.0. La comparación tampoco ha sido equitativa, ya que el bloque IP del proyecto SHyLoC se trata de un producto comercial que ha tenido un proceso de desarrollo y optimización más largo que el realizado en el proyecto presentado en este documento. Si lo comparamos con resultados previos realizados sobre HLS, también se aprecia una pérdida notable en rendimiento. Los resultados de la comparación con las implementaciones nombradas se encuentran en la Figura 65. Destaca principalmente un aumento en el uso de DSPs, por diferencias en la arquitectura de este proyecto, y de la escasa frecuencia de funcionamiento, pendiente de una futura optimización.

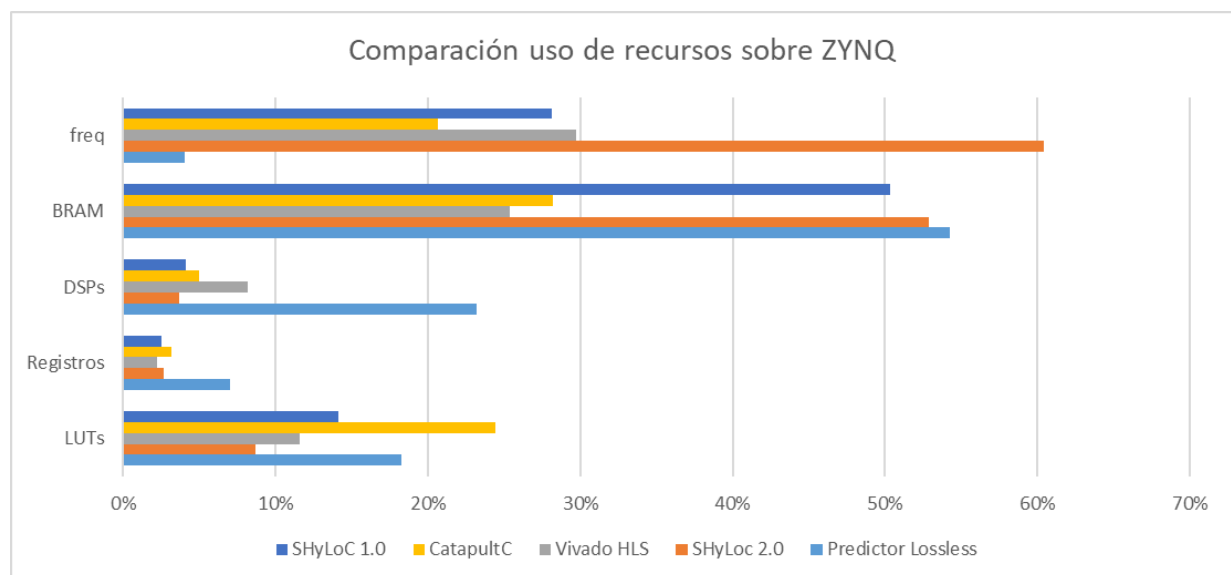


Figura 65: Gráfico comparación resultados con otras implementaciones del estándar CCSDS 123.0-B-1.

Finalmente, comentar que el desarrollo del bloque IP que se ha generado para este proyecto se ha enfocado únicamente en su correcto funcionamiento y en realizar una primera aproximación a la generación de modelos para FPGA siguiendo el flujo de diseño de Matlab-Simulink, que no había sido previamente abordado por parte del grupo de investigación. Por tanto, el bloque IP desarrollado debe ser optimizado para

ser comparable con el usado como referente. Con un posterior trabajo de optimización deberíamos acercarnos a los resultados que se presentan para la herramienta de CatapultC, que introduce una penalización del 26% en la frecuencia máxima.

6. Conclusiones y líneas futuras

Este proyecto se inició con el objetivo de implementar las novedades que el estándar CCSDS 123.0-B-2 ofrecía frente a la versión previa (CCSDS 123.0-B-1). En esta nueva versión se incluyen dos nuevos bloques, *quantizer* y *sample representative*, que son los que permiten introducir pérdidas en el proceso de compresión. Para realizar una implementación en VHDL del predictor del estándar, se optó por un flujo de trabajo basado en la suite de Matlab, y en concreto Simulink y HDL-Coder.

Este flujo de trabajo, que no había sido probado antes dentro del grupo, está despertando mucho interés por empresas del sector y aporta muchas ventajas y originalidad al proyecto:

1. Gracias a este flujo de diseño se ha obtenido un mayor dominio del estándar de compresión, ya que Matlab permite profundizar en los aspectos matemáticos y algorítmicos del estándar.
2. El bloque predictor, que ha sido desarrollado tanto en su versión *lossless* como *lossy-near lossless*, puede ser migrado a otros entornos y lenguajes como: GPUs (*Graphics processing unit*), WEB Apps, sistemas embebidos o aplicaciones de escritorio. Matlab ofrece múltiples paquetes para la síntesis de sus diseños sobre diversos equipos y dispositivos.
3. El código generado por el HDL-Coder no está sujeto a una FPGA concreta ni a una plataforma concreta. Transferirlo a otras implementaciones, por ejemplo, en FPGAs tolerantes a la radiación, es inmediato.
4. Se ha obtenido un *set-up* de pruebas completo, lo que ha permitido verificar tanto el predictor del estándar sin pérdidas como el predictor del estándar con pérdidas.
5. Se ha validado que tenemos una plena compatibilidad con el *software* de referencia en términos de comportamiento algorítmico y se ha comprobado que el bloque generado es implementable satisfactoriamente sobre FPGA.

Si volvemos a los objetivos con los que se inició este proyecto, se aprecia que se han cumplido y superado todos y cada uno de ellos:

- Estudio en profundidad de los algoritmos CCSDS 123.0-B-1 y CCSDS 123.0-B-2 para la compresión con pérdidas de imágenes hiperespectrales.
- Implementación del estándar CCSDS 123.0-B-1 para compresión *lossless* en VHDL.
- Implementación de las novedades aportadas por el estándar CCSDS 123.0-B-2 para compresión *near-lossless* en VHDL.
- Verificación de las implementaciones realizadas mediante simulación.
- Validación completa del sistema sobre FPGA

Finalmente, como se aprecia a lo largo de todo el documento, la elección del flujo de trabajo ha definido cómo se ha realizado todo el desarrollo. Con la alternativa seleccionada nos hemos beneficiado de un flujo flexible, que ha permitido acelerar todo el proceso de desarrollo, frente a realizar la implementación inicial en C y posteriormente dar el salto a VHDL. Tras haberla aplicado, se considera que la estrategia utilizadas es interesante cuando se inicia por primera vez la implementación del algoritmo, para lo que primará inicialmente la funcionalidad y tiempos de desarrollo. Sin embargo, hay que considerar que también conlleva ciertos desafíos a medida que la complejidad del diseño aumenta, limitando en gran medida sus prestaciones sin son comparadas con implementaciones realizadas directamente en RTL.

Volviendo a nuestros resultados, el bloque IP que se ha generado es funcional pero aún se encuentra lejos de ser un producto comercial y optimizado, como el que se generó durante el proyecto SHyLoC [15]. Aun así, se considera un buen punto de partida que, aplicando las respectivas optimizaciones, se puede obtener una solución capaz y competitiva, en términos temporales y de área.

Como alternativa, el código que se ha sintetizado en VHDL es plenamente funcional y legible. A partir de la descripción sintetizada en HDL, se puede realizar el salto del sistema a las FPGAs de la familia NanoXplore.

6.1. Trabajos futuros

En esta sección se comentan las posibles ampliaciones de este proyecto, para poder ofrecer el bloque IP como una solución cerrada, a un nivel equivalente al del bloque visto en el proyecto de SHyLoC. Se considera que hay tres líneas de trabajo esenciales que deberían de realizarse:

En primer lugar, debería implementarse la configuración en *runtime* por el bus AXI4-Lite. Hasta el momento, el bloque se configura con el fichero de inicialización en Simulink, y todos los parámetros son integrados en el código como constantes. Ciertos parámetros de comprensión, ajuste de la actualización de pesos, *sample representative*, etc. deberían ser configurables. Aunque hay parámetros como el tamaño de la imagen y bandas usadas en la predicción que siempre serían constantes, ya que alteran el uso de recursos sobre la FPGA.

Además, es esencial una labor de optimización. El factor más crítico es la optimización temporal. Es necesario analizar la ruta crítica para incluir retardos u otras estrategias. También hay zonas del código que son paralelizables y mejorarían a su vez los tiempos de predicción. Como última etapa de optimización, también se plantea retocar el código VHDL generado para ciertos bloques críticos. Con esta última etapa se podrían alcanzar niveles de optimización cercanos a una implementación pura sobre VHDL.

Finalmente, el estándar recoge distintas opciones del comportamiento de algunos bloques. Estas no son esenciales para el funcionamiento del predictor, pero en determinadas circunstancias pueden aportar un incremento en las ratios de compresión. Las siguientes funcionalidades no se encuentran recogidas en este trabajo:

- Cómputo de las *directional local differences*.
- Empleo de errores relativos en lugar de absolutos.
- Inicialización personalizada de los pesos.
- Introducción de múltiples algoritmos de *local sum*, por ejemplo: *narrow column-oriented*.

7. Presupuesto

En este capítulo se expone el presupuesto asociado a la realización de este Trabajo Fin de Grado. Para su desarrollo han sido necesarios diversos recursos de tipo humano y material.

7.1. Recursos humanos

En este apartado se calcula el coste que supone el desarrollo del proyecto por parte de un graduado en ingeniería de las telecomunicaciones. La retribución se calcula a partir de la tabla de clasificación y retribución del personal contratado con cargo a proyectos, programas, convenios y contratos de la ULPGC, definida en BOULPGC del 3 de junio de 2019 [49]. El contrato sería para un técnico titulado de grado, grupo TCP4, y una dedicación semanal de 20 horas. A partir de los datos planteados, se obtiene un coste mensual de 711,90 €.

El trabajo presentado ha llevado aproximadamente 260 horas de desarrollo, lo que si lo dividimos por una dedicación de 20 horas semanales, supone un total de 13 semanas. Pasado a meses supone un total de 3 meses y 1 semana. Para la generación de esta memoria se ha empleado otras 40 horas de trabajo, incluyendo redacción y figuras. Por tanto, asumiremos que el trabajo total en recursos humanos equivale a un contrato de 4 meses. En la Tabla 12 se encuentra el desglose del trabajo dedicado por tareas y el precio total del desarrollo.

Tabla 12: Costes en recursos humanos del proyecto

Concepto	Tiempo empleado	Meses	Coste
Documentación y estudio	40 horas	0,50	355,95 €
Desarrollo sobre Matlab	60 horas	0,75	533,93 €
Desarrollo sobre Simulink	60 horas	1,00	711,90 €
Implementación Vivado	60 horas	0,75	533,93 €
Validación	40 horas	0,50	355,95 €
Informe	40 horas	0,50	355,95 €
Coste Total	300 horas	4	2847,61 €

El coste final del trabajo tarifado por el tiempo empleado en el desarrollo es de DOS MIL OCHOCIENTOS CUARENTA Y SIETA EUROS SESENTA Y UN CÉNTIMOS

7.2. Recursos hardware

En la Tabla 13 se enumeran los recursos *hardware* empleados en el proyecto. Dentro de estos se incluye su precio inicial de compra y a partir de este se calcula su coste anual basándonos en una amortización a tres años. El coste estimado parte de dividir el coste de amortización entre los meses de un año y multiplicarlo por los meses empleados para el proyecto.

Tabla 13: Coste recursos hardware

Recurso	Coste total adquisición	Tiempo de empleo	Coste anual	Coste estimado
Pynq-Z1	232,19 €	1,5 meses	77,40 €	9,68 €
Workstation de diseño y programación	1400,00 €	4 meses	466,67 €	155,56 €
Laptop de documentación y redacción	640,00 €	4 meses	213,33 €	71,11 €
Cables y equipos auxiliares	40,00 €	1,5 meses	13,33 €	1,67 €
Coste Total	2312,19 €			238,02 €

El coste total por el hardware empleado en el desarrollo es de DOSCIENTOS TREINTA Y OCHO EUROS DOS CÉNTIMOS.

7.3. Recursos software

En la Tabla 14 se encuentran los recursos *software* que han sido utilizados durante el desarrollo del TFG. Al estar bajo licencia universitaria y de estudiante, los costes se reducen notoriamente.

Tabla 14: Coste recurso software

Recurso	Tipo de Licencia	Coste Licencia	Mantenimiento Anual
Xilinx Vivado Suite	Universitaria	Donación	214,00 €
Matlab Suite	Estudiante	Donación	-
Office 365	Personal	-	69,00 €
Entorno Eclipse programación C	Pública	-	-
Coste Total			283,00 €

El coste del software en el desarrollo es de DOSCIENTOS OCHENTA Y TRES EUROS.

7.4. Material fungible

El costo en material fungible ha sido residual. Incluyendo los costes de papel e impresión, el total es de aproximadamente QUINCE EUROS.

7.5. Presupuesto total del proyecto

Finalmente, en la Tabla 15: Coste total del proyecto, se recoge el coste total del proyecto subdividido en los conceptos declarados previamente.

Tabla 15: Coste total del proyecto

Concepto	Coste
Recursos hardware	238,02 €
Recursos software	283,00 €
Recursos humanos	2847,61 €
Fungible C	15,00 €
Coste Total	3383,63 €

Por tanto, el coste total del proyecto asciende a TRES MIL TRESCIENTOS OCHENTA Y TRES EUROS SESENTA Y TRES CÉNTIMOS.

BIBLIOGRAFÍA

- [1] “ESA - Going hyperspectral.” [Online]. Available: https://www.esa.int/Applications/Observing_the_Earth/Proba-1/Going_hyperspectral. [Accessed: 22-Jan-2020].
- [2] “Sobre Copernicus | Copernicus.” [Online]. Available: <https://www.copernicus.eu/es/sobre-copernicus>. [Accessed: 12-Oct-2020].
- [3] “Missions - Sentinel Online.” [Online]. Available: <https://sentinel.esa.int/web/sentinel/missions>. [Accessed: 04-Feb-2020].
- [4] “ESA - Copernicus High Priority Candidates.” [Online]. Available: https://www.esa.int/Applications/Observing_the_Earth/Copernicus/Copernicus_High_Priority_Candidates. [Accessed: 08-Oct-2020].
- [5] “ESA UNCLASSIFIED-For Official Use Copernicus Hyperspectral Imaging Mission for the Environment-Mission Requirements Document.”
- [6] C. Leonard, “Challenges for Electronic Circuits in Space Applications.”
- [7] CCSDS, “Recommendation for Space Data System Standards LOW-COMPLEXITY LOSSLESS AND NEAR-LOSSLESS MULTISPECTRAL AND HYPER SPECTRAL IMAGE COMPRESSION RECOMMENDED STANDARD,” 2019.
- [8] “Lossy Compression - an overview | ScienceDirect Topics.” [Online]. Available: <https://www.sciencedirect.com/topics/computer-science/lossy-compression>. [Accessed: 04-Feb-2020].
- [9] “CCSDS.org - The Consultative Committee for Space Data Systems (CCSDS).” [Online]. Available: <https://public.ccsds.org/default.aspx>. [Accessed: 30-Jan-2020].
- [10] CCSDS, “Recommendation for Space Data System Standards LOSSLESS MULTISPECTRAL & HYPER SPECTRAL IMAGE COMPRESSION RECOMMENDED STANDARD CCSDS HISTORICAL DOCUMENT CCSDS HISTORICAL DOCUMENT,” 2012.
- [11] “What is an FPGA? Field Programmable Gate Array.” [Online]. Available: <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>. [Accessed: 30-Jan-2020].
- [12] “ESA - SHyLoC IP Core.” [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Microelectronics/SHyLoC_IP_Core. [Accessed: 22-Jan-2020].
- [13] Xilinx and Inc, “General Description Radiation-Hardened, Space-Grade Virtex-5QV Family Data Sheet: Overview Product Specification Table 1: Virtex-5QV FPGA Family Members Device Configurable Logic Blocks (CLBs) DSP48E Slices (2) Block RAM Blocks CMTs (4) Endpoint Blocks for PCI Express Ethernet MACs (5) Max RocketIO GTX Transceivers,” 2018.

-
- [14] “RTG4 Radiation-Tolerant FPGAs | Microsemi.” [Online]. Available: <https://www.microsemi.com/product-directory/rad-tolerant-fpgas/3576-rtg4>. [Accessed: 12-Oct-2020].
- [15] L. Santos, “SHyLoC Product Datasheet,” *Univ. Las Palmas Gran Canar. Inst. Appl. Microelectron. Spain*, no. October, 2017.
- [16] “ESA - High Density European Rad-Hard SRAM-Based FPGA- First Validated Prototypes – BRAVE.” [Online]. Available: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Shaping_the_Future/High_Density_European_Rad-Hard_SRAM-Based_FPGA-First_Validated_Prototypes_BRAVE. [Accessed: 01-Feb-2020].
- [17] “What is MATLAB? - MATLAB & Simulink.” [Online]. Available: <https://es.mathworks.com/discovery/what-is-matlab.html>. [Accessed: 12-Oct-2020].
- [18] “CCSDS.org - About CCSDS.” [Online]. Available: <https://public.ccsds.org/about/default.aspx>. [Accessed: 27-Sep-2020].
- [19] “CCSDS.org - Publications.” [Online]. Available: <https://public.ccsds.org/Publications/default.aspx>. [Accessed: 29-Sep-2020].
- [20] “CCSDS.org - Standards Development Process.” [Online]. Available: <https://public.ccsds.org/Publications/StandardsDevProcess.aspx>. [Accessed: 29-Sep-2020].
- [21] B. Karlik, *COMPARISON OF IMAGE COMPRESSION TECHNIQUES*. 2015.
- [22] R. Kaur, “A Review of Image Compression Techniques,” *Int. J. Comput. Appl.*, vol. 142, pp. 8–11, May 2016.
- [23] “The JPEG 2000 standard (ISO/IEC 15444-1).” [Online]. Available: <https://w3.ual.es/~vruiz/Docencia/Apuntes/Coding/Image/04-JPEG2000/index.html#x1-70006>. [Accessed: 10-Oct-2020].
- [24] F. Dufaux, G. J. Sullivan, and T. Ebrahimi, “The JPEG XR image coding standard [Standards in a Nutshell],” *IEEE Signal Process. Mag.*, vol. 26, no. 6, pp. 195–204, 2009.
- [25] B. Book, “Recommendation for Space Data System Standards IMAGE DATA COMPRESSION RECOMMENDED STANDARD,” 2017.
- [26] “Recommendation for Space Data System Standards BLUE BOOK SPECTRAL PREPROCESSING TRANSFORM FOR MULTISPECTRAL AND HYPERSPECTRAL IMAGE COMPRESSION RECOMMENDED STANDARD,” 2017.
- [27] “What is Wavelet and How We Use It for Data Science | by Muhammad Ryan | Towards Data Science.” [Online]. Available: <https://towardsdatascience.com/what-is-wavelet-and-how-we-use-it-for-data-science-d19427699cef>. [Accessed: 10-Oct-2020].

- [28] “everything you always wanted to know JPEG 2000 about.”
- [29] S.- Medouakh and Z.-E. Baarir, “Entropy Encoding EBCOT (Embedded Block Coding with Optimized Truncation) In JPEG2000.”
- [30] K. Hattori, H. Tsutsui, H. Ochi, and Y. Nakamura, “An architecture of photo core transform in HD photo coding system for embedded systems of various bandwidths,” in *APCCAS 2008 - 2008 IEEE Asia Pacific Conference on Circuits and Systems*, 2008, pp. 1592–1595.
- [31] “HDL Coder - MATLAB & Simulink.” [Online]. Available: <https://es.mathworks.com/products/hdl-coder.html>. [Accessed: 29-Sep-2020].
- [32] A. R. M. Limited, “DUI0534B_amba_4_axi4_protocol_assertions_ug,” 2012.
- [33] “What is MATLAB? - MATLAB & Simulink.” [Online]. Available: <https://es.mathworks.com/discovery/what-is-matlab.html>. [Accessed: 27-Sep-2020].
- [34] “MATLAB - Simulink - Tutorialspoint.” [Online]. Available: https://www.tutorialspoint.com/matlab/matlab_simulink.htm. [Accessed: 27-Sep-2020].
- [35] “Include MATLAB code in models that generate embeddable C code - Simulink - MathWorks España.” [Online]. Available: <https://es.mathworks.com/help/simulink/slref/matlabfunction.html>. [Accessed: 13-Oct-2020].
- [36] “Functions Supported for HDL Code Generation - MATLAB & Simulink - MathWorks España.” [Online]. Available: <https://es.mathworks.com/help/hdlcoder/ug/functions-supported-for-hdl-code-generation-alphabetical-list.html>. [Accessed: 27-Sep-2020].
- [37] Xilinx and Inc, “Zynq-7000 SoC Family Product Selection Guide,” 2014.
- [38] “Vitis Unified Software Platform Overview.” [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/vitisoverview.html. [Accessed: 13-Oct-2020].
- [39] CCSDS, “Lossless Multispectral and Hyperspectral Image Compression,” *Ccsds 120.2-G-1*, no. December, 2015.
- [40] “HDL Code Generation from hdl.RAM System Object - MATLAB & Simulink - MathWorks España.” [Online]. Available: <https://es.mathworks.com/help/hdlcoder/ug/hdl-code-generation-from-hdl-ram-system-object.html>. [Accessed: 29-Sep-2020].
- [41] Xilinx and Inc, “Zynq-7000 SoC First Generation Architecture,” 2012.
- [42] “Board Support Packages.” [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/SDK_doc/concepts/sdk_c_bsp.htm. [Accessed: 12-Oct-2020].

-
- [43] “AVIRIS - Airborne Visible / Infrared Imaging Spectrometer.” [Online]. Available: <https://aviris.jpl.nasa.gov/>. [Accessed: 29-Sep-2020].
- [44] “AVIRIS-NG - eoPortal Directory - Airborne Sensors.” [Online]. Available: <https://earth.esa.int/web/eoportal/airborne-sensors/aviris-ng>. [Accessed: 29-Sep-2020].
- [45] L. Santos, A. Gómez, and R. Sarmiento, “Implementation of CCSDS Standards for Lossless Multispectral and Hyperspectral Satellite Image Compression,” *IEEE Trans. Aerosp. Electron. Syst.*, vol. 56, no. 2, pp. 1120–1138, 2020.
- [46] Y. Barrios, A. J. Sánchez, L. Santos, and R. Sarmiento, “SHyLoC 2.0: A Versatile Hardware Solution for On-Board Data and Hyperspectral Image Compression on Future Space Missions,” *IEEE Access*, vol. 8, pp. 54269–54287, 2020.
- [47] Y. Barrios, A. Sanchez, L. Santos, S. López, J. F. López, and R. Sarmiento, “Hardware Implementation of the CCSDS 123.0-B-1 Lossless Multispectral and Hyperspectral Image Compression Standard by means of High Level Synthesis Tools,” in *2018 9th Workshop on Hyperspectral Image and Signal Processing: Evolution in Remote Sensing (WHISPERS)*, 2018, pp. 1–5.
- [48] “Catapult C/C++/SystemC HLS - Mentor Graphics.” [Online]. Available: <https://www.mentor.com/hls-ip/catapult-high-level-synthesis/c-systemc-hls>. [Accessed: 22-Nov-2020].
- [49] ULPGC, “Boletín Oficial de la Universidad de Las Palmas de Gran Canaria 3 de junio de 2019,” Jun. 2019.