ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo y validación de un codificador entrópico de un compresor de vídeo monocromo usando el estándar H.264 para la misión EROSS-IOD

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas de Telecomunicación

Autor/a: David Nielsen Ojeda

Tutores/as: Dr. Roberto Sarmiento Rodríguez

Dr. Felipe Machado Sánchez

Fecha: enero de 2026

Resumen

El proyecto EROSS-IOD es un proyecto financiado por la Unión Europea dentro del programa marco Horizon Europe, que tiene el objetivo de demostrar en órbita las capacidades de mantenimiento robótico de satélites. En el contexto de este proyecto, se pretende implementar una versión hardware del formato de compresión de vídeo del estándar H.264.

En específico este Trabajo de Fin de Grado se centra en la etapa CABAC del estándar. Esta etapa optimiza la compresión de datos mediante el uso de la codificación aritmética binaria adaptativa basada en el contexto. Este proceso se divide en tres etapas fundamentales: la binarización, el modelado de contexto y la codificación aritmética binaria.

En concreto el proyecto parte de una implementación desarrollada anteriormente y tiene como objetivo aumentar las capacidades de esta. Para ello, se ha extendido su funcionalidad de modo que, además de procesar vídeo con predicción intra 4, sea capaz de procesar también vídeo con predicción intra 16. El diseño final ha sido descrito en el lenguaje de descripción hardware *Very high speed integrated circuit Hardware Description Language* (VHDL), y se ha desarrollado usando el software Vivado. Asimismo, el diseño ha sido implementado en la FPGA Nexys 4 DDR.

Abstract

The EROSS-IOD project is a project funded by the European Union under the Horizon Europe framework programme, with the objective of demonstrating in-orbit robotic satellite servicing capabilities. Within the context of this project, the implementation of a hardware version of the H.264 video compression standard is pursued.

Specifically, this Bachelor's Thesis focuses on the CABAC stage of the standard. This stage optimizes data compression through the use of context-adaptive binary arithmetic coding. This process is divided into three fundamental stages: binarization, context modeling, and binary arithmetic coding.

In particular, the project builds upon a previously developed implementation and aims to extend its capabilities. To this end, its functionality has been expanded so that, in addition to processing video with intra 4 prediction, it is also capable of processing video with intra 16 prediction. The final design has been described using the hardware description language *Very High Speed Integrated Circuit Hardware Description Language* (VHDL) and has been developed using the Vivado software. Furthermore, the design has been implemented on the Nexys 4 DDR FPGA.

INDICE DE CONTENIDO

INDICE DE FIGURAS

INDICE DE TABLAS

Listado de acrónimos

BAC *Binary Arithmetic Coder*, codificador aritmético binario.

CABAC Context-adaptive binary arithmetic coding.

CALM1 *Coeff_abs_level_minus1[i]*, Nivel de coeficiente absoluto menos 1.

CAVLC Context-Adaptive Variable Length Coding.

CBF *Coded_Block_Flag*, indica si el bloque de coeficientes 4x4 tiene algún coeficiente distinto de 0.

CBP Coded_Block_Pattern, indica si alguno de los bloques de coeficientes 4x4 que componen el macrobloque de 16x16 tiene coeficientes distintos de 0.

CSF *Coeff_Sign_Flag[i]*, Bandera que indica el símbolo del coeficiente.

EDIF *Electronic Design Interchange Format*, Formato de Intercambio para Diseños Electrónicos.

EROSS-IOD *European Robotic Orbital Support Services In-Orbit Demonstration*, Servicios Europeos de Apoyo Orbital Robótico – Demostración en Órbita.

ESA *European Space Agency*, Agencia espacial europea.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

HD *High Definition*, Alta definición.

LSCF *Last_Significant_Coeff_flag[i]*, Bandera que indica que es el último coeficiente significativo.

MBT *Macroblock_Type*, tipo de macrobloque.

MEF Máquina de Estados Finito.

PMF Prev_intra4x4_pred_Mode_Flag, indica si el bloque de 4x4 anterior se codifico con el mismo tipo de predicción (H, V, DC) o no.

QP *Quantization Parameter, Parametro de cuantización*.

QPD *Mb_QP_Delta*, delta del parámetro de cuantización del macrobloque.

REM REM_intra4x4_pred_mode, indica el tipo de predicción usado para codificar el bloque de coeficientes 4x4 que se está codificando.

RGB Red, Green, Blue.

ROM *Read-Only Memory*, memoria de solo lectura.

SCF *Significant_coeff_flag[i]*, Bandera que indica que es un coeficiente significativo.

SE *Syntax Element*, Elemento sintáctico.

TFT Trabajo de Fin de Título.

TU *Truncated Unary*, Unario Truncado.

UART *Universal Asynchronous Receiver-Transmitter*, Transmisor-Receptor Asíncrono Universal.

Capítulo 1: Introducción y objetivos del trabajo

1.1 Antecedentes

EROSS-IOD (acrónimo de European Robotic Orbital Support Services In-Orbit Demonstration) es un proyecto financiado por la Unión Europea dentro del programa marco Horizon Europe, que tiene el objetivo de demostrar en órbita las capacidades de mantenimiento robótico de satélites [1]. Su propósito es validar tecnologías clave como el acercamiento autónomo, la captura, el acoplamiento, el reabastecimiento y la sustitución de cargas útiles, utilizando brazos robóticos avanzados. Estas capacidades permitirán extender la vida útil de los satélites, reducir la generación de desechos espaciales y facilitar la evolución hacia infraestructuras espaciales modulares y sostenibles [2].

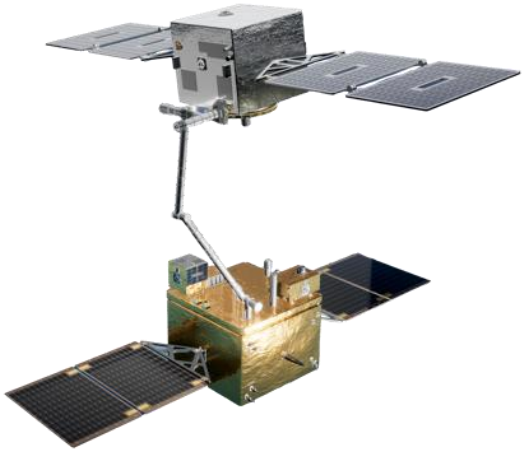

Figura 1.1: Ilustración del satélite del proyecto EROSS-IOD reparando otro satélite. Fuente [2].

El sistema completo lleva varias cámaras monocrómas, tanto en el cuerpo central del satélite como en el brazo articulado. Las imágenes y vídeo producidas por las cámaras deben ser comprimidas para reducir el flujo de datos. En concreto, dentro del marco de este proyecto, se está implementando el estándar H.264 en una FPGA (acrónimo de Field Programmable Gate Array) con el fin de reducir el tamaño de la información visual captada por las cámaras de vídeo y hacer más eficiente su transmisión. Este estándar define un método de compresión y descompresión de vídeo que disminuye la cantidad de datos necesarios para codificar una señal digital de vídeo antes de su transmisión o almacenamiento [3].

El proceso que describe el estándar está compuesto por una serie de etapas: predicción, transformación, cuantización y codificación entrópica. Esta última fase también se divide en dos posibles opciones: CAVLC (acrónimo de Context-Adaptive Variable Length Coding), opción que ofrece menor tasa de compresión, pero una mayor rapidez y sencillez de desarrollo. La otra opción es CABAC (acrónimo de Context-Adaptive Binary Arithmetic

Coding), esta ofrece mayor compresión, pero con una mayor complejidad de procedimientos [3].

El desarrollo de este TFT (acrónimo de Trabajo de Fin de Título) está enfocado en la implementación de CABAC en hardware siguiendo las especificaciones del estándar. Para ello se parte de una versión de CABAC desarrollada anteriormente por David Martín para la realización de su TFT [4]. En esta versión se implementaba la codificación de vídeo que tenía predicción intra 4x4, mientras que en la nueva propuesta se amplía la funcionalidad para permitir también la codificación con predicción intra 16×16. Asimismo, cabe recalcar que ambas versiones son monocromas, lo que quiere decir que no usan pixeles de color, sino pixeles con escalas de grises.

El estándar H.264 permite distintos modos de operación, pudiendo mantener la calidad de la imagen constante, mediante el uso de un valor fijo de un parámetro denominado QP (acrónimo de Quantization Parameter), o manteniendo un bitrate constante, variando el valor de QP con el fin de obtener una tasa de bits aproximadamente constante. En esta implementación se ha decidido mantener la calidad de la imagen constante y por lo tanto trabajar con QPs fijos.

1.2 Objetivos

El objetivo principal de este TFT es el desarrollo e implementación de un codificador entrópico CABAC y la salida del estándar H.264 en FPGA. Este objetivo general se desglosa en los siguientes objetivos específicos:

- **O1**: Comprender el funcionamiento del estándar H.264 y su codificador entrópico CABAC, incluyendo su estructura y principios de compresión.
- **O2**: Diseñar e implementar el codificador CABAC junto con la etapa de salida del estándar H.264, asegurando su integración con los bloques ya desarrollados.
- **O3**: Verificar el funcionamiento adecuado del diseño a través de simulaciones, y posteriormente validar y evaluar su rendimiento mediante su implementación en FPGA.

1.3 Metodología usada

Para el desarrollo de este TFT, se ha utilizado una metodología que divide el proyecto en las siguientes tareas:

- **Investigación del funcionamiento del estándar H.264 y CABAC:** se ha estudiado el funcionamiento del estándar y más concretamente de la etapa de codificación entrópica CABAC a través de la documentación del estándar y de su software de referencia (Joint Model).
- **Análisis de la versión inicial existente:** Se realizará un diseño incremental utilizando lo máximo posible el desarrollo existente, añadiendo o modificando las partes necesarias. Por ese motivo, es importante conocer en detalle lo realizado en [4].

- **Desarrollo de una nueva arquitectura del sistema incluyendo la nueva funcionalidad:** Partiendo de la versión existente realizar una nueva arquitectura añadiendo o modificando los bloques que sean necesarios.
- **Descripción VHDL del diseño:** se han diseñado todas bloques que conforman CABAC en intra 16x16 mediante el lenguaje de descripción de hardware VHDL. También se ha realizado la descripción del sistema global.
- **Verificación del correcto funcionamiento de los dos diseños creados de CABAC**: se ha comprobado el correcto funcionamiento de los diseños creados en las anteriores tareas mediante simulación en Vivado y Questa. Como modelo de referencia se ha usado la versión disponible de JM.
- **Implementación en FPGA de los diseños creados y verificación de su correcto funcionamiento**: se ha validado la arquitectura desarrollada mediante su implementación en FPGA y el procesamiento de archivos de vídeo.

1.4 Organización del documento

El contenido de este documento se organiza en un total de cuatro capítulos más el presupuesto. A continuación, se indican los contenidos abordados en cada uno de estos apartados:

- **Capítulo 1:** se define el objetivo del proyecto en el que se abarca este trabajo de fin de título y se aclaran los objetivos y tareas llevadas a cabo.

- **Capítulo 2:** En este capítulo se describe el funcionamiento general del estándar, incluyendo los procedimientos previos al proceso CABAC, y se detalla posteriormente el proceso de codificación CABAC, explicando las etapas de binarización, modelado de contexto y codificación aritmética.

- **Capítulo 3** se explica el diseño de la arquitectura realizada, haciendo hincapié en cada bloque uno de los bloques que la compone.

- **Capítulo 4:** se describe cómo se ha llevado a cabo la verificación y validación de la arquitectura desarrollada.

- **Capítulo 5:** se expone una conclusión y líneas futuras de trabajo.

- **Presupuesto:** se elabora el análisis económico de los costes y amortizaciones derivados de la realización del presente proyecto.

Capítulo 2: Antecedentes y estado del arte

El estándar H.264 se empieza a utilizar en 2003 como un formato que ofrecía una gran compresión de vídeo, manteniendo la calidad de la imagen original. En el momento en el que se creó, el estándar se popularizó bastante debido a la necesidad de optimizar el ancho de banda en aplicaciones como la televisión digital [3], [4], [5].

A pesar de su antigüedad, sigue siendo un estándar bastante usado a nivel mundial. Esto se debe a la creciente necesidad de compresión que ha surgido con el aumento del consumo de vídeo en directo a través de diversas plataformas en internet. Asimismo, con el paso del tiempo han surgido nuevos estándares que también son muy usados hoy en día, como el estándar H265, que ofrecen mayores tasas de compresión a costa de una mayor complejidad computacional [3], [4], [5].

El estándar H.264 es conocido también como Advanced Video Coding (AVC) o MPEG-4 Part 10. A partir de este punto, y con el objetivo de simplificar la nomenclatura, en este documento se hará referencia al estándar únicamente como H.264 [3], [4], [5].

2.1 Introducción

Un vídeo está compuesto por una secuencia de imágenes que se muestran de forma continua a gran velocidad. Estas imágenes se denominan fotogramas. Para generar la sensación de movimiento, se requieren entre 25 y 30 fotogramas por segundo, de manera que el ojo humano perciba fluidez y no identifique cada imagen por separado [3].

Asimismo, los fotogramas están formados por píxeles, que son los pequeños puntos que componen la imagen y le otorgan brillo y color. A mayor número de píxeles, mayor será la resolución de la imagen. Por ejemplo, el formato conocido como 1080p hace referencia a una resolución de 1920 × 1080 píxeles, lo que proporciona una imagen de alta definición. En comparación, un formato como 480p corresponde a una resolución de 640 × 480 píxeles y ofrece una calidad visual significativamente inferior [3].

En el sector del vídeo digital, existen varios formatos para representar el color y el brillo de una imagen. Uno de los más conocidos es el modelo RGB (acrónimo de Red, Green, Blue), que asigna a cada píxel una cierta intensidad de los tres colores primarios: rojo, verde y azul. Al combinar estas intensidades en distintas proporciones, es posible generar una amplia gama de colores [4].

No obstante, el estándar de compresión H.264 utiliza otro formato de color: YCrCb. Este formato aprovecha la mayor sensibilidad del ojo humano a las variaciones de luminancia (la intensidad de luz) frente a las de crominancia (el color). Gracias a esta característica, es posible representar las imágenes de manera más eficiente, ya que se puede reducir la resolución de los canales de crominancia sin que se perciba una pérdida significativa de calidad visual. De este modo, se disminuye el volumen de información necesaria para almacenar o transmitir el vídeo [4], [6].

2.2 Elementos del estándar H.264

En este apartado se describen algunos de los elementos fundamentales del estándar necesarios para comprender su funcionamiento.

2.2.1 Macrobloque

Al procesar un fotograma, este se divide en bloques de 256 píxeles organizados en una matriz de 16x16, denominados macrobloques. Por ejemplo, en la Figura 2.1, donde cada cuadro representa un píxel del fotograma, se muestra la extracción de un macrobloque como ejemplo ilustrativo [3].

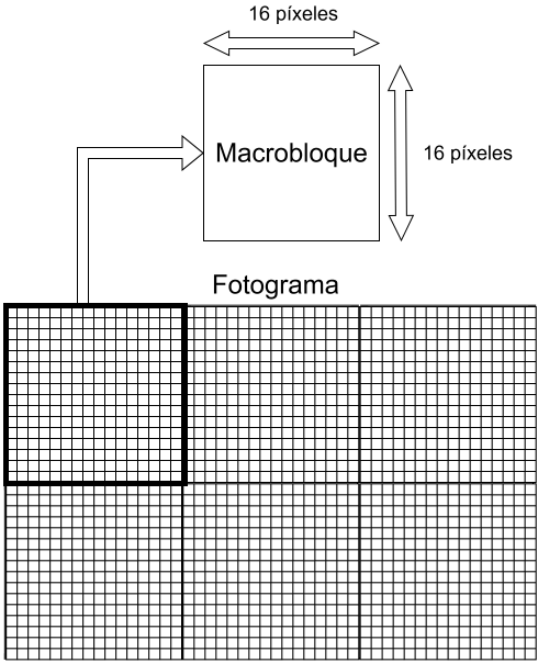

Figura 2.1: Ejemplo visual de un macrobloque

En el estándar, los macrobloques se procesan de forma secuencial, comenzando en la esquina superior izquierda del fotograma y recorriéndolo línea por línea, de izquierda a derecha, hasta llegar a la esquina inferior derecha, tal como se ilustra en la Figura 2.2.

1.°	2.°	3.°	4.°	5.°	6.°
7.°	8.°	9.°	10.°	11.°	12.°
13.°	14.°	15.°	16.°	17.°	18.°

Figura 2.2: Orden de procesamiento de los macrobloques en un fotograma

2.2.2 Slices

Asimismo, los macrobloques pueden agruparse en slices, que son conjuntos de macrobloques dentro de un mismo fotograma. Estos slices se forman con el fin de subdividir la imagen en secciones independientes, lo que permite aplicar diferentes tipos de predicción y facilita el procesamiento paralelo, además de mejorar la tolerancia a errores en la transmisión [3], [4]. En la Figura 2.3 se presenta un ejemplo visual de la división de un fotograma, en el que se ha supuesto que cada fila corresponde a un slice. En la imagen puede observarse cómo cada slice está compuesto por distintos macrobloques, y cómo, a su vez, cada macrobloque está formado por 16×16 píxeles.

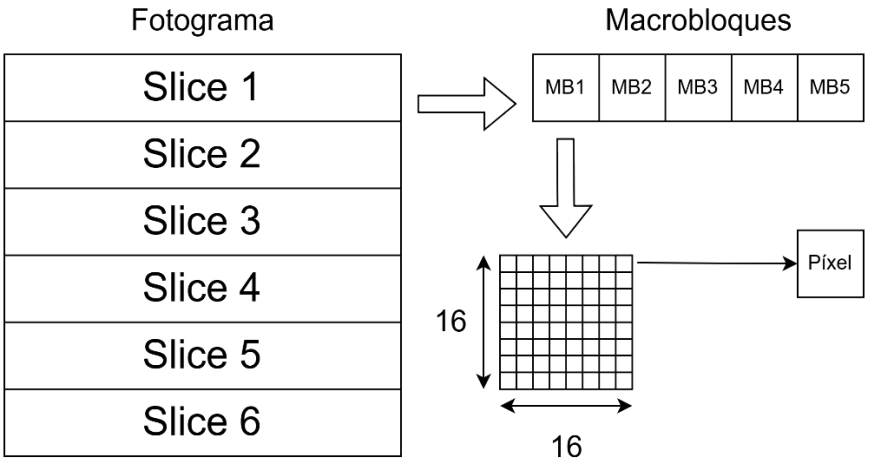

Figura 2.3: Ilustración de los elementos que forman el fotograma. Adaptada de [3].

2.3 El proceso de compresión del estándar H.264

El proceso del estándar H.264 se puede clasificar en 4 grandes pasos. El primero de ellos es la predicción, cuyo objetivo es reducir la cantidad de información que debe codificarse aprovechando la redundancia espacial y temporal entre macrobloques [7]. En este paso, se genera para cada macrobloque una estimación de su contenido a partir de información de macrobloques previamente codificados.

Posteriormente, la información generada en la estimación se sustrae del macrobloque original, dando lugar a los denominados datos residuales. Estos datos son los que no va a ser capaz de generar el decodificador a través de datos codificados previamente y, por tanto, resultan necesarios para que la imagen que se genere en el decodificador sea lo más parecida a la original [3].

Este proceso cambia según el tipo de predicción que se aplica, cuando se aplica predicción intra, la estimación es generada a partir de macrobloques cercanos al que se está procesando pertenecientes al mismo fotograma. Mientras que, si se trata de predicción inter, la estimación es generada a partir de macrobloques de fotogramas previamente codificados mediante procesos de estimación y compensación de movimiento [3].

Es importante destacar que, en esta fase, la estimación no se genera a partir de macrobloques de la imagen original, sino que se genera a partir de macrobloques estimados generados anteriormente. Esto se debe a que tanto el codificador como el decodificador solo disponen de esta información estimada para generar las predicciones [3].

La obtención de esta imagen reconstruida requiere un proceso interno de reconstrucción, que incluye la "descuantización", la transformada entera inversa para revertir la transformación aplicada durante la codificación y la aplicación de un filtro como se muestra en la Figura 2.4.

El siguiente paso es la transformación, que transforma los datos residuales obtenidos del paso de la predicción al dominio de la frecuencia para su posterior procesamiento en pasos siguientes [7].

El paso siguiente es la cuantización, que reduce la precisión de los coeficientes obtenidos en la transformada, consiguiendo disminuir la cantidad de información necesaria para representar esos bloques [7].

Finalmente se lleva a cabo la codificación entrópica, último paso del estándar en donde se termina de comprimir la información y en donde se centra el desarrollo de este TFT [7]. Se puede observar también en la Figura 2.4.

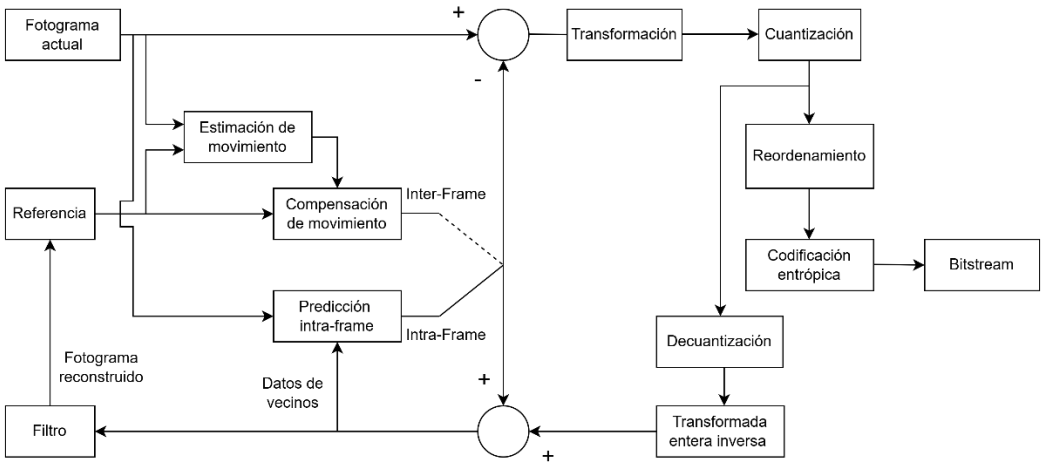

Figura 2.4: Diagrama de bloques del proceso del estándar H.264. Adaptada de [7].

2.3.1 Predicción

En el estándar H.264, el proceso de codificación utiliza técnicas de predicción para reducir la cantidad de datos que deben transmitirse. El codificador genera una predicción del macrobloque actual a partir de datos previamente codificados, ya sea dentro del mismo fotograma mediante predicción intra, o a partir de otros fotogramas mediante predicción inter [3].

2.3.1.1 Predicción intra

En el caso de la predicción intra, el codificador genera una estimación del contenido del macrobloque actual basándose únicamente en datos ya codificados dentro del mismo slice. Esta predicción se basa en la gran similitud que suelen tener los macrobloques cercanos en un mismo fotograma. Por ello, se utilizan los valores ya codificados de los bloques adyacentes para anticipar el contenido del bloque que se está procesando [4].

Los macrobloques intra pueden codificarse subdividiéndose en bloques de menor tamaño, estos bloques pueden ser de 4×4 píxeles, 8×8 píxeles o usando el bloque entero de 16×16 píxeles. En la implementación de CABAC para este trabajo solo se usarán los tamaños 4×4 y 16×16 [3].

La elección del tamaño del bloque va a influir tanto en la calidad de la predicción como en la cantidad de bits necesarios para su codificación. Esto se debe a que los bloques 4×4 tienden a generar predicciones más precisas y ajustadas a los detalles del contenido, lo que reduce el residuo (la diferencia entre el valor real y la predicción). Sin embargo, esta precisión también implica que es necesario transmitir más información al decodificador sobre qué modo de predicción se utilizó en cada bloque [3].

Por otro lado, los bloques más grandes, como los de 16×16, suelen ofrecer una predicción menos precisa, pero requieren menos bits para describir el modo de predicción, lo que puede resultar ventajoso en términos de compresión [3].

Por lo tanto, la elección del tamaño de bloque depende de qué opción produzca una representación más eficiente: bloques de 4×4, con residuos más pequeños pero que exigen especificar más modos de predicción, o bloques de 16×16, con residuos mayores, pero menos información de predicción. Normalmente cuando la imagen es homogénea y tiene muchos pixeles iguales unos cerca de otros se usan los bloques de 16×16, mientras que si la imagen tiene mucho nivel de detalle se usan bloques 4×4 [4].

Además, la predicción intra puede realizarse mediante distintos modos de predicción, cada uno de los cuales define una forma específica de estimar los valores del bloque en función de sus vecinos. Los modos usados en esta implementación son los siguientes:

- Vertical: las predicciones del bloque se obtienen extrapolando verticalmente los valores de luminancia más cercanos del bloque superior al bloque que se está procesando [4].

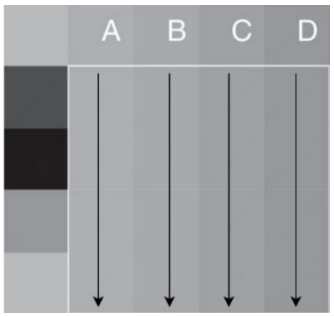

Figura 2.5: Predicción vertical aplicada a bloque de 4×4. Adaptada de [3].

- Horizontal: las predicciones del bloque se obtienen extrapolando horizontalmente los valores de luminancia más cercanos del bloque ubicado a la izquierda del bloque que se está procesando [4].

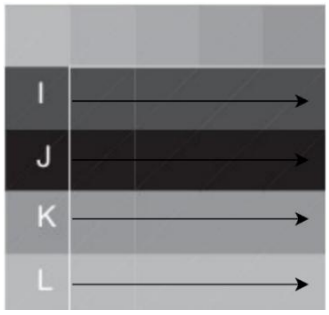

Figura 2.6: Predicción horizontal aplicada a bloque de 4×4. Fuente [3].

- DC: las predicciones del bloque se obtienen haciendo una media de los valores de luminancia tanto de los píxeles superiores como de los píxeles a la izquierda del bloque que se está procesando [4].

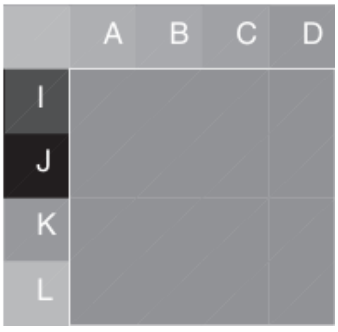

Figura 2.7: Predicción DC aplicada a bloque de 4×4. Fuente [3].

2.3.2 Transformación

El siguiente paso dentro del proceso que aplica el estándar H.264 es la transformación, la cual se aplica a los bloques residuales generados tras la etapa de predicción. Esta etapa permite representar los datos del bloque en el dominio de la frecuencia, facilitando la posterior compresión. Para ello, se utiliza una transformada entera, que es una versión aproximada y escalada de la Transformada Discreta del Coseno (DCT), diseñada para operar con números enteros, lo que reduce la complejidad computacional y evita errores de redondeo entre el codificador y el decodificador. [4], [8].

Después de aplicar esta transformada, se obtiene un bloque de coeficientes transformados, donde cada valor representa una frecuencia espacial dentro del bloque. El coeficiente DC, ubicado en la esquina superior izquierda, representa la frecuencia más baja, es decir, la frecuencia cero o componente continua. Esto significa que contiene la mayor parte de la energía del bloque y, por tanto, es el coeficiente más significativo en términos visuales, ya que refleja el nivel medio de brillo o color del área representada. El resto de los coeficientes, llamados AC, representan frecuencias más altas, que capturan detalles, bordes y texturas [3], [8].

En el caso de los macrobloques codificados con predicción Intra 16×16, se aplica un tratamiento especial a los coeficientes DC de los bloques 4×4. Concretamente, se extraen los coeficientes DC individuales de los 16 bloques 4×4 que componen la región de luminancia del macrobloque. Estos coeficientes se agrupan en un nuevo bloque de 4×4 y se les aplica una transformada Hadamard, ya que suelen estar altamente correlacionados. Este paso adicional mejora la eficiencia de la compresión, ya que esta transformada, basada únicamente en sumas y restas, permite representar la información de forma más compacta, concentrando la energía en unos pocos coeficientes [3], [8].

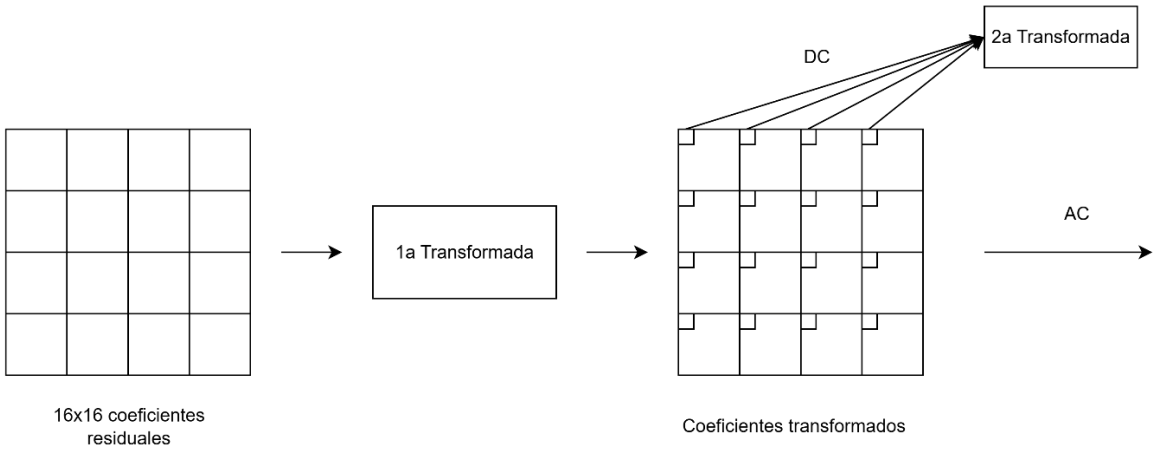

Figura 2.8: Diagrama de bloques mostrando el paso de transformación en bloques codificados con predicción Intra 16×16. Adaptada de [3].

2.3.3 Cuantización

Una vez aplicada la transformada a los bloques residuales, el siguiente paso en el proceso de codificación es la cuantización. Este procedimiento consiste en reducir la precisión de los coeficientes obtenidos en la transformada, lo cual permite disminuir la cantidad de información necesaria para representar esos bloques. La cuantización es una operación con pérdida, pero es fundamental para lograr una compresión efectiva [3], [4].

El proceso se basa en dividir cada coeficiente por un valor determinado llamado paso de cuantización, que está relacionado con QP. Luego, el resultado se redondea al entero más cercano. Esta operación reduce la magnitud de los coeficientes y, en muchos casos, convierte los valores pequeños, que son menos significativos, en ceros. Como consecuencia, se eliminan detalles que el ojo humano apenas percibe, pero se consigue una notable reducción en la cantidad de datos que deben almacenarse o transmitirse. Este paso se aplica tanto a los coeficientes AC como a los DC [3], [4].

2.3.4 Codificación entrópica

El último paso dentro del estándar H.264 es la codificación entrópica, en ella se comprime la información necesaria para representar el vídeo con el decodificador H.264, esta información contiene parámetros, identificadores y códigos de delimitación, así como tipos de predicción, vectores de movimiento codificados diferencialmente y coeficientes de transformada. El estándar H.264 especifica dos métodos para codificar estos símbolos, es decir, para convertir cada símbolo en un patrón binario que se transmite o se almacena como parte del flujo de bits (bitstream) [3].

Los dos métodos son CAVLC y CABAC, el primero se caracteriza por ofrecer menor tasa de compresión, pero una mayor rapidez y sencillez de desarrollo. Mientras que CABAC ofrece mayor compresión a cambio de una mayor complejidad de procedimientos [3]. Como se mencionó en el Capítulo 1 este trabajo se centra en el desarrollo de CABAC que se explica más detenidamente en el siguiente capítulo.

2.4 Descripción del proceso de CABAC

El objetivo de CABAC, en el estándar H.264, es optimizar la compresión de datos mediante una codificación aritmética binaria adaptativa basada en el contexto. Este proceso se divide en tres etapas fundamentales: la binarización, el modelado de contexto y la codificación aritmética binaria. Además, CABAC puede operar en dos modos distintos: el modo normal, que utiliza el modelado de contexto y la codificación aritmética completa, y el modo bypass, que se aplica a elementos con probabilidad uniforme, es decir, cuando la probabilidad de que un bit sea 0 o 1 es aproximadamente igual. Esto permite una codificación más simple y rápida al prescindir del modelado de contexto [8], [9].

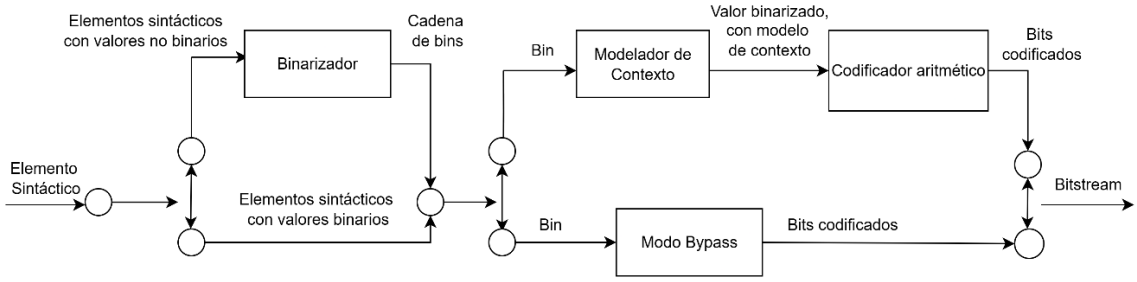

Figura 2.9: Diagrama de los procesos del codificador CABAC. Adaptada de [8].

2.4.1 Descripción de los elementos sintácticos

Antes de explicar en detalle el proceso de CABAC, es importante destacar que en esta etapa se reciben los coeficientes y la información relacionada con la predicción, como el tipo de predicción o si corresponde al modo de predicción más probable y el valor del QP. A partir de estos valores se generan los elementos sintácticos que a partir de ahora serán nombrados como SE (acrónimo de Syntax Element) en singular y SEs en plural. Estos contienen la información necesaria para que el decodificador pueda reconstruir correctamente la imagen que está siendo codificada [8], [9].

Asimismo, cabe recalcar que existen dos tipos de SEs, los residuales y no residuales, los primeros son los elementos que tienen información directamente relacionada con los coeficientes que ya han sido cuantizados y llegan a CABAC. Estos están relacionados con una posición de escaneo [i] que indica la posición dentro de la matriz del coeficiente que se está codificando. Y los elementos sintácticos no residuales son aquellos que aportan otro tipo de información no relacionada con los coeficientes, como el tipo de predicción que ha sido usada y su modo o información relacionada con el parámetro de cuantización [8].

2.4.1.1 SEs no residuales

- **MBT (acrónimo de Macroblock_type)**: indica el tipo de macrobloque, su valor depende de si se usa predicción intra 4 o intra 16 en su codificación, de si contiene coeficientes no nulos y del modo con que se codificó el macrobloque (V, H o DC) [10].

- **PMF (acrónimo de Prev_intra4×4_pred_mode_flag)**: indica si el bloque de coeficientes 4×4 utiliza el modo de predicción intra más probable. Toma el valor 1 cuando el modo de predicción coincide con el modo más común, determinado a partir de bloques previamente procesados, y el valor 0 en caso contrario, especificándose entonces el modo de predicción en el SE REM. Este SE no se emplea cuando el macrobloque utiliza predicción intra 16×16 [10].

- **REM (acrónimo de Rem_intra4×4_pred_mode)**: indica el tipo de predicción que usa el bloque 4×4 actual, si es vertical, horizontal o DC. Si el macrobloque usa predicción intra 16×16 no se usa, ya que viene definido en el tipo de macrobloque (MBT) [10].

- **QPD (acrónimo de Mb_qp_delta)**: especifica la diferencia entre el QP utilizado en el macrobloque actual y el macrobloque anterior. El QPD del primer macrobloque en cada slice especifica la diferencia entre el QP del primer macrobloque y el QP del slice [10].

2.4.1.2 SEs residuales

- **CBP (acrónimo de Coded_Block_Pattern)**: en el caso de predicción intra 4, especifica cuáles de los cuatro bloques 8×8 que componen un macrobloque contiene al menos un coeficiente que no es 0. En el caso de intra 16, indica si al menos hay un coeficiente distinto de 0 en la parte de coeficientes AC del macrobloque [10].

  Este SE está formado por un total de 4 bits, simbolizando cada uno de estos un bloque 8×8 del macrobloque como se muestra en la Figura 2.10. Asimismo, en esta figura podemos ver un ejemplo de cómo se vería el CBP cuando hay coeficientes distintos de 0 en el primer bloque (B0) de 8×8 y en el B2 en el caso de intra 4 e intra 16 [10].

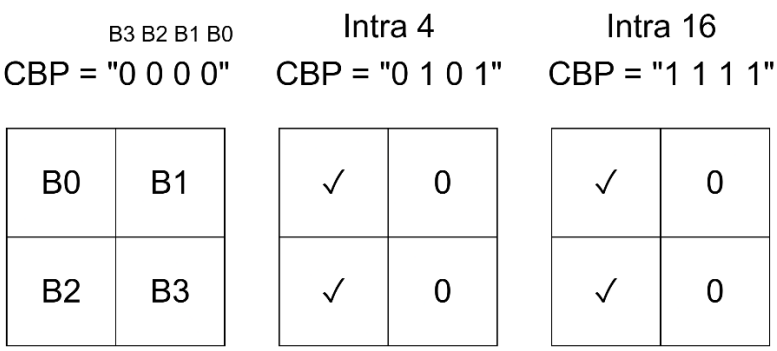

Figura 2.10: Demostración del SE CBP. Fuente [3].

- **CBF (acrónimo de Coded_Block_Flag)**: indica si un bloque de 4×4 contiene coeficientes distintos de cero. Por lo tanto, si el valor de CBF es 1, al menos uno de los coeficientes que conforman el bloque es distinto a 0, mientras que, si el valor del elemento sintáctico es 0, ninguno de los coeficientes será distinto de 0 [10].

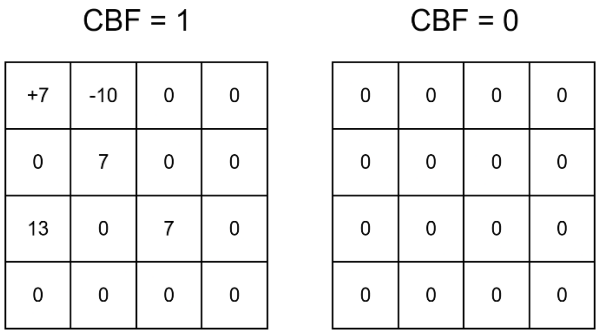

Figura 2.11: Ejemplo del SE de CBF. Adaptado de [3].

- **SCF (acrónimo de Significant_coeff_flag[i])**: indica si el coeficiente de transformación en una posición de escaneo específica es distinto de cero: si SCF es 0, el coeficiente es 0; si es 1, el coeficiente tiene un valor diferente de cero [10].

- **LSCF (acrónimo de Last_significant_coeff_flag[i])** indica si, a partir de la posición de escaneo i, existen coeficientes de transformación distintos de cero en posiciones posteriores. Si LSCF[i] es 1, todos los coeficientes siguientes son 0; si es 0, aún hay coeficientes diferentes de cero más adelante en el escaneo [10].

- **CALM1 (acrónimo de Coeff_abs_level_minus1[i])**: es el valor absoluto de un coeficiente de transformación menos 1. Es decir, si el valor original es x, pues el valor de CALM1 será x-1 [10].

- **CSF (acrónimo de coeff_sign_flag[i])**: especifica el signo de un coeficiente de transformación, cuando es igual a 0 el coeficiente es positivo y cuando es igual a 1 el coeficiente es negativo [10].

2.4.2 Binarización

Una vez explicados todos los elementos sintácticos, ya se puede proceder a la descripción del primer proceso, la binarización. Este proceso se encarga de transformar todos los elementos sintácticos en bins, es decir conjuntos de 1 o 0. A diferencia de los bits, a estos denominados bins se les asignará una probabilidad en el paso de modelado de contexto [8]. En el proceso de binarización se usan distintos métodos, los cuales son escritos en los siguientes párrafos.

2.4.2.1 Codificación Unaria y Unaria Truncada (TU)

La codificación unaria es un método sencillo para representar enteros sin signo. Para un valor entero m, se genera una secuencia compuesta por m bins con valor "1", seguidos de un bin de terminación con valor "0". En la tabla se muestra un ejemplo sencillo de una codificación unaria [8].

Valor	**Binarización final**
0	0
1	10
2	110
3	1110
4	11110

Tabla 2.1: Ejemplo de codificación unaria. Adaptada de [8].

La versión truncada de este código, denominada código TU (acrónimo de unario truncado), se define para valores en el rango $0 \leq m \leq cMax$, donde cMax es un valor límite. Si $m<cMax$, el código TU es idéntico al código unario estándar. Sin embargo, cuando $m=cMax$, se omite el bin de terminación "0", y el código consiste únicamente en cMax bins con valor "1". Esta modificación permite reducir la longitud del código en casos donde el valor máximo es frecuente, contribuyendo a una mayor eficiencia de compresión [8].

Valor	**Binarización final**
0	0
1	10
2	110
3	1110
4	11110
5	111110
6	111111

Tabla 2.2: Ejemplo de codificación unaria truncada con cMax igual a seis.

2.4.2.2 Codificación Fixed-length

FL (acrónimo de Fixed_length) es un tipo de binarización que, como su propio nombre indica, usa longitud fija para todos los valores de un mismo SE. Esta longitud será el resultado de la operación "fixedLength = Ceil(Log2(cMax + 1))" donde ceil es la operación de redondeo hacia arriba y cMax es el máximo valor que puede tener el SE [10]. En la Tabla 2.3 se puede apreciar un ejemplo de implementación de este tipo de binarización con dos cMax distintos:

SE_val	Bins (cMax = 3)	Bins (cMax = 7)
0	00	000
1	01	001
2	10	010
3	11	011
4	-	100
5	-	101
6	-	110
7	-	111

Tabla 2.3: Ejemplo de binarización Fixed_length con cMax = 3 y cMax = 7. Adaptada de [4]

2.4.2.3 Codificación Exponential-Golomb de orden k

La binarización Exponential-Golomb de orden k es una técnica avanzada diseñada para representar números enteros de forma eficiente [8].

Esta codificación se compone de dos partes:

- **Prefijo**: Está compuesto por una secuencia de l bins con valor 1, seguida de un bin con valor 0, que actúa como separador entre el prefijo y el sufijo [8]. El valor de l viene determinado por la siguiente expresión:

$$l(x) = \left\lfloor \log_2\left(\frac{x}{2^k} + 1\right)\right\rfloor$$

- **Sufijo**: el sufijo se obtiene como la representación binaria del valor:

$$x + 2^k\,(1 - 2^{l(x)})$$

  empleando k + l(x) bits significativos, donde l(x) es la longitud del prefijo.

Valor	$l(x) = \left\lfloor \log_2\left(\frac{x}{2^k} + 1\right) \right\rfloor$	Prefijo	Sufijo	Binarización final
0	0	0	-	0
1	1	10	0	100
2	1	10	1	101
3	2	110	00	11000
4	2	110	01	11001
5	2	110	10	11010
6	2	110	11	11011

Tabla 2.4: Ejemplo de binarización Exp_Golomb de orden 0

2.4.2.4 Codificación Concatenated Unary/k-eth order Exp-Golomb (UEGK)

La codificación UEGK (acrónimo de Concatenated Unary/k-eth order Exp-Golomb) es un tipo de binarización que combina la codificación unaria con la codificación exponencial-golomb de orden k, para lograr un método eficiente y que adapta su mecanismo según el valor a binarizar. Para ello se define el parámetro "S", este define el valor máximo con el que la binarización se compone por la codificación TU.

Si el valor a binarizar es mayor a "S", la binarización va a estar conformada por un prefijo que será la parte codificada con TU y el resto de bins serán el sufijo que provendrán de la codificación Exponential-Golomb aplicada al valor obtenido tras restar el parámetro "S" al valor original. Si se trata de un valor pequeño, se codifica con codificación unaria corta, mientras si se codifica un valor grande se usa codificación Exponential-Golomb adicionalmente [8].

Valor	Prefijo (TU)	Sufijo (Exp_Golomb (Valor - "S"))	Binarización final
0	0	-	0
1	10	-	10
2	110	-	110
3	1110	-	1110
4	1111	1	11111
5	1111	010	1111010
6	1111	011	1111011

Tabla 2.5: Ejemplo de binarización UEGK de orden 0 y S igual a 3. Adaptada de [8].

2.4.2.5 Codificación de macroblock_type

Además de los tres tipos de binarización mencionados, existe otro que se aplica al elemento sintáctico MBT. A diferencia de los anteriores, este elemento se binariza directamente mediante valores definidos en tablas preestablecidas por el estándar [10]. En la Tabla 2.6 se puede apreciar un ejemplo de estas, donde se relaciona los distintos valores posibles del SE mb_type explicado en 2.4.1.1 con su binarización.

Valor de mb_type	**Binarización**						
0	0						
1	1	0	0	0	0	0	
2	1	0	0	0	0	1	
3	1	0	0	0	1	0	
4	1	0	0	0	1	1	
5	1	0	0	1	0	0	0
6	1	0	0	1	0	0	1
7	1	0	0	1	0	1	0
8	1	0	0	1	0	1	1
9	1	0	0	1	1	0	0
10	1	0	0	1	1	0	1
11	1	0	0	1	1	1	0
12	1	0	0	1	1	1	1
13	1	0	1	0	0	0	
14	1	0	1	0	0	1	
15	1	0	1	0	1	0	
16	1	0	1	0	1	1	
17	1	0	1	1	0	0	0
18	1	0	1	1	0	0	1
19	1	0	1	1	0	1	0
20	1	0	1	1	0	1	1
21	1	0	1	1	1	0	0
22	1	0	1	1	1	0	1
23	1	0	1	1	1	1	0
24	1	0	1	1	1	1	1
binIdx	0	1	2	3	4	5	6

Tabla 2.6: Valores de binarización para macroblock_type de predicción intra. Fuente [10].

2.4.3 Modelado de contexto

Una vez realizada la binarización de los elementos sintácticos, el siguiente paso fundamental en el proceso de CABAC es el modelado de contexto. Esta etapa tiene como objetivo otorgar dos valores a cada bin, uno el "valMPS" que indica si ese bin tiene más probabilidad de ser 0 o 1, y el "pState" que indica el porcentaje de probabilidad de que ese bin sea el "valMPS". A través de este mecanismo, se asignan modelos de probabilidad adaptativos que permiten una codificación aritmética más eficiente, al dar menor longitud binaria a los valores más probables y mayor a los menos probables [8].

En este apartado se explica el concepto de tablas de contexto y su proceso de inicialización, así como el término ctxIdx, que es el índice que apunta a los valores de contexto ("valMPS" y "pState") de los bins. Asimismo, se explican los valores para obtener este ctxIdx (CtxIdxOffset, CtxIdxIncrement, CtxBlockCat) y cómo se van actualizando las tablas a medida que se codifican bins [8].

2.4.3.1 Inicialización de las tablas de contexto

Al principio, como no se ha codificado ningún dato, se le debe otorgar valores por defecto a cada bin de cada elemento sintáctico. Para ello, en el estándar se han establecido unas tablas de contexto, estas están compuestas por índices que van del 0 al 1023, donde cada índice almacena un par de valores denominados *m* y *n*. Con estos valores y el QP, se obtienen los valores de "valMPS" y "pState" a través de las fórmulas que se muestran en la Figura 2.12 [10], [11].

```
preCtxState = Clip3(1, 126, (( m * Clip3(0, 51, SliceQPY)) >> 4) + n);
if(preCtxState <= 63)  {
   pStateIdx = 63 - preCtxState;
   valMPS = 0;
} else {
   pStateIdx = preCtxState - 64;
   valMPS = 1;
}
```

Figura 2.12: Pseudo-código para la inicialización de contextos. Adaptada de [10].

Nota: Clip3(x,y,z) es una función definida en [10], siendo su salida:

- **x**: si z < x.
- **y**: si z > y.
- **z** en caso de que no se cumpla ninguna de las condiciones anteriores.

En CABAC, los contextos pueden inicializarse al comienzo de cada slice o al inicio de cada fotograma. En la implementación llevada a cabo en este TFT cada slice coincide con un fotograma, por lo tanto, al inicio de cada fotograma se lleva a cabo este procedimiento.

2.4.3.2 Indice CtxIdx

Cuando ya se han creado las tablas de contexto, los índices de los elementos que son procesados, que se denominan CtxIdx, se obtienen con las siguientes fórmulas dependiendo de si el elemento es residual o no [10]. Las variables de las fórmulas son descritas en los subapartados 2.4.3.3, 2.4.3.4, 2.4.3.5:

- **No residual**: CtxIdx= ctxIdxOffset + ctxIdxIncrement
- **Residual**: CtxIdx = ctxIdxOffset + ctxIdxIncrement(ctxBlockCat) + ctxIdxBlockCatOffset(ctxBlockCat)

2.4.3.3 CtxIdxOffset

El parámetro ctxIdxOffset es obtenido a partir de la siguiente tabla en donde se le asigna un valor fijo a cada elemento sintáctico:

Elemento sintáctico	Valor de ctxIdxOffset
MBT	3
PMB	68
REM	69
CBP	73
CBF	85
SCF	105
LSCF	166
CALM1	227
CSF	276

Tabla 2.7: Elementos sintácticos con sus valores de ctxIdxOffset. Fuente [10].

2.4.3.4 CtxIdxIncrement

CtxIdxIncrement es obtenido según los valores de cada elemento sintáctico de los bloques codificados anteriormente. Como se muestra en la Figura 2.13, A es el bloque situado justo a la izquierda del bloque que está siendo procesado y B el que tiene justo encima [10].

D	B	C
A	Macrobloque actual o bloque (8x8 o 4x4)	

Figura 2.13: Representación gráfica de los bloques A y B. Adaptada de [10].

2.4.3.5 CtxBlockCat

Por otro lado, en los elementos residuales, la obtención del valor de ctxIdx no solo depende del tipo de elemento sintáctico, sino también del tipo de bloque al que pertenece dicho elemento. Según el tipo de bloque, se le asignará un valor de ctxBlockCat, que a su vez será utilizado para obtener el ctxIdxIncrement de estos elementos, así como el ctxIdxBlockCatOffset [10].

En la Tabla 2.8 se muestran los tipos de bloques que se tienen en cuenta en esta implementación de CABAC, junto con los valores de ctxBlockCat que se les asignan:

Tipo de bloque	Valor de ctxBlockCat
Bloque de coeficientes de luminancia DC usando predicción intra 16×16	0
Bloque de coeficientes de luminancia AC usando predicción intra 16×16	1
Bloque de coeficientes de luminancia usando predicción intra 4×4	2

Tabla 2.8: Valores de ctxBlockCat relacionados con el tipo de bloque. Adaptada de [10].

2.4.3.6 Actualización de los contextos

Cuando ya se han inicializado los contextos y se ha calculado el ctxIdx del bin que se va a codificar, se procede a extraer los valores de "valMPS" y "pState" de dentro de la tabla de contexto mediante el índice ctxIdx. Habiendo extraído estos valores, se procede a comprobar si el "valMPS" es igual al valor del bin que se está procesando o no. Si es igual, el valor de "pState", que va en un rango del 0 al 63, incrementa. Por contrario, si no es igual, es decir, si corresponde al valor "LPS" (acrónimo de Least Probable Symbol), "pState" disminuye.

El incremento y disminución de "pState" depende del valor de este y esta transición está descrita por el estándar en la Tabla 9-45 de [10]. Asimismo, se puede ver gráficamente en la Figura 2.14 extraída de [8].

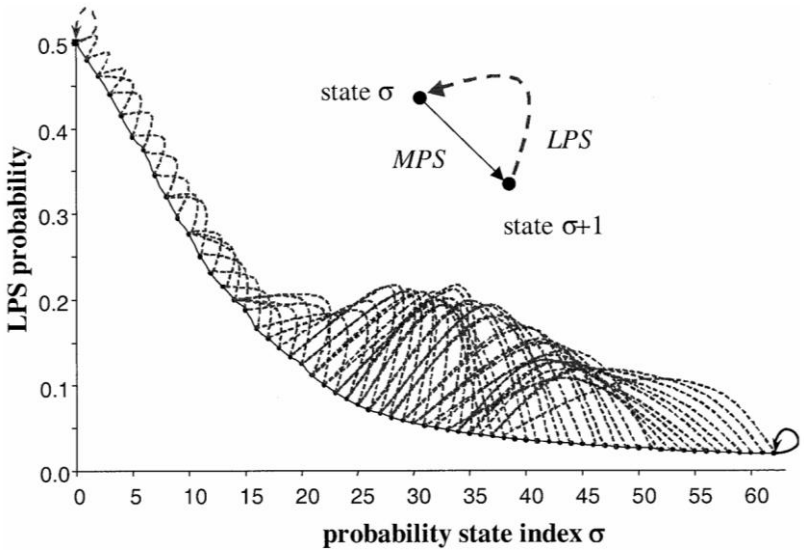

Figura 2.14: Transiciones del valor de "pState" dependiendo del valor del bin a codificar. Fuente [8].

2.4.4 Codificación aritmética binaria

En este apartado se describe la etapa final del proceso CABAC, correspondiente a la codificación aritmética binaria, encargada de transformar los bins y los modelos de contexto en el modo normal y los bins sin contexto en el modo “bypass”, en cadenas de bits, es decir, en el “bitstream” [10], [11].

2.4.4.1 Codificación aritmética

A diferencia de otro tipo de métodos de codificación, como la codificación de Huffman, la codificación aritmética tiene la peculiaridad de que permite lograr tasas de compresión cercanas al límite teórico [3]. Esto lo consigue a través de la transformación de las secuencias de símbolos que recibe, en un número fraccionario, lo que posibilita aproximarse al número óptimo de bits necesarios para representar cada símbolo en formato binario [3]. Para hacer más fácil el entendimiento de este tipo de codificación, se propone el siguiente ejemplo:

Supongamos que tenemos la siguiente secuencia de símbolos: ACBC, el primer paso para realizar la codificación es definir el alfabeto y las probabilidades de cada símbolo. Como se puede ver en la secuencia, se puede distinguir un alfabeto de 3 símbolos, estos son A, B y C. El siguiente paso tras obtener el alfabeto es obtener las probabilidades de cada símbolo. Estas se muestran en la Tabla 2.9.

Símbolos	Frecuencia	Probabilidad
A	1	1/4 = 0,25
B	1	1/4 = 0,25
C	2	2/4 = 0,5

Tabla 2.9: Obtención de las probabilidades de los símbolos del ejemplo de codificación aritmética.

Una vez obtenidas las probabilidades, se define el intervalo de [0,0 a 1,0) para posteriormente dividirlo en subintervalos, utilizando las probabilidades de cada símbolo. Como se muestra en la Figura 2.15.

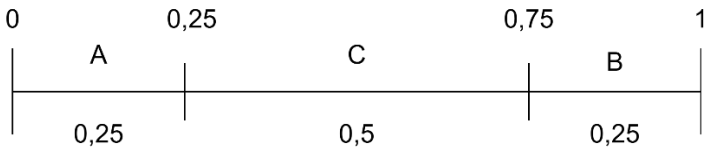

Figura 2.15: Representación de los rangos del ejemplo de codificación aritmética.

Cuando los intervalos ya están definidos, se procede a realizar el proceso de codificación, en donde se buscan los nuevos límites inferior y superior, dando como resultado la reducción del intervalo. Aplicando las siguientes fórmulas a cada símbolo:

- Límite inferior = Antiguo inferior + (Antiguo superior – Antiguo inferior) × límite inferior del símbolo
- Límite superior = Antiguo inferior + (Antiguo superior – Antiguo inferior) × límite superior del símbolo

Por lo tanto, aquí tenemos el proceso completo de la codificación aplicada a la secuencia:

Primer símbolo A con intervalo [0,0; 0,25]

- Límite inferior = 0,0 + (1,0 – 0,0) × 0,0 = 0,0
- Límite superior = 0,0 + (1,0 – 0,0) × 0,25 = 0,25

Segundo símbolo C con intervalo [0.25; 0.75]

- Límite inferior = 0,0 + (0,25 – 0,0) × 0,25 = 0,0625
- Límite superior = 0,0 + (0,25 – 0,0) × 0,75 = 0,1875

Tercer símbolo B con intervalo [0,75; 1,0]

- Límite inferior = 0,0625 + (0,1875 – 0,0625) × 0,75 = 0,140625
- Límite superior = 0,0625 + (0,1875 – 0,0625) × 1,0 = 0,1875

Cuarto símbolo C con intervalo [0,25; 0,75]

- Límite inferior = 0,140625 + (0,1875 – 0,140625) × 0,25 = 0,0406875
- Límite superior = 0,140625 + (0,1875 – 0,140625) × 0,75 = 0,140625

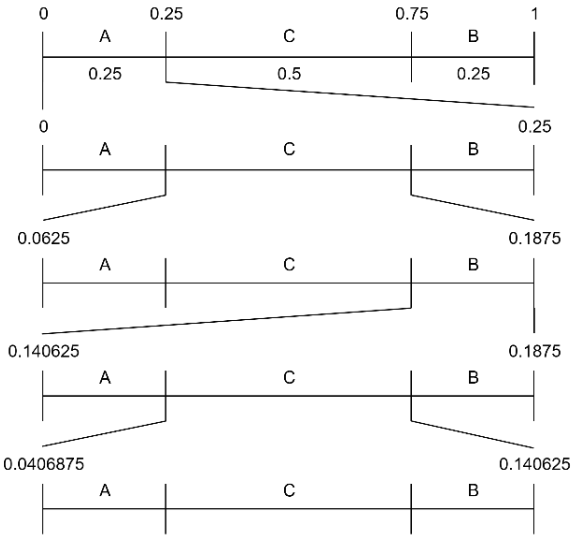

Figura 2.16: Proceso de codificación aplicado al intervalo de forma gráfica

Para tener un resultado final y pasarlo a binario, debemos elegir cualquier valor del rango final que hemos obtenido, es decir, de [0,0406875; 0,140625), como por ejemplo el número 0,1, este sería nuestro número fraccionario final.

2.4.4.2 Modo normal

El proceso descrito en el subapartado anterior no es exactamente el que se aplica en CABAC, ya que, en este caso, cada elemento codificado posee una probabilidad adaptativa asociada a las tablas de contexto, como se mencionó en la sección 2.4.3. Además, el ejemplo mostrado no es binario ya que tenía un alfabeto de tres elementos. Por ello es importante la binarización para convertir los valores de los elementos sintácticos en bins [8], [11].

Este procedimiento tiene la particularidad de que no utiliza multiplicaciones; en su lugar, emplea una tabla definida por el estándar con el fin de optimizar el cálculo. El intervalo que se va dividiendo se denomina codIRange. Dicho intervalo no contiene valores decimales y se encuentra en el rango (0, 510). Además, cuenta con dos límites, como en el ejemplo anterior; sin embargo, en este caso, el límite inferior recibe el nombre de codILow [10], [11].

El intervalo está conformado por CodIRangeMPS y CodIRangeLPS. En la figura correspondiente se muestran de forma visual las distintas variables: codIRange y codILow, con sus valores por defecto, es decir, los que presentan al iniciar el codificador, y CodIRangeMPS y CodIRangeLPS, con valores arbitrarios [11].

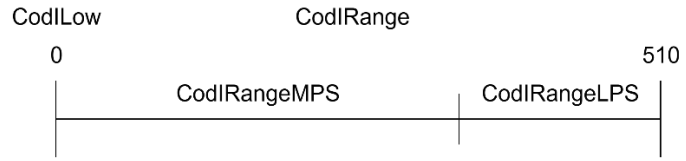

Figura 2.17: Representación gráfica del intervalo CodIRange y sus variables. Adaptada de [9].

Como ya se mencionó anteriormente, en el modo normal el codificador recibe el valor del bin a codificar (binVal) y el índice ctxIdx, el cual indica la posición en la tabla de contextos donde se encuentran los valores de “pState” y “valMPS”, necesarios para llevar a cabo la codificación [9].

Al recibir estos valores, el codificador procede a realizar una serie de pasos definidos por el estándar. Primero obtiene los bits más significativos de la variable codIRange (mediante un desplazamiento de los bits de izquierda a derecha y enmascaramiento). Posteriormente se usa la tabla rangeTabLPS, esta tabla fue definida en el estándar y tiene todos los posibles valores de LPS calculados para no tener que hacer operaciones matemáticas [10].

Por lo tanto, haciendo uso de los bits más significativos se accede a la columna de la tabla rangeTabLPS, para determinar el valor de codIRangeLPS ver Figura 2.17, mientras que la fila de la tabla se selecciona a partir del valor de pState asociado al ctxIdx. A continuación, se actualiza codIRange restándole el valor de codIRangeLPS [10], [11].

En este punto, el algoritmo evalúa si binVal es distinto de “valMPS”. En caso afirmativo, se sigue la rama LPS, donde se recalculan las variables internas: codILow se incrementa y codIRange toma el valor de codIRangeLPS. Luego, el valor de “pState” se actualiza mediante la tabla transIdxLPS definida por el estándar. Además, si el nuevo estado es igual a 0, se intercambia el valor de “valMPS” (el LPS pasa a ser el MPS y viceversa) [8], [10], [11].

Por el contrario, si binVal coincide con “valMPS”, se toma la rama MPS. En este caso se conservan los valores previamente calculados de codIRange y codIRangeLPS, y se actualiza el estado pState usando la tabla transIdxMPS, también especificada por el estándar [10], [11].

Finalmente, en ambos casos se ejecuta la etapa de renormalización, encargada de garantizar que las variables internas permanezcan dentro del rango válido, y se incrementa el contador de bins en la unidad de acceso (BinCountsInNALUnits). El proceso concluye cuando el bin ha sido correctamente codificado [8], [10], [11]. En la Figura 2.18 se puede apreciar el proceso gráficamente.

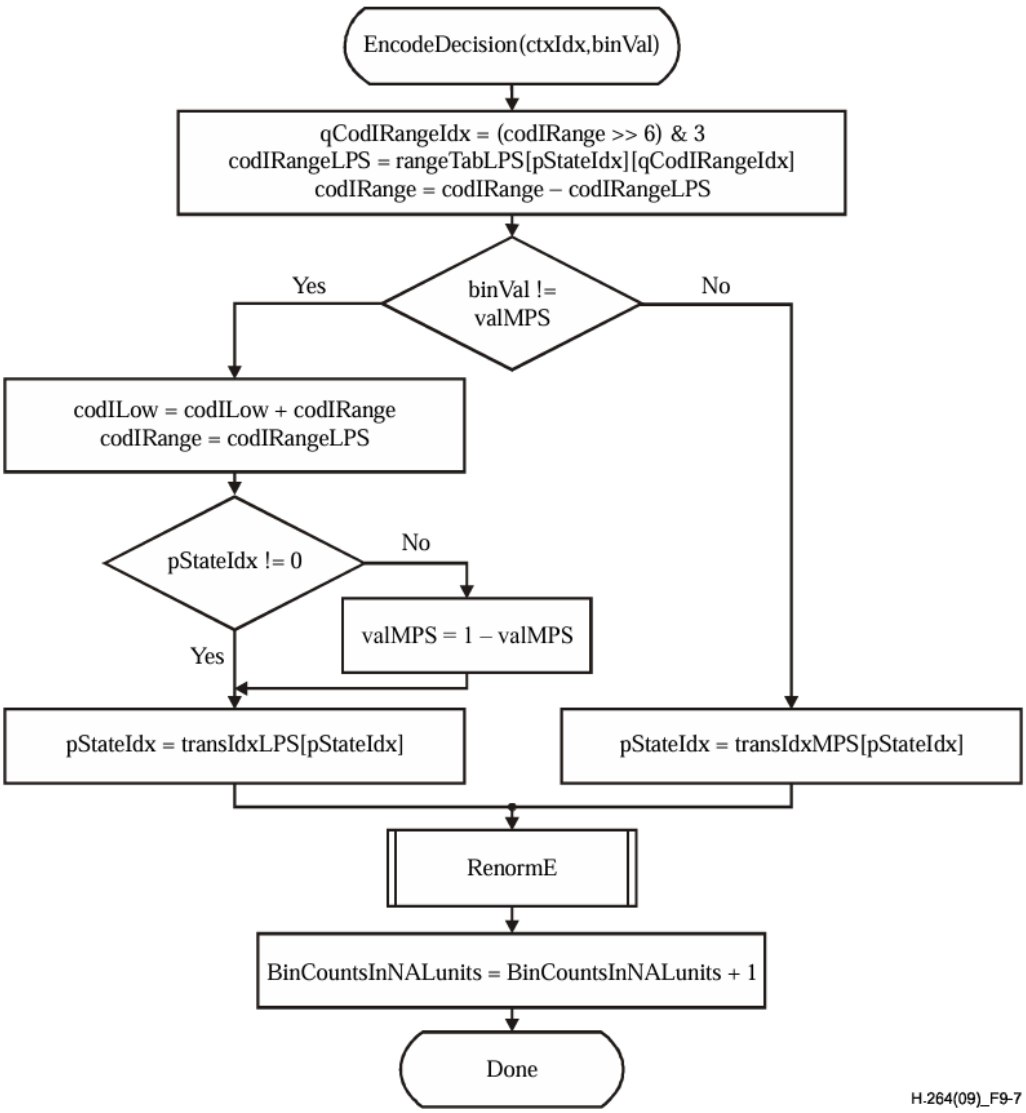

Figura 2.18: Diagrama de flujo de la codificación de un bin mediante proceso normal. Fuente [10].

2.4.4.3 Renormalización

Como se mencionó en el subapartado 2.4.4.2, tras la codificación de cada bin se lleva a cabo un proceso denominado renormalización que viene definido en el estándar. En este proceso, se realizan comprobaciones sucesivas sobre las variables que definen el estado interno del codificador aritmético, concretamente codIRange y codILow [10].

La primera variable que se comprueba es codIRange, si el valor de esta es mayor de 256 el proceso de renormalización finaliza. En cambio, si el valor de esta es menor, se procede a comprobar el valor de codILow:

- Si codILow es menor a 256 se ejecuta la función putBit, que será descrita en el apartado siguiente, con entrada 0.
- Si codILow es mayor a 256 y menor a 512 se le resta 256 a su valor y se incrementa la variable interna bitsOutstanding que será usada en la función putBit.
- Si codILow es mayor o igual a 512 se le resta 512 a su valor y se ejecuta la función putBit con entrada 1.

En todos los casos, cuando codIRange es menor a 256 y ya se ha llevado a cabo cada condición, se le duplica el valor a codIRange y codILow y se vuelve a comprobar el valor de codIRange como se muestra en la Figura 2.19 donde aparece todo el proceso de forma gráfica [10].

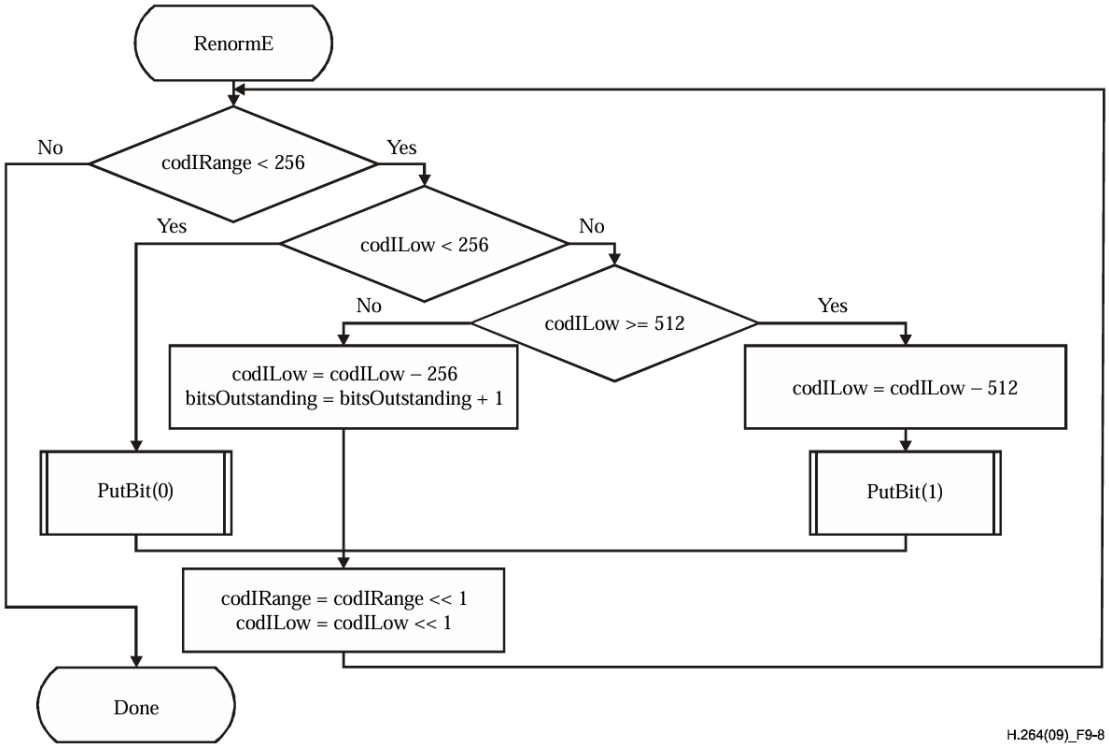

Figura 2.19: Diagrama de flujo del proceso de renormalización del modo normal. Fuente [10].

2.4.4.4 Escritura de bits

El proceso “Putbit” se emplea en la codificación binaria aritmética, tanto en el modo normal como en el modo bypass. Su propósito es gestionar la escritura de bits en el bitstream, es decir, en la salida del codificador [10].

La función recibe como entrada un bit, cuyo valor puede ser 0 o 1. En primer lugar, se evalúa la señal firstBitFlag, la cual indica si el bit recibido corresponde al primero que se escribe en el bitstream. En caso afirmativo, dicha señal se pone a 0, de modo que el codificador reconozca que los bits siguientes ya no constituyen el primero. En caso contrario, es decir, si la señal ya se encuentra en 0, el bit de entrada se escribe directamente en el bitstream [10].

Posteriormente, se verifica el valor de la variable interna bitsOutstanding, la cual ha sido modificada en fases anteriores del proceso de codificación binaria aritmética. Si el valor de esta variable es igual a 0, el procedimiento concluye. En cambio, si es distinto de cero, se escriben en el bitstream tantos bits como el valor de la variable, pero con polaridad opuesta a la del bit de entrada [10].

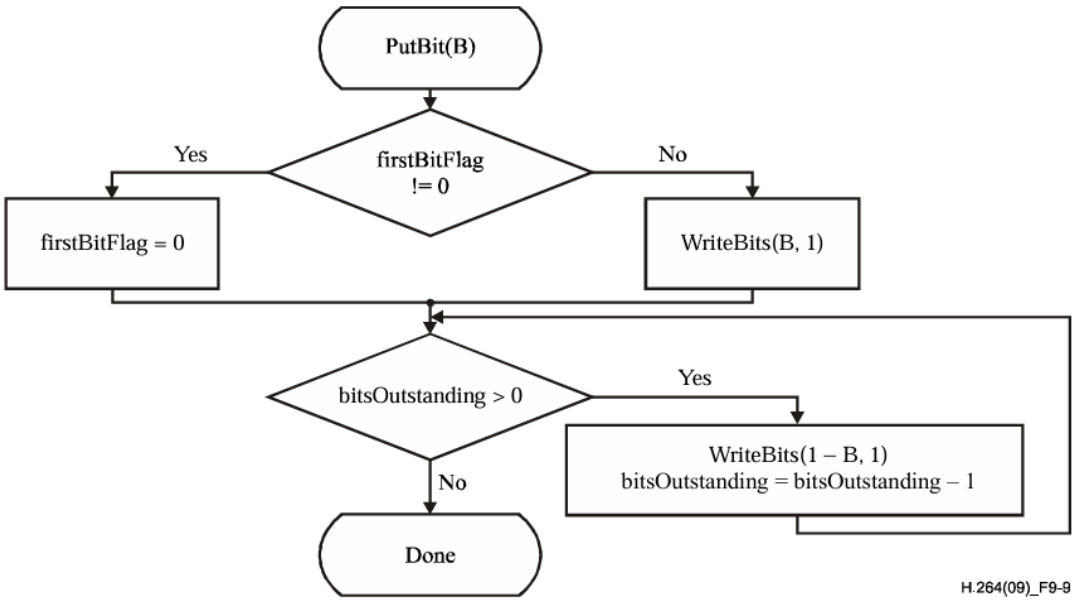

Figura 2.20: Diagrama de flujo del proceso de Putbit. Fuente [10].

2.4.4.5 Modo Bypass

Como se mencionó previamente, el modo Bypass no emplea modelos de contexto. Esto se debe a que determinados bins asociados a elementos sintácticos presentan una equiprobabilidad de tomar los valores 0 o 1. Para la codificación de este tipo de bins se recurre al modo Bypass, reduciendo así la complejidad del proceso sin comprometer la eficiencia de la codificación [10].

El primer paso consiste en duplicar el valor de la variable codILow, posteriormente si el valor del bin que llega a la entrada del proceso es igual a 1, se procede a aumentar el valor de codILow nuevamente sumándole el valor de codIRange [10].

Tras este paso, se procede a comprobar el valor de codILow siguiendo las siguientes condiciones:

- Si codILow es mayor o igual a 1024 se ejecuta el proceso de Putbit con entrada 1, y se le resta 1.024 a codILow
- Si codILow es menor a 512 se ejecuta el proceso de Putbit con entrada 0.
- Si codILow es mayor o igual a 512 y menor a 1.024 se le resta 512 a codILow y se le suma 1 a la variable bitsOutstanding.

Finalmente, se incrementa la cuenta de bits en unidades NAL.

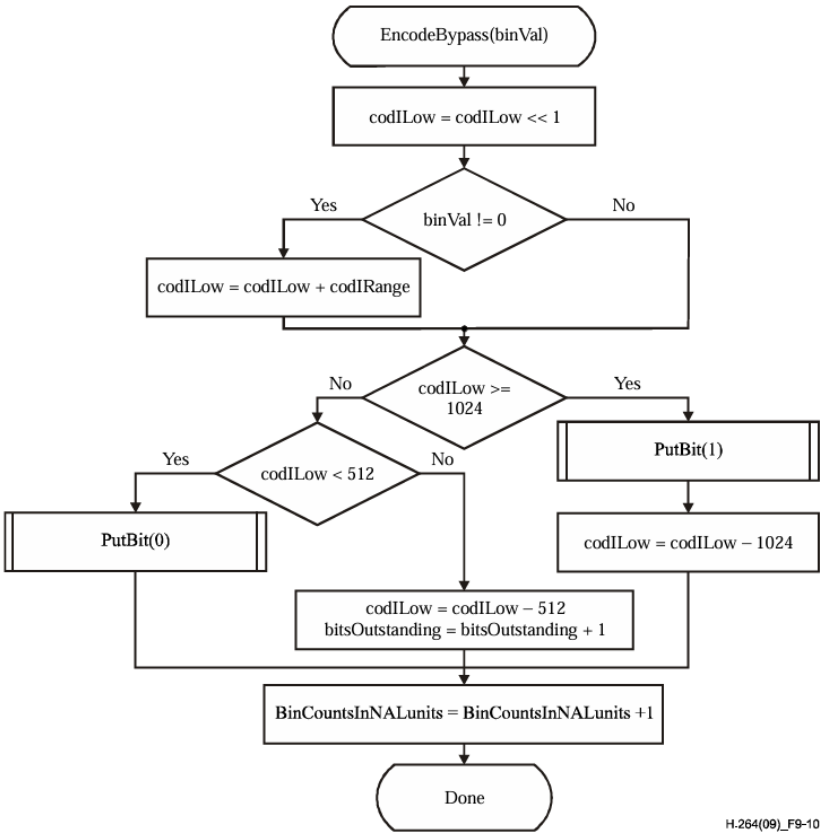

Figura 2.21: Diagrama de flujo del proceso del modo de codificación aritmética binaria Bypass. Fuente [10].

2.4.4.6 Proceso EncodeTerminate

El proceso EncodeTerminate se ejecuta cuando se han terminado de codificar los elementos de un macrobloque y se va a comenzar a codificar otro. Este proceso recibe un bin con un valor que nos indica si se trata del último bloque del slice o no [10].

El primer paso que aplica el proceso es restarle 2 al valor de codIRange. Posteriormente se comprueba el valor del bin de la entrada, si este es 0 significa que no es el último macrobloque del slice y se ejecuta el proceso de renormalización. Por otro lado, si el valor del bin es 1 y se trata del último macrobloque del slice, se le suma a codILow el valor de codIRange y se ejecuta la función EncodeFlush explicada en 2.4.4.7. Tras llevar a cabo alguna de las posibilidades se incrementa la cuenta de bits en unidades NAL [10].

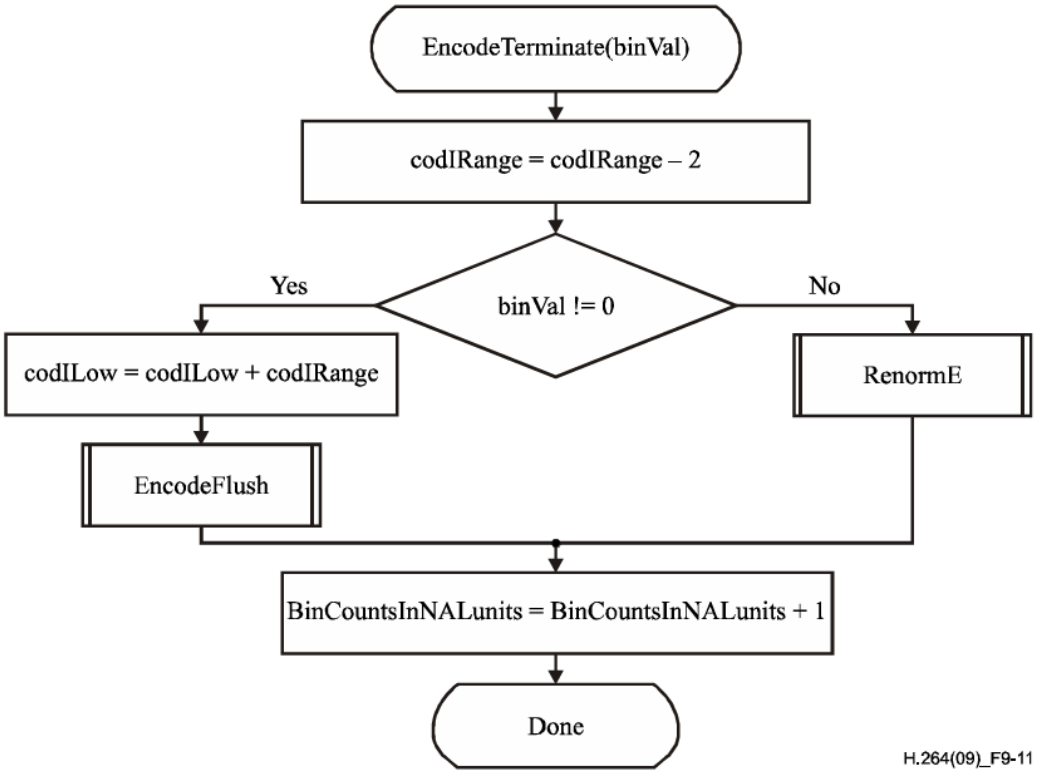

Figura 2.22: Diagrama de flujo del proceso de EncodeTerminate. Fuente [10].

2.4.4.7 Proceso EncodeFlush

Como se indicó en 2.4.4.6, el proceso EncodeFlush se lleva a cabo cuando se finaliza la codificación de todos los elementos sintácticos del último macrobloque del slice.

Este proceso comienza dándole el valor 2 a codIRange, posteriormente lleva a cabo el proceso de renormalización explicado en 2.4.4.3. Seguidamente se procede a llevar a cabo el proceso de PutBit explicado en 2.4.4.4, dándole como entrada al bit 9 de codILow. Cabe señalar que los índices de los bits se consideran en orden decreciente, de izquierda a derecha. Por último, se escribe al bitstream el bit 8 de codILow seguido de un '1' [10].

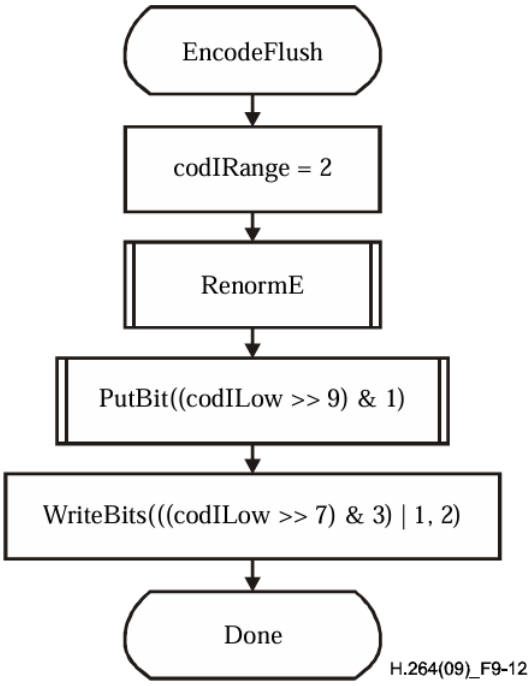

Figura 2.23: Diagrama de flujo del proceso EncodeFlush. Fuente [10].

Capítulo 3: Diseño e implementación de la arquitectura

3.1 Introducción

Habiendo explicado el funcionamiento del estándar H.264 y más concretamente del compresor CABAC, en este capítulo se expone la solución implementada. En esta solución se ha desarrollado una versión hardware de CABAC, capaz de procesar coeficientes de macrobloques codificados usando predicción intra 4 e intra 16 de imágenes monocromáticas. Esta arquitectura se basa y amplia la versión presentada [4] que se desarrolló para codificar únicamente macrobloques de intra 4.

En el capítulo se explicará de forma extendida el punto de partida de la arquitectura y los cambios realizados para poder implementar la posibilidad de codificar ambos tipos de coeficientes.

3.2 Tipos de coeficientes en la entrada de la arquitectura

Como se comentó en 2.3.2, cuando se trabaja con predicción intra 16, se trabaja con dos tipos de coeficientes, los coeficientes AC y DC. En CABAC se procesan ambos coeficientes de la misma manera, solo teniendo en cuenta que primero se deben codificar los coeficientes AC y posteriormente los DC. También se debe tener en cuenta que al separar los coeficientes tenemos a la entrada de la arquitectura una matriz de 4×4 de coeficientes DC y 16 matrices de 15 coeficientes AC cada una.

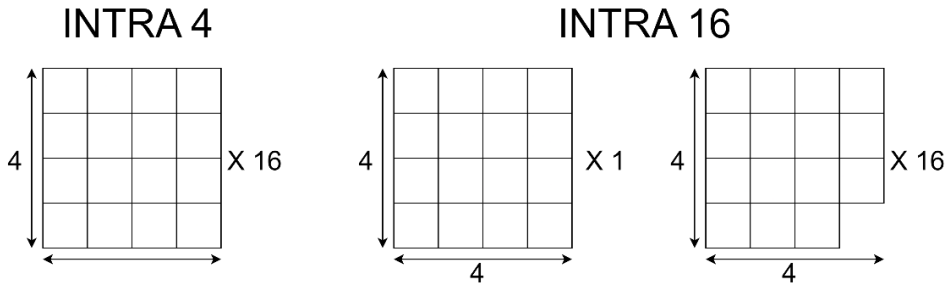

Figura 3.1: Coeficientes en la entrada de CABAC según el tipo de predicción usada en su codificación.

3.3 Arquitectura

En la Figura 3.2 se puede apreciar la arquitectura implementada, compuesta por un total de 10 bloques diferentes con sus distintas funciones. Como se comentó anteriormente, esta arquitectura parte de la implementada por David Martín para su TFT explicada en [4], que procesaba coeficientes codificados con predicción intra 4. En la nueva versión, se amplía la funcionalidad para permitir también el procesamiento de coeficientes con predicción intra 16. Para ello, se modificaron todos los bloques de la arquitectura de [4] menos BAC (acrónimo de Binary Arithmetic Coder) y el escritor de bits. La nueva implementación tiene un total de 14 entradas y 10 salidas que se explican en 3.3.1 y 3.3.2 respectivamente.

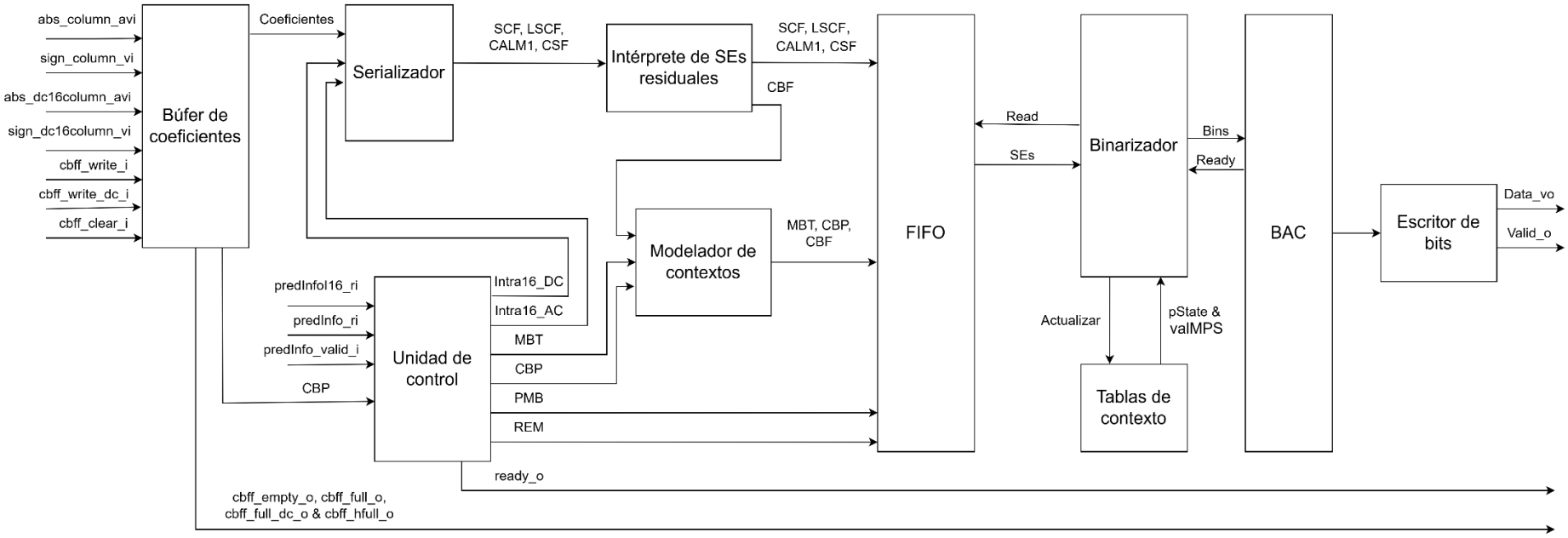

Figura 3.2: Arquitectura de CABAC. Adaptada de [4]

3.3.1 Entradas

Las entradas de la arquitectura tienen distintos sufijos que indican su formato:

- _i entrada de un bit.
- _ui entrada unsigned.
- _vi entrada std_logic_vector.
- _avi entrada array de std_logic_vector.
- _ri entrada tipo record.

A continuación, se presentan las definiciones de cada una de las entradas:

clk_i: clock para sincronizar con el reloj de la FPGA.

rst_n_i: reset para poder reiniciar el sistema y establecer todas las señales a sus valores predeterminados.

valid_qp_i: indica si el valor de la entrada slice_qp_ui es válida o no.

slice_qp_ui: valor de QP del slice que se está codificando.

predInfoI16_ri: indica si la predicción más eficiente es intra 16 o intra 4, y que nos indica en caso de que sea intra 16 el modo de predicción, es decir, horizontal, vertical o DC.

predInfo_ri: indica el modo de predicción (horizontal, vertical o DC) que se usó en la predicción de los distintos bloques 4×4 en caso de que los coeficientes hayan sido codificados con predicción intra 4.

predInfo_valid_i: indica si predInfo_ri es válida.

abs_column_avi: Array de cuatro coeficientes en valor absoluto correspondientes a una columna de un bloque 4×4 de predicción intra 4 o a una columna de un bloque 4×4 de coeficientes AC de predicción intra 16.

sign_column_vi: signos de los cuatro coeficientes de abs_column_avi.

cbff_write_i: indica que hay nuevos valores válidos en las entradas abs_column_avi e sign_column_vi.

abs_dc16column_avi: similar a abs_column_avi pero para los coeficientes DC de intra 16.

sign_dc16column_vi: signos de los 4 coeficientes de abs_dc16column_avi.

cbff_write_dc_i: similar a cbff_write_i, pero para las entradas abs_dc16column_i, sign_dc16column_vi.

cbff_clear_i: indica que debemos eliminar los coeficientes que nos habían llegado debido a que vamos a usar predicción intra 16 en vez de intra 4.

3.3.2 Salidas

Las salidas de la arquitectura son las siguientes:

ready_o: indica al resto de módulos del diseño que la arquitectura está lista para recibir coeficientes.

cbff_empty_o: indica que no se han recibido coeficientes desde el último macrobloque.

cbff_full_o: indica que el bloque buffer de coeficientes ha recibido todos los coeficientes AC de un macrobloque que ha usado predicción intra 16, o todos los coeficientes de un bloque que ha usado predicción intra 4, y que no está listo para recibir más datos hasta terminar de procesarlos.

cbff_full_dc_o: indica el bloque buffer de coeficientes ha recibido todos los coeficientes DC de un macrobloque que ha usado predicción intra 16, y que no está listo para recibir más hasta terminar de procesarlos.

cbff_hfull_o: indica que el bloque buffer de coeficientes ha recibido la mitad de los coeficientes AC de un macrobloque que ha usado predicción intra 16, o la mitad de los coeficientes de un bloque que ha usado predicción intra 4.

valid_o: indica si data_vo es válida o no.

data_vo: bitstream de salida de CABAC tiene longitud de 8 bits, es decir, un byte.

3.3.3 Búfer de coeficientes

Los módulos encargados de realizar la parte de predicción del estándar no saben qué tipo de predicción es más eficiente hasta haber procesado el bloque con predicción intra 4. Cuando se llega a ese punto, el módulo de predicción determina si va a ser más eficiente usar predicción intra 4 o intra16. Por este motivo, se creó el bloque búfer de coeficientes dentro de CABAC, para almacenar los coeficientes hasta conocer qué tipo de predicción es más eficiente.

Este bloque recibe los coeficientes de predicción intra 4 de cuatro en cuatro por la entrada abs_column_i y los va a almacenando en módulos FIFO (acrónimo de First In First Out), que son estructuras de memoria que proporcionan los datos a la salida con el mismo orden en el que entraron.

Cuando el módulo de predicción del estándar sabe qué tipo de predicción es más eficiente, envía la señal predInfoI16_ri a la arquitectura de CABAC. Esta entrada está conformada por los siguientes elementos:

- **i16best**: indica si la predicción intra 16 es la que se va a utilizar, o si por el contrario se va a utilizar predicción intra 4.
- **I16predmod_v**: en caso de que se use predicción intra 16 indica el tipo de predicción (H, V o DC).
- **Valid**: indica si los valores de i16best y i16predmod_v son válidos.

El bloque encargado de recibir esta entrada es la unidad de control, explicada en 3.3.5, que gestiona el valor de sus elementos. Cuando estos indican que se va a usar predicción intra 4, la unidad de control gestiona que se reciban los últimos coeficientes del macrobloque y se codifiquen todos en el resto de los bloques de la arquitectura. Por el contrario, cuando se recibe que se va a usar predicción intra 16 se procede a vaciar la FIFO que había guardado los coeficientes con predicción intra 4 y se procede a almacenar los coeficientes AC y DC de predicción intra 16 en sus respectivas FIFOs.

Asimismo, el módulo búfer de coeficientes obtiene el valor de CBP explicado en 2.4.1.2. Este SE se calcula comprobando el valor de cada uno de los coeficientes que entran en el bloque. En caso de que el valor de alguno de los coeficientes sea distinto de cero, si se trata de predicción intra 4, se obtiene el bloque 8×8 al que pertenece el coeficiente y se le añade un uno en la posición correspondiente del CBP. Por el contrario, si se trata de un macrobloque codificado con predicción intra 16, basta con que haya un solo coeficiente distinto de cero en todo el macrobloque para poner todas las posiciones del CBP a uno.

Cabe recalcar que los coeficientes se obtienen de cuatro en cuatro a través de la entrada abs_column_i. Por lo tanto, al calcular el valor de CBP, el bloque búfer de coeficientes procesa estos cuatro coeficientes en paralelo, aplicando el mismo procedimiento mostrado en Figura 3.3 para los cuatro valores recibidos en el búfer de coeficientes.

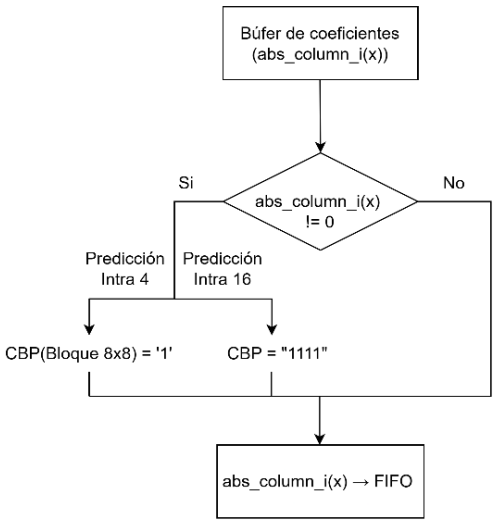

Figura 3.3: Diagrama de flujo explicando el procedimiento de CBFF cuando recibe abs_column_i.

Nota: La variable x apunta a alguno de los coeficientes del array de la entrada abs_column_i, esta variable puede ser un valor entre cero y 3.

3.3.3.1 Modificaciones respecto a la arquitectura original

Se ha adaptado el bloque para que sea capaz de obtener el valor correcto de CBP cuando se procesan coeficientes de un macrobloque codificado con predicción intra 16. Ya que en este caso, con que haya un coeficiente distinto de 0 todos los elementos de CBP pasan a valer ‘1’. Asimismo, se han incluido las entradas abs_dc16column_avi, sign_dc16column_vi y cbff_write_dc_i en el bloque para recibir los coeficientes DC de macrobloques intra 16.

3.3.4 Serializador

El módulo serializador tiene la tarea de reordenar los coeficientes para que sean escaneados como indica el estándar, pero con orden invertido para poder calcular el SE LSCF, es decir el que nos indica cual es el último coeficiente significativo. Ya que el módulo CBFF recibe columnas de coeficientes de derecha a izquierda y necesita mandarlos con otro orden como se muestra en la Figura 3.4.

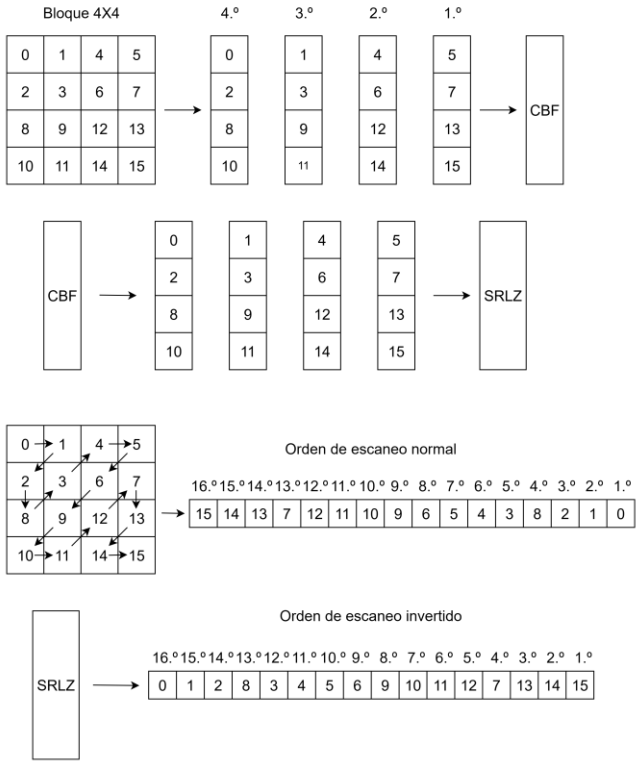

Figura 3.4: Demostración gráfica del orden de los coeficientes en la entrada y salida de los módulos CBFF y SRLZ.

3.3.4.1 Modificaciones respecto a la arquitectura original

Se han creado varias entradas indicando qué conjunto de coeficientes va a ser procesado en el bloque. Pudiendo tratarse de coeficientes de un macrobloque intra 4, o coeficientes DC o AC de un macrobloque intra 16. Se necesita saber el tipo de bloque 4×4 que va a ser procesado, debido a que en el caso de que se procesen coeficientes intra 16 AC se debe de tener en cuenta que en cada bloque 4×4 hay 15 coeficientes. Por lo tanto, el bloque tiene que estar preparado para que en estos casos solo mande los 15 coeficientes con el orden de escaneo inverso. En la figura siguiente se puede apreciar esta disposición, donde también se pueden observar las diferencias respecto al caso de intra 4 mostrado en la Figura 3.4.

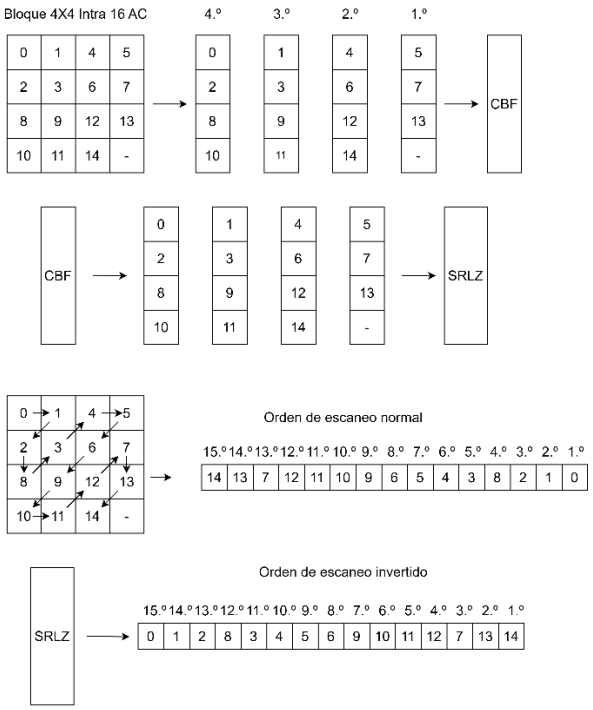

Figura 3.5: Orden de escaneo asignado en el bloque serializador para bloques 4×4 Intra 16 AC.

Además, se han implementado salidas en el bloque que indican al resto de módulos el tipo de bloque 4×4 al que pertenece cada coeficiente. Ya sea perteneciente a un bloque 4×4 de un macrobloque codificado con predicción intra 4 o bloques DC o AC de un macrobloque codificado con predicción intra 16.

3.3.5 Unidad de control

Para finalizar con los primeros bloques de la arquitectura, la unidad de control se encarga de dirigir el módulo búfer de coeficientes, indicándole cuándo debe llevar a cabo sus distintas funciones, como eliminar coeficientes o enviar los coeficientes al serializador. Asimismo, es el bloque encargado de saber en qué momento se codifica intra 4 o intra 16, y comunicárselo al resto de módulos.

El bloque opera con una MEF (acrónimo de Máquina de Estados Finitos), que tiene un total de 9 estados como se puede apreciar en la Figura 3.6. El estado inicial se llama IDLE_E, en este estado el bloque está esperando a recibir el modo de predicción de los 16 bloques que componen el macrobloque codificado con predicción intra 4×4 por la entrada predInfo_ri, para guardarlos en un array. Ya que no se sabe el tipo de predicción que se va a usar, si intra 16 o intra 4, hasta que se reciban todos los modos de predicción.

Cuando se han recibido todos los modos de predicción, se confirma que el macrobloque se va a codificar usando predicción intra 4, y el sistema pasa al estado “I4_LASTB4_E”. En cambio, si antes de recibir el último modo de predicción del macrobloque intra 4 se determina que es más eficiente usar predicción intra 16, el módulo recibe la señal predInfoI16_ri, que nos indica el modo de predicción y se procede a pasar al estado “I16_CBFF_E”.

El estado “I4_LASTB4_E” está diseñado para esperar a que llegue el último bloque de coeficientes del macrobloque intra 4×4, cuando llegan los coeficientes se pasa al estado “I4_CBP_E”. En este estado se obtiene el CBP calculado por el búfer de coeficientes y se envía a los módulos Intérprete de SEs residuales y modelador de contextos. Se hace en este momento debido a que en este estado ya han sido procesados todos los coeficientes en búfer de coeficientes y sabemos que el valor de CBP es correcto.

Una vez enviado el CBP se pasa al estado “I4_E”. En este estado, la unidad de control le indica al búfer de coeficientes si el bloque serializador aún está procesando coeficientes y, por lo tanto, no debe enviar nuevos datos, o, por el contrario, si el bloque serializador ya ha procesado los coeficientes anteriores y está listo para recibir más. Una vez mandados todos los coeficientes al bloque serializador se retorna al estado IDLE.

Por otro lado, el estado “I16_CBFF_E”, está diseñado para esperar a que le lleguen todos los coeficientes AC y DC del macrobloque codificado con intra 16 al bloque búfer de coeficientes. Asimismo, en este estado, la unidad de control le indica a este bloque que se está codificando un macrobloque con predicción intra 16 y por lo tanto debe calcular el SE CBP de distinta manera a como lo hace con intra 4.

Cuando se han recibido todos los coeficientes se pasa al estado “I16_CBP_E”, donde se obtiene el valor de CBP calculado por el bloque búfer de coeficientes y se procede a obtener el valor de MBT del macrobloque, ya que este valor es dependiente de su CBP. Una vez obtenido este valor se manda el CBP y el MBT a los módulos siguientes para que estos sean procesados y se pasa al estado “I16_SRLZ_DC_E”.

En el estado “I16_SRLZ_DC_E”, se le indica al módulo búfer de coeficientes que ya puede enviar los coeficientes DC a serializador, mientras este esté preparado para recibir. Una vez enviados todos los coeficientes DC se cambia al estado “I16_SRLZ_AC_E”, en este estado se realiza lo mismo que en el estado “I16_SRLZ_DC_E”, pero con coeficientes AC. Una vez enviados estos últimos se pasa al estado “I16_SRLZ_FINISH_E” en el que se espera a que serializador termine de procesar el resto de los coeficientes que le ha mandado búfer de coeficientes. Una vez finalizado se retorna al primer estado “IDLE_E”. En el diagrama mostrado en la Figura 4.6 se pueden apreciar las distintas fases de la MEF.

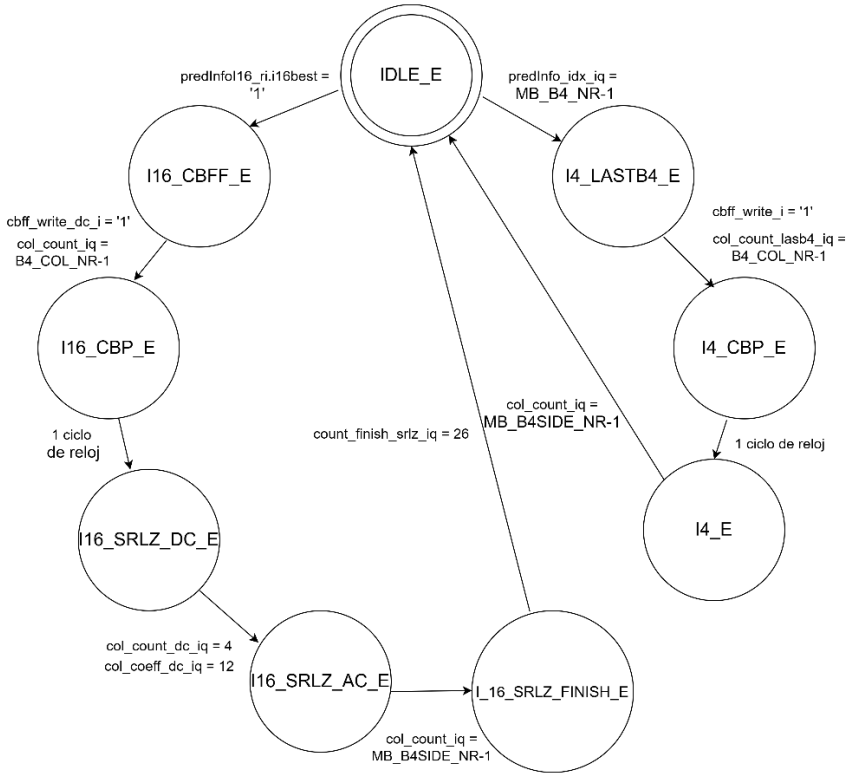

Figura 3.6: Diagrama de flujo del comportamiento de la MEF del bloque unidad de control.

Las señales que se muestran en él son:

- **Col_count_iq**: se usa para contar el número de columnas de un bloque 4×4. En el estado "I16_CBFF_E" sirve para saber cuándo se han terminado de mandar los coeficientes DC y AC ya que los DC son los últimos en ser recibidos. Asimismo, es usado en el estado "I16_SRLZ_AC_E" para contar que se manden las columnas de todos los bloques de 4×4 de coeficientes AC del macrobloque. A su vez es usado en el estado I4_E para comprobar que se mandan todos los coeficientes del bloque de búfer de coeficientes al bloque serializador.
- **Col_count_dc_iq**: creada para esperar a que se manden las 4 columnas del bloque de coeficientes DC del bloque búfer de coeficientes a serializador.
- **Col_coeff_dc_iq**: usada en el estado "I16_SRLZ_DC_E" para contar los coeficientes DC que son procesados. Se espera a llegar a 12 porque, mientras se mandan las columnas del bloque búfer de coeficientes al serializador, también se van procesando coeficientes; por lo tanto, le quedan otros 12 coeficientes, ya que en ese momento ya se han codificado cuatro.
- **Count_finish_srlz_iq:** usada en "I_16_SRLZ_FINISH_E" para contar el resto de los coeficientes que faltan por procesar en el SRLZ.
- **predInfo_idx_iq**: cuenta que lleguen todos los modos de predicción de los bloques 4×4 que forman el macrobloque de intra 4.
- **Col_count_lastb4_iq**: se encarga de contar que entren las últimas columnas de coeficientes en CBFF para proceder al estado I4_CBP y estar seguros de que tenemos calculado el CBP correctamente.

3.3.5.1 Modificaciones respecto a la arquitectura original

Se ha adaptado el bloque para que obtenga el valor del SE MBT siguiendo las condiciones que se establecen en el estándar. Debido a que la arquitectura original usa solo un valor de MBT de forma constante, los macrobloques con predicción intra 4 solo tienen un valor de este SE.

En la siguiente tabla se pueden visualizar los distintos valores de MBT dependiendo del tipo de predicción y el valor de CBP. Esta tabla se ha adaptado ya que en la tabla original se tiene en cuenta también el valor de CBP relacionado a los coeficientes de crominancia (color), pero la arquitectura de este TFT solo procesa vídeo monocromo es decir sin coeficientes de crominancia.

Valor de MBT	**Tipo de predicción**	**Modo de predicción**	**CBP**
0	Intra 4	-	-
1	Intra16	V	0
2	Intra16	H	0
3	Intra16	DC	0
13	Intra16	V	1
14	Intra16	H	1
15	Intra16	DC	1

Tabla 3.1: Valores de MBT relacionados con el tipo de predicción, modo de predicción y valor de CBP. Adaptada de [10] .

Asimismo, se han creado todos los estados relacionados con los macrobloques intra 16 explicados en 3.3.5. Además, se han creado las salidas que le indican al bloque búfer de coeficientes cuándo debe eliminar los coeficientes de intra 4 para recibir los coeficientes de intra 16, así como las salidas que indican al bloque serializador qué tipo de bloque 4×4 se está procesando.

3.3.6 Intérprete de SEs residuales

El bloque intérprete de SEs residuales, como el propio nombre indica, es el encargado de obtener los valores de los SEs residuales explicados en 2.4.1.2. Asimismo, también se encarga de obtener el valor de la variable ctxIdxIncrement asociada a CALM1 y asignar el valor de ctxBlockCat a cada bloque procesado, para su posterior uso en el cálculo de ctxIdx explicado en el subapartado 2.4.3.2.

Este bloque obtiene el valor de varios SEs de manera simultánea, para ello en primer lugar se comprueba el valor de CBP. Cuando este SE nos indica que el bloque 8×8 relacionado al coeficiente que se está codificando tiene algún elemento no nulo, se procede a analizar su valor y asignarles valores a sus respectivos SEs. Para ello se crean variables a las que se les dan los siguientes valores:

- CALM1: valor absoluto del coeficiente menos uno.
- CSF: signo del coeficiente.
- SCF: '1' si el coeficiente es distinto de cero, '0' en caso contrario.
- LSCF: '1' si el coeficiente es el último elemento significativo del bloque 4×4 al que pertenece, '0' en caso contrario.

Asimismo, también se le asigna valor al SE CBF, que será '1' si algún coeficiente de un bloque 4×4 es distinto de cero, '0' en caso contrario. Además, se debe de tener en cuenta que el codificador CABAC usa modo de escaneo inverso, por lo tanto, el primer coeficiente que sea distinto de cero de un bloque 4×4 será el que tenga el LSCF a '1'.

Por otra parte, el último coeficiente de cada bloque 4×4 no tiene asignados los SEs SCF ni LSCF. Esto se debe a que esta información es redundante, ya que el CBF nos indica si hay o no coeficientes distintos de cero en el bloque y LSCF si el coeficiente en el orden de escaneo es el último no nulo o no. Por lo tanto, si el último elemento del bloque es el último distinto de cero, se sabrá debido a que el CBF es '1' y, antes de llegar al último coeficiente, no ha habido otro LSCF igual a uno. Mientras que, si es el último y es igual a cero, también se sabrá debido a que, antes de llegar a este, ha habido otro LSCF igual a uno. Asimismo, se sabrá también su SCF, ya que va relacionado con el valor de LSCF.

En la Figura 3.7 se puede apreciar lo explicado en este párrafo, en el caso de la izquierda, el último coeficiente del bloque es el último elemento distinto de cero. Mientras que en el bloque de la derecha el último elemento del bloque es igual a cero.

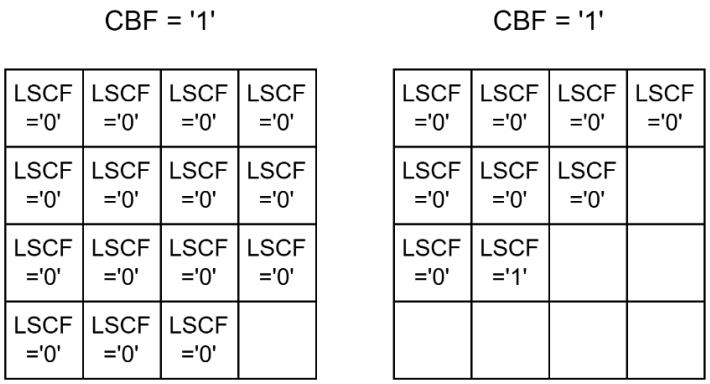

Figura 3.7: Demostración de la redundancia de LSCF en el último coeficiente de un bloque 4×4.

En el bloque interprete de SEs residuales se calculan los ctxIdxIncrements de los 14 primeros bins que componen el SE CALM1. Se calculan solo hasta el 14 debido a que el resto de bins son codificados en modo Bypass, es decir, sin contexto. Asimismo, el ctxIdxIncrement del bin 0 se calcula con una formula distinta al resto de bins. Para el cálculo de ambos se debe tener en cuenta que la variable numDecodAbsLevelGt1 indica el

número de coeficientes previos dentro del mismo B4 con valor absoluto mayor a 1. Mientras que la variable numDecodAbsLevelEq1 indica el número de coeficientes previos dentro del mismo B4 con valor absoluto igual a 1 [10].

Fórmula para el cálculo de ctxIdxIncrement para bin 0:

- **ctxIdxIncrement = ((numDecodAbsLevelGt1 ≠ 0)? 0 : Min(4,1+ numDecodAbsLevelEq1))**

Fórmula para el cálculo de ctxIdxIncrement para bin ϵ [1,13]:

- **ctxIdxIncrement = 5 + Min (4 − ((ctxBlockCat = = 3)? 1 : 0), numDecodAbsLevelGt1)**

Como en esta implementación no se da el caso en el que ctxBlockCat es 3 como se explica en 2.4.3.5 la fórmula queda así:

- **ctxIdxIncrement = 5 + Min (4, numDecodAbsLevelGt1)**

Para el mejor entendimiento del funcionamiento del bloque intérprete de residuales, a continuación, se muestra un ejemplo. Supongamos que se va a procesar el bloque 4×4 siguiente:

23	- 7	2	1
- 3	- 4	0	- 1
0	0	0	0
1	0	- 1	1

Figura 3.8: Bloque 4×4 inicial del ejemplo del proceso del bloque Interprete de SEs residuales.

En la Figura 3.9 se puede apreciar cómo el bloque procesa los coeficientes. El primer coeficiente en entrar en el bloque es el que tiene valor 1, que se encuentra abajo a la derecha, y el último, el de valor 23, siguiendo el orden de codificación inverso comentado anteriormente. El primer SE es CBF, que tiene valor '1' debido a que hay por lo menos un coeficiente distinto de cero. El siguiente SE es SCF, la cual está asignada a todos los coeficientes e indica con un '1' si su valor es distinto de cero y con un '0' si es igual a cero.

Por otro lado, a los coeficientes con SCF igual a '1' se les asigna valores a sus SEs LSCF, CALM1 y CSF. Como se mencionó anteriormente, el primer coeficiente en orden de escaneo inverso, es decir, el último elemento del bloque 4×4 no tiene asignado ni SCF ni LSCF, ya que esta información es redundante. Sin embargo, sí se le asignan valores a sus CALM1 y CSF relacionados.

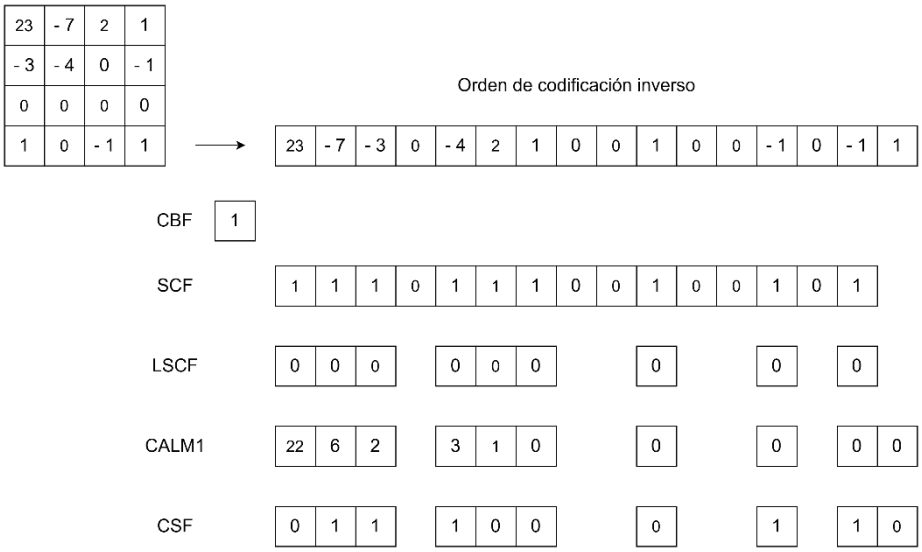

Figura 3.9: Asignación de SEs residuales al bloque ejemplo.

El siguiente proceso que se lleva a cabo es la asignación del valor del ctxIdxIncrement de los 14 primeros bins del SE CALM1, esto se lleva a cabo con el uso de las fórmulas comentadas anteriormente. En la imagen se pueden apreciar los valores que van tomando tanto las variables numDecodAbsLevelGt1 y numDecodAbsLevelEq1como los valores asociados a cada coeficiente bin0_ctxIdxIncrement y bin1-13_ctxIdxIncrement.

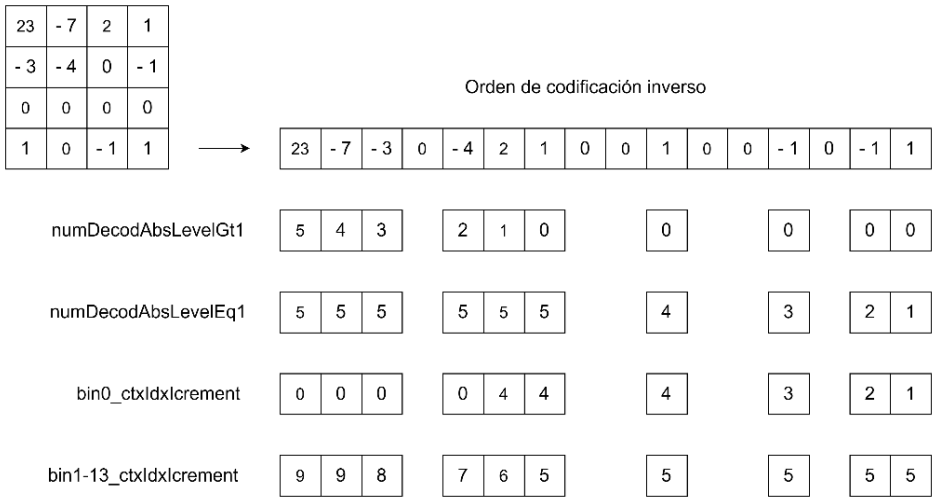

Figura 3.10: Ejemplo de asignación de los distintos valores relacionados al ctxIdxIncrement de CALM1.

3.3.7 Modelador de contextos

El bloque modelador de contextos es el encargado del cálculo de los valores de ctxIdxIncrement relacionados a CBP, CBF y MBT. CBP tiene un total de 4 bins, cada uno representa un bloque 8×8 de un macrobloque, y cada uno de ellos tiene su propio ctxIdxIncrement. Para calcular el valor de ctxIdxIncrement se necesita obtener el valor de los bloques 8×8 que se encuentran encima (B) y a la izquierda (A) del bloque correspondiente al bin de CBP que se está procesando [10]. En la Figura 2.13 se puede observar un ejemplo visual de a qué bloques corresponden A y B.

Una vez obtenido el valor de estos bloques vecinos, se procede a asignar un valor a la variable condTermFlagN (donde *N* puede ser A o B, dependiendo del bloque vecino correspondiente). Esta variable se utiliza posteriormente en la fórmula para calcular el ctxIdxIncrement [10].

El valor de condTermFlagN se establece en 0 si se cumple alguna de las siguientes condiciones [10]:

- El macrobloque N no está disponible, es decir, el bloque vecino no existe (por ejemplo, porque el macrobloque actual está pegado a los bordes de la imagen y por ello no tiene vecinos).
- El bloque 8×8 vecino tiene coeficientes distintos de cero, independientemente de si este pertenece al macrobloque actual o a un macrobloque vecino.

Si no se cumple ninguna de las condiciones citadas, el valor de condTermFlagN se establece a 1. Una vez obtenido el valor de ambos condTermFlag se procede a calcular el valor de ctxIdxIncrement a través de la siguiente formula extraída de [10]:

- $ctxIdxInc = condTermFlagA + 2 \times condTermFlagB$

En la siguiente imagen se pueden apreciar varios casos en los que se obtiene el valor de ctxIdxIncrement a partir del valor de CBP relacionado a los bloques 8×8 que están encima y en el lado izquierdo del bloque que se está procesando.

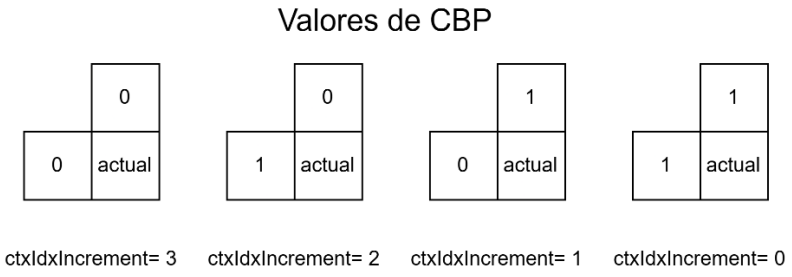

Figura 3.11: Ejemplos de valores de la variable ctxIdxIncrement del SE CBP.

El cálculo del valor de ctxIdxIncrement asociado a CBF es bastante parecido, se basa también en los valores de CBF de los bloques vecinos, pero esta vez en bloques 4×4. El cálculo sigue dependiendo de la variable condTermFlagN (siendo N el vecino A o B). Pero además el valor de ctxIdxIncrement también dependerá del tipo de bloque que se está codificando, es decir, de su ctxBlockCat. Ya que se van a codificar bloques de intra 4×4 y bloques AC y DC de intra 16×16, en todos estos casos el valor de condTermFlagN será 0 cuando:

- El macrobloque está disponible, pero el bloque 4×4 no, esta condición se da cuando se está codificando un bloque intra 16×16 y los macrobloques que lo rodean son intra 4×4. Debido a que a la hora de codificar el bloque DC, este no tiene bloques DC vecinos, pero sí macrobloques vecinos.

Por otro lado, será 1 cuando:

- El macrobloque N no está disponible.

Si no se cumplen ninguna de las dos condiciones, condTermFlagN toma el valor de coded_block_flag del bloque 4×4 N que fue decodificado para el macrobloque N [10]. Finalmente se aplica la formula, que en este caso depende de ctxBlockCat:

- ctxIdxInc(ctxBlockCat) = condTermFlagA + 2 × condTermFlagB

Para finalizar, el cálculo de ctxIdxIncrement de MBT se lleva a cabo de la misma manera, pero en este caso condTermFlag se establece a 0 cuando:

- El macrobloque N tiene predicción intra 4.

Si no se cumple la condición, el valor de condTermFlag es 1. Una vez obtenido el valor de condTermFlag se aplica la fórmula:

- ctxIdxInc = condTermFlagA + condTermFlagB

Para llevar a cabo estos procedimientos se crean una serie de arrays en los que se almacenan los distintos valores de MBT, CBP y CBF correspondientes a los bloques que se van procesando.

El tamaño de estos arrays varía según el tipo de dato. En el caso de los arrays de MBT y CBP, el número de elementos que contienen corresponde al número máximo de macrobloques en la dimensión horizontal de la resolución utilizada. Es decir, su tamaño es equivalente al número de macrobloques por fila del fotograma, lo que sería equivalente a dividir el número total de columnas entre 16, ya que cada macrobloque ocupa 16 pixeles por columna. Esta cantidad es la necesaria para poder acceder a la información de los bloques situados encima y a la izquierda del bloque que se está procesando.

En la Figura 3.12Figura 3.12 se muestra un ejemplo, los macrobloques marcados con líneas representan aquellos cuyos valores se mantienen almacenados en los arrays, ya que serán necesarios para el cálculo de futuros ctxIdxIncrement de otros macrobloques. Por el contrario, los macrobloques que no presentan líneas ya no son requeridos, por lo que sus valores han sido eliminados del array.

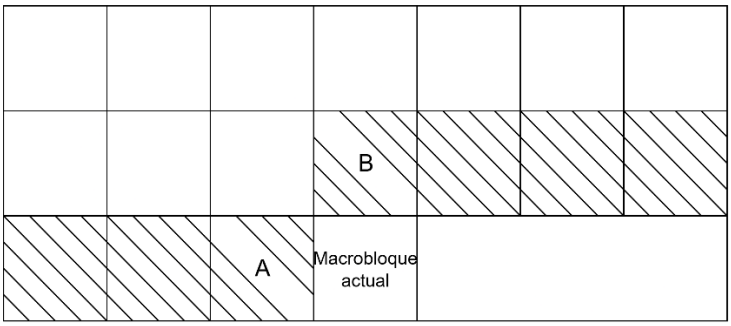

Figura 3.12: Valores almacenados en el array para el cálculo del ctxIdxIncrement de MBT y CBP.

Por otro lado, para realizar el cálculo del valor de ctxIdxIncrement de CBF se utilizan varios sistemas de almacenamiento. El primero es un array que va a almacenar los valores de CBF de bloques ya codificados dentro del mismo macrobloque que se está codificando. Mientras que para almacenar los valores de los macrobloques vecinos se va a utilizar un array encargado de almacenar los valores de los bloques (10, 11, 14, 15) que van a ser usados cuando se comprueben las condiciones del bloque B. Y un vector que almacene los valores del bloque que se codificó justo antes del que se está procesando ahora (5, 7, 13, 15) que van a ser usados cuando se comprueben las condiciones del bloque A.

En la Figura 3.13 se puede apreciar un ejemplo de qué valores de bloques serían almacenados para el cálculo de ctxIdxIncrement del SE CBF. En este caso se pone un ejemplo en el que se está codificando el bloque 10 del último macrobloque que se puede apreciar. Los bloques marcados con líneas son los bloques que aún se guardan en sistemas de almacenamiento para calcular los valores de ctxIdxIncrement.

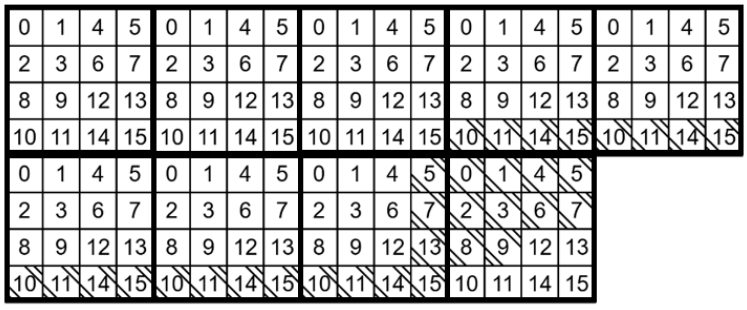

Figura 3.13: Valores de CBF almacenados para el cálculo de ctxIdxIncrement.

3.3.7.1 Modificaciones respecto a la arquitectura original

Se ha adaptado el bloque para que calcule el parámetro ctxIdxIncrement del SE MBT, ya que en la arquitectura original este valor era constante. El cálculo de este está explicado en 3.3.7.

3.3.8 FIFO y Tablas de contexto

El siguiente bloque dentro de la arquitectura desarrollada es el bloque FIFO. Este se encarga de almacenar los distintos SEs, junto con sus correspondientes valores de ctxIdxIncrement obtenidos en los bloques anteriores, hasta que el bloque binarizador esté preparado para procesarlos.

En esta arquitectura, el bloque FIFO ha sido modificado respecto a la arquitectura original mediante la creación de módulos adicionales de memoria, con el objetivo de poder almacenar el valor del SE MBT.

Por otro lado, el bloque de tablas de contexto se encarga de inicializar las tablas de contexto al empezar la codificación, así como de almacenarlas y actualizar sus valores cada vez que se codifica un nuevo bin. En este bloque se han añadido las tablas de contexto necesarias para permitir la codificación de coeficientes pertenecientes a macrobloques codificados con predicción *Intra 16*.

3.3.9 Binarizador

El bloque binarizador es el encargado de llevar a cabo el proceso de binarización y asignar contexto a cada uno de los bins que van a ser codificados con el modo normal del BAC. Este bloque está compuesto por una MEF para poder codificar cada SE en el orden establecido por el estándar. Asimismo, este recibe el valor de un SE junto a su ctxIdxIncrement y ctxBlockCat, lo convierte en bins usando los métodos de binarización expuestos en 2.4.2 según el tipo de SE y le da contexto a cada bin, para finalmente mandar esta información al siguiente módulo (BAC) y ahí ser codificado aritméticamente [4].

En los siguientes subapartados se explican las entradas, salidas y funcionamiento del bloque binarizador.

3.3.9.1 Entradas

Las entradas que recibe el bloque binarizador son las siguientes:

- Entradas con los valores de cada SE junto con entradas que confirman que el valor es válido (mbt_i, pmf_i, rem_i, cbp_i, cbf_i, scf_i, lscf_i, calm1_i).
- Tablas de contexto provenientes del bloque tablas de contexto que van siendo actualizadas a medida que se procesan los SEs junto con las entradas que indican que ya han sido actualizadas para saber que sus valores son válidos.
- Entrada con el valor de QP para pasárselo al bloque tablas de contexto a principio de cada fotograma y crear las tablas de contexto.

- Entrada que indica que el bloque BAC está listo para recibir las salidas del bloque binarizador.

3.3.9.2 Salidas

Las salidas del bloque binarizador son las siguientes:

- Salidas indicando a la FIFO que ya ha procesado el valor del SE que tenía en la entrada para que lo elimine y obtenga el siguiente (mbt_fifo_read_o, predinfo_fifo_read_o, cbp_fifo_read_o, qpd_fifo_read_o, cbf_fifo_read_o, sigmap_fifo_read_o, levelinfo_fifo_read_o).
- Salidas indicando la posición (ctxIdx) y el valor con el que se tiene que actualizar las tablas de contexto.
- Salidas con los valores que necesita el bloque BAC para realizar la codificación aritmética: valor de mps (0 o 1), valor del bin, si se codifica o no en bypass, la fila calculada a través de pState de la tabla rangeTabLPS (ver Figura 2.18) y la salida que indica si el proceso EncodeTerminate se tiene que llevar a cabo, explicado en 2.4.4.6.

3.3.9.3 Funcionamiento

La MEF está diseñada para que en cada estado se codifique un SE, el orden es el establecido por el estándar, en la Figura 3.14 se puede visualizar un diagrama con los estados de la MEF.

- MBT: el primer SE que se codifica es MBT, que indica el tipo de macrobloque que se está codificando. Este SE se binariza mediante las tablas definidas por el estándar (sección 2.4.2.5). Cuando se procesa un macrobloque codificado con predicción intra 4, se asigna el bin '0', mientras que los valores pState y valMPS se obtienen a partir de las tablas de contexto, empleando el valor de la variable ctxIdxIncrement, calculada en el bloque modelador de contextos. Tras esto se pasa al estado PMF.

  Por otro lado, si se codifica un macrobloque codificado con predicción intra 16, se le asignan varios bins según el tipo de macrobloque, y a cada uno de estos bins se les asignan los valores de "pState" y "valMPS" del mismo modo que en el proceso de un macrobloque codificado con predicción intra 4. Tras esto se pasa al estado CBP.

- PMF: en este estado se codifica PMF, que indica si el modo de predicción usado (V, H o DC) para cada bloque 4×4 es el más común o no. Por lo tanto, en este estado se van a codificar 16 PMF ya que el macrobloque tiene un total de 16 bloques 4×4. Este SE se binariza con binarización FL con un cMax igual a uno (ver sección 2.4.2.2), por lo tanto, si su valor es cero se le asigna un bin igual a '0' y si es uno se le asigna un bin igual a '1'.

  Asimismo, este SE tiene la peculiaridad de solo tener un contexto en la lista de contextos por lo tanto se obtiene los valores de "pState" y "valMPS" sin necesidad de cálculos previos del ctxIdx. Para finalizar, si el valor de PMF es igual a uno se

codifica el siguiente PMF o se pasa al estado CBP si no quedan más por codificar, mientras que si es igual a cero se pasa al estado REM.

- REM: en este estado se codifica REM, que indica el modo de predicción usado cuando no se trata del más común. La binarización se realiza mediante el método FL, con un valor cMax igual a siete, esto es debido a que en esta implementación solo se usan 3 modos de predicción (V, H o DC), mientras que en el estándar se usan más.

  Por lo tanto, al ser cMax igual a siete la binarización tiene una longitud fija de tres bins, estos son obtenidos a través del proceso de FL. Asimismo, al igual que PMF, REM solo tiene un valor en su tabla de contexto, por lo que no requiere el uso de la variable ctxIdx para obtener los valores de "pState" y "valMPS". Una vez finalizada la binarización y obtenidos los valores de contexto se retorna al estado PMF si aún quedan bloques 4×4 por procesar, o se avanza al estado CBP si ya este es el último.

- CBP: en este estado se codifica CBP, que indica qué bloques 8×8 del macrobloque contienen coeficientes no nulos. Este SE es binarizado con binarización FL con cMax igual a 15, por lo tanto, su longitud fija será de 4 bins que corresponde a los 4 elementos que se obtienen en el bloque búfer de coeficientes.
  Para obtener los valores de "valMPS" y "pState" se usa la variable ctxIdx que se obtiene a partir de la variable ctxIdxIncrement obtenida en el bloque modelador de contextos.

  Una vez obtenidos los bins y los respectivos valores de "valMPS" y "pState" se avanza al siguiente estado. Si el valor de todos los bins es cero, es decir, no hay coeficientes en ninguno de los bloques 8×8 del macrobloque, se pasa al estado ETT. En cambio, con que haya algún coeficiente en alguno de los bloques se pasa al estado QPD.

- QPD: En este estado se codifica QPD, que indica la diferencia de valor entre el parámetro de cuantización del macrobloque anterior y el actual. Como en esta implementación se mantiene constante la calidad de la imagen, el QP permanece fijo y, por lo tanto, la diferencia entre el QP actual y el QP de referencia es siempre nula. Asimismo, este SE se binariza con codificación unaria, por lo que el valor de su bin siempre es '0'. A su vez, su valor de ctxIdx también es constante para obtener los valores de "valMPS" y "pState". Una vez obtenidos todos los valores se avanza al estado CBF.

- CBF: en este estado se codifica CBF, que indica si hay algún coeficiente distinto de cero en el macrobloque actual o no. Como hay 16 bloques 4×4 en el macrobloque, se codifican 16 valores de CBF. Este SE se binariza con codificación FL, con cMax igual a uno, por lo tanto, si hay coeficientes distintos de cero su bin es igual a '1', mientras que si no hay, su bin es igual a '0'. Los valores de "pState" y "valMPS" se obtienen con el uso de la variable ctxIdxIncrement obtenida en el bloque modelador de contextos y la variable ctxBlockCat, que depende del tipo de bloque que se codifica.

Si el valor del bin de CBF es igual a '1', se se avanza al estado SCF. Por el contrario, si el valor del bin de CBF es igual a '0', se codifica el valor de CBF del siguiente bloque de coeficientes 4×4 en el mismo estado, en el caso de que ya se hayan codificado todos los bloques 4×4 se avanza al estado ETT.

- SCF: en este estado se codifica el valor de SCF de los coeficientes que componen los bloques con un CBF igual a uno. SCF indica si el coeficiente es cero o no, la binarización de este SE se lleva a cabo con la codificación FL, con un cMax igual a uno. Por lo tanto, cuando el coeficiente es igual a cero su bin tiene el valor '0' y si el coeficiente es distinto de cero su bin tiene el valor '1'.

  Para obtener el valor de "pState" y "valMPS", se usan las variables ctxIdxIncrement, que en este caso corresponde con la posición del coeficiente dentro del bloque 4×4, y la variable ctxBlockCat, que depende del tipo de bloque al que pertenece el coeficiente. Una vez obtenidos todos los valores, se avanza al estado LSCF, en el caso de que el valor del bin de SCF sea '1'. En cambio, si el valor del bin es '0', la máquina de estados se mantiene en el estado SCF para procesar el valor del SE del siguiente coeficiente, excepto cuando no queden valores de coeficientes por procesar, en ese caso, se avanza al estado CALM1.

- LSCF: en este estado se codifica el valor de LSCF de los coeficientes con un SCF igual a '1', este SE indica si el coeficiente es el último coeficiente del bloque distinto de cero. Se binariza con la codificación FL, con un cMax igual a 1. Por lo tanto, si es el último coeficiente distinto de cero su bin es '1', en caso contrario su bin es '0'.

  Por otro lado, para obtener el valor de "pState" y "valMPS", se aplica el mismo procedimiento que en el estado SCF. Una vez obtenidos los valores de las variables, se avanza al estado de CALM1 si el coeficiente procesado es el último del bloque 4×4. En caso contrario se retrocede al estado SCF.

- CALM1: en este estado se codifica el valor de CALM1 de todos los coeficientes distintos de cero de cada bloque 4×4. Este SE equivale al valor absoluto del coeficiente menos uno. El SE se binariza con la codificación UEGK, explicada en el subapartado 2.4.2.4. Asimismo, para obtener los valores de "pState" y "valMPS" relacionados con los 14 primeros bins del SE, se usan los valores de ctxIdxIncrement obtenidos en el bloque intérprete de SEs residuales y ctxBlockCat. Las posiciones de bins posteriores a los 14 primeros se codifican en modo Bypass, es decir, sin usar contexto. Una vez procesados todos los bins del SE, se pasa al estado CSF.

- CSF: En este estado se codifica el valor de CSF, que representa el signo del coeficiente. Para binarizarlo se usa la codificación FL, con un valor de cMax igual a uno, por lo tanto, cuando el coeficiente tiene signo negativo se le asigna el bin '1' y cuando tiene signo positivo se le asigna el bin '0'. Por otro lado, este SE se codifica en modo ByPass por lo que no necesita valores de contexto.

Una vez codificado el bin de este SE, la máquina de estados retrocede al estado CALM1 para codificar el siguiente coeficiente o se retrocede al estado CBF cuando se trata del último coeficiente del bloque 4×4, con el fin de codificar el siguiente bloque. Asimismo, si además de ser el último coeficiente del bloque 4×4 también lo es del macrobloque se avanza al estado ETT.

- ETT: en este estado se le indica al siguiente bloque (BAC), que lleve a cabo el proceso de EncodeTerminate explicado en 2.4.4.6. Asimismo, se lleva a cabo un contador para saber cuándo se llega al último macrobloque e indicárselo a BAC también. Finalizado este proceso, la MEF pasa al estado MBT para codificar un nuevo macrobloque.

En la Figura 3.14 se puede visualizar el comportamiento de la MEF de forma gráfica, en esta se han incorporado menos entradas y salidas de las que se usan en el bloque, con el objetivo de facilitar la compresión de esta. En ella se muestran las entradas con los valores de los SEs, las salidas indicando al bloque FIFO que ya puede obtener el siguiente valor.

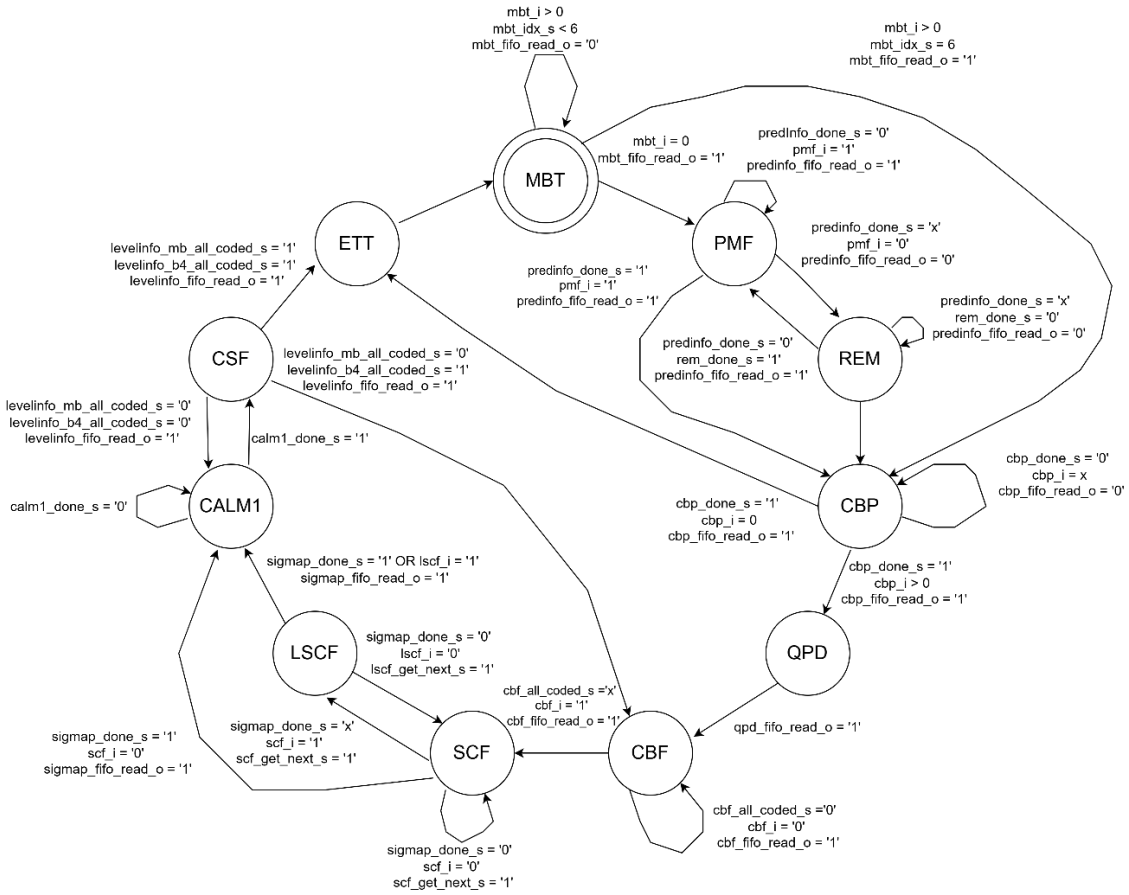

Figura 3.14: Diagrama de estados de la MEF implementada en el bloque binarizador. Adaptada de [4].

Las señales que se usan en el diagrama de la *Figura 3.14* son las siguientes:

- Mbt_idx_s: cuenta el número de bins que se han codificado en MBT.
- Señales con sufijo _done: indican que se ha terminado de codificar ese SE.
- Levelinfo_b4_all_coded_s: indica si se ha finalizado de codificar los coeficientes de todos los b4 de un bloque 16×16.
- Levelinfo_mb_all_coded_s: indica si se ha finalizado de codificar todos los coeficientes del macrobloque.

3.3.9.4 Cambios realizados respecto a la arquitectura original

Se ha modificado el estado MBT para permitir la codificación de los bins de este SE. Ya que anteriormente solo se tenía en cuenta el valor de MBT para macrobloques intra 4, que solo usan un bin como se muestra en la Tabla 2.6. En esta nueva implementación se tiene en cuenta el resto de los casos.

Asimismo, se ha adaptado la máquina de estados para que tenga en cuenta que cuando se trata de un macrobloque con predicción intra 16 no pase por los estados PMF y REM, debido a que estos son exclusivos de intra 4. Además, se han modificado los estados para que se obtenga el valor de ctxIdx a partir de la variable ctxBlockCat. Esta variable se calculada en el bloque Intérprete de SEs residuales para obtener los parámetros de contexto de cada SE.

3.4 BAC

BAC es el bloque encargado de la codificación aritmética binaria, lleva a cabo la mayoría de los procedimientos explicados en 2.4.4, concretamente EncodeDecision(ctxIdx,binVal), renormalización, EncodeBypass(binVal), EncodeTerminate(binVal), EncodeFlush. Asimismo, es el encargado de indicar al último bloque WB (acrónimo de Write Bits) las órdenes para escribir los bits del bitstream. Sin embargo, se debe aclarar que el primer proceso ya ha sido empezado por el bloque binarizador, ya que este obtenía la fila de rangeTabLPS a través del valor de pStateIdx. En los siguientes apartados se explican las entradas y salidas del bloque [4].

3.4.1 Entradas

Las entradas del bloque son:

- Valid_i: indica que el resto de las entradas son válidas, ‘1’ si son válidas y ‘0’ si no.
- Bin_i: el valor del bin a codificar (‘1’ o ‘0’).
- Mps_i: indica cual es el bin más probable, es ‘1’ si el bin introducido se corresponde con el más probable y ‘0’ si no.
- Bp_I: indica si el bin introducido va a codificarse en modo Bypass, ‘1’ si se codifica en Bypass y ‘0’ si no.
- Rlps_row_aii: fila de la tabla range TabLPS asociada al pStateIdx relativo al contexto del bin introducido.

- Ett_i: indica si el bin que llega está relacionado con el proceso de EncodeTerminate, ‘1’ cuando está relacionado y ‘0’ cuando no.

3.4.2 Salidas

Las salidas del bloque son:

- Ready_o: indica a BNRZ si el bloque está listo para recibir un nuevo bin, ‘1’ cuando está listo y ‘0’ cuando no.
- Valid_o: indica a WB si los datos son válidos, ‘1’ si son válidos y ‘0’ si no lo son.
- Write_bits_ro: indica qué bits debe escribir el bloque WB.
- Ett_done_o: indica si se ha terminado de codificar el bin relacionado con el proceso de EncodeTerminate, ‘1’ cuando ha finalizado y ‘0’ cuando no.

3.4.3 Funcionamiento

Este bloque implementa tres Máquinas de Estados Finitos (MEF) destinadas a los procesos de codificación aritmética binaria. Dos de ellas son MEF sencillas, correspondientes a los procesos EncodeDecision y EncodeBypass, explicados respectivamente en 2.4.4.2 y 2.4.4.5, mientras que la tercera es una MEF más compleja que integra los procesos EncodeTerminate y EncodeFlush. Cabe destacar que el algoritmo de renormalización explicado en 2.4.4.3 se implementa tanto en la MEF de EncodeTerminate como en la de EncodeDecision [4].

Esta última MEF consta de un total de 5 estados ETT, ETT_RN, FLUSH, FLUSH RN y FLUSH_WR. El estado inicial es ETT, en este se evalúa si la renormalización está relacionada con la operación EncodeFlush y se trata del final del Slice o si por el contrario no está relacionado y se lleva a cabo el procedimiento de renormalización normal. Cuando se ha evaluado qué tipo de renormalización se lleva a cabo, se lleva a cabo en sus respectivos estados ya sea ETT o FLUSH. Cuando se ejecuta ETT y la renormalización se lleva a cabo en un único ciclo de reloj, la MEF se mantiene en este estado, pero cuando se necesita más de un ciclo, la MEF cambia de estado a ETT_RN para finalizar el procedimiento de renormalización [4].

En el caso de que se ejecute el procedimiento de renormalización durante el proceso EncodeFlush y la MEF transite al estado FLUSH, se aplica el mismo procedimiento que en el estado ETT. Se realiza la renormalización y, si son necesarias más iteraciones, se pasa al estado FLUSH_RN hasta finalizar el proceso, tras lo cual se retorna al estado FLUSH.

Una vez completada la renormalización, se ejecuta el proceso PutBit((codILow ≫ 9) & 1) en el estado FLUSH. A continuación, se pasa al estado FLUSH_WR, donde se ejecuta el proceso WriteBits(((codILow ≫ 7) & 3) | 1, 2) y se activa la señal que indica que esta es la última operación del proceso EncodeFlush. Finalmente, el sistema vuelve al estado ETT [4].

En la Figura 3.15 se puede visualizar el comportamiento de la MEF de forma gráfica, en esta se han incorporado menos entradas y salidas de las que se usan para darle una mayor sencillez. Se muestra la entrada bin_i definida en 3.4.1, la salida ett_done_o definida en 3.4.2.

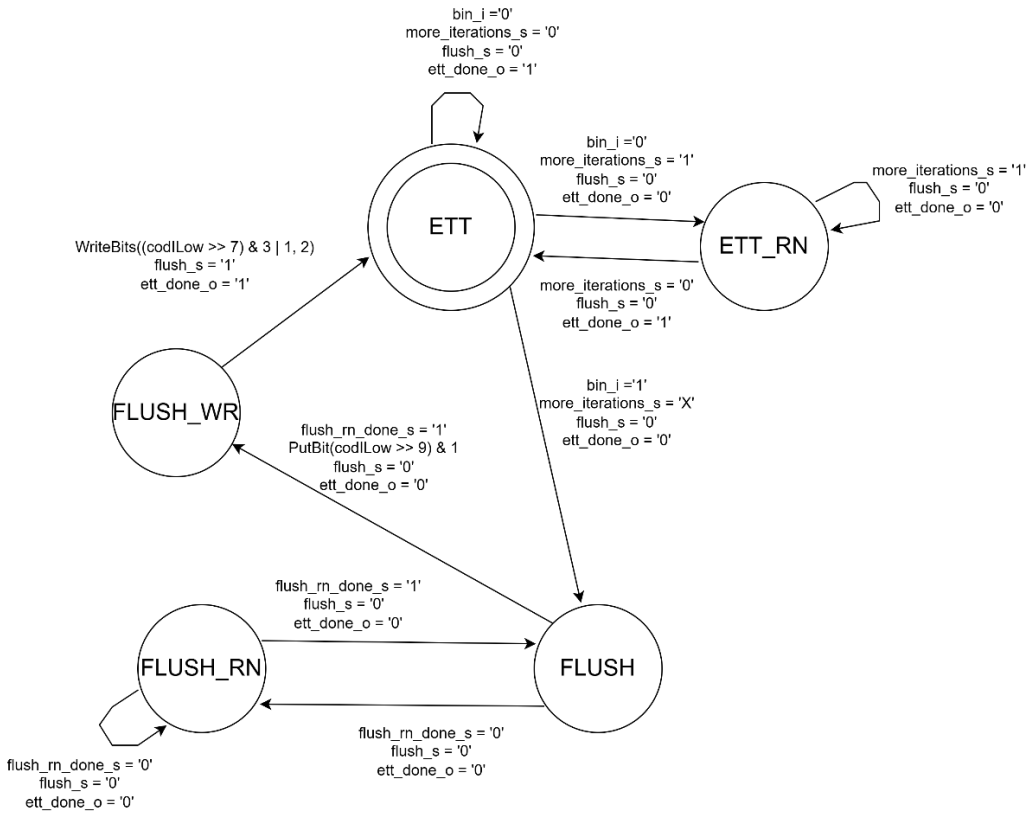

Figura 3.15: Diagrama de estados de la MEF implementada en BAC. Adaptada de [4].

Las señales que se usan en el diagrama son las siguientes:

- more_iterations_s: indica si se requieren más iteraciones de renormalización, si es '1' necesita más iteraciones, '0' en caso contrario.
- flush_rn_done_s: indica si la renormalización asociada a EncodeFlush ha terminado, si es '1' ha terminado, '0' en caso contrario.
- flush s: indica si la orden de escritura generada está asociada a EncodeFlush, si es '1' esta asociada, '0' en caso contrario.

3.5 Escritor de bits

Este es el último bloque de la arquitectura y se encarga de escribir los bits que le indica el bloque de BAC y sacarlos de byte en byte (es decir en agrupaciones de 8 bits) por el bitstream. Este bloque lleva a cabo las implementaciones de los procesos PutBit y WriteBits explicados en 2.4.4. En resumen, WriteBits(B, N) se encarga de escribir los bits en el bitstream y este proceso se lleva a cabo al final de los procedimientos EncodeFlush y PutBit. Por lo tanto, en este bloque se ha creado una MEF que ejecuta el proceso con las características de cada procedimiento [4].

3.5.1 Entradas

Para saber si se trata de un proceso o de otro se ha creado la entrada de tipo record write_bits_ri. Esta señal contiene la siguiente información:

- Nr_i: número de bits que deben ser escritos en el bitstream (es equivalente a la N del proceso).
- Flush_i: indica si la llamada está relacionada al proceso EndodeFlush, '1' si está relacionada y '0' si no.
- Val_i: conjunto de dos bits que indican cómo deben escribirse los bits en el bitstream, estos operan de la siguiente manera:
    - Modo de funcionamiento a través de PutBit: en este modo el MSB (acrónimo de Most Significant Bit, *bit más significativo*) se corresponde con el inverso del valor firstBitFlag del proceso 2.4.4.4. Cuando este bit es '1', se realizan dos escrituras en el bitstream, la primera consiste en un bit con la polaridad del LSB (acrónimo de Least Significant Bit, *bit menos significativo*). Mientras que en la segunda se escriben los bits indicados por la entrada Nr_i con la polaridad opuesta al LSB.
    - Modo de funcionamiento a través de EncodeFlush: en este modo se escriben los bits que se indican en Val_i directamente.

3.5.2 Salidas

En la salida, los bits se escriben byte a byte; es decir, cada vez que se completan 8 bits, se envían y se comienza un nuevo byte. Cuando los bits son escritos por el proceso EncodeFlush (es decir, cuando flush = '1'), es necesario rellenar los bits restantes del byte con ceros. Por ejemplo, si solo se escriben dos bits en un byte, se añaden ceros para completar los seis bits restantes antes de enviar el byte [4].

3.5.3 Funcionamiento

La MEF implementada depende de la entrada Flush_i explicada en 3.5.1, de la señal done_s, que indica si la orden de escritura indicada en Val_i se ha finalizado, y de la señal read_o, que indica a una FIFO implementada dentro del bloque que la MEF está lista para procesar otra orden de escritura. Es decir, read_o indica si el bloque está listo para recibir otro valor de Val_i [4].

Los estados de la MEF son los siguientes:

- First: este es el primer estado de la MEF, en él se realiza la primera escritura de bits que indique Val_i. Si la orden se finaliza y caben todos los bits indicados por Val_i en el conjunto de 8 bits, las señales done_s y read_o toman el valor '1', y la MEF se mantiene en el estado. En caso contrario, si no ha terminado la operación de escritura porque no cabe en el byte, la MEF avanza a otro estado. En caso de que WriteBits se produzca tras EncodeFlush (flush_i = '1') se pasa al estado Flush, mientras que si se produce tras PutBit se pasa al estado Rest.

- Flush: en este estado se escribe el bit que falta por escribir que no ha cabido en el byte anterior y el resto de los bits a ‘0’ para añadir el relleno. Esto se ejecuta en un ciclo de reloj por lo tanto independientemente de las señales se vuelve a First.

- Rest: en este estado se escriben los bits o bit que faltan por escribir y no cupieron en el byte anterior. Una vez finalizada la escritura de todos los bits se pasa al estado First.

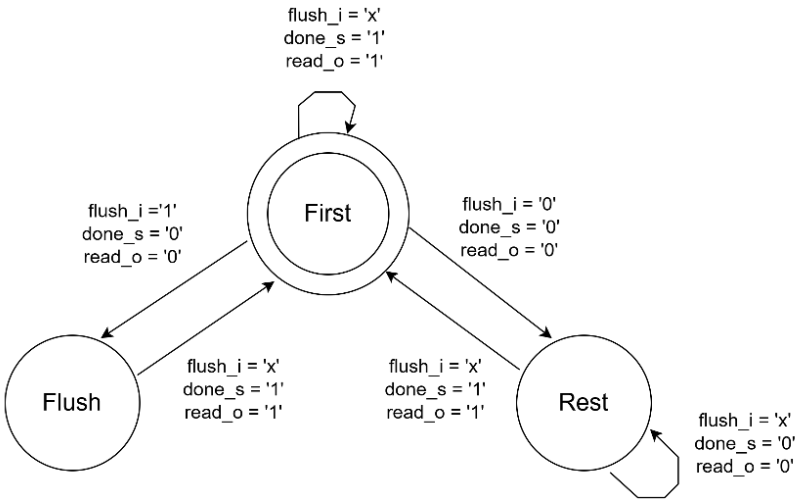

Figura 3.16: Diagrama de estados de la MEF implementada en WB. Adaptada de [4]

Capítulo 4: Verificación e implementación de CABAC

En este capítulo se explica el proceso llevado a cabo para la verificación e implementación de CABAC y los resultados obtenidos.

4.1 Vídeos utilizados en la verificación y validación

Tanto para la verificación como para la validación del módulo de CABAC se han usado varios vídeos. En la Tabla 4.1 se muestran las dimensiones de los fotogramas de cada vídeo, así como, el número de fotogramas que los componen. En general, se han utilizado vídeos sintéticos para comprobar condiciones específicas y vídeos disponibles de distinto tipo, debido a que los vídeos del proyecto EROSS-IOD son confidenciales y propiedad de Thales Alenia Space en Francia.

Nombre del Archivo	Tamaño en pixeles (ancho x alto)	Número de fotogramas
Black	1024×1024	3
Mov_small_squares	96×80	9
Mov_squares	112×96	5
Soccer	176×144	5
City	704x576	4
Syntrawvid	1024×1024	10

Tabla 4.1: Características de los vídeos usados para la simulación de CABAC.

En el vídeo "Black" todos los fotogramas son idénticos y están compuestos únicamente por píxeles negros, lo que da lugar a una imagen completamente oscura. Este vídeo, al igual que "Syntrawvid", es de carácter sintético, ya que han sido generados mediante scripts, es decir, programas que crean automáticamente los fotogramas siguiendo patrones definidos. En el caso de "Syntrawvid, el contenido no se mantiene exactamente igual en todos los fotogramas, en la *Figura 4.1* se muestra el primer fotograma. Este vídeo está formado por píxeles con distintos tonos de gris que generan patrones diagonales a lo largo del fotograma. Además, existe una gran variación entre píxeles vecinos, lo que hace que la predicción intra sea poco favorable en este caso.

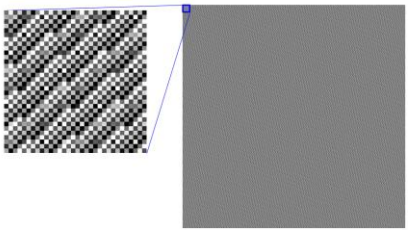

Figura 4.1: Fotograma del archivo de vídeo "Syntrawvid".

Los vídeos “Mov_small_squares”, “Mov_squares” han sido creados manualmente y están conformados por fotogramas distintos entre sí, ya que en ellos se representa movimiento. En “Mov_small_squares” se muestran pequeños cuadrados, que se mueven en el fotograma, en “Mov_squares” también se muestran cuadrados moviéndose en el fotograma, pero, en este caso los cuadrados son más grandes. Los vídeos “Soccer” y “City”, por su parte, corresponde a una secuencia real descargada de un repositorio de vídeos utilizado habitualmente para pruebas y evaluaciones experimentales [13]. En el vídeo “Soccer” se muestra a unas personas jugando fútbol y en el vídeo “City” se muestran unos edificios de una ciudad. En las siguientes imágenes se pueden ver fotogramas de cada vídeo:

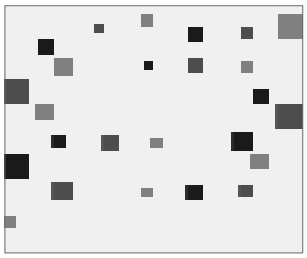

Figura 4.2: Fotograma del archivo de vídeo "Mov_small_squares".

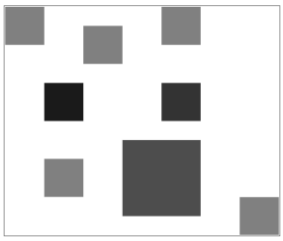

Figura 4.3: Fotograma del archivo de vídeo "Mov_squares"

Figura 4.4: Fotograma del archivo de vídeo "Soccer".

Figura 4.5: Fotograma del archivo de vídeo “City”

Tras la descripción del contenido de los distintos vídeos, es importante detallar el formato en el que se encuentran almacenados. Todos los vídeos utilizados son monocromos, por lo que únicamente se considera la componente de luminancia, con una profundidad de 8 bits por píxel, lo que implica que cada muestra puede tomar valores de intensidad comprendidos entre 0 y 255. Las secuencias se almacenan en ficheros binarios, donde los valores de los píxeles se organizan de forma secuencial por filas dentro de cada fotograma, y los fotogramas se disponen consecutivamente a lo largo del fichero.

4.2 Verificación

Para verificar el correcto funcionamiento del sistema desarrollado, se han creado una serie de bancos de pruebas, es decir, simulaciones diseñadas para verificar el comportamiento del diseño bajo ciertas condiciones de operación. Los primeros bancos de pruebas han sido creados con el fin de verificar el funcionamiento de algunos bloques de la arquitectura de CABAC de manera individual. Para ello se han asignado valores a las entradas de los bloques, y se ha observado si las salidas obtenidas con la simulación son las esperadas. En caso de que las salidas no coincidieran con las esperadas, se analizaba el diagrama de ondas con el fin de identificar el origen del error y corregir la parte del bloque responsable del fallo.

Una vez comprobado el correcto funcionamiento de algunos de los bloques de forma individual, se ha desarrollado un banco de pruebas que permite verificar el funcionamiento conjunto de todo el sistema CABAC. Para la generación de estímulos de este banco de pruebas se han utilizado los archivos que se describen en la Sección 5.2.1, los cuales contienen los valores necesarios para el funcionamiento de la arquitectura, obtenidos a partir de las etapas anteriores a CABAC. Finalmente, se ha realizado el conexionado de la arquitectura CABAC con el resto de las etapas del estándar, concretamente las de predicción, transformación y cuantización, verificándose así el correcto funcionamiento del sistema completo. Esta última verificación se detalla en 4.2.2.

4.2.1 Verificación de la arquitectura desarrollada CABAC

Para obtener los valores de los coeficientes y predicciones que se introducen en las entradas de la arquitectura, se han usado archivos .dat proporcionados por la división de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA). Estos archivos contienen las predicciones y coeficientes obtenidos en los procedimientos anteriores a CABAC, es decir, predicción, transformación y cuantización. Estos tienen el formato que se muestra en la Figura 4.6.

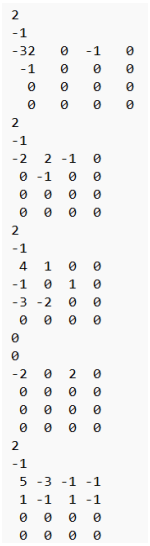

Figura 4.6: Estímulos necesarios para simular y validar la arquitectura.

En la imagen se puede visualizar el patrón que sigue el fichero: primero hay dos números y después se ve un conjunto de 4×4 números. Estos son los datos que nos hacen falta en la entrada de la arquitectura, el primer número es el modo de predicción que se usa, este puede ser 0 (V), 1 (H) o 2 (DC). Y el segundo número indica si es el modo más común o no, -1 si lo es y 1 si no. El resto de los números son los coeficientes de cada bloque 4×4 del macrobloque.

Este patrón se repite 16 veces, ya que contiene la información de los 16 bloques 4×4 del macrobloque procesado con predicción intra 4. Justo después de aparecer el último bloque 4×4 aparece una letra, esta puede ser una i o una v. Si es una i es que se va a usar predicción intra 16 finalmente, por lo tanto, la arquitectura debe desechar los coeficientes anteriores y se recibirán los coeficientes de un macrobloque intra 16. En cambio, si aparece una v, significa que la predicción más eficiente es intra 4, por lo tanto, los coeficientes anteriores son los correctos y pueden ser procesados.

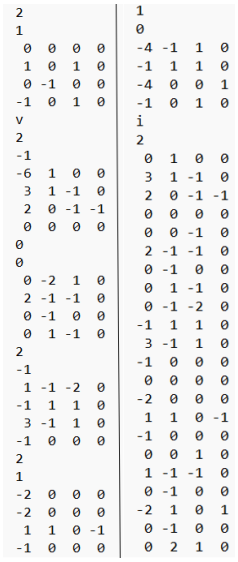

Figura 4.7: Ejemplos de las letras que se reciben al finalizar de recibir los bloques 4×4 de un macrobloque con predicción intra 4.

En la Figura *4.7* se pueden visualizar dos ejemplos separados por una línea negra, en el de la izquierda se muestra la letra v, seguida de los datos del macrobloque siguiente. Por otro lado, en el ejemplo de la derecha aparece la letra i, la secuencia continúa con un número, que es el modo de predicción usado (H, V o DC) y seguidamente aparecen los coeficientes del macrobloque. En este caso, los coeficientes no están divididos en bloques de 4×4 debido a que estos solo usan un modo de predicción para todo el macrobloque.

Por lo tanto, el banco de pruebas funciona siguiendo los siguientes pasos:

- Se introduce el valor de QP a través de la entrada slice_qp_ui y se le da el valor '1' a valid_qp_i.

- Se leen las dos primeras líneas y se obtienen los datos de la predicción del bloque 4×4 perteneciente al macrobloque codificado con predicción intra 4, esos datos se introducen en la arquitectura por la entrada predInfo_ri y se le da el valor '1' a predInfo_valid_i.

- Se leen las cuatro líneas siguientes y se obtienen los valores de los coeficientes, estos valores son introducidos en la arquitectura a través de la entrada abs_column_avi y se le da el valor '1' a cbff_write_i.

- Se siguen los dos primeros pasos para los 15 primeros bloques 4×4. Al llegar al bloque número 16, se siguen también los dos primeros pasos, pero estos datos no son introducidos en la arquitectura, en su lugar, se guardan en variables dentro del módulo del banco de pruebas. Debido a que antes de introducir estas variables a la arquitectura se debe saber si se va a usar finalmente la predicción 4×4 o 16×16.

- Se lee la siguiente línea para obtener la letra, si la letra es v, la predicción intra 4 es la más eficiente. Por lo tanto, los coeficientes enviados a la arquitectura son correctos, así que se introduce el último bloque 4×4 de coeficientes en la arquitectura con su modo de predicción y se indica con la entrada predInfoI16_ri que se va a usar predicción intra 4.

  Por otro lado, si la letra es i, se eliminan los datos guardados del último bloque 4×4 recibido, y se le indica a la arquitectura que se tienen que eliminar los bloques de coeficientes recibidos anteriormente con la entrada cbf_clear_i a '1'. Posteriormente, se lee la siguiente línea para obtener el valor del modo de predicción (H, V o DC) que se usa en el macrobloque, esta predicción es enviada a través de la entrada predInfoI16_ri. Tras esto, se leen los coeficientes del macrobloque y se introducen en la arquitectura a través de las entradas abs_column_avi, sign_column_i, cbff_write_i, cbff_write_dc_i, abs_dc16column_avi, sign_dc16column_vi.

- Tras enviar todos los coeficientes y los modos de predicción a la arquitectura, se espera a recibir las salidas de la arquitectura bac_ett_o y ready_o con valor '1', que indican que ya se han procesado todos los coeficientes enviados a la arquitectura.

- Se repiten todos los pasos menos el primero, hasta terminar de procesar todo el archivo .dat.

Por otro lado, para verificar las salidas de la arquitectura, es decir el bitstream generado, se ha usado el software oficial del estándar, JM 19.0. Este software implementa la codificación y decodificación de vídeo según el estándar H.264, para su ejecución se necesitan 2 archivos: el vídeo a codificar en formato .raw y un archivo de texto para la configuración con extensión .cfg.

El archivo .cfg contiene toda la información necesaria para que el software del JM codifique el vídeo con los parámetros deseados, como el tamaño del vídeo, el valor de QP con el que se quiere codificar, los tipos de predicción que se quieren usar, en nuestro caso serían intra 4 e intra 16, y muchos otros parámetros. Ambos archivos han sido otorgados por la división de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA).

En esta simulación de la arquitectura de CABAC desarrollada se han utilizado los siguientes vídeos:

- "Black" con valores de QP iguales a 0, 24 y 51.
- "Mov_smallsquares" con valores de QP iguales a 0 y 51.
- "Mov_squares" con un valor de QP = 21.
- "Soccer" con valores de QP iguales a 0 y 51.
- "City" con valores de QP iguales a 0, 24 y 51.
- "Syntrawid" con valores de QP iguales a 24 y 51.

4.2.2 Verificación del sistema completo del estándar H.264

Una vez verificado el correcto funcionamiento de la arquitectura CABAC desarrollada en este TFT, se comprobó su funcionamiento conjunto con el resto de los módulos del estándar H.264, correspondientes a los pasos previos a CABAC. Estos módulos fueron desarrollados por la División de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA).

Para la integración de estos bloques en el diseño, se proporcionó una descripción en formato EDIF (acrónimo de Electronic Design Interchange Format), un formato estándar de intercambio utilizado para describir la estructura y la conectividad de diseños electrónicos de forma independiente de la herramienta de diseño empleada [14].

El uso del formato EDIF permitió disponer de una representación estructural de los pasos previos a CABAC, que no incluía información confidencial sobre su implementación interna. De este modo, no se tuvo acceso al código fuente original de los bloques, pero sí fue posible integrar estos pasos previos con la arquitectura de CABAC desarrollada y simularlos e implementarlos en FPGA, garantizando la confidencialidad del diseño proporcionado por el IUMA.

Para poder integrar correctamente estos bloques en la arquitectura global, fue necesario desarrollar un wrapper en VHDL cuya función principal fue adaptar las interfaces del diseño descrito en EDIF a un conjunto de entradas y salidas en VHDL. Este wrapper permitió encapsular los bloques proporcionados y conectarlos con el resto de los módulos del sistema, asegurando una correcta interconexión y sincronización con los bloques desarrollados en este trabajo.

En la Figura 4.8 se muestra un diagrama simplificado con los distintos bloques que conforman el conexionado final empleado para esta verificación. Todos los bloques mostrados en dicho diagrama han sido descritos en VHDL. Puede observarse que todos los módulos se encapsulan dentro de bloques con el sufijo _TOP, los cuales se encargan de instanciar las correspondientes entradas y salidas de cada bloque y del sistema completo.

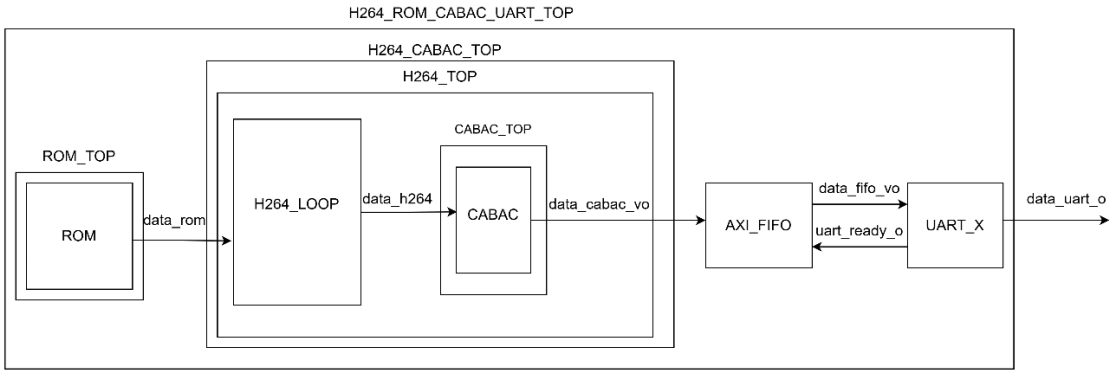

Figura 4.8: Diagrama de bloques del diseño usado en la verificación del sistema completo del estándar H.264.

De izquierda a derecha, los bloques que componen el sistema son los siguientes:

- ROM_TOP: bloque encargado de proporcionar píxel a píxel el contenido del vídeo raw al bloque de H264_CABAC_TOP. Incorpora un contador configurable que permite enviar los píxeles a mayor o menor velocidad, para darle tiempo a la arquitectura completa comprendiendo CABAC y sus pasos anteriores a procesar y codificar la información.
    - ROM (acrónimo de Read-Only Memory): memoria ROM en la que se almacena el vídeo raw utilizado como entrada del sistema. Este fichero se genera mediante un script de Python a partir del fichero raw.

- H264_CABAC_TOP: bloque que integra tanto la arquitectura CABAC como el resto de las etapas previas del estándar H.264. En él se instancian las entradas y salidas que lo conectan con los bloques ROM_TOP y AXI_FIFO.
    - H264_TOP: bloque que instancia las entradas y salidas que unen el bloque CABAC_TOP con el bloque H264LOOP.
        - H264_LOOP: *wrapper* que contiene los procedimientos del estándar H.264 previos a CABAC en formato EDIF y realiza la conversión de las entradas y salidas de estos a formato VHDL para conectarlos con el resto de los bloques, incluye las etapas de predicción, transformación y cuantización.

        - CABAC: bloque que implementa la arquitectura CABAC desarrollada en este TFT.

- AXI_FIFO: memoria intermedia que recibe el bitstream generado por CABAC y lo almacena hasta que la interfaz UART (acrónimo de Universal Asynchronous Receiver-Transmitter) se encuentra lista para su transmisión.

- UART: bloque encargado de transmitir los datos a través de este sistema de comunicación, permitiendo la verificación de los resultados que se obtienen en la implementación.

Para ayudar a la comprensión del diagrama se ha simplificado, incluyendo solo unas cuantas salidas de cada bloque:

- data_rom: creada para introducir poco a poco la información de los pixeles en el sistema del estándar, la memoria ROM está creada para mandar esta información cada cierto número de ciclos de reloj y así darle tiempo al resto de bloques del sistema para procesar los datos.
- data_H264: contiene la información necesaria para que CABAC codifique los coeficientes, esto incluye el tipo de predicción, valor absoluto de los coeficientes, su signo y todas las entradas explicadas en 3.3.1.
- data_cabac_vo: contiene el bitstream generado por la arquitectura CABAC.
- data_fifo_vo: bitstream almacenado por el bloque AXI_FIFO.
- uart_ready_o: indica si el bloque UART_X está listo para recibir más bytes del bitstream.
- data_uart_o: salida usada para enviar bit a bit el bitstream generado en formato UART.

Una vez realizadas todas las conexiones, se creó un banco de pruebas que genera únicamente las señales de reloj y reset, con el fin de observar las salidas del sistema, ya que este se encuentra configurado para funcionar de manera autónoma usando la memoria que contiene el vídeo que se quiere codificar.

Para esta verificación final se han utilizado los siguientes vídeos:

- “Black” con un valor de QP = 21.
- “mov_squares” con un valor de QP = 21.
- “soccer” con un valor de QP = 51.

4.2.3 Resultados de las verificaciones

Los bancos de pruebas han servido para verificar el correcto funcionamiento tanto de la arquitectura CABAC de manera individual como del sistema completo del estándar H.264. Para ello, el bitstream generado por la salida de la arquitectura se ha comparado con el bitstream generado por el software oficial del estándar, JM 19.0, lo que ha permitido detectar si alguno de los bits no coincidía.

Cuando el bitstream obtenido no coincidía con el esperado, se aplicaba el procedimiento habitual de depuración, que consiste en analizar las señales internas de la arquitectura mediante el diagrama de ondas con el fin de localizar el origen del error y corregir la parte del diseño responsable del fallo. Tras realizar las modificaciones necesarias, se comprobó que el bitstream generado por la arquitectura coincidía con el obtenido mediante el software de referencia.

Las simulaciones se han realizado utilizando los simuladores Vivado Simulator y QuestaSim de ModelSim, ambos ejecutados en un entorno Linux. El uso de ambos simuladores fue necesario ya que, en algunos casos, los resultados obtenidos no coincidían. Concretamente, se detectó un error en el bloque escritor de bits de la arquitectura CABAC descrito en el apartado 3.5 que no aparecía en las simulaciones realizadas con Vivado Simulator, donde el bitstream generado era correcto. Sin embargo, al implementar el diseño, el bitstream obtenido no coincidía con el esperado. Pero como el error no aparecía en la simulación no podíamos encontrar el fallo en la arquitectura, por eso se decidió implementar las simulaciones en otro simulador más restrictivo. Al llevar a cabo las simulaciones en QuestaSim, el error sí se reproducía, lo que permitió localizar y corregir el origen del fallo.

Una vez verificada la arquitectura CABAC de manera individual, se realizó la verificación del sistema completo integrando todas las etapas del estándar H.264. En esta última verificación, el sistema se configuró para funcionar de manera autónoma utilizando una memoria ROM que contiene el vídeo a simular.

En todos los casos, el bitstream generado en las simulaciones ha coincidido con el obtenido mediante el software de referencia, confirmando así el correcto funcionamiento del sistema desarrollado.

4.3 Implementación en FPGA y validación

Una vez comprobado el correcto funcionamiento del sistema completo, incluyendo CABAC y el resto de las etapas del estándar H.264 en simulación, se procede a su validación en FPGA. La FPGA que se ha usado en la implementación es la Nexys 4 DDR, algunas de las características de esta son [15]:

- FPGA Xilinx Artix-7 XC7A100T, con 15.850 slices lógicos, 4.860 Kbits de BRAM y 240 bloques DSP.
- Memoria externa DDR2 de 128 MB con interfaz de 16 bits.
- Seis bloques de gestión de reloj (PLL/MMCM) y frecuencia interna superior a
- 450 MHz.
- Memoria Flash SPI.
- Interfaz USB-UART integrada para comunicación.

El diseño implementado en FPGA corresponde al explicado en 4.2.2 y que se muestra en la Figura 4.8. Para visualizar el bitstream generado por este diseño se ha conectado la FPGA a un ordenador por medio de conexión UART, ya que el sistema está configurado con este método de comunicación. Para ello se ha utilizado un cable USB que conecta la FPGA por el puerto configurado para la comunicación UART con uno de los puertos USB del ordenador. Para ver el bitstream dentro del ordenador se ha hecho uso del software Cutecom [16], que desde el ordenador lee lo que se recibe por el puerto serie y lo muestra por pantalla.

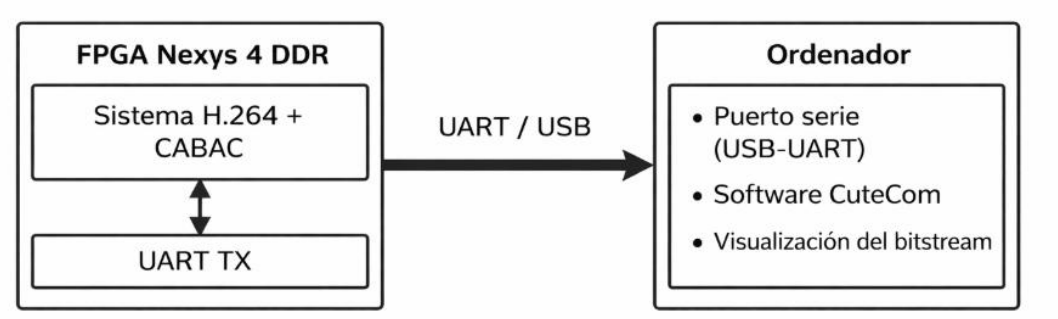

Figura 4.9: Diagrama de la implementación del sistema final.

4.3.1 Resultados de la implementación

Una vez realizada la verificación detallada en 4.2.2 y corregidos los errores encontraron l se procedió a realizar la implementación sobre una tarjeta FPGA estándar. La principal dificultad encontrada inicialmente es que para la validación en la tarjeta el resto de los bloques que conforman el estándar H.264 estaban disponibles como un bloque EDIF. Este bloque EDIF debió conectarse adecuadamente con los bloques CABAC y ROM del diseño final.

Para resolver el problema, se desarrolló un *wrapper* que transformaba las entradas VHDL al formato que identificaba el archivo EDIF y las salidas del archivo EDIF a formato VHDL. Tras varios ajustes en el *wrapper* y una verificación completa en simulación del diseño integrado, que incluía la memoria ROM, la arquitectura CABAC con todos sus pasos previos y la interfaz UART, se procedió a implementar el sistema completo en FPGA. Tras este último cambio, se consiguió generar correctamente el bitstream.

La validación del diseño fue llevada a cabo utilizando los siguientes vídeos:

- "Black" con un valor de QP = 21.
- "mov_squares" con un valor de QP = 21.
- "soccer" con un valor de QP = 51.

En las siguientes imágenes se muestran algunos fragmentos del bitstream en formato hexadecimal generado por el JM oficial, así como el obtenido con la implementación del sistema final en la FPGA, presentado en pantalla mediante Cutecom. Se puede apreciar como ambos bitstream son idénticos.

```
0b b6 dc 91 79 be da aa   96 39 fb 59 45 02 ef ac
a8 29 f7 31 f1 84 69 98   53 26 67 f7 06 4d 6c ef
4d ab c9 03 59 c5 8a ca   09 ca 5c 58 64 e0 dc 05
c7 0c 95 3d 60 da a4 76   f6 01 37 10 8d 4c a4 8b
6f 62 3b b9 7a 24 5d ad   cf ab 7a 35 94 b9 b2 69
eb d1 84 21 63 9a 4f 23   b6 09 96 ab f6 55 c9 9b
```

Figura 4.10: Extracto del bitstream generado por la FPGA y presentado en pantalla con Cutecom.

```
0b b6   dc 91   79 be   da aa   96 39   fb 59   45 02   ef ac
a8 29   f7 31   f1 84   69 98   53 26   67 f7   06 4d   6c ef
4d ab   c9 03   59 c5   8a ca   09 ca   5c 58   64 e0   dc 05
c7 0c   95 3d   60 da   a4 76   f6 01   37 10   8d 4c   a4 8b
6f 62   3b b9   7a 24   5d ad   cf ab   7a 35   94 b9   b2 69
eb d1   84 21   63 9a   4f 23   b6 09   96 ab   f6 55   c9 9b
```

Figura 4.11: extracto del bitstream generado por el software oficial del estándar.

4.3.2 Frecuencia

En cuanto a la frecuencia de operación, el diseño ha sido validado funcionalmente utilizando un reloj de 100 MHz. En la Figura 5.12 se muestran los resultados del análisis temporal tras la implementación, donde se observa un valor de Worst Negative Slack (WNS) de − 0,297 ns. Este valor negativo indica que el diseño no cumple completamente las restricciones temporales establecidas para una frecuencia de 100 MHz, por lo que, desde el punto de vista temporal, sería necesario reducir la frecuencia de operación para garantizar el cumplimiento de dichas restricciones.

No obstante, a pesar de este incumplimiento temporal, la verificación y validación del sistema, incluyendo todas las simulaciones e implementaciones, se realizaron a una frecuencia de 100 MHz, confirmando su correcto funcionamiento a dicha frecuencia.

Figura 4.12: Tabla de tiempos obtenida tras la implementación del diseño.

4.3.3 Recursos

En cuanto al uso de recursos, en la *Tabla 4.2* se muestran los recursos usados sobre la AMD Artix™ 7 que incluye la Nexys 4 usada en la implementación del diseño. Se puede apreciar como ningún recurso sobrepasa el 30 % de recursos usados, a pesar de tratarse de un dispositivo de capacidad moderada. Lo que indica que existe margen suficiente para implementar mejoras en el diseño y otorgarlo de más prestaciones como la codificación de imagen cromática.

Recurso utilizado	Total Utilizado	Porcentaje (%)
LUT como lógica	63.400	22,86
LUT como memoria	184	0,96
Slice flip-flops	30.311	23,9
Multiplexores F7	2.089	6,58
Multiplexores F8	899	5,67

Tabla 4.2: Utilización de recursos en la implementación.

Capítulo 5: Conclusiones

Este TFT ha supuesto una experiencia de aprendizaje muy relevante, ya que se ha trabajado en un proyecto de carácter industrial, utilizando herramientas profesionales de diseño y verificación hardware, y contribuyendo al desarrollo de un sistema real y funcional. A lo largo del trabajo ha sido necesario estudiar en profundidad un estándar industrial complejo como H.264 y llevar a cabo su implementación en FPGA, lo que ha permitido adquirir conocimientos tanto sobre compresión de vídeo como sobre diseño de arquitecturas digitales.

Durante el desarrollo del proyecto también se han encontrado las dificultades propias del diseño real, en el que las cosas no siempre funcionan a la primera. En varias ocasiones, los resultados obtenidos en simulación no coincidían con los de la implementación en FPGA, lo que hizo que la fase de validación fuese especialmente costosa y obligó a utilizar simuladores más estrictos, como QuestaSim, para localizar y corregir los errores detectados. En conjunto, esta experiencia ha permitido consolidar varios de los conocimientos adquiridos durante el grado y aplicarlos en un entorno práctico y real, aportando una visión más completa del proceso de diseño e implementación de sistemas de compresión de vídeo sobre FPGA.

En los siguientes subapartados se analizan los resultados obtenidos y el grado de consecución de los objetivos propuestos, además de las posibles líneas futuras de investigación.

5.1 Trabajo realizado y resultados obtenidos

En este Trabajo de Fin de Título se ha ampliado con éxito el diseño realizado en [4], de tal forma que este nuevo diseño amplía las capacidades del anterior, pudiendo codificar tanto vídeos con predicción intra 4 como con predicción intra 16. Esto se ha conseguido realizando varios cambios en el diseño:

- Se han creado las diversas entradas necesarias para introducir datos relacionados con predicción intra 16. Asimismo, se han creado diversas señales dentro de la arquitectura que conectan todos los bloques para llevar a cabo uno u otro proceso dentro de esto según el tipo de predicción (intra4 o intra 16).

- Modificación del bloque búfer de coeficientes para obtener correctamente el valor del SE CBP cuando se usa intra 16.

- Modificación del bloque serializador para que se envíen los coeficientes escaneados por el bloque de una forma u otra según el tipo de predicción.

- Modificación del bloque unidad de control para que se calcule dentro de este el valor del SE MBT ya que a este en la arquitectura original se le daba un valor constante. Asimismo, se han creado numerosos estados en la MEF que compone este bloque para tener en cuenta los casos en los que la predicción es intra 16.

- Ampliación del bloque interprete de SEs residuales para que en este se calculen los valores de las variables ctxBlockCat necesaria para el cálculo del índice de contexto de cada bin.

- Ampliación del bloque modelador de contexto para que en este se calcule el valor de la variable ctxIdxIncrement del SE MBT.

- Creación de más módulos de memoria para poder almacenar los valores de ctxIdxIncrement relacionados con el SE MBT.

- Creación de las tablas de contexto necesarias para la codificación de coeficientes obtenidos por predicción intra 16.

- Ampliación del bloque Binarizador y más concretamente se han ampliado los estados de la MEF que compone este bloque para poder binarizar los SEs relacionados a macrobloques codificados con predicción intra 16.

Tras realizar los cambios, se comprobó el correcto funcionamiento del diseño mediante procesos de verificación, empleando bancos de prueba. Dichos bancos de prueba comparaban el bitstream generado por la arquitectura CABAC desarrollada con el generado por el software oficial del estándar H.264. Una vez que las simulaciones produjeron bitstreams idénticos a los obtenidos mediante el software de referencia, quedó validado el correcto funcionamiento de la arquitectura ampliada del codificador CABAC implementada en este TFT.

Una vez completada la verificación individual de la arquitectura CABAC desarrollada, se procedió a su integración en un diseño completo del estándar H.264, que incluía las etapas previas a CABAC proporcionadas por la División de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA). Estas etapas previas fueron suministradas en formato EDIF, por lo que fue necesario el uso de un wrapper para permitir el adecuado conexionado de señales entre el código desarrollado en VHDL y el diseño en formato EDIF.

Tras realizar la interconexión completa entre el codificador CABAC y las etapas previas del estándar, se incorporó una memoria ROM que contenía los píxeles de los vídeos a codificar. En total, se emplearon tres memorias ROM, correspondientes a los tres vídeos utilizados durante las fases de verificación y validación final del diseño integrado. Asimismo, se implementó una interfaz UART conectada a la salida del codificador CABAC, encargada de transformar el bitstream final generado al formato UART, lo que permitió su visualización en un ordenador mediante conexión USB y el software Cutecom.

Finalmente, tanto la verificación como la validación del sistema completo se llevaron a cabo mediante simulación y mediante la implementación de la arquitectura en la FPGA Nexys 4 DDR, confirmando el correcto funcionamiento del diseño integrado conforme a los objetivos planteados.

Tras la verificación y validación, se ha demostrado que el diseño funciona correctamente a una frecuencia de 100 MHz, a pesar de presentar un slack negativo en los resultados del análisis temporal. La correcta operación se confirma además porque el bitstream generado coincide con el producido por el software oficial del estándar. Estas pruebas se realizaron utilizando varios vídeos con diferentes características, algunos de los cuales se codificaron

con distintos valores de QP, lo que permitió evaluar y verificar el funcionamiento del sistema en múltiples escenarios.

Por lo tanto, se puede considerar que el diseño final creado en este TFT cumple con el propósito, ya que proporciona una implementación completa del estándar H.264 en FPGA. Esta implementación incluye los pasos previos a CABAC (predicción, transformación y cuantificación), así como la versión ampliada del codificador CABAC desarrollada en este TFT, capaz de codificar vídeos monocromos utilizando tanto predicción intra 4×4 como intra 16×16.

5.2 Cumplimiento de los objetivos

El objetivo principal del proyecto es el desarrollo e implementación de un codificador entrópico CABAC y la salida del estándar H.264 en FPGA. Este objetivo general se desglosa en los siguientes objetivos específicos:

- O1: Comprender el funcionamiento del estándar H.264 y su codificador entrópico CABAC, incluyendo su estructura y principios de compresión.
- O2: Diseñar e implementar el codificador CABAC junto con la etapa de salida del estándar H.264, asegurando su integración con los bloques ya desarrollados.
- O3: Verificar el funcionamiento adecuado del diseño a través de simulaciones, y posteriormente validar y evaluar su rendimiento mediante su implementación en FPGA.

A partir del trabajo desarrollado y de los resultados obtenidos, puede concluirse que los objetivos O1 y O3 han sido cumplidos satisfactoriamente. En la memoria se presenta un estudio detallado del estándar H.264 y del funcionamiento del codificador CABAC, así como la verificación del sistema tanto en simulación como en FPGA, comprobando la correcta generación del bitstream y su equivalencia con el codificador de referencia.

Asimismo, el objetivo O2 ha sido cumplido, puesto que se ha implementado la salida conforme al estándar, obteniendo un bitstream funcionalmente correcto. No obstante, el diseño podría haberse mejorado incorporando las cabeceras generales del estándar H.264 responsables de estructurar el bitstream en unidades NAL. Estas cabeceras no se implementaron por limitaciones de tiempo, si bien su integración sería compatible con el diseño desarrollado y no supondría un esfuerzo significativo, ya que bastaría con añadirlas como una capa superior al flujo generado.

Finalmente hay que comentar que el diseño satisface las necesidades del proyecto EROSS-IOD, ofreciendo un sistema funcional y eficiente que permite comprimir las imágenes y vídeos generados por las cámaras del satélite y del brazo robótico. Esto contribuye a reducir el tamaño de la información visual captada por las cámaras de vídeo y hacer más eficiente su transmisión.

5.3 Líneas futuras

A partir del diseño finalizado en este TFT se proponen las siguientes líneas de trabajo basadas en su mejora y su ampliación:

- Modificar el diseño para que pueda codificar macrobloques procesados con predicción inter, es decir, predicción usando datos de otros frames.
- Modificar el diseño para que pueda codificar vídeos cromáticos, es decir, vídeos con color.
- Optimizar el diseño para que se pueda implementar con una frecuencia mayor.
- Implementar las unidades NAL conforme al estándar H.264.
- Implementar en hardware el decodificador del estándar H.264, capaz de decodificar el bitstream generado por el codificador.

Anexo A

Presupuesto

El presupuesto correspondiente a este Trabajo de Fin de Grado se estructura en distintas categorías con el fin de separar los diversos costes relacionados con su realización. Este está conformado por recursos humanos, recursos materiales, redacción del documento, derechos de visado del Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones (COITT), gastos de tramitación y envió, material fungible.

A.1 Recursos humanos

Para calcular los recursos humanos, se va a tener en cuenta que el TFT se ha realizado durante 6 meses, dedicando 37,5 horas semanales. Asimismo, considerando el salario correspondiente a un graduado en ingeniería en tecnologías de la telecomunicación y teniendo en cuenta que el proyecto propuesto se ha realizado en la Universidad de las Palmas de Gran Canaria (ULPGC). La tarifa aplicada es la asociada al personal técnico con titulación mínima exigida de graduado o equivalente, de acuerdo con [17]. El resultado del coste de los recursos humanos se halla en la siguiente tabla:

Personal	**Coste total mensual (€)**	**Tiempo**	**Total (€)**
Ingeniero técnico	1.926,89	6 meses	11.561,34

Tabla A.1: Coste de los recursos humanos.

El coste final del trabajo tarifado por tiempo empleado es de ONCE MIL QUINIENTOS SESENTA Y UN EUROS CON TREINTA Y CUATRO CÉNTIMOS.

A.2 Recursos materiales

Para el cálculo del coste relativo a los recursos materiales empleados se tiene en cuenta un periodo de amortización de tres años de carácter lineal. El cálculo de la cuota de amortización anual se realiza mediante la siguiente expresión siendo C la cuota de amortización anual, Vad el valor de adquisición, Vres el valor residual y N el número de años considerados para la amortización de cada producto. Nótese que el valor residual se supone 0 e para todos los casos.

$$C = \frac{Vad - Vres}{N}$$

A.2.1 Recursos software

Dado que las herramientas software utilizadas en el desarrollo del proyecto, detalladas a continuación, son de uso gratuito, disponen de licencias académicas para estudiantes o se encuentran disponibles en los laboratorios de la ULPGC, no se deriva ningún coste económico asociado a los recursos software, siendo este de CERO EUROS.

- Paquete ofimático Microsoft-Office 365.
- Vivado™ Design Suite 202X.
- VSCodium.
- Doxygen.
- Zotero.
- Cutecom

A.2.2 Recursos hardware

Dado que la duración del trabajo ha sido inferior al periodo establecido para la amortización, se ha considerado una amortización proporcional al tiempo efectivo de realización del proyecto (6 meses). Por otro lado, se estima que los recursos de hardware cuentan con un periodo de amortización de 3 años de uso. En la siguiente tabla se recoge el coste total asociado a los recursos hardware:

Descripción	**Unidades**	**Tiempo de uso**	**Vad (€)**	**Coste anual (€)**	**Coste final (€)**
PC personal	1	6 meses	1.700,00	566,67	283.34
Estación de trabajo del laboratorio	1	6 meses	1.400,00	466,67	233.34
Nexys 4 DDR	1	6 meses	306,84	102,28	51.14
Total					**567.82**

Tabla A.2: Coste de los recursos hardware.

A.3 Redacción del documento

El coste asociado a la redacción del documento se ha calculado mediante la expresión que se presenta a continuación, donde R representa dicho coste, P corresponde al presupuesto del proyecto y C_n es el coeficiente de ponderación del presupuesto. Dado que el coste total del proyecto no supera los 30.050,00 €, se ha considerado C_n con valor unitario.

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

Por otra parte, el presupuesto total se obtiene a partir de la suma del coste de las amortizaciones y del coste del trabajo tarifado en función del tiempo empleado. El resultado de este cálculo se presenta en la siguiente tabla:

Descripción	**Coste (€)**
Coste recursos humanos	11.561,34
Amortización de los recursos software	0
Amortización de los recursos hardware	567,82
Total	**12129,16**

Tabla A.3: Total de las amortizaciones y los recursos humanos.

Una vez hallado el presupuesto, finalmente pueden determinarse los costes asociados a la redacción del documento aplicando la expresión previamente expuesta:

$$R = 0{,}07 \times P \times C_n = 0{,}07 \times 12.129{,}16 \times 1 \approx 849{,}04\,€$$

El coste derivado de la redacción del documento es de OCHOCIENTOS CUARENTA Y NUEVE EUROS CON CUATRO CÉNTIMOS.

A.4 Derechos de visado del COITT

En cuanto a los derechos de visado del COITT, estos pueden calcularse aplicando la siguiente expresión según [18]:

$$V = 0{,}007 \times P \times Cr$$

En cuanto a los derechos de visado del COITT, estos pueden calcularse aplicando la siguiente expresión según: Nótese que V hace alusión a los costes de los derechos de visado, P al presupuesto de ejecución sin impuestos1 y Cr al coeficiente reductor. Por ser P < 6000 e, el valor de Cr es 1. Asimismo, en [18] se menciona que el mínimo importe para los derechos de visado de un documento de estas características es de 40 e. Finalmente, los costes de los derechos de visado son:

$$V = 0{,}007 \times P \times Cr = 0{,}007 \times (12.129{,}16 + 849{,}04) \times 1 \approx 90{,}85\,€$$

El coste de los derechos de visado es de NOVENTA EUROS Y OCHENTA Y CINCO CÉNTIMOS.

A.5 Gastos de administración

De acuerdo con [18], los gastos de administración por documento son de nueve o doce euros dependiendo de si el visado es digital o manual, respectivamente. Por tanto, suponiendo visado digital, los gastos de administración para este documento ascienden a NUEVE EUROS.

A.6 Material fungible

No se ha empleado otro recurso aparte de recursos hardware y software.

A.7 Presupuesto final del proyecto

Al desarrollo del presente TFT se le aplica el Impuesto General Indirecto Canario (IGIC), el cual representa un siete por ciento del valor del presupuesto. En la siguiente tabla se recoge el presupuesto final:

Descripción	Coste (€)
Recursos humanos	11.561,34
Amortización de los recursos software	0,00
Amortización de los recursos hardware	567.82
Redacción del documento	849,04
Derechos de visado del COITT	90,85
Gastos de administración	9,00
Subtotal	13.078,05
Total (+7% IGIC)	**13.993,51**

Tabla A.4: Coste total del proyecto

El valor del presupuesto final para el presente TFT es de TRECE MIL NOVECIENTOS NOVENTA Y TRES EUROS CON CINCUENTA Y UN CÉNTIMOS.

Bibliografía

[1] European Robotic Orbital Support Services In-Orbit Demonstration | EROSS IOD | Project | Fact Sheet | HORIZON | CORDIS | European Commission, [En línea]. Disponible en: https://doi.org/10.3030/101082464 [Último acceso: 19 de noviembre de 2025].

[2] What is EROSS-SC- EROSS SC, [En línea]. Disponible en: https://eross-sc.eu/about/what-is-eross-sc/. [Último acceso: 19 de noviembre de 2025].

[3] Ian Richardson, E. *The H. 264 advanced video compression standard*. John Wiley & Sons, 2011.

[4] David Martín Martín, "Diseño e implementación en FPGA del codificador entrópico CABAC para su empleo a bordo de satélites". Universidad de Las Palmas de Gran Canaria, 2025 [En línea].
Disponible en: https://accedacris.ulpgc.es/handle/10553/140298 [Último acceso: 19 de noviembre de 2025].

[5] Lih-Jen Kau, Chin-Kun Tseng, Ming-Xian Lee, Perception-Based H.264/AVC Video Coding for Resource-Constrained and Low-Bit-Rate Applications, [En línea]. Disponible en: https://www.mdpi.com/1424-8220/25/14/4259 [Último acceso: 19 de noviembre de 2025].

[6] G.Sullivan y T. Wiegand, "Video Compression- From Concepts to the H.264/AVC Standard," [En línea]. Disponible en: https://doi.org/10.1109/JPROC.2004.839617 [Último acceso: 19 de noviembre de 2025].

[7] R. Chen, "Architecture and hardware for a 1 bin per cycle context-adaptive binary arithmetic coder (CABAC) encoder," 2019.

[8] D. Marpe, H. Schwarz y T. Wiegand, "Context-based adaptive binary arithmetic coding in the h.264/AVC video compression standard," IEEE Transactions on Circuits and Systems for Video Technology, [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/1218195 [Último acceso: 19 de noviembre de 2025].

[9] Dajiang Zhou, Jinjia Zhou, Ultra-high-throughput VLSI architecture of H.265/HEVC CABAC encoder for UHDTV applications, [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/6851145 [Último acceso: 19 de noviembre de 2025].

[10] Itu-t, ITU-T SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services-Coding of moving video Advanced video coding for generic audiovisual services Recommendation ITU-T H.264, [En línea]. Dispobible en: https://www.itu.int/rec/T-REC-H.264-202408-I [Último acceso: 19 de noviembre de 2025].

[11] R. Osorio, Javier D. Bruguera, High-throughput architecture for H.264/AVC CABAC compression system, [En línea]. Disponible en: https://ieeexplore.ieee.org/abstract/document/4012010 [Último acceso: 19 de noviembre de 2025].

[12] Xiaohua Tian, Thinh M. Le, Yong Lian, Entropy Coders of the H.264/AVC Standard: Algorithms and VLSI Architectures, 2011.

[13] Xiph.org: Derf's Test Media Collection, [En línea] https://media.xiph.org/video/derf/ [Último acceso: 26 de diciembre de 2025].

[14] EDIF | Wikipedia, [En línea] https://en.wikipedia.org/wiki/EDIF [Último acceso: 9 de enero de 2026].

[15] Nexys 4 DDR Reference Manual - Digilent Reference [En línea] https://digilent.com/reference/programmable-logic/nexys-4-ddr/reference-manual [Último acceso: 1 de enero de 2026].

[16] CuteCom [En línea] https://cutecom.sourceforge.net/ [Último acceso: 9 de enero de 2026].

[17] S. DE General La Universidad De Las Palmas De Gran Canaria, «MODIFICACIÓN RETRIBUCIONES PERSONAL PROYECTOS», 2024. [En línea]. Disponible en: https://www.ulpgc.es/sites/default/files/ArchivosULPGC/vinvestigacion/20240709_acg_modificaciones_retribuciones_personal_investigador_tecnico_apoyo_proyectos_.pdf [Último acceso: 13 de enero de 2026].

[18] Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones, El visado de documentos en la ingeniería de telecomunicaciones, 2020.