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

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO DE FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN EN FPGA DEL CODIFICADOR ENTRÓPICO CABAC PARA SU EMPLEO A BORDO DE SATÉLITES

Titulación:	Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención:	Sistemas Electrónicos
Autor:	David Martín Martín
Tutores:	Dr. Roberto Sarmiento Rodríguez
	Dr. Felipe Machado Sánchez
Fecha:	mayo de 2025

Resumen

Este Trabajo de Fin de Grado (TFG) presenta el diseño e implementación en *Field Programmable Gate Array* (FPGA) del sistema de codificación entrópica *Context Adaptative Binary Arithmetic Coding* (CABAC) conforme al estándar H.264/AVC, con el propósito de ser utilizado en plataformas satelitales. La motivación principal del proyecto surge de la necesidad de optimizar tanto la transmisión de vídeo desde el espacio como su almacenamiento, donde los recursos de ancho de banda y procesamiento son especialmente limitados. En este contexto, CABAC representa una solución avanzada por su alta eficiencia en compresión de datos, aunque supone desafíos tanto conceptuales como a nivel de implementación *hardware*.

El documento comienza con una revisión de los fundamentos de codificación de vídeo, centrándose en la estructura y funcionamiento a grandes rasgos del estándar H.264. Se profundiza en el módulo de interés, CABAC, detallándose cada una de las etapas que lo componen: binarización, modelado, de contextos y codificación binaria aritmética.

En cuanto a la implementación, el diseño ha sido descrito en lenguaje *Very high speed integrated circuit Hardware Description Language* (VHDL) y se ha desarrollado utilizando herramientas de síntesis y simulación como Vivado™. El sistema ha sido verificado mediante la comparación con trazas generadas por el *software* de referencia JM 19.0, asegurando así que la implementación *hardware* reproduce correctamente el comportamiento de dicho *software*. Finalmente, se ha realizado la implementación para una FPGA AMD Artix™ 7, demostrando la factibilidad técnica de utilizar este diseño en un entorno real.

Abstract

This project presents the design and implementation of the H.264/AVC video compression standard Context Adaptative Binary Arithmetic Coding (CABAC) entropy encoding system on a Field Programmable Gate Array (FPGA). The primary motivation behind the project lies in the need to optimize both video transmission and storage in space-based systems, where bandwidth and processing capabilities are notably constrained. In this context, CABAC emerges as an advanced solution offering high compression efficiency. However, due to its nature, it implies significant conceptual and hardware implementation challenges.

The document begins with a theoretical review of video coding fundamentals, followed by a general overview of the H.264 standard. It then focuses on the CABAC module, thoroughly describing its internal stages —binarization, context modeling, and binary arithmetic coding— while explaining the operational modes of each block based on the type of data being encoded.

The implementation has been developed in Very high speed integrated circuit Hardware Description Language (VHDL) and synthesized using tools such as Vivado™. The system was verified through comparison with reference traces generated by the JM 19.0 software, ensuring that the hardware implementation accurately replicates the behavior of the reference model. Finally, the design was implemented on an AMD Artix™ 7 FPGA, demonstrating the technical feasibility of this solution in a real environment.

Índice

Índice de figuras

Índice de tablas

Lista de acrónimos

AC *Arithmetic Coding*, codificación aritmética.

ASIC *Application Specific Integrated Circuit*, circuito integrado de aplicación específica.

AVC *Advanced Video Coding*, codificación avanzada de vídeo.

B4 Bloque 4x4.

B8 Bloque 8x8.

BAC *Binary Arithmetic Coder*, codificador binario aritmético.

CABAC *Context Adaptative Binary Arithmetic Coding*.

CAVLC *Context Adaptative Variable Length Coding*.

CLB *Configurable Logic Block*, bloque lógico configurable.

CM *Context Modeler*, modelador de contextos.

COITT Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones.

DCT *Discrete Cosine Transform*, transformada discreta del coseno.

DSI Diseño de Sistemas Integrados.

DUT diseño bajo prueba (del inglés *Design Under Test*).

DVFS *Dynamic Voltage and Frequency Scaling*, tensión dinámica y escalado de frecuencia.

EGk Exp-Golomb de orden k.

FIFO *First In First Out*.

FL *Fixed-Length*.

FPGA *Field Programmable Gate Array*.

FPS Fotogramas Por Segundo (del inglés *Frames Per Second*).

IGIC Impuesto General Indirecto Canario.

IOB *I/O Block*, bloque de E/S.

IUMA Instituto Universitario de Microelectrónica Aplicada.

JVT *Joint Video Team*.

LPS *Least Probable Symbol*, símbolo menos probable.

LSB *Least Significant Bit*, bit menos significativo.

LUT *Look-Up Table*.

MB Macrobloque.

MEF Máquina de Estados Finitos.

MMCM *Mixed-Mode Clock Managers*.

MPEG *Moving Picture Experts Group*, grupo de expertos en imágenes en movimiento.

MPM Modo Más Probable.

MPS *Most Probable Symbol*, símbolo más probable.

MSB *Most Significant Bit*, bit más significativo.

NAL *Network Abstraction Layer*, capa de abstracción de red.

NRE *Non-Recurring Engineering*, costes de ingeniería no recurrentes.

PLL bucle de enganche de fase (del inglés *Phase-Locked Loops*).

QP *Quantization Parameter*, parámetro de cuantización.

RGB *Red Green Blue*.

RTL *Register Transfer Level*, nivel de transferencia de registros.

SE *Syntax Element*.

TFG Trabajo de Fin de Grado.

TU *Truncated Unary*.

U *Unary*.

UEGk *Concatenated Unary/k-th order Exp-Golomb*.

UHD *Ultra High Definition*.

ULPGC Universidad de las Palmas de Gran Canaria.

UVVM *Universal VHDL Verification Methodology*.

VCEG *Video Coding Experts Group*, grupo de expertos en codificación de vídeo.

VHDL *Very high speed integrated circuit Hardware Description Language*.

VLC *Variable Length Coding*.

Capítulo 1

Introducción

En este capítulo se exploran los motivos subyacentes a la realización de este proyecto. Asimismo se incluyen las necesidades y objetivos que pretende cubrir, concluyendo con una breve descripción de la organización de este documento.

1.1. Antecedentes

La codificación de vídeo se corresponde con el proceso de comprimir y descomprimir una señal digital de vídeo [1]. Este proceso juega un papel crucial en la sociedad moderna, ya que, con los avances tecnológicos, son cada vez más demandadas las aplicaciones relacionadas con el *streaming* de vídeo de alta calidad. En la codificación y decodificación de vídeo, la demanda de computación es bastante alta, puesto que se trata de buscar correlaciones dentro de los distintos píxeles y fotogramas (*frames*) de una secuencia de vídeo [2]. Ello toma especial importancia en equipos que cuentan con limitaciones de almacenamiento y de velocidad de transmisión, como puede ser en los satélites.

Los datos proporcionados por las imágenes satelitales son ampliamente utilizados en varios campos de investigación. Algunos de ellos son la agricultura, geología, monitorización de la degradación de la tierra, estudios del océano y meteorología, entre muchos otros [3].

La compresión explota el hecho de que las imágenes captadas dentro de una secuencia de vídeo cuentan con las ya mencionadas correlaciones o redundancias. Dichas redundancias se manifiestan significativamente en imágenes capturadas por satélites de observación de la Tierra. Por ejemplo, debido a la vasta distancia que existe entre la superficie terrestre y los propios satélites, aparecen en gran medida las redundancias espaciales. Como se muestra en la Figura 1.1, en este tipo de imágenes pueden aparecer patrones geográficos repetitivos, como en la observación de un bosque, una masa de agua o incluso una ciudad. Por otro lado, las diferencias entre fotogramas sucesivos son mínimas, pues el movimiento de objetos dentro del campo de visión es prácticamente inapreciable [4].

Figura 1.1: Ejemplo de imagen satelital con alta redundancia espacial.[1]

Además, es importante mencionar que el ancho de banda de las conexiones de los satélites con las estaciones terrestres (*downlink*) es relativamente limitado y cada vez está más saturado. Igualmente, la capacidad de almacenamiento de estos vehículos espaciales también está condicionada en cierto modo, pues no siempre es posible transmitir datos dependiendo de la ubicación en la que se encuentren. Por ende, es esencial reducir la cantidad de bits a transmitir a través de dicho enlace, reforzándose así la necesidad de aplicar técnicas de compresión a los vídeos enviados.

Todo lo mencionado hasta ahora, sumado también al coste computacional que supone la compresión de vídeo, obligan a la implementación de un *hardware* para este fin. Para la realización del mismo, surgen dos principales posibilidades a valorar: el uso de un circuito integrado de aplicación específica (del inglés *Application Specific Integrated Circuit*) (ASIC) o la utilización de una *Field Programmable Gate Array* (FPGA).

Cabe destacar que tanto las FPGA como los ASIC son tipos diferentes de circuitos integrados, siendo la principal diferencia entre ellos que los primeros son reprogramables tras su fabricación, mientras que los segundos no. Ambos se utilizan para desempeñar tareas específicas, donde la paralelización es clave. Asimismo, las FPGA son frecuen-

[1]Imagen: *Zaslawskaye Reservoir satellite photo.*
Autor: Agencia Espacial Europea.
Licencia: Creative Commons Attribution-ShareAlike 3.0 IGO.
Fuente: WikiMedia.

temente usadas en la fase de prototipado de los ASIC, debido a la flexibilidad que la mencionada capacidad de reprogramarse les otorga.

A continuación, se incluye una breve descripción de ambas tecnologías, señalando sus ventajas e inconvenientes de acuerdo con [5]-[7].

FPGA:

1. Están formadas por bloques lógicos programables que implementan lógica secuencial y lógica combinacional. Dichos bloques pueden ser de varios tipos y finalidades diferentes. Los tipos básicos que pueden encontrarse en una FPGA son: bloques lógicos configurables (del inglés *Configurable Logic Blocks*) (CLB), bloques de E/S (del inglés *I/O Blocks*) (IOB), memorias y multiplicadores, entre otros. Todos estos bloques se hallan rodeados de una matriz de interconexión que admite un sinfín de configuraciones posibles.

2. Como previamente se mencionaba, el hecho de ser reconfigurables les otorga una gran flexibilidad, que, además, implica un menor tiempo de desarrollo.

3. Como consecuencia de lo expuesto en 2), los diseños realizados sobre FPGA cuentan con menores costes costes de ingeniería no recurrentes (del inglés *Non-Recurring Engineering*) (NRE).

ASIC:

1. La principal ventaja de los ASIC sobre las FPGA es que únicamente se fabrica el *hardware* necesario para desempeñar la tarea para la cual fue diseñado. Ello implica un menor coste unitario y en torno a 30 o 40 veces menor área ocupada, aunque su escasez de flexibilidad supone unos costes NRE significativamente mayores respecto al diseño en FPGA.

2. Por otro lado, con este tipo de circuitos integrados se hace posible implementar técnicas avanzadas de gestión de energía como, por ejemplo, tensión dinámica y escalado de frecuencia (del inglés *Dynamic Voltage and Frequency Scaling*) (DVFS). Estas técnicas suponen un ahorro energético entre 12 y 14 veces mayor al que se consigue con FPGA.

3. En cuanto a prestaciones se refiere, también se alcanza entre 4 y 5 veces mayor velocidad en un ASIC que en una FPGA.

Dada la naturaleza del proyecto, se pretende diseñar e implementar la etapa de codificación entrópica *Context Adaptative Binary Arithmetic Coding* (CABAC), del estándar H.264/AVC mediante el lenguaje *Very high speed integrated circuit Hardware Description Language* (VHDL). En el anteproyecto se indicó que la implementación se realizaría en la tarjeta AMD Zynq 7000 SoC ZC706 *Evaluation Kit*, ya que suele utilizarse en la fase de prototipado de la AMD Xilinx XCU060, cualificada para su uso a bordo de satélites. Sin embargo, la tarjeta finalmente utilizada fue la Nexys 4 DDR como se expone en el Capítulo 4.

1.2. Objetivos

El objetivo de este proyecto se corresponde con describir en VHDL, verificar e implementar el algoritmo CABAC, del estándar de codificación H.264, en una FPGA para su empleo a bordo de satélites. La resolución de vídeo máxima será *Ultra High Definition* (UHD) (3840x2160) a una tasa de 10 Fotogramas Por Segundo (del inglés *Frames Per Second*) (FPS). Dicho objetivo general se desglosa en los siguientes objetivos específicos:

O1. Estudiar el algoritmo CABAC y algunas arquitecturas usadas.

O2. Realizar la descripción *hardware* en VHDL del algoritmo CABAC.

O3. Verificar que el sistema cumple con las especificaciones proporcionadas.

O4. Implementar el algoritmo en una FPGA conjuntamente con el resto de etapas que conforman el estándar H.264.

O5. Generar la documentación del proyecto realizado.

Además, cabe señalar que el desarrollo del algoritmo mencionado se realiza bajo una serie de restricciones que surgen tanto de cuestiones contractuales del proyecto de

investigación del que forma parte este Trabajo de Fin de Grado (TFG), *Efficient Video Compression for space (ESA AO/1-1-10954/21/NL/MGu)* (vinculado a *Thales Alenia Space en España (Sector Espacial)*) como a la gran extensión y complejidad de CABAC. Debido a ello, y para limitar la extensión temporal del proyecto, se han establecido las siguientes restricciones que se comentan a continuación y más detenidamente en la Subsección 2.3.2:

- Imágenes monocromas.
- Predicción *intraframe*.
- Tres modos de predicción Intra 4x4.
- No se considera la codificación en modo *field*.
- Un *slice* por fotograma.
- El parámetro de cuantización (del inglés *Quantization Parameter*) (QP) para todos los Macrobloques (MB) siempre es igual al del *slice* al que pertenecen.

1.3. 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 explica brevemente el trasfondo y la motivación que desencadenan en la realización de este TFG.
- **Capítulo 2:** se exponen los conceptos básicos de codificación de vídeo necesarios para facilitar la comprensión del presente documento. A continuación, se realiza una breve introducción a H.264, haciendo una descripción detallada de CABAC. También se describen los elementos sintácticos relacionados con dicho codificador, las relaciones entre ellos, y las restricciones del diseño y su efecto sobre la utilización de algunos de estos elementos.

- **Capítulo 3:** se aborda la solución *hardware* implementada acorde a las funcionalidades, a las restricciones y a los elementos sintácticos descritos en el Capítulo 2. De igual manera, se especifican los procesos y herramientas empleados en la verificación del diseño.

- **Capítulo 4:** se incluyen las prestaciones obtenidas por el diseño. Asimismo, se recoge la valoración del trabajo realizado, las lecciones aprendidas y las líneas futuras de trabajo que surgen de la realización de este proyecto.

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

Capítulo 2

Análisis del estado del arte

En este capítulo se exponen conceptos básicos relativos a la codificación y compresión de vídeo. Se incluye un breve resumen del estándar H.264, mencionando algunos aspectos claves del mismo, y se presenta una vista global del codificador entrópico *Context Adaptative Binary Arithmetic Coding* (CABAC). Posteriormente, se detalla cada una de las etapas en las que se subdivide dicho codificador: el binarizador, el modelador de contextos y el codificador binario aritmético (del inglés *Binary Arithmetic Coder*) (BAC).

A continuación, se tratan en profundidad todos los aspectos de necesario conocimiento acerca de los Elementos Sintácticos (del inglés *Syntax Elements*) (SE) haciendo especial hincapié en aquellos relacionados con CABAC. De igual manera, se enumeran los SE utilizados en el proyecto, atendiendo a las restricciones introducidas en el Capítulo 2 y posteriormente detalladas en la Subsección 2.3.2. Finalmente, se abordan cuestiones como el orden de codificación de los SE seleccionados, así como el cálculo de sus índices de contexto.

2.1. Codificación de vídeo

Un vídeo digital está compuesto de una sucesión de imágenes, donde, a su vez, cada imagen está formada por un conjunto de puntos de muestreo llamados píxeles [8]. El muestreo de vídeo típico suele emplear el espacio de color *Red Green Blue* (RGB) [8]. Cada uno de estos colores primarios, rojo, azul y verde, se representan con un número determinado de bits. Dicho número se corresponde con la profundidad de color. Según se incrementa este parámetro, también lo hace el número de colores a captar. Generalmente, la profundidad de color es uniforme para todos los canales, aunque en ocasiones puede variar como es el caso del RGB565. Como su propio nombre indica, para el rojo y el azul se utilizan 5 bits, mientras que para el verde se emplean 6.

Por otro lado, el tamaño y relación de aspecto de la imagen vienen determinadas por su resolución espacial, que expresa las dimensiones de la misma en píxeles [9]. A mayor resolución, más píxeles tendrá la imagen y, por tanto, un nivel de detalle superior.

Por ejemplo, la tasa de bits, R_b, necesaria para transmitir un vídeo crudo monocromo en *Ultra High Definition* (UHD) (3840x2160) a 10 Fotogramas Por Segundo (del inglés *Frames Per Second*) (FPS), con una profundidad de color de 10 bits utilizando RGB es[1]:

$$píxeles_{frame} = 3\ 840 \cdot 2\ 160 = 8\ 294\ 400\ píxeles/frame$$

$$píxeles_s = píxeles_{frame} \cdot FPS = 8\ 294\ 400 \cdot 10 = 82\ 944\ 000\ píxeles/s$$

$$bits_{píxel} = num\_bits_{profundidad\_color} = 10\ bits/píxel$$

$$R_b = píxeles_s \cdot bits_{píxel} = 82\ 944\ 000 \cdot 10 \rightarrow \boxed{R_b = 829,44\ Mbps}$$

Esta cantidad de información resulta muy alta, tanto para ser transmitida como para almacenarse. Por tanto, surge la necesidad de aplicar técnicas de compresión de vídeo, pudiendo de esta manera representar los mismos datos empleando muchos menos bits. La compresión puede ser con pérdidas (*lossy*) o sin pérdidas (*lossless*). Si se emplea

[1] Algunas cámaras cuentan con sensores tipo *Bayer Grid*, donde cada píxel solamente registra una componente de color. En ocasiones se envía este *bitstream*, como se plantea en este ejemplo, para posteriormente procesarlo y obtener la imagen completa a color.

la primera, se podrá reconstruir una aproximación de los datos originales, mientras que si se utiliza la segunda, se obtendrá una copia exacta de estos datos [1]. Sin embargo, la técnica que implica pérdidas es capaz de conseguir mayores ratios de compresión que la otra alternativa mencionada, aunque ha de considerarse el empeoramiento en calidad de imagen que ello conlleva [1].

2.1.1. Conceptos de Macrobloque y *Slice*

Al comienzo de la esta sección se comentó que una imagen estaba formada por un número determinado de píxeles. En este contexto, un Macrobloque (MB) se corresponde a una región de 16x16 píxeles, siendo la unidad básica para la predicción con compensación de movimiento en varios estándares de codificación de vídeo como MPEG-1, MPEG-2, MPEG-4 Visual, H.261, H.263 y H.264 [1]. Asimismo, estos MB pueden dividirse de varias formas, siendo de interés para este proyecto las subdivisiones en Bloques 8x8 (B8) y Bloques 4x4 (B4), tal como aparecen en la Figura 2.1.

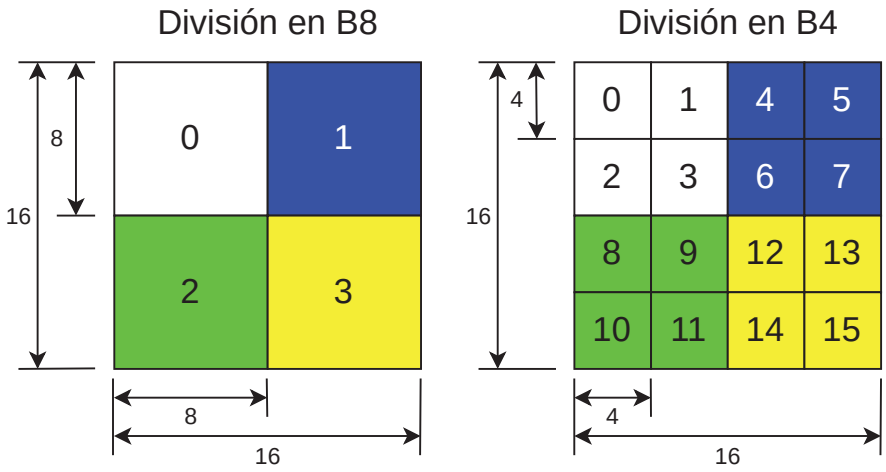

Figura 2.1: Ejemplos de subdivisión de un macrobloque.

Por otro lado, los *slices* son particiones independientes que pueden realizarse dentro de un fotograma, donde cada uno está formado por un grupo de MB pertenecientes al mismo *frame* [2], [10].

Además, cabe resaltar que cada imagen puede hacer referencia a, efectivamente un fotograma completo, o a un campo [11]. Generalmente, un *frame* puede dividirse en

dos campos: superior (*top field*) e inferior (*bottom field*). El superior abarca las filas de píxeles pares, mientras que el inferior contiene las impares. Es decir, cada campo comprende la mitad de información de un *frame* [1]. Si ambos campos son capturados en el mismo instante temporal, se estará empleando el modo progresivo. Sin embargo, si se muestrean en instantes distintos, el modo de codificación utilizado será el entrelazado [11], [12]. En la Figura 2.2 se representa una secuencia de vídeo en modo entrelazado.

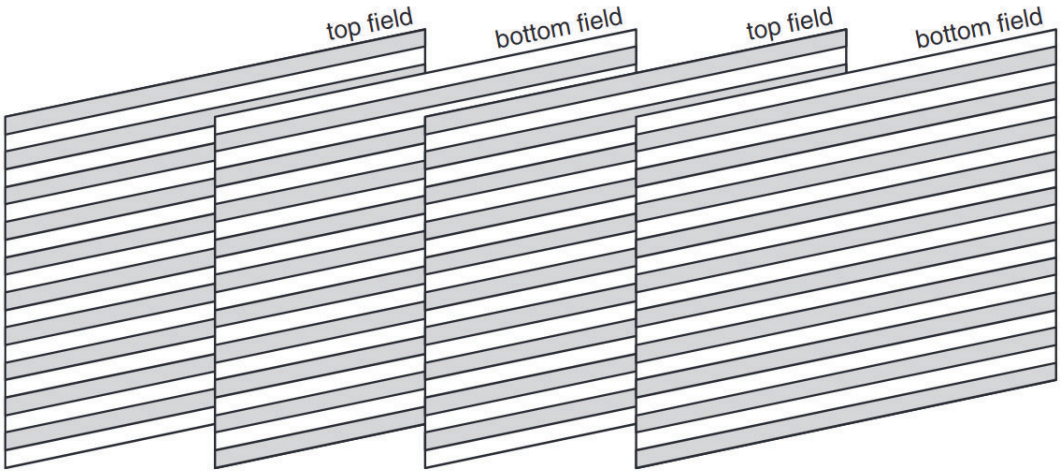

Figura 2.2: Secuencia de fotogramas en modo entrelazado. Fuente [1].

2.1.2. Tipos de redundancias

Como se mencionaba previamente en la Sección 1.1, la compresión de vídeo consiste en eliminar las redundancias existentes en las imágenes [1]. Dichas redundancias, pueden manifestarse en varias formas. Según [1] y [2], son:

-Redundancia espectral: ocurre cuando en una imagen hay información similar entre los diferentes canales de color o componentes de luminancia y crominancia, como es el caso de YUV o YCbCr.

YCbCr es un espacio de color, más eficiente que RGB, utilizado en H.264. Este espacio de color, separa la luminancia de la crominancia. Las señales de crominancia, *Cb* y *Cr*, se obtienen al calcular la diferencia entre las componentes azul, `B`, y roja, *R*, de RGB con la señal de luminancia, *Y*. Por ende, dichos modelos cromáticos se relacionan entre sí de la siguiente manera [2], [8]:

$$Y = 0{,}299 \cdot R + 0{,}587 \cdot G + 0{,}114 \cdot B$$

$$Cb = 0{,}564 \cdot (B - Y)$$

$$Cr = 0{,}713 \cdot (R - Y)$$

YCbCr incluye varios formatos de muestreo, siendo los más típicos 4:4:4, 4:2:2 y 4:2:0. En el primer caso, las tres componentes (*Y, Cb* y *Cr*) cuentan con la misma resolución, por lo que existe una muestra de cada una por cada píxel [1].

No obstante, aprovechando que el ojo humano es más sensible a los cambios de iluminación que de color, en 4:2:2 y 4:2:0 se realiza un submuestreo de las componentes cromáticas [8]. Por consiguiente, en 4:2:2 se tienen únicamente dos muestras de *Cb* y *Cr* por cada cuatro muestras de luminancia en la dirección horizontal. Por otro lado, en 4:2:0 la resolución de las señales de croma es la mitad tanto en dirección horizontal como en dirección vertical respecto a la de luminancia [1]. En la Figura 2.3 se comparan gráficamente estos dos últimos esquemas de color.

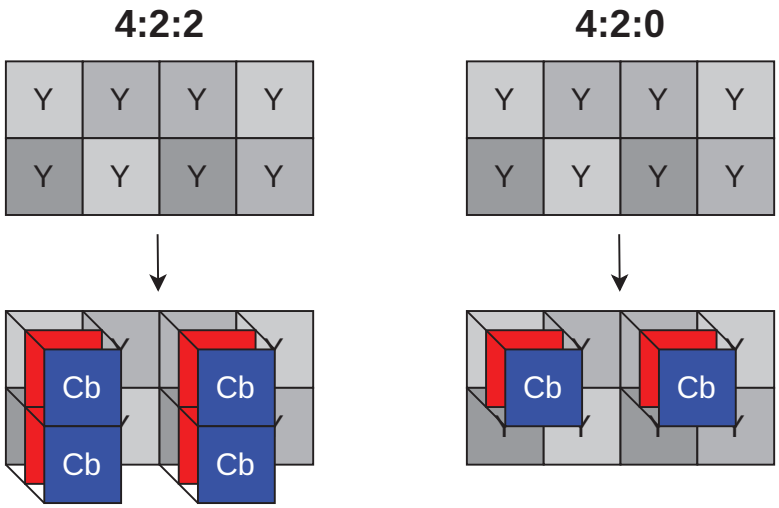

Figura 2.3: Comparativa entre 4:2:2 y 4:2:0. Adaptada de [1].

-Redundancia espacial: es aquella que aparece dentro de una imagen (*intraframe*) debido a la similitud entre píxeles adyacentes, como puede ser en los ejemplos ya mencionados en la Sección 1.1 de observación de masas de agua o bosques. Por tanto, en lugar de transmitir toda la información de cada píxel, es más eficiente enviar la

diferencia entre el valor de un píxel y el valor estimado a partir de sus vecinos.

-Redundancia temporal: tiene lugar entre fotogramas sucesivos (*interframe*), pues es común que las diferencias entre estos *frames* sean mínimas, especialmente si la frecuencia de muestreo es alta. Por ende, en lugar de enviar un fotograma prácticamente idéntico al anterior, resulta una mejor alternativa enviar las diferencias entre ellos. En la Figura 2.4 se representa gráficamente un ejemplo que compara este tipo de redundancia con la anterior.

Figura 2.4: Redundancia temporal y espacial. Fuente [1].

-Redundancia estadística: aparece tras las compresiones espaciales y/o temporales. Guarda relación con la probabilidad de aparición de ciertos símbolos en base a otros. Considerando esto, las técnicas de reducción de redundancia estadística tratan de representar dichos símbolos de manera eficiente, siendo común emplear menos bits para aquellos elementos más frecuentes.

2.1.3. Introducción a H.264

Tras haber introducido algunos conceptos relativos a la codificación de vídeo, ahora se expone un breve resumen acerca de los aspectos más importantes del estándar en torno al cual gira este proyecto: H.264.

H.264, MPEG-4 parte 10 o codificación de vídeo avanzada (del inglés *Advanced Video Coding*) (AVC) es un estándar de compresión de vídeo que surge de la alianza estratégica entre el grupo de expertos en imágenes en movimiento (del inglés *Moving Picture Experts Group*) (MPEG) de la ISO/IEC, y el grupo de expertos en codificación de vídeo (del inglés *Video Coding Experts Group*) (VCEG) de la ITU-T. El equipo resultante, y por tanto encargado de desarrollar el estándar, se conoce por el nombre de *Joint Video Team* (JVT) [13].

Este estándar ha tenido y tiene gran peso en la transmisión de vídeo y en electrónica de consumo, entre otros [1]. Ello es debido a las técnicas de codificación que permiten obtener bajas tasas binarias para altas resoluciones [8]. Hoy por hoy, es tal su importancia que los nuevos chips de la serie M4 de Apple, entre muchos otros, incluyen aceleración por *hardware* para H.264 y su evolución, H.265.

Como es lógico, H.264 cuenta con mejoras respecto a sus antecesores. Una de ellas es la mayor robustez frente a errores de transmisión, pues está diseñado de tal forma que cada paquete transmitido es independiente al resto, por lo que puede ser decodificado por sí mismo [13]. Además, también son de importancia las mejoras que se consiguen en predicción por compensación de movimiento, gracias a los nuevos *slices* tipo B que se comentan en el Apartado 2.1.3.3. Asimismo, introdujo avances en eficiencia de codificación, ratios de compresión y soporte para diferentes tasas de bits, entre otros [13].

2.1.3.1. Flujo de datos

Entrando en materia, en la Figura 2.5 se expone el flujo de datos del codificador propuesto en el estándar. Este se halla divido en cuatro principales partes resaltadas en color: predicción (azul), transformación (rosa), cuantización (amarillo) y codificación entrópica (verde). De acuerdo con [1], [8], [14], el funcionamiento es el siguiente:

En primer lugar, se generan las señales de predicción, que pueden ser el resultado de la predicción *interframe* o *intraframe* según corresponda. Nótese que en el primer

caso se toman uno o dos fotogramas de referencia previamente codificados, mientras que en el segundo únicamente se utilizan datos de macrobloques vecinos. Este último aspecto se desarrolla más adelante en el Apartado 2.3.4.1.

Acto seguido, a la diferencia entre la imagen original y el resultado de la predicción (conocida como datos residuales) se le aplica la transformada discreta del coseno (del inglés *Discrete Cosine Transform*) (DCT). Es decir, se realiza una transformación en el dominio del espacio al de frecuencia, obteniendo coeficientes. Es preciso señalar que la DCT es un proceso completamente reversible y no produce pérdidas por sí sola.

A continuación, se dividen los coeficientes resultantes por los de una matriz de cuantización, relacionada con el parámetro de cuantización (del inglés *Quantization Parameter*) (QP). Cuanto mayor es el QP, que varía entre 0 y 51, mayores serán los coeficientes de dicha matriz, eliminándose mayor cantidad de información.

Finalmente, los coeficientes residuales cuantizados son reordenados e introducidos al codificador entrópico que corresponda. Concretamente, H.264 cuenta con dos: CABAC y *Context Adaptative Variable Length Coding* (CAVLC). En la Sección 2.2 se profundiza en el primero, pues es el desarrollado en este proyecto.

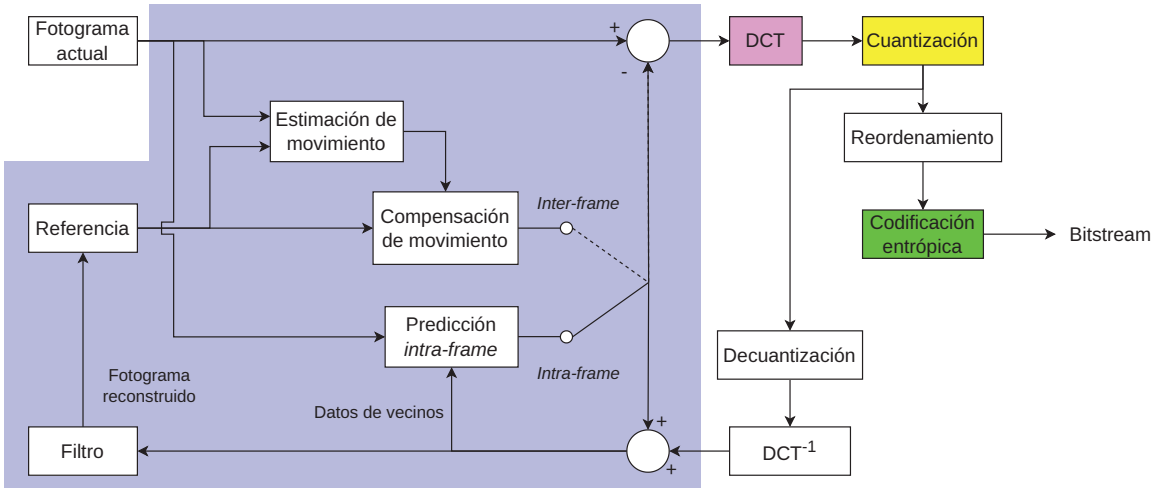

Figura 2.5: Flujo de datos de un codificador H.264. Adaptado de [8].

2.1.3.2. Perfiles

Debido a la complejidad y extensión de H.264, este se divide en una serie de perfiles, que no son más que subconjuntos de herramientas que pretenden simplificar las implementaciones del estándar a la vez que tratan de garantizar su compatibilidad entre dispositivos y aplicaciones. Existen un total de once perfiles más sus respectivas variaciones como se puede encontrar en [12].

Seguidamente se incluye un breve resumen de los tres perfiles más importantes: *baseline* o base, *main* o principal y *extended* o extendido. Según [8], [11], [15] se tiene:

- ***Baseline:*** está destinado a aplicaciones de baja complejidad, tales como conferencias de vídeo. No incluye *slices* tipo B, SP ni SI, ni codificación en modo campo (vídeo entrelazado), ni CABAC, entre otros.
- ***Main:*** es ampliamente utilizado en *streaming* de vídeo. En este perfil ya se añade lo mencionado en el punto anterior, excepto los *slices* SP y SI.
- ***Extended:*** combina los perfiles base y principal, de manera que aprovecha tanto la flexibilidad del primero, como la eficiencia del segundo.

Resumiendo y precisando, en la Figura 2.6 se halla la representación gráfica de las especificaciones de estos tres perfiles:

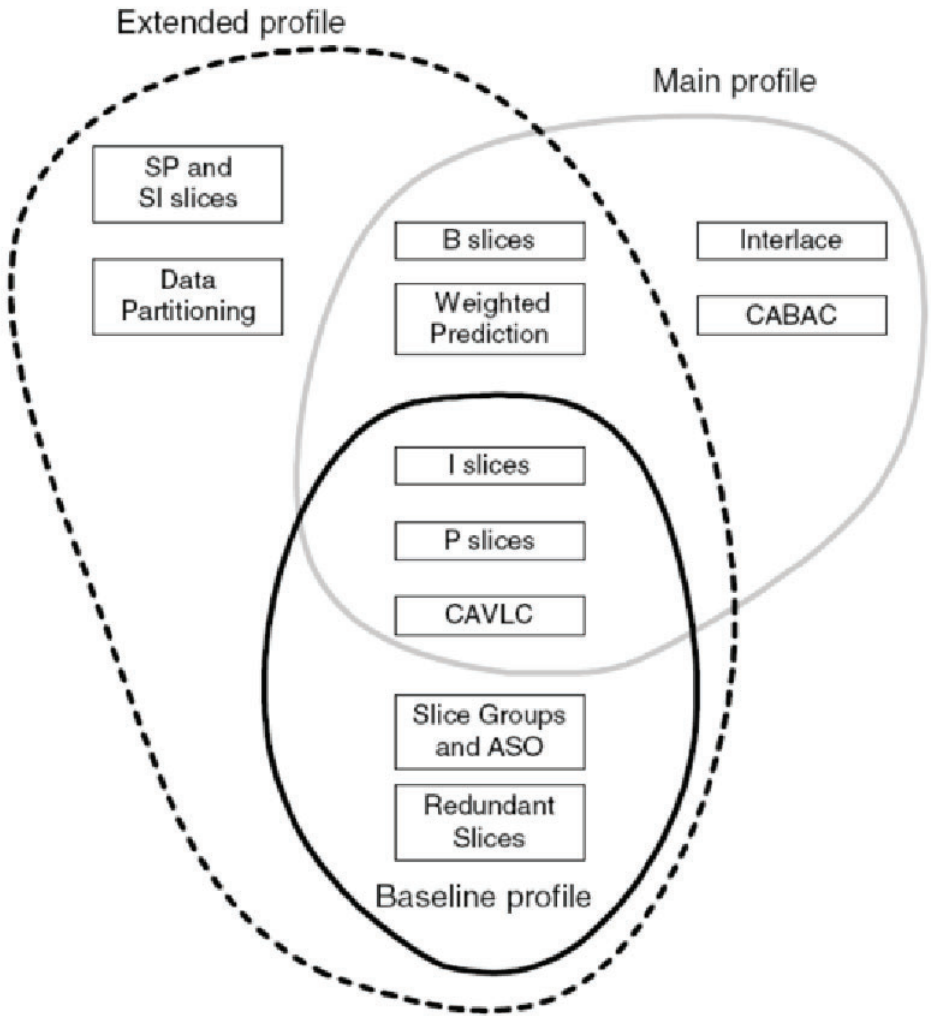

Figura 2.6: Principales perfiles de H.264. Fuente [8].

2.1.3.3. Tipos de *slices*

Como bien se mencionaba en el apartado anterior, en H.264 existen varios tipos de *slices*. De acuerdo a [8], [10], [11], estos son:

- **I:** únicamente contiene MB codificados mediante predicción *intraframe*. En otras palabras, el tipo de predicción empleado para aquellos MB pertenecientes a este tipo de *slices* es la que se enfoca en aprovechar la redundancia espacial.
- **P:** están formados por MB codificados mediante tanto predicción *intraframe* como

interframe. Cabe señalar que este segundo tipo de predicción es el encargado de explotar la redundancia temporal.

- **B:** acogen los mismos tipos de MB que los P. Sin embargo, a diferencia de los anteriores, la predicción por compensación de movimiento puede realizarse empleando datos procedentes de dos referencias: además de los de un *frame* previo, también se permite el uso de datos pertenecientes a un *frame* posterior[2].

- **SP y SI:** son tipos de *slices* especiales orientados al control de tasa de bits y al control de errores.

2.2. CABAC

CABAC es uno de los dos codificadores entrópicos definidos en H.264/AVC, el cual cuenta con una tasa de bit en torno a un 10-15 % menor para la misma calidad que la alternativa restante, CAVLC [10]. Dicha mejora en compresión se consigue, de acuerdo a [1], gracias a:

- Seleccionar los modelos de probabilidad para cada SE.

- Adaptar las estimaciones de probabilidad basadas en estadísticas locales.

- Emplear codificación aritmética (del inglés *Arithmetic Coding*) (AC), en lugar de *Variable Length Coding* (VLC).

Sin embargo, CABAC presenta dos principales desventajas: mayor complejidad de implementación frente a CAVLC y también mayor lentitud de cómputo causada por las dependencias de datos, propias de la naturaleza serial y recursiva del algoritmo [10], [16], [17].

Atendiendo a su estructura, en la Figura 2.7 se observan los distintos bloques que la componen: el binarizador, el modelador de contextos (del inglés *Context Modeler*) (CM) y el BAC, que a su vez se divide en el *regular coding engine* y en el *bypass coding*

[2]El orden de visualización no necesariamente coincide con el orden de codificación. En este caso, los fotogramas mencionados, anterior y posterior, lo son en orden de visualización.

engine. Dichos bloques serán posteriormente tratados en detalle en las Secciones 2.2.1, 2.2.2 y 2.2.3, respectivamente.

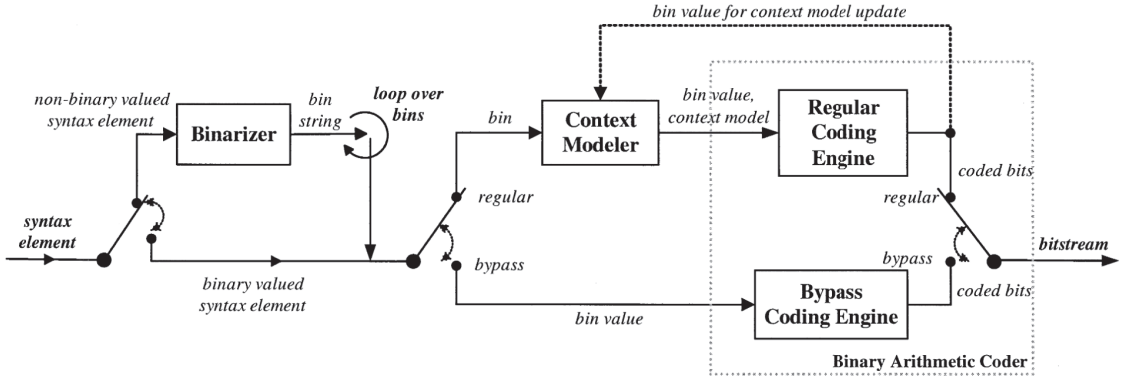

Figura 2.7: Diagrama de bloques de CABAC. Fuente [16].

2.2.1. Binarizador

La función del binarizador consiste en convertir a binarios los símbolos o SE que originalmente no lo son, como es el caso de los vectores de movimiento o los coeficientes residuales cuantizados de transformadas, entre otros [1]. Cada SE se binariza, formando un *binstring*, que no es más que un conjunto de longitud variable de bits, comúnmente conocidos *bins* para diferenciarlos de aquellos que todavía no han atravesado este proceso. De esta manera se simplifica el modelado de contextos, ya que el tamaño del alfabeto se reduce a tan solo dos elementos: 0 y 1. Así, se posibilita el uso de un codificador binario aritmético en lugar de un codificador M-ario siendo M el número de elementos que componen el alfabeto. Esto reduce la complejidad computacional a la vez que se facilita una estimación rápida y precisa de las probabilidades condicionales de cada símbolo[3] [16].

En [12] se especifican los distintos modos de binarización que existen para CABAC en H.264. Estos son:

- *Fixed-Length (FL)*
- *Unary (U)*

[3]Nótese que la estimación de probabilidades se realiza a nivel de subsímbolo siendo la probabilidad del símbolo original no binario igual al producto de las probabilidades de los *bins* individuales de la cadena de *bins* [16].

- *Truncated Unary (TU)*
- *Concatenated Unary/k-th order Exp-Golomb (UEGk)*
- Tablas con *binstrings* predefinidos y concatenación tipos de codificación.

2.2.1.1. Fixed-Length (FL)

Este modo de binarización es aplicado a SE con una distribución de probabilidades prácticamente uniformes [16]. Más adelante, en la Subsección 2.3.3 se concreta el tipo de binarización que le corresponde a cada elemento sintáctico.

Como su propio nombre indica, *Fixed-Length* se basa en la generación de códigos de longitud fija. Dichos códigos se corresponden con el valor en binario del elemento, `SE_val`, escrito con un determinado número de *bins*, el cual viene dado por $log_2(\texttt{cMax})$, siendo `cMax` el mayor valor a reperesentar [1], [12], [16]. Algunos ejemplos pueden observarse en la Tabla 2.1.

SE_val	**binstring**	
	cMax = 7	**cMax = 15**
0	000	0000
1	001	0001
2	010	0010
3	011	0011
4	100	0100
5	101	0101

Tabla 2.1: Ejemplo de codificación Fixed-Length.

2.2.1.2. Unary (U) y Truncated Unary (TU)

La binarización *Unary* (U), representa los valores de los SE, `SE_val`, como un *binstring* compuesto por tantos unos como unidades tenga `SE_val`, y un 0 para el *bin* con índice de *bin*, `binIdx`, igual a `SE_val` [12]. Es decir, si `SE_val` = 2, el *binstring*

resultante será "110". Nótese que el *bin* con `binIdx` = 0 es el primero que se encuentra comenzando por la izquierda.

Por otro lado, *Truncated Unary* (TU) se puede definir como una ligera modificación de U, donde nuevamente aparece `cMax`, actuando en este caso, como un valor de corte a partir del cual la binarización de cualquier SE con `SE_val` $\geq$ `cMax`, consiste en un *binstring* de tantos unos como unidades tenga `cMax`, pero sin un '0' al final [12]. En la Tabla 2.2 se muestran algunos ejemplos.

SE_val	Unary (U)	Truncated Unary (TU) cMax = 5
0	0	0
1	10	10
2	110	110
3	1110	1110
4	11110	11110
5	111110	11111
6	1111110	11111
7	11111110	11111
8	111111110	11111
9	1111111110	11111
10	11111111110	11111

Tabla 2.2: Ejemplo de codificación U y TU.

2.2.1.3. Concatenated Unary/k-eth order Exp-Golomb (UEGk)

Los *binstrings* producidos por UEGk se emplean en la binarización de coeficientes residuales y cuantificados de transformadas, `coeff_abs_level_minus1`, y en la de vectores de movimiento, `mvd_lX`. Estos *binstrings* constan de dos partes: un prefijo y un sufijo. El prefijo se construye a partir de un código TU invocado con `cMax = uCoff`, siendo `uCoff` el valor de corte para cada SE. El sufijo se corresponde con un el resultado de aplicar una variante adaptada para codificación de vídeo de Exp-Golomb de orden k (EGk), que puede variar dependiendo del SE [12], [16].

En la Tabla 2.3 se recogen algunos ejemplos de binarización UEG0, la cual emplea

EG0 para los sufijos. Nótese que dichos modos de codificación hacen referencia a UEGk y EGk con $k = 0$, respectivamente. En el Algoritmo 1 se halla el pseudo-código mediante el cual se puede implementar este tipo de binarización.

SE_val	Prefijo (TU) uCoff=5	Sufijo (EG0)
0	0	-
1	10	-
2	110	-
3	1110	-
4	11110	-
5	11111	0
6	11111	100
7	11111	101

Tabla 2.3: Ejemplo de codificación UEG0 con `uCoff` = 5.

Algoritmo 1 Pseudo-código Exp-Golomb de orden k. Adaptado de [12].

```
if(abs(signedVal) >= uCoff){
  sufs = abs(signedVal) - uCoff;
  stopLoop = 0;
  do {
    if(sufs >= (1<<k)){
      put(1);
      sufs = sufs - (1<<k);
      k++;
    } else{
      put(0);
      while(k--){
        put((sufs >> k) & 1);
      }
      stopLoop = 1;
    }
  } while(!stopLoop);
}
if(signedValFlag && (signedVal != 0)){
  if(signedVal > 0){
    put(0);
  } else{
    put(1);
  }
}
```

2.2.1.4. Tablas con binstrings predefinidos y concatenación de tipos de codificación

H.264 emplea modos de binarización especiales para ciertos SE, como ocurre con `mb_type` y `coded_block_pattern`. Para el primer SE, se utiliza la Tabla 2.4 de *binstrings* predefinidos, obtenidos a partir de un árbol de Huffman [16]. En cuanto al segundo, se emplea codificación FL para la parte asociada a la luminancia, mientras que para la relativa a la crominancia, se utiliza la binarización TU.

Valor de `mb_type`	Binstring
0	0
1	100000
2	100001
3	100010
4	100011
5	1001000

Tabla 2.4: Binarización de `mb_type`. Adaptada de [12].

2.2.2. Modelador de Contextos

El CM, es el bloque responsable de asignar los contextos a todos los *bins* de los *binstrings* que serán codificados por el *regular coding engine*. El codificador aritmético y los tipos de codificación se abordan en la Subsección 2.2.3.

2.2.2.1. Concepto de modelo de contexto

Como se mencionaba al inicio de este capítulo, un motivo por el cual CABAC consigue un mayor rendimiento de compresión que CAVLC era la selección y actualización de modelos de probabilidad para cada SE. Dichos modelos de probabilidad se representan en H.264 por medio de los modelos de contexto, los cuales se nombrarán a partir de

ahora *contextos*.

Un contexto es un conjunto de siete bits, donde seis son destinados al índice de estado de la probabilidad, `pStateIdx`, y el restante al valor del símbolo más probable (del inglés *Most Probable Symbol*) (MPS), `valMPS` [16]. En el estándar existen más de mil contextos, utilizándose un subconjunto de ellos en cada *slice*. Dicho subconjunto viene determinado por el estado de la codificación en cada momento, accediéndose a cada contexto a través de su índice de contexto, `ctxIdx`. En las Figuras 2.8 y 2.9 se muestra la representación gráfica de un contexto:

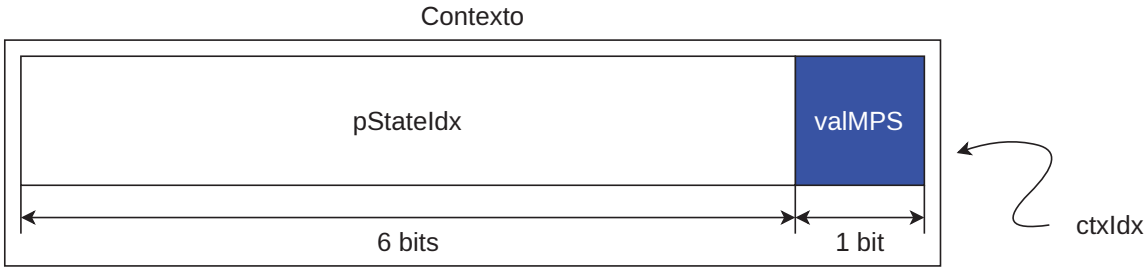

Figura 2.8: Representación gráfica de un contexto.

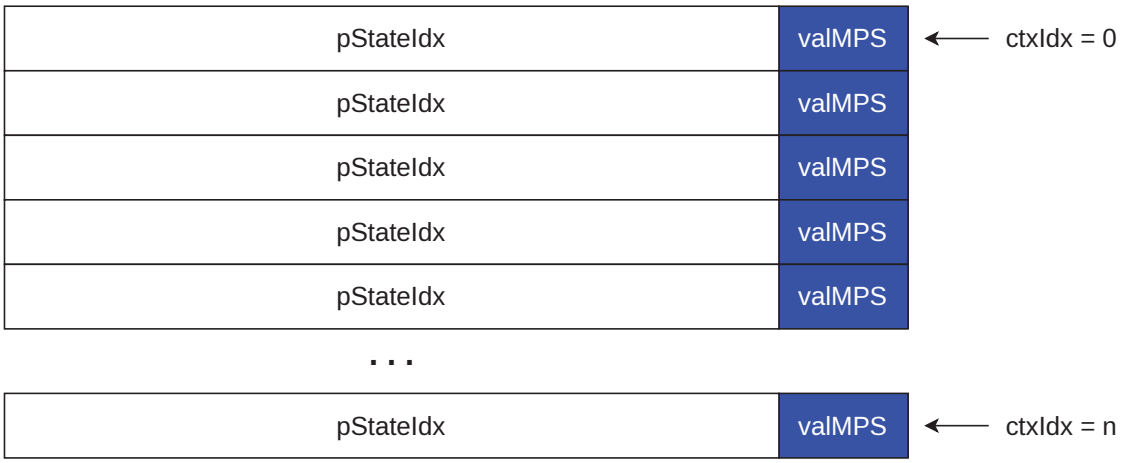

Figura 2.9: Representación gráfica de una tabla de contextos.

2.2.2.2. Inicialización de contextos

Al principio de cada *slice*, todos los contextos a utilizar en este han de ser inicializados. Para ello, se sigue el proceso descrito en el pseudo-código del Algoritmo 2. El proceso mencionado requiere de tres parámetros: `m, n` y $\texttt{SliceQP}_Y$. Los dos primeros se hallan

tabulados en el estándar, y se seleccionan según el valor de `ctxIdx` y del tipo de *slice*. El tercero, simplemente se identifica con el QP del *slice* referido a luminancia.

Algoritmo 2 Pseudo-código para la inicialización de contextos. Adaptado de [12].

```
1  preCtxStateIdx = clip3(1,126,((m * clip3(0,51,SliceQPy))>>4) + n);
2  if(preCtxStateIdx <= 63){
3    pStateIdx = 63 - preCtxStateIdx:
4    valMPS = 0;
5  } else{
6    pStateIdx = preCtxStateIdx - 64;
7    valMPS = 1;
8  }
```

-Nota: `Clip3(x,y,z)` es una función definida en [12], siendo su salida:

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

2.2.2.3. Actualización de contextos

Una vez inicializados, cada contexto se actualiza tras el paso de cada *bin* por el codificador aritmético. Concretamente, `pStateIdx` varía entre 0 y 63, considerando si el *bin* asociado al contexto es el MPS, es decir, si coincide con `valMPS`, o por contra, si su valor se corresponde con el del símbolo menos probable (del inglés *Least Probable Symbol*) (LPS). Las transiciones de `pStateIdx` se especifican en la Tabla 9-45 de [12], siendo su representación gráfica la de la Figura 2.10.

No obstante, la actualización recursiva de contextos previamente mencionada es uno de los factores que dificulta la codificación simultánea de dos o más *bins* pertenecientes al mismo *slice*. Además, en la Subsección 2.2.3, se expone cómo el estado del propio codificador aritmético fluctúa en función a los contextos, influyendo directamente sobre el *bitstream* de salida.

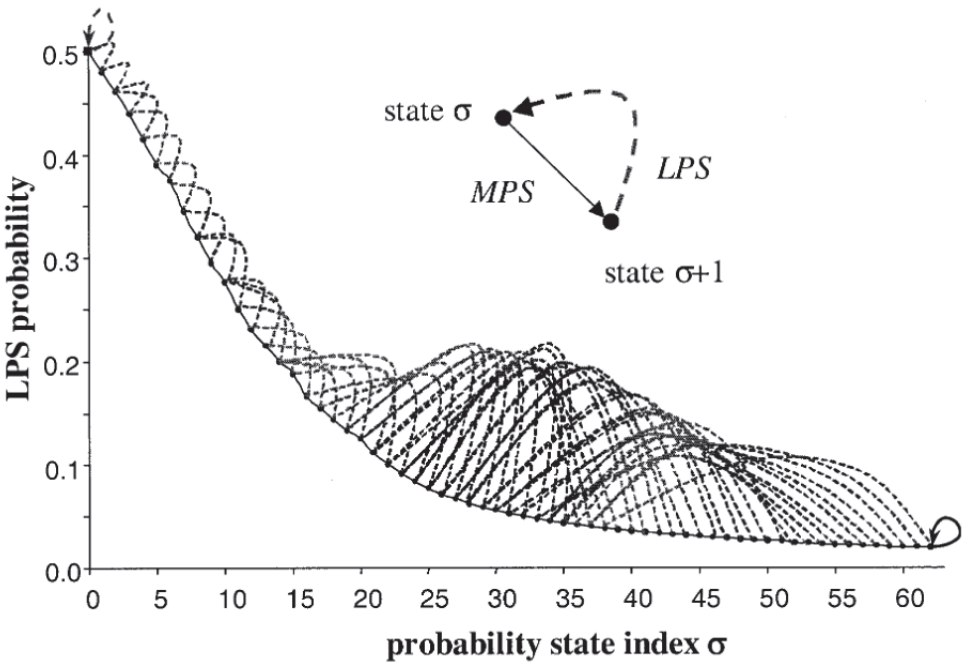

Figura 2.10: Transiciones de `pStateIdx` según el valor del *bin* codificado. Fuente [16].

2.2.3. Codificador Binario Aritmético

La codificación aritmética constituye una alternativa práctica a la implementación de algoritmos de codificación Huffman, permitiendo aproximar los resultados de compresión a los máximos teóricos [1]. Para ello, el codificador aritmético ha de convertir una secuencia de datos en un único número fraccionario, donde además se pueda aproximar al número de bits óptimo para representar cada símbolo en binario [1].

Este proceso se descompone en los siguientes pasos:

1. Se define el alfabeto y las probabilidades de cada símbolo.
2. Se redimensionan los rangos de probabilidad, tomando como base el rango actual del símbolo codificado.
3. Se repite el paso 2) hasta que no queden más símbolos por codificar.

Para ilustrar el procedimiento, se añade el ejemplo de la Figura 2.11. En él, se codifica la secuencia "1100", partiendo de un alfabeto de dos símbolos, '0' y '1', con probabilidades 0,3 y 0,7, respectivamente. Cualquier valor perteneciente al rango sombreado en azul, representa a la secuencia completa:

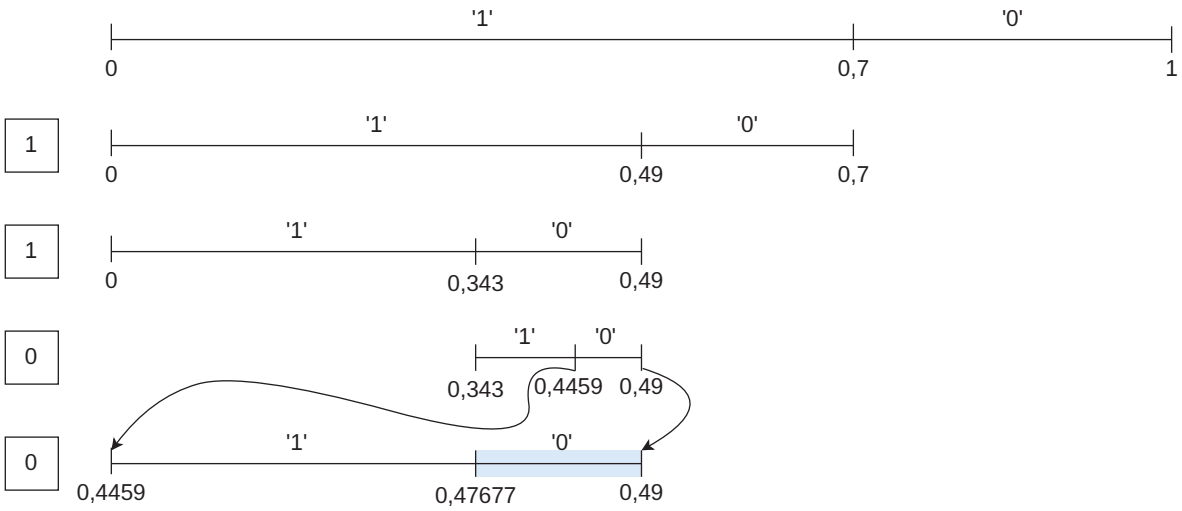

Figura 2.11: Ejemplo de codificación aritmética. Adaptado de [17].

Sin embargo, la técnica previamente descrita se implementa en CABAC mediante el uso de probabilidades adaptativas intrínsecas a los modelos de contexto [17]. Con el fin de agilizar el cómputo, la mencionada probabilidad adaptativa se lleva a cabo a través de tablas de rangos de probabilidad previamente calculados, en lugar de realizar multiplicaciones [16]. Cada uno de estos rangos, `codIRangeLPS`, es seleccionado de la tabla que los contiene en función del valor del índice de contexto, `pStateIdx`, y de una de las variables de las que definen el estado interno del codificador aritmético. Dichas variables son: `codIRange` y `codILow`, de al menos 9 y 10 bits de tamaño, respectivamente [12], [16]. La primera de ellas indica el rango actual utilizado de probabilidad, verificándose que `codIRange` ϵ $[256, 512)$ antes de codificar un nuevo *bin*, mientras que la segunda señala el punto más bajo de dicho intervalo. En la Figura 2.12 se incluye un ejemplo gráfico de dichas variables con valores arbitrarios sobre el rango total de probabilidades:

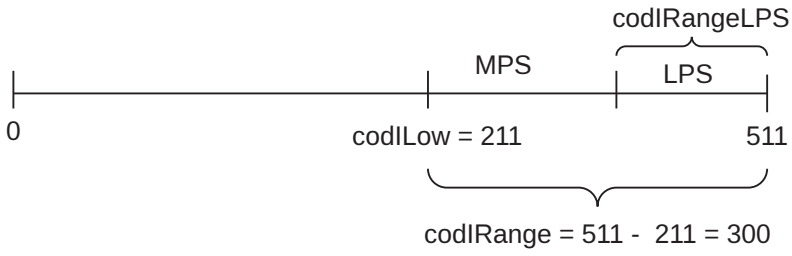

Figura 2.12: Representación gráfica de las variables de estado del BAC. Adaptado de [17].

Por otro lado, cabe destacar que `codIRange` y `codILow` han de ser inicializadas, respectivamente, a 510 y 0 al comienzo de cada *slice* [12]. Además, son comunes a los dos motores de codificación en los que se subdivide el BAC, representados previamente en la Figura 2.7.

Asimismo, cabe señalar que la escritura de nuevos bits al *bitstream* se realiza en función de los valores de las variables internas del BAC. Concretamente, pueden añadirse bits tras la codificación de un *bin* tanto en el *regular coding engine* como en el *bypass coding engine* si se dan ciertas condiciones. De igual manera, también pueden generarse bits tras la ejecución de `EncodeTerminate` como se expone en el Apartado 2.2.3.4.

2.2.3.1. Regular Coding Engine

El *regular coding engine* es el motor de codificación que realmente desempeña la codificación adaptativa, pues al contrario que el *bypass coding engine*, utiliza contextos [16]. Las entradas de este proceso se corresponden con el valor del *bin* a codificar, `binVal`, y su índice de contexto asociado, `ctxIdx`, a través del cual se obtienen `valMPS` y `pStateIdx`. A continuación, se selecciona el, a priori, nuevo valor del rango de codificación propio del LPS, `codIRangeLPS`. Dicho valor se halla en una tabla, `rangeTabLPS`, donde el índice de la fila escogida es fijado por `pStateIdx`, mientras que la columna la determinan los bits 6 y 5 de `codIRange` (suponiendo que el bit de mayor índice es el primero que se encuentra comenzando por la izquierda). Acto seguido, se comprueba si el *bin* a codificar coincide con el valor de `valMPS`. En caso afirmativo, se actualiza `pStateIdx` utilizando la tabla de transición para este caso, `transIdxMPS`. Sin embargo, si su valor no se corresponde con `valMPS` se lleva a cabo la siguiente secuencia:

1. Se actualiza `codILow`.
2. Se recalcula `codIRange`.
3. Se actualiza `valMPS` únicamente si `pStateIdx` es 0.
4. Se actualiza `pStateIdx` utilizando la tabla de transición correspondiente a este caso, `transIdxLPS`.

Finalmente, se realiza el proceso de renormailzación, `RenormE`, y se incrementa la cuenta de *bins* en unidades de capa de abstracción de red (del inglés *Network Abstraction Layer*) (NAL).

En el diagrama de flujo de la Figura 2.13 se resume la funcionalidad, ya descrita, de este motor de codificación. Además, en él se invoca al proceso `PutBit(B)`, explicándose su funcionamiento en el Apartado 2.2.3.4.

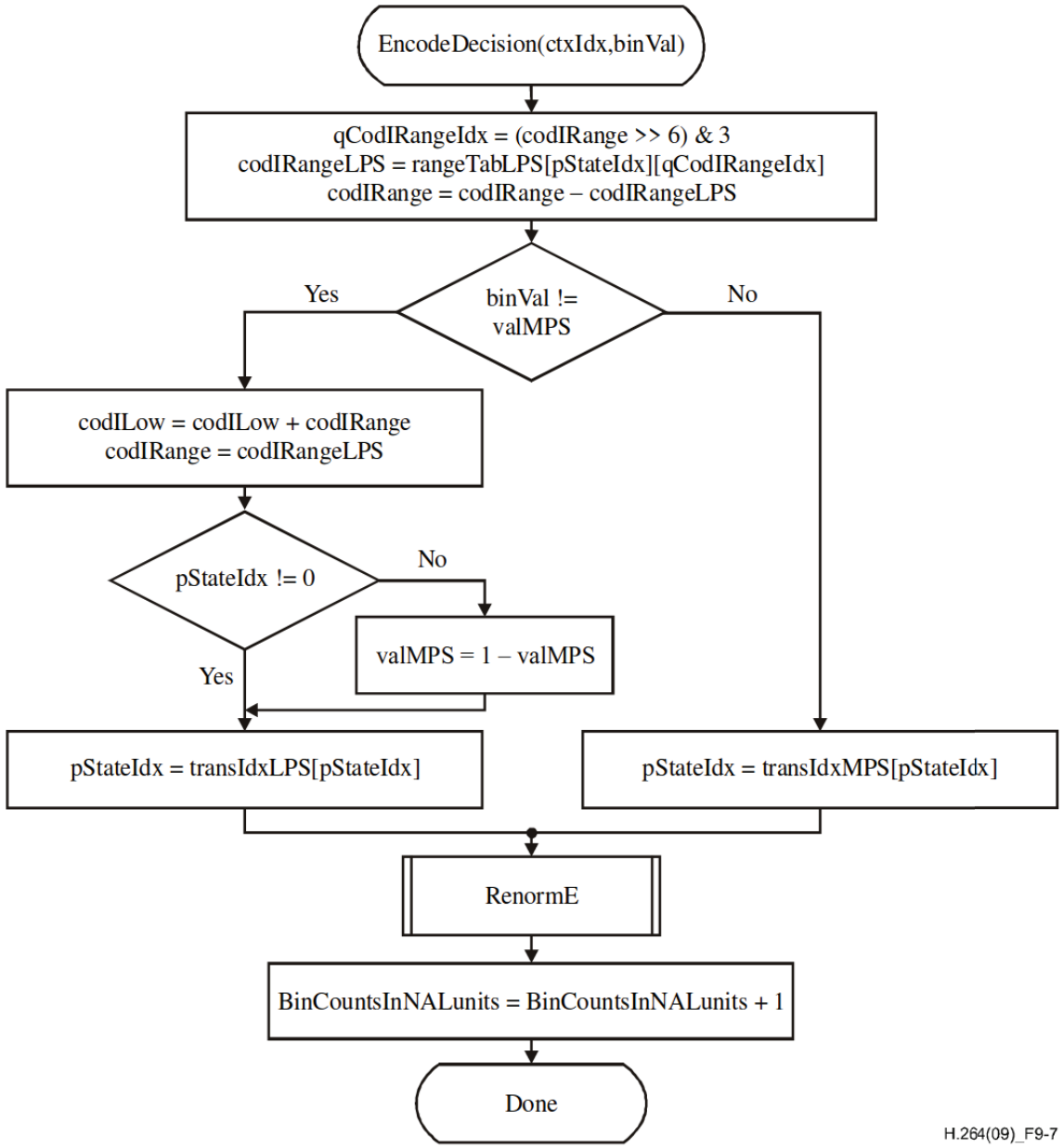

Figura 2.13: Diagrama de flujo de la codificación de un *bin* del *regular coding engine*. Fuente [12].

2.2.3.2. Proceso de renormalización

Como se mencionaba previamente en la Subsección 2.2.3, antes de codificar cada *bin*, ha de verificarse que `codIRange` pertenezca al rango $[256, 512)$. Por consiguiente, siempre se ejecuta este proceso al final del ya explicado en el Apartado 2.2.3.1.

Atendiendo a la descripción del proceso en sí, consiste en una serie de comprobaciones sucesivas sobre las variables que definen el estado interno del codificador aritmético: `codIRange` y `codILow`. En función de sus valores tras la codificación de cada *bin* se realizan diferentes acciones:

1. Si `codIRange` es mayor a 256, se da por finalizado su ajuste. En caso contrario:

   a) Si `codILow` es menor a 256 se invoca a `PutBit(0)`.

   b) Si `codILow` es mayor a 256 y menor a 512, se le restan 256 a su valor actual y se incrementa la cuenta de `bitsOutstanding`[4].

   c) Si `codILow` es mayor o igual a 512, se le sustrae 512 de su valor actual y se invoca a `PutBit(1)`.

   d) Por último, se duplica el valor de `codIRange` y `codILow`, volviendo al punto 1.

En la Figura 2.14 se halla el diagrama de flujo que muestra gráficamente este procedimiento.

[4]bits cuya polaridad todavía es desconocida, y que se determinará según avance la codificación

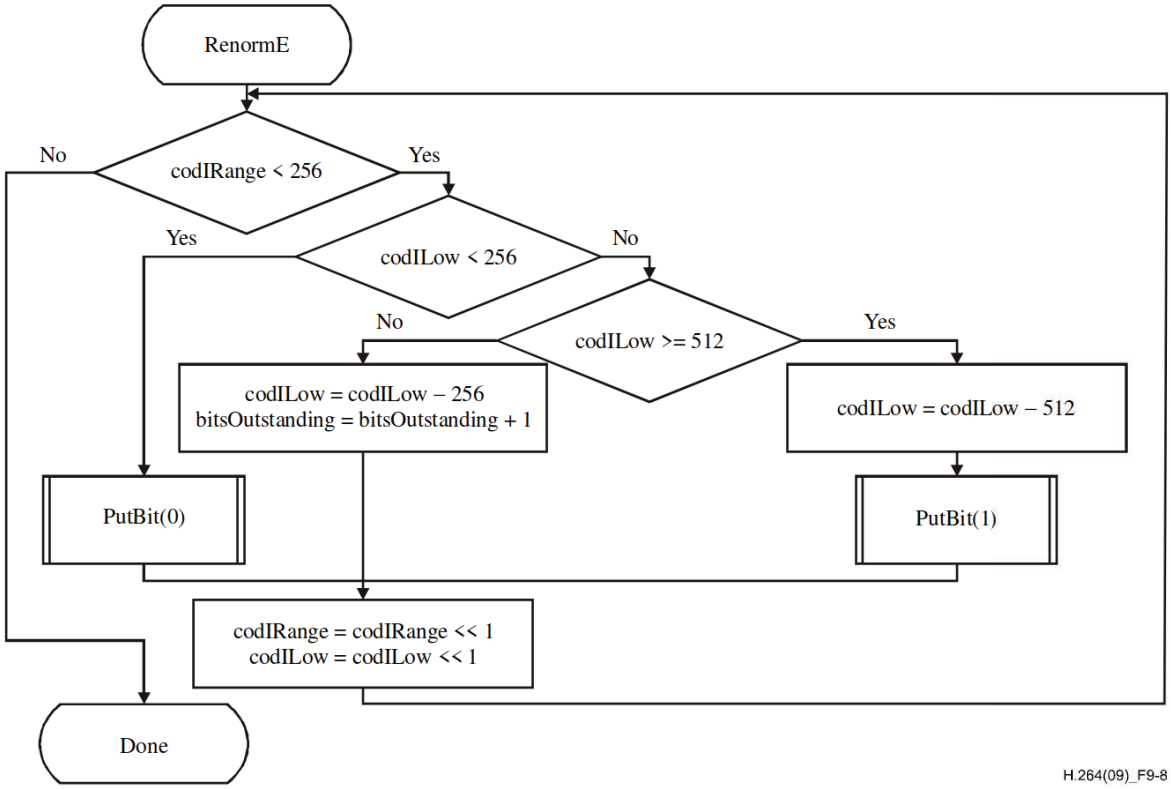

Figura 2.14: Diagrama de flujo del proceso de renormalización del *regular coding engine*. Fuente [12].

2.2.3.3. Bypass Coding Engine

En cuanto al *bypass coding engine*, este motor de codificación opera sin modelos de contexto, pues codifica aquellos *bins* que siguen una distribución de probabilidad prácticamente uniforme [16]. Además, a diferencia del *regular coding engine*, este no modifica el valor de `codIRange`, pues utiliza una variante del proceso de renormalización expuesto en el apartado anterior.

Prestando atención a su funcionamiento, los pasos a seguir para codificar un *bin* son los siguientes:

1. Se duplica el valor de `codILow`.
2. Si el *bin* a codificar es '1', se actualiza el valor de `codILow`.
3. Ahora se realiza la renormalización de este proceso, donde se desarrollan diferentes acciones según el valor de `codILow` en este punto:

a) Si `codILow` es mayor o igual a 1024[5], se le sustrae esta cantidad a su valor actual y se invoca a `PutBit(1)`.

b) Si está entre 512 y 1024 (no incluido), se le resta 512 a su valor actual y se incrementa la cuenta de `bitsOutstanding`.

c) Si es menor a 512, se invoca a `PutBit(0)`.

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

En la Figura 2.15 se encuentra el diagrama de flujo que resume el funcionamiento de este motor de codificación.

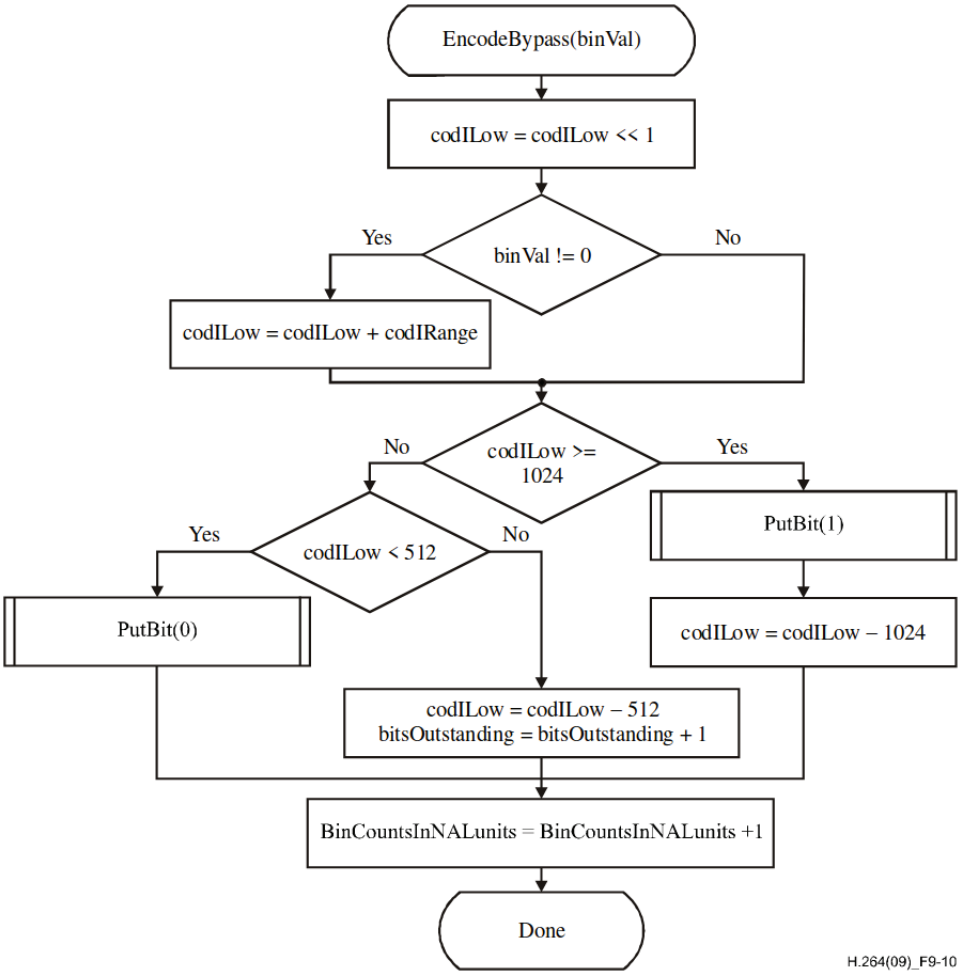

Figura 2.15: Diagrama de flujo de la codificación de un *bin* del *bypass coding engine*. Fuente [12].

[5]Para que pueda darse esta situación, han de utilizarse más de 10 bits para `codILow`.

2.2.3.4. Procesos auxiliares

Para concluir con la descripción del funcionamiento del BAC en su conjunto resta exponer una serie de procesos auxiliares. Estos son: `PutBit`, `EncodeTerminate` y `EncodeFlush`.

`-PutBit` (ver Figura 2.16): tiene como entrada la polaridad del bit a escribir en el *bitstream*. Primeramente, se comprueba si el flag de primer bit del *slice*, `firstBitFlag`, está activo. En caso afirmativo, se desactivará, mientras que si su valor es '0', se escribe un bit al *bitstream* con la polaridad mencionada previamente. Tras haber realizado cualquiera de estas dos acciones, se procede a escribir, con polaridad opuesta a la introducida, tantos bits como indique `bitsOutstanding`. Es decir, en este punto se determina el valor de aquellos bits cuya polaridad era inicialmente desconocida.

Nótese que la función `WriteBits(B,N)`, que aparece en la Figura 2.16, escribe `N` bits al *bitstream* con el valor `B` indicado.

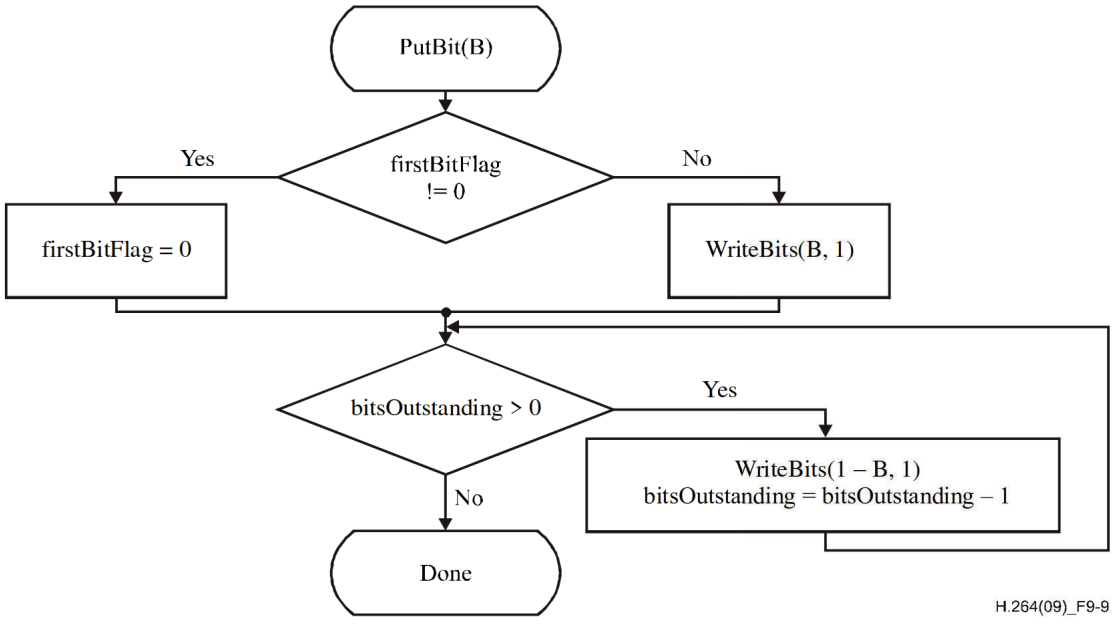

Figura 2.16: Diagrama de flujo de `PutBit`. Fuente [12].

`-EncodeTerminate` (ver Figura 2.17): este proceso se ejecuta tras haber codificado todos los elementos sintácticos de cada MB, justo antes de comenzar a codificar los del siguiente. En primer lugar, se restan 2 unidades al valor actual de `codIRange`.

Seguidamente, si el valor del *bin* que recibe como entrada, `binVal`, es '1', se interpreta que este era el último MB del *slice*, por lo que se procede a sumar `codIRange` a `codILow` y a ejecutar `EncodeFlush`. En caso contrario, se lleva a cabo el ya descrito `RenormE` en el Apartado 2.2.3.2. Tras haber realizado cualquiera de las acciones descritas previamente, se incrementa el número de *bins* en unidades NAL.

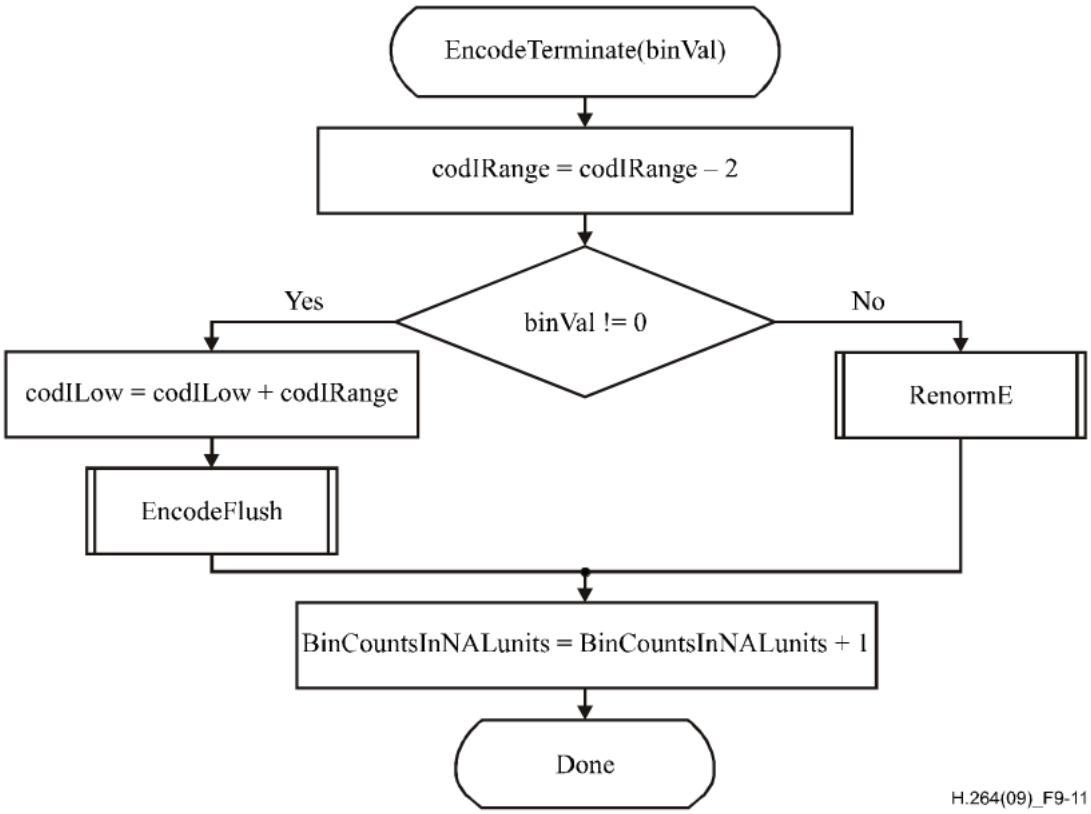

Figura 2.17: Diagrama de flujo de `EncodeTerminate`. Fuente [12].

`-EncodeFlush` (ver Figura 2.18): como se comentaba anteriormente, únicamente es necesario llevarlo a cabo tras haber codificado todos los SE del último MB del *slice*. Se comienza por establecer el valor de `codIRange` en 2. A continuación, se ejecuta el ya mencionado proceso de renormalización. Luego, se invoca a `PutBit(B)`, siendo `B` en este caso el bit 9 de `codILow`. Nótese que se supone que estos índices se asignan en orden decreciente de izquierda a derecha. Finalmente, se escribe al *bitstream* el bit 8 de `codILow` seguido de un '1'. Esto es equivalente a la siguiente operación,

```
WriteBits(((codILow ≫ 7) & 3) | 1 , 2)
```

ya que el bit menos significativo de los dos resultantes (el bit 7 de `codILow`) siempre se fuerza a '1' mediante la `OR` del resultado con 1.

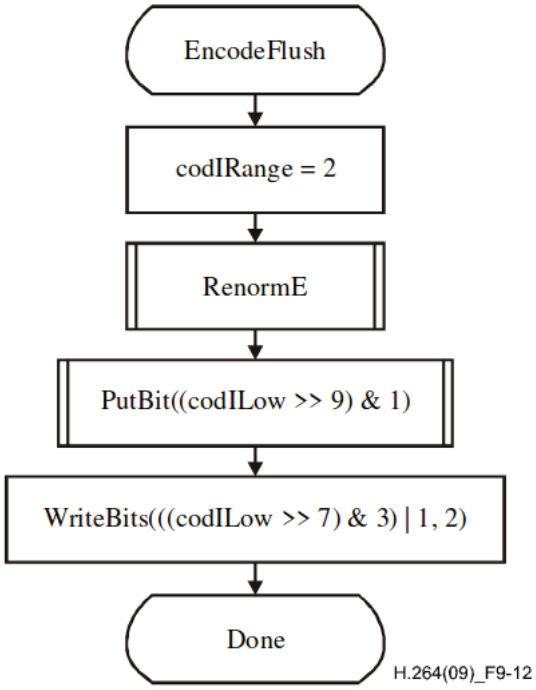

Figura 2.18: Diagrama de flujo de `EncodeFlush`. Fuente [12].

2.3. Elementos Sintácticos relacionados con CABAC

Según figura en el estándar, existen un total de veinte tipos de símbolos o SE diferentes que son codificados empleando CABAC. Este conjunto de SE, se recoge en la Tabla 2.5:

Capa	Tipo de SE
Slice data	mb_skip_flag
	mb_field_decoding_flag
	end_of_slice_flag
Macroblock layer	mb_type
	transform_size_8x8_flag
	coded_block_pattern
	mb_qp_delta
MB prediction	prev_intra4x4_pred_mode_flag
	rem_intra4x4_pred_mode
	prev_intra8x8_pred_mode_flag
	rem_intra8x8_pred_mode
	intra_chroma_pred_mode
MB and sub MB prediction	ref_idx_lX
	mvd_lX
Sub MB prediction	sub_mb_type
Residual block CABAC	coded_block_flag
	significant_coeff_flag
	last_significant_coeff_flag
	coeff_abs_level_minus1
	coeff_sign_flag

Tabla 2.5: SE relacionados con CABAC.

2.3.1. Definiciones

Cada uno de estos SE tiene una interpretación dentro de H.264. De acuerdo con [12] y [16], los significados de dichos SE son los que se recogen a continuación:

`-mb_skip_flag`: indica si el MB actual ha de ser omitido. Esta situación únicamente puede tener lugar para MB de tipo P o B.

`-mb_field_decoding_flag`: se utiliza cuando se codifica vídeo en modo entrelazado. Sirve para indicar si el MB actual ha de ser tratado como un MB de campo (*field*), o como uno de fotograma completo (*frame*).

`-end_of_slice_flag`: como su propio nombre sugiere, marca el final de cada *slice*.

`-mb_type`: indica el tipo de MB. Toma diferentes nombres y valores dependiendo de varios parámetros, como pueden ser el tipo de *slice* y los modos de predicción, entre otros.

`-transform_size_8x8_flag`: señala la utilización de Bloques 8x8 (B8) de luminancia y crominancia en el MB actual.

`-coded_block_pattern`: advierte cuáles de los B8 de luminancia y los asociados de crominancia en los que se divide un MB contienen coeficientes distintos de cero. Cuando se están codificando imágenes a color, `coded_block_pattern` está compuesto por un total de seis bits. Los dos más significativos, b_5b_4, están ligados a las señales de crominancia. Según [1], de sus valores se deduce:

- 00: no existen coeficientes de croma distintos de cero.
- 01: existen coeficientes DC de croma distintos de cero y todos los de AC son cero.
- 10: existen coeficientes de AC no nulos y posiblemente también de DC.

Por otro lado, cada uno de los bits restantes, $b_3b_2b_1b_0$, está ligado a un B8 de luminancia diferente. En caso de que en un B8 exista al menos un coeficiente distinto de cero, su bit asociado se pondrá a '1' [1]. En la Figura 2.19 se muestran algunos ejemplos.

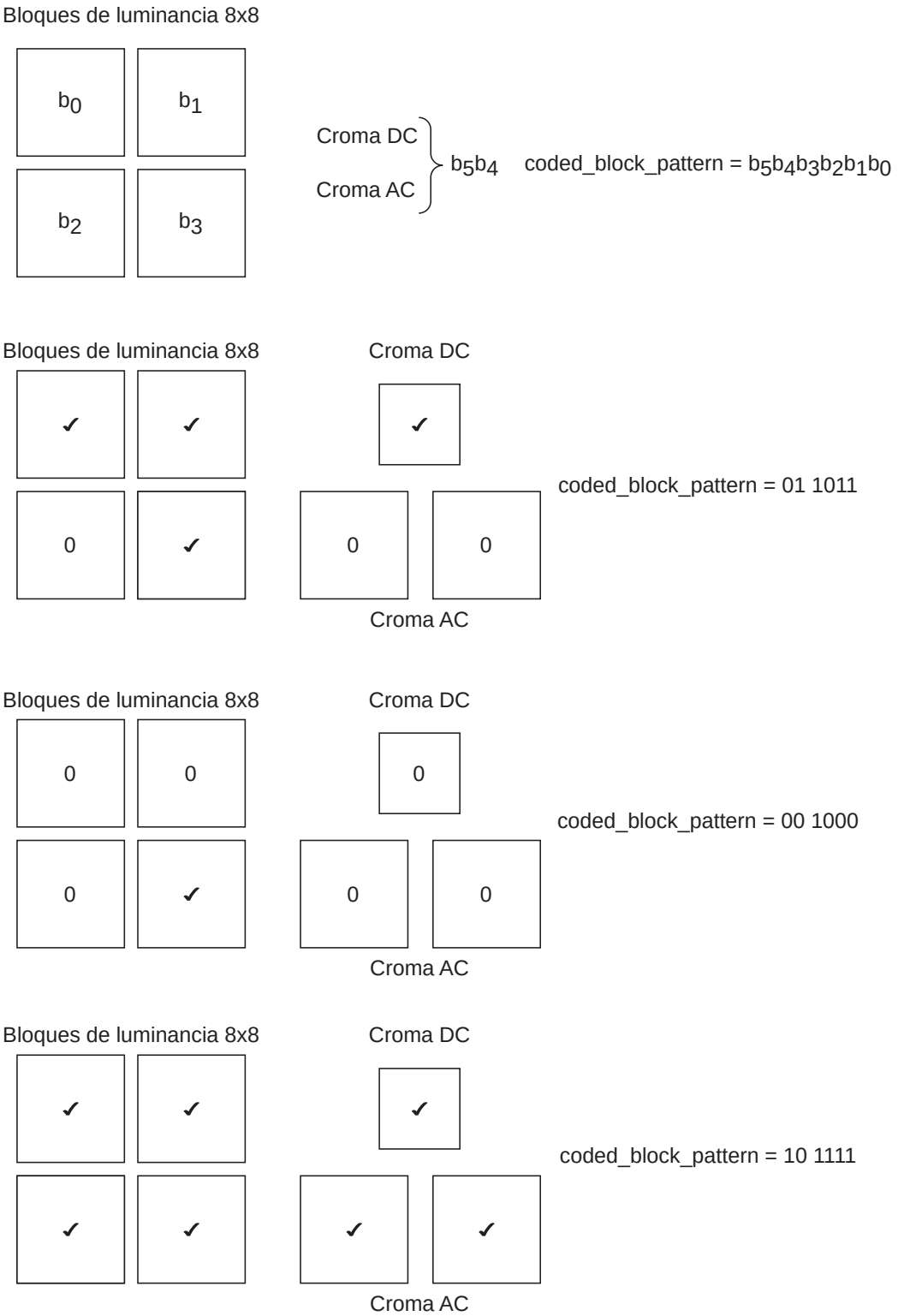

Figura 2.19: Ejemplos de `coded_block_pattern`. Fuente [1].

`-mb_qp_delta`: representa la diferencia entre el QP configurado a nivel de *slice*, $\texttt{SliceQP}_Y$, y el escogido para un MB en concreto. Por tanto, su rango de valores es el siguiente:

$$\texttt{mb\_qp\_delta}\ \epsilon\ \left[-\left(26+\frac{\texttt{QpBdOffset}_Y}{2}\right)\ ,\ \left(25+\frac{\texttt{QpBdOffset}_Y}{2}\right)\right]$$

Nótese que $\texttt{SliceQP}_Y$ está comprendido entre -26 y 25, lo que es equivalente al rango [0,51] si se comienza por 0. Sin embargo, para el primer intervalo existe un *offset*, $\texttt{QpBdOffset}_Y$, que modifica los extremos del mismo. De ahí que aparezca en la expresión anterior.

`-prev_intra4x4_pred_mode_flag`: advierte si el modo de predicción utilizado para cada B4 de luminancia se corresponde con el más probable. Si `ChromaArrayType`[6] = 3, el valor de este SE también hace referencia a los modos de predicción de los bloques de croma asociados.

`-rem_intra4x4_pred_mode`: indica el modo de predicción empleado para el B4 de lumninancia actual, en caso de que no se corresponda con el más probable. Es decir, únicamente es necesario codificar este SE cuando `prev_intra4x4_pred_mode_flag` = 0.

`-intra_chroma_pred_mode`: señala el tipo de predicción parcial utilizado para coeficientes de croma. Únicamente está presente si `ChromaArrayType` es igual a 1 ó 2 en Intra 4x4, Intra 8x8 o Intra 16x16.

`-ref_idx_lX`: especifica el índice de la imagen de referencia en la lista X, así como la pareja del campo dentro de la imagen de referencia usada, si procede. Dichas imágenes de referencia solo tienen cabida en predicción *interframe*.

`-mvd_lX`: representa la diferencia entre el vector de movimiento estimado para el bloque que se está codificando y un vector de movimiento de referencia. Por otro lado, cabe señalar que, al igual que el SE anterior, este también es específico de la predicción

[6] `ChromaArrayType` es un parámetro definido en [12]. Su valor aporta información adicional acerca de la codificación de la croma, influyendo en varios SE expuestos en este apartado.

interframe.

`-sub_mb_type`: expone el tipo de sub-MB utilizado. Solo se utiliza para MB de tipo P y B, en caso de ser necesario.

`-coded_block_flag`: indica si el B4 actual contiene coeficientes distintos de cero. Si todos los coeficientes son nulos, se da por finalizada la codificación del B4 en cuestión, pasando al siguiente.

`-significant_coeff_flag`: indica qué coeficientes de un B4, en orden de escaneo (ver Figura 2.20), son distintos de 0, es decir, significativos.

`-last_significant_coeff_flag`: señala cuál es el último coeficiente significativo en orden de escaneo. Solo es necesario si `significant_coeff_flag` = '1'. En la Figura 2.20 se muestra un ejemplo de ello:

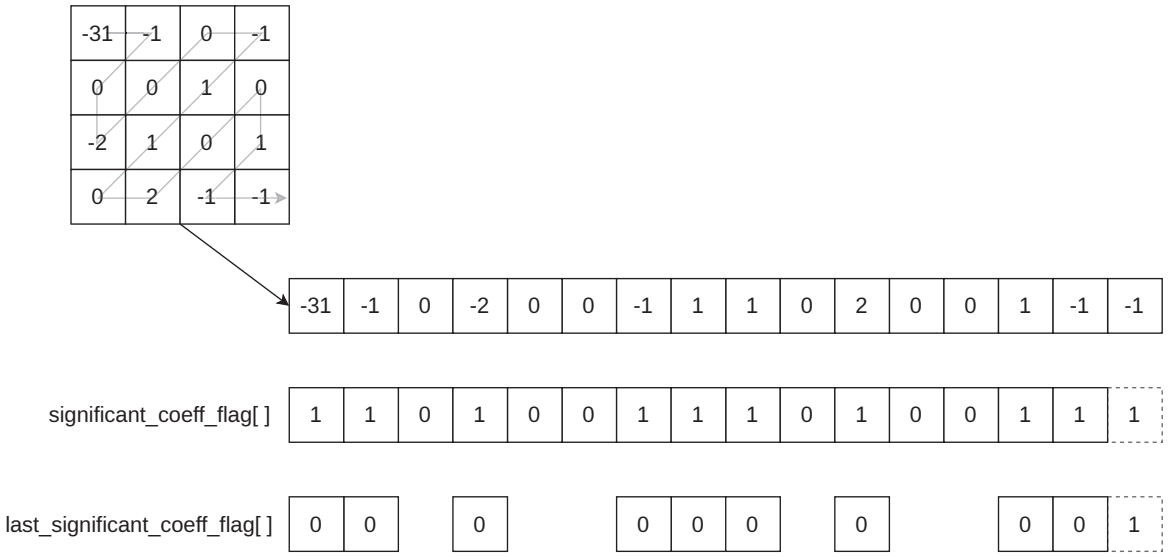

Figura 2.20: Cálculo de SCF y LSCF

-Nota: para `significant_coeff_flag` y `last_significant_coeff_flag`, nunca se codifica el *flag* referido al último coeficiente del B4, pues en caso de llegar a él, se sobreentiende que ha de ser significativo. De lo contrario, para $i <$ `maxBlockIdx` siendo i el índice actual de escaneo y `maxBlockIdx` el mayor índice que puede encontrarse en el tipo de B4 escaneado, existirá un $\texttt{last\_significant\_coeff\_flag}_i$ = 'i'.

`-coeff_abs_level_minus1`: se corresponde con el valor absoluto menos una unidad de cada coeficiente.

`-coeff_sign_flag`: representa el signo de los coeficientes: '1' si son negativos '0' en el caso contrario.

2.3.2. Restricciones del diseño

Habiendo previamente expuesto los diferentes tipos de SE relacionados con CABAC, ha de hacerse hincapié en que la selección y uso final de los mismos depende de la configuración elegida para cada codificador particular. De hecho, como ya se comentó en la Sección 1.3, en este Trabajo de Fin de Grado (TFG) se implementa CABAC considerando tres modos de predicción Intra 4x4 e imágenes monocromas. Por tanto, este tipo de predicción únicamente contempla *slices* tipo I. Tampoco se considera la codificación en modo *field* ni más de un *slice* por *frame*. La variación del QP a nivel de MB, `mb_qp_delta`, será constante, siendo su valor siempre 0. Es decir, el QP de todos los MB será igual al del *slice* al que pertenezcan.

Todas estas restricciones de diseño vienen dadas por dos principales motivos: la acentuada complejidad y extensión de CABAC, y cuestiones contractuales del proyecto de investigación *Efficient Video Compression for space (ESA AO/1-1-10954/21/NL/MGu)*, vinculado a la empresa *Thales Alenia Space en España (Sector Espacial)*.

Por tanto, debido al alcance de este TFG, no se consideran aquellos SE que guardan relación con imágenes a color o predicción *interframe*, quedando descartados. En consecuencia, los SE finalmente utilizados son los de la Tabla 2.6.

Capa	Tipo de SE
Slice data	`end_of_slice_flag`
Macroblock layer	`mb_type`
	`coded_block_flag`
	`mb_qp_delta`
MB prediction	`prev_intra4x4_pred_mode_flag`
	`rem_intra4x4_pred_mode`
Residual block CABAC	`coded_block_flag`
	`significant_coeff_flag`
	`last_significant_coeff_flag`
	`coeff_abs_level_minus1`
	`coeff_sign_flag`

Tabla 2.6: Listado de los SE utilizados en el ámbito de este TFG.

2.3.3. Binarización

Tras haber concretado los SE finalmente utilizados en base a las restricciones expuestas en la Subsección 2.3.2, se expone el tipo de binarización a utilizar para cada uno de ellos según se indica en [8] y [12]. Esta información se encuentra, de manera resumida, en la Tabla 2.7.

Atendiendo a dicha tabla, cabe señalar que los tamaños máximos de los *binstrings* de `mb_type`, `coded_block_pattern` y `mb_qp_delta` han experimentado una variación respecto a lo definido en el estándar. En el caso del primer SE, se tiene que su valor siempre será 0, pues únicamente se emplean MB con predicción Intra 4x4. En las tablas de *binstrings* predefinidos de `mb_type` se encuentra que el *binstring* para el valor 0 es "0", de ahí que solo sea necesario un *bin*. Por otro lado, `coded_block_pattern` realmente se subdivide en dos SE diferentes: uno para la crominancia y otro para la luminancia. Como las imágenes a codificar son monocromas, solo se considera la parte relativa a luminancia, necesitándose ahora cuatro bits en lugar de seis.

Tipo de SE	Binarización	Longitud máxima del binstring (bins)
`end_of_slice_flag`	FL, cMax = 1	1
`mb_type`	Predefinida en tablas	1
`coded_block_pattern` (luma)	FL, cMax = 15	4
`mb_qp_delta`	U	1
`prev_intra4x4_pred_mode_flag`	FL, cMax = 1	1
`rem_intra4x4_pred_mode`	FL, cMax = 7	3
`coded_block_flag`	FL, cMax = 1	1
`significant_coeff_flag`	FL, cMax = 1	1
`last_significant_coeff_flag`	FL, cMax = 1	1
`coeff_abs_level_minus1`	UEG0, signedValFlag = 0, uCoff = 14	43
`coeff_sign_flag`	FL, cMax = 1	1

Tabla 2.7: Binarización asignada a cada tipo de SE.

Finalmente, `mb_qp_delta` necesita un *bin* para representar su *binstring*, pues al igual que `mb_type`, este siempre es "0". En el caso de que su valor fuese variable y/o distinto de cero, sería necesario convertirlo a un número natural, pues como se expuso en la Subsección 2.3.1, este puede oscilar entre -26 y +25 más el $\texttt{QpBdOffset}_Y$ correspondiente. En la Tabla 2.8 se recoge la forma en la que se realiza dicha conversión, donde `SE_val` es el valor del `mb_qp_delta` y `codeNum` el número natural resultante a codificar.

codeNum	SE_val
0	0
1	1
2	-1
3	2
4	-2
5	3
6	-3
k	$(-1)^{k+1}\ Ceil(k/2)$

Tabla 2.8: `codeNum` asignado al valor del SE. Fuente [12].

2.3.4. Cálculo de índices de contexto

Los SE de la Tabla 2.6 pueden dividirse en tres categorías si se clasifican por la manera en la que se calcula su índice de contexto, `ctxIdx` [16]:

- **Índices de contexto fijos:** `end_of_slice_flag`, `prev_intra4x4_pred_mode_flag` y `rem_intra4x4_pred_mode`
- **SE no residuales:** `ctxIdx` = `ctxIdxInc` + `ctxIdxOffset`
- **SE residuales**[7]**:** `ctxIdx` = `ctxIdxInc` + `ctxIdxOffset` + `ctxIdxBlockCatOffset`

En los dos últimos grupos aparecen ahora por primera vez tres parámetros nuevos. Sus significados según [16] son:

- `ctxIdxInc:` incremento de índice de contexto.
- `ctxIdxOffset:` *offset* del índice de contexto. Impone el `ctxIdx` más bajo que cada tipo de SE puede tomar en cada momento. Los valores utilizados, acorde a las restricciones expuestas en la Subsección 2.3.2, se hallan en la Tabla 2.9.

[7]Los SE residuales hacen referencia a los SE derivados de datos residuales. Nótese que a lo largo del documento aparecerán como SE residuales.

- `ctxIdxBlockCatOffset:` incremento del índice de contexto debido a la categoría de contexto del B4 escaneado. Este tipo de *offset* también está tabulado, y se añade más información acerca de él en el Apartado 2.3.4.3.

SE	ctxIdxOffset
mb_type	3
coded_block_pattern	73
mb_qp_delta	60
prev_intra4x4_pred_mode_flag	68
rem_intra4x4_pred_mode	69
coded_block_flag	85
significant_coeff_flag	105
last_significant_coeff_flag	166
coeff_abs_level_minus1	227
end_of_slice_flag	276

Tabla 2.9: `ctxIdxOffset` según el tipo de SE [12].

Para los elementos sintácticos `end_of_slice_flag`, `prev_intra4x4_pred_mode_flag` y `rem_intra4x4_pred_mode` el valor de `ctxIdxOffset` es fijo y además coincide con el `ctxIdx`.

Nótese que `coeff_sign_flag` no aparece en la Tabla 2.9. Ello se debe a que es codificado por el *bypass coding engine*, por lo que no requiere contextos y tampoco de `ctxIdx`.

Por otro lado, también resulta posible agrupar los SE acorde a otro criterio, siendo en esta ocasión la necesidad de valores de SE previamente codificados. En este caso, dichos grupos están formados de la manera siguiente:

- **Necesitan valores de SE previamente codificados:**
  - `mb_type`
  - `coded_block_pattern`
  - `mb_qp_delta`

- `coded_block_flag`

- **No necesitan valores de SE previamente codificados:**

  - `significant_coeff_flag`
  - `last_significant_coeff_flag`
  - `coeff_abs_level_minus1`

Para los del primer grupo, los valores requeridos de SE previos no son seleccionados arbitrariamente. Se utilizan como referencia ciertos "vecinos", cuya determinación se explica en el Apartado 2.3.4.1.

Además, ha de ponerse énfasis en el hecho de que existen SE para los que la forma de obtener `ctxIdxInc` varía dependiendo del índice de *bin*, `binIdx`[8]. En este documento únicamente se recogen los procesos de obtención de `ctxIdxInc` a utilizar dadas las restricciones previamente expuestas en la Subsección 2.3.2.

2.3.4.1. Cálculo de vecinos

Los mencionados bloques "vecinos" son aquellos MB, B8 o B4 donde alguno de sus SE asociados se emplean en el cálculo de índices de contexto de elementos sintácticos de otros bloques. Concretamente, como se verá en el Apartado 2.3.4.2, los vecinos de interés se corresponden con aquellos que se encuentran a la izquierda del bloque actual, *vecinos A*, y los que se hallan justo sobre él, *vecinos B*.

En la Figura 2.21 aparece una representación de los MB vecinos dentro de un *slice* para el MB que se está codificando actualmente, `MB actual`. Cabe resaltar que no todos los bloques cuentan con vecinos, pues aquellos que se encuentren en los bordes izquierdo y/o superior tendrán alguno o ambos vecinos como "no disponibles".

[8]El *bin* con `binIdx = 0`, generalmente, es el primero que se encuentra comenzando por la izquierda coincidiendo con el bit más significativo (del inglés *Most Significant Bit*) (MSB) del *binstring*. En ciertos SE, como `coded_block_pattern` y `rem_intra4x4_pred_mode`, se corresponde con el primero comenzando por el extremo opuesto, que además es el bit menos significativo (del inglés *Least Significant Bit*) (LSB).

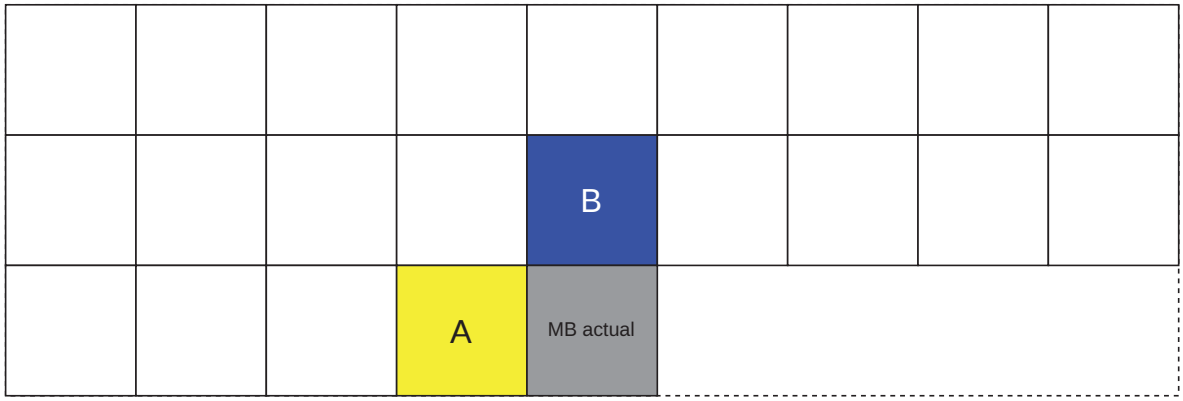

Figura 2.21: Macrobloques vecinos.

En cuanto a los SE utilizados en este trabajo de fin de grado, los vecinos de `mb_type` y `mb_qp_delta` son a nivel de MB. En otras palabras, los datos necesarios para el cálculo de `ctxIdxInc` para estos SE están ligados a los MB vecinos.

Sin embargo, existen otros SE que, por definición de los mismos, operan a nivel de B8 o B4. Este es el caso de `coded_block_pattern` y `coded_block_flag`, respectivamente. Esta circunstancia implica que los vecinos también son referidos a estas subdivisiones de MB. Es decir, también se toman referencias que pertenecen a otros MB, pero solo de los B8 o B4 pertinentes.

En las Figuras 2.22 y 2.23 se representan gráficamente cómo son estos vecinos para los SE mencionados. En ambas imágenes, primeramente se observa la comparativa entre los macrobloques `B` y *actual* frente a su subdivisión en B8 y B4 con sus respectivos índices. Los sub-bloques difuminados en dicha comparativa indican que no son vecinos para ninguno de los pertenecientes al MB que se está codificando.

Posteriormente, en cada imagen se han incluido dos ejemplos de cálculo de vecinos. Con ellos se refleja que dichos bloques vecinos pueden pertenecer al MB *actual* o a los MB contiguos, dependiendo de la disposición del sub-bloque que se esté codificando en cada momento. Al igual que ocurría con `mb_type` y `mb_qp_delta`, también existen casos en los que hay vecinos "no disponibles". Ello ocurre cuando los sub-bloques se encuentren en los bordes superior y/o izquierdo de MB que a su vez se hallen sobre los mismos márgenes del *slice*.

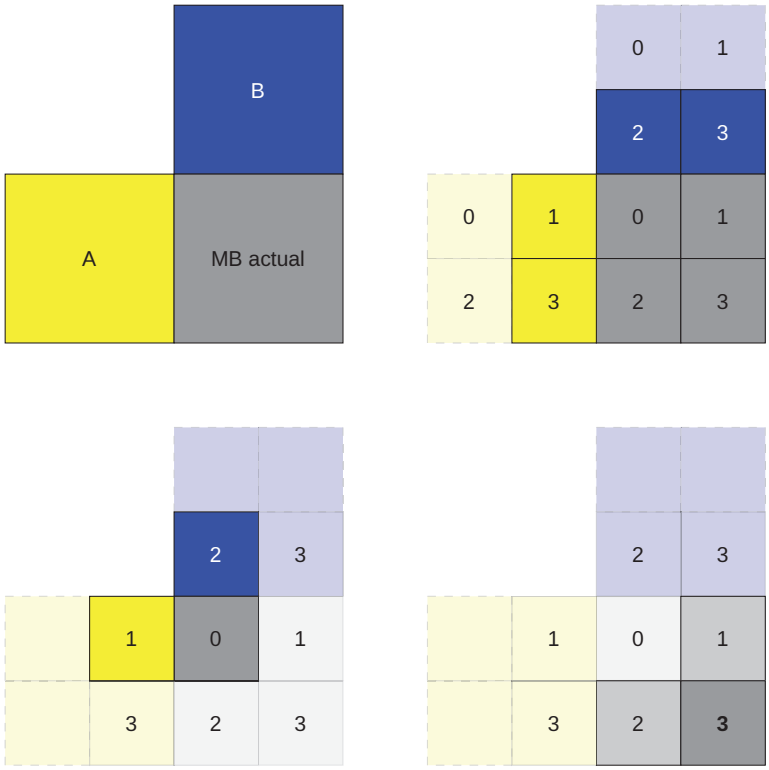

Figura 2.22: Bloques 8x8 vecinos.

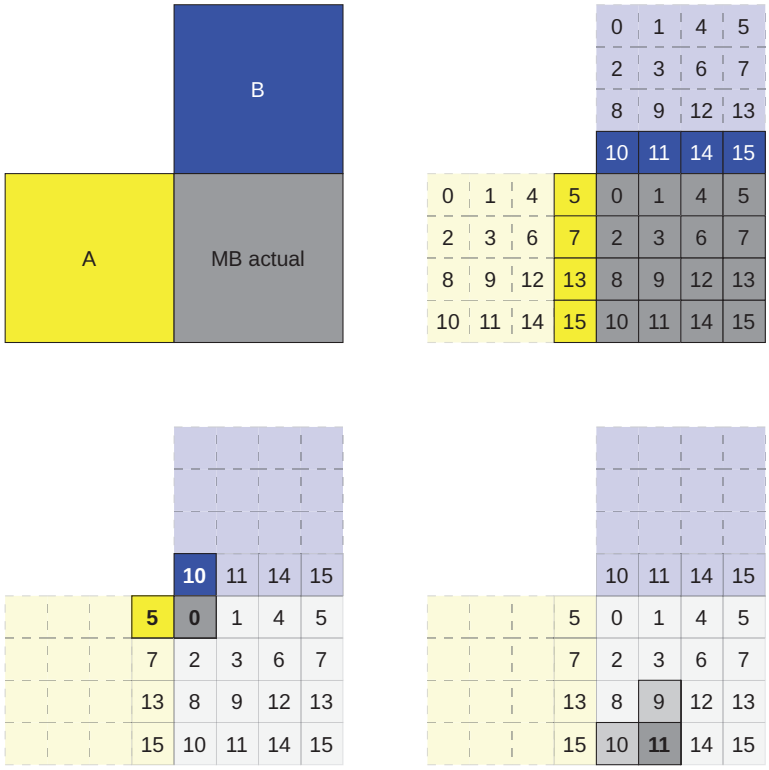

Figura 2.23: Bloques 4x4 vecinos.

2.3.4.2. ctxIdxInc

Habiendo expuesto cómo se calculan los vecinos, en esta subsección se incluyen los procedimientos para determinar el incremento de índice de contexto, `ctxIdxInc`, de los SE utilizados que así lo requieran. En este apartado, los bloques vecinos izquierdo y superior aparecerán siempre como `A` y `B`, respectivamente.

-mb_type: el `ctxIdxInc` para el *bin* 0 de este SE viene determinado por la expresión `ctxIdxInc = condTermFlagA + condTermFlagB`. El valor de los *flags* condicionales, `condTermFlagA` y `condTermFlagB`, será 0 si se verifica cualquiera de las siguientes condiciones:

1. No está disponible el MB vecino, $\texttt{MB}_N$ (siendo `N`: `A` o `B`).
2. ($\texttt{ctxIdxOffset} = 0$) && ($\texttt{mb\_type}_N = \texttt{SI}$)
3. ($\texttt{ctxIdxOffset} = 3$) && ($\texttt{mb\_type}_N = \texttt{I\_NxN}$)
4. ($\texttt{ctxIdxOffset} = 27$) && ($\texttt{mb\_type}_N = \texttt{B\_Skip} \mid\mid \texttt{B\_Direct\_16x16}$)

Considerando las circunstancias ya descritas, se deduce que siempre se cumple la tercera condición. Ello se debe a que 3 es el `ctxIdxOffset` que ha de utilizarse para *slices* tipo I, mientras que `I_NxN` es el nombre que recibe `mb_type` cuando se emplea predicción Intra 4x4, que es la única que se utiliza en este proyecto. Por consiguiente, `condTermFlagA = condTermFlagB = 0` siendo `ctxIdxInc = 0`.

-coded_block_pattern: el `ctxIdxInc` para cada *bin* asociado a la luminancia se obtiene con `ctxIdxInc = condTermFlagA + 2*condTermFlagB`. El valor por defecto de los *flags* condicionales es 1. Sin embargo será 0 en caso de que se cumpla cualquiera de las condiciones:

1. No está disponible el B8 vecino, $\texttt{B8}_N$ (siendo `N`: `A` o `B`).
2. $\texttt{mb\_type}_N = \texttt{I\_PCM}$
3. $\texttt{B8}_N$ no pertenece al MB actual, $\texttt{mb\_type}_N$ no es ni `P_Skip` ni `B_Skip`, y el *bin* de $\texttt{coded\_block\_pattern}_N$ ligado a $\texttt{B8}_N$ es '1'.

4. $\texttt{B8}_N$ pertenece al MB actual y el *bin* de `coded_block_pattern` ligado a $\texttt{B8}_N$ es '1'.

Para este SE no es necesario considerar la condición 2, ya que en la implementación realizada se emplea exclusivamente la codificación Intra 4x4. Además, también cabe destacar que en la tercera expresión no es necesario verificar el valor de $\texttt{mb\_type}_N$, pues, como se mencionó previamente, se sabe que siempre será `I_NxN`.

-mb_qp_delta: en el caso de este SE se tiene la certeza de que el tamaño máximo de su *binstring* es de un *bin*. Por tanto, solo resulta de interés el proceso para calcular el `ctxIdxInc` para `binIdx = 0`. Dicho `ctxIdxInc` será 0 si alguna de las siguientes condiciones se cumple:

1. El MB previamente codificado, `prevMB`, no está disponible.
2. $\texttt{mb\_type}_{prevMB}$ = `P_Skip` ó `B_Skip`
3. $\texttt{mb\_type}_{prevMB}$ = `I_PCM`
4. El modo de predicción utilizado para el MB previo no es Intra 16x16 y los `coded_block_pattern` referidos a luminancia y crominancia para el MB anterior son iguales a 0.
5. $\texttt{mb\_qp\_delta}_{prevMB} = 0$

Como `mb_qp_delta` siempre es 0, la última condición, o en su defecto la primera para el primer MB, siempre se cumple. Esto implica que el `ctxIdxInc` del *bin* con `binIdx = 0` de este SE siempre será 0.

-`coded_block_flag`: la fórmula para obtener `ctxIdxInc` es idéntica a la utilizada para `coded_block_pattern`: `ctxIdxInc = condTermFlagA + 2*condTermFlagB`. No obstante, para la obtención de los *flags* condicionales, aparece un *flag* auxiliar `transBlockN`, que a su vez puede variar en función de la categoría de contexto del B4 codificado, `ctxBlockCat`. A continuación, se incluyen los procedimientos, de manera simplificada, para determinar las variables mencionadas.

Cálculo de `transBlockN`, pudiendo ser N tanto A como B:

- Si el B4 vecino, $\texttt{B4}_N$, está disponible y el *bin* de $\texttt{coded\_block\_pattern}_N$ asociado a $\texttt{B4}_N$ es '1', se asigna este vecino a `transBlockN`.
- En caso contrario, `transBlockN` se marca como no disponible.

Cálculo de `condTermFlagN`, pudiendo ser N tanto A como B:

- `condTermFlagN` será 0 si el B4 vecino, $\texttt{B4}_N$, está disponible y `transBlockN` no está disponible.
- `condTermFlagN` será 1 si el B4 vecino no está disponible.
- En caso de que no se cumpla ninguna de las condiciones anteriores, `condTermFlagN` será igual al valor del `coded_block_flag` asociado a `transBlockN`.

`-significant_coeff_flag` y `last_significant_coeff_flag`: el `ctxIdxIncInc` de sendos SE se corresponde con el índice de escaneo del coeficiente en cuestión (véase la Figura 2.24).

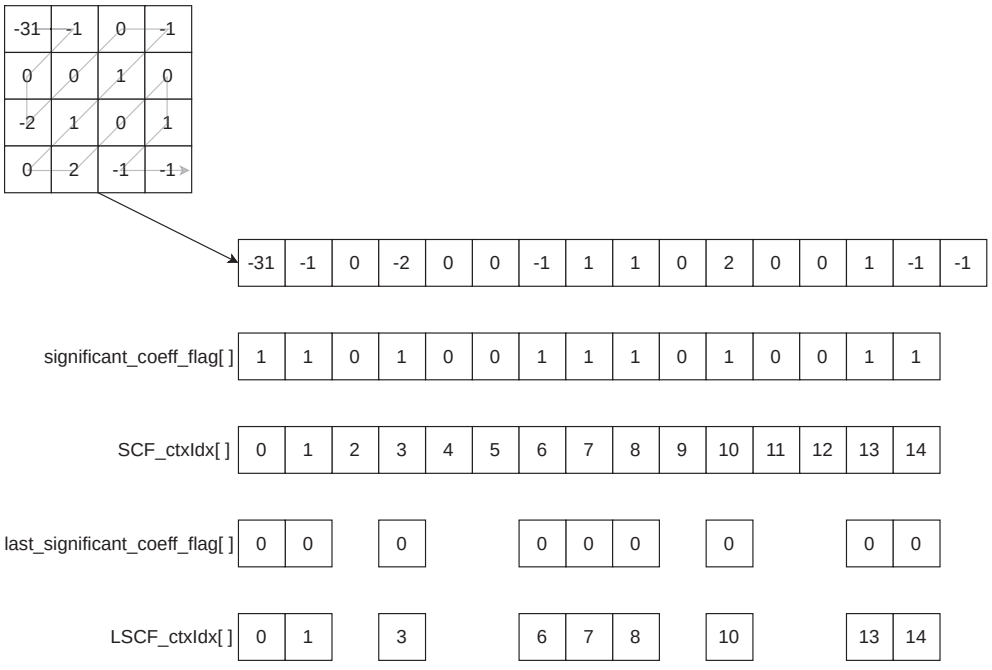

Figura 2.24: Ejemplo de cálculo de `ctxIdxInc` para el mapa de significancia.

-coeff_abs_level_minus1: en primer lugar, cabe hacer un inciso en que este SE se codifica en orden de escaneo inverso, y por consiguiente, `coeff_sign_flag` también [16]. En cuanto al cálculo de `ctxIdxInc` para `coeff_abs_level_minus1`, son necesarias dos variables auxiliares:

- **ngt1s:** número de coeficientes previos dentro del mismo B4 con valor absoluto mayor a 1.
- **n1s:** número de coeficientes previos dentro del mismo B4 con valor absoluto igual a 1.

Además, cabe señalar que los *bins* de `coeff_abs_level_minus1` se diferencian entre aquellos con `binIdx = 0` y `binIdx` ϵ $[1, 13]$, determinándose `ctxIdxInc` con un procedimiento diferente en cada caso. Nótese que los *bins* con `binIdx` > 13 no requieren `ctxIdxInc`, dado que son codificados por el *bypass coding engine*. Entonces, los `ctxIdxInc` pertinentes se codifican de la siguiente manera:

- *Bins* con `binIdx = 0`:

$$\texttt{ctxIdxInc} = ((\texttt{ngt1s} \neq 0)\ ?\ 0 : \texttt{min}(4, 1 + \texttt{n1s}))$$

- *Bins* con `binIdx` ϵ $[1, 13]$:

$$\texttt{ctxIdxInc} = 5 + \texttt{min}(4, \texttt{ngt1s})$$

En la Figura 2.25, también se muestra un ejemplo del cálculo de `ctxIdxInc` para `coeff_abs_level_minus1`. Asimismo, se incluyen los valores de `coeff_sign_flag` a codificar.

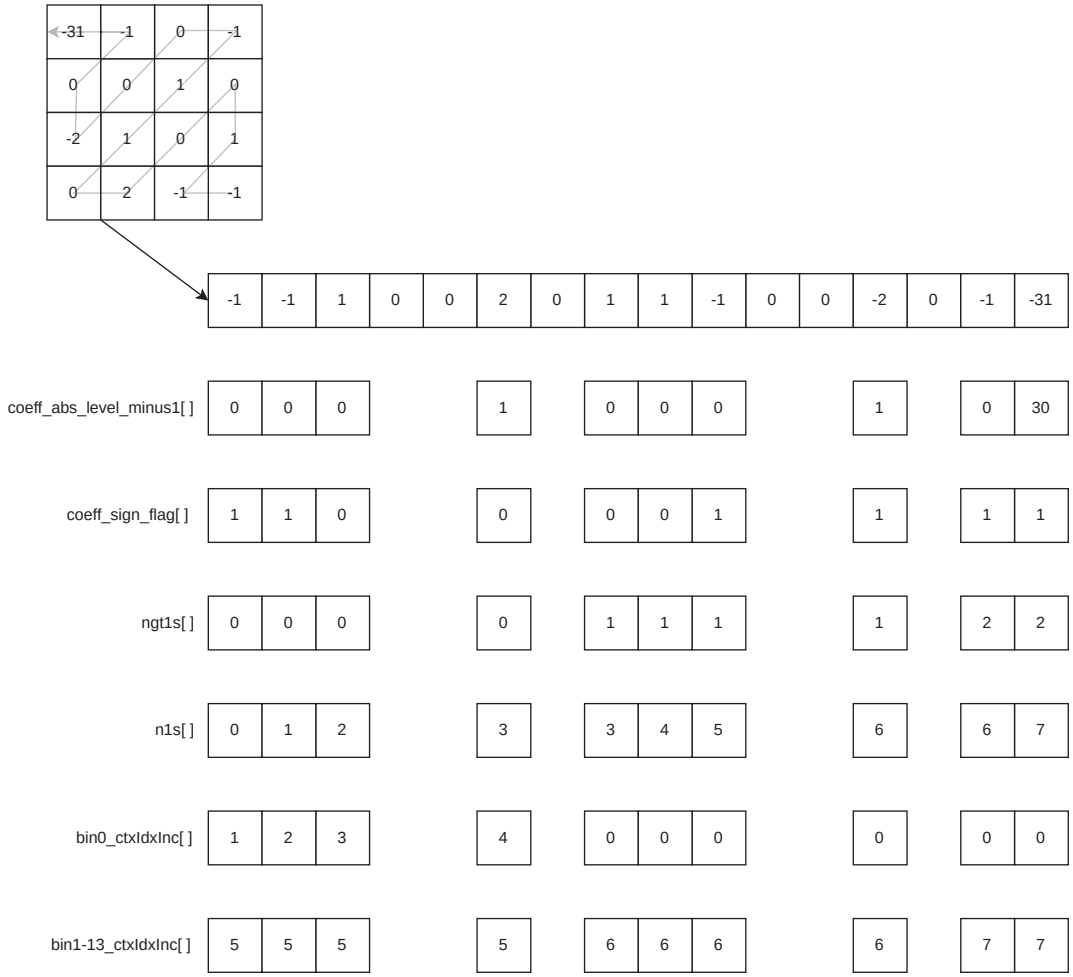

Figura 2.25: Ejemplo de cálculo de `ctxIdxInc` para `coeff_abs_level_minus1`.

2.3.4.3. `ctxIdxBlockCatOffset`

Una vez calculado el `ctxIdxInc`, para los SE residuales resta determinar el ya mencionado `ctxIdxBlockCatOffset`. El valor de esta variable depende tanto de la categoría de contexto del B4, `ctx_cat`, así como del tipo de SE a codificar.

En primer lugar, ha de determinarse el valor de `ctx_cat` a usar. Considerando las limitaciones del diseño ya mencionadas, de la Tabla 2.10 se deduce que el valor de `ctx_cat` será siempre 2, y que el máximo número de coeficientes por B4 será 16. Esto se debe a que únicamente se emplea codificación Intra 4x4 para imágenes monocromas, por lo que los B4 a codificar serán todos del tipo `Luma-Intra4-Intra16`.

ctx_cat		Máximo número de coeficientes por B4
Nombre	Valor	
Luma-Intra16-DC	0	16
Luma-Intra16-AC	1	15
Luma-Intra4-Intra16	2	16
Chroma-DC	3	4
Chroma-AC	4	15
...		

Tabla 2.10: Valor de `ctx_cat` por tipo de B4. Adaptado de [16].

Tras determinar el valor de `ctx_cat`, finalmente puede hallarse el *offset* asociado a la categoría de contexto, `ctxIdxBlockCatOffset`, para cada SE. Dichos valores se encuentran tabulados en [16], recogiéndose exclusivamente en la Tabla 2.11 aquellos utilizados en este diseño:

SE	ctxIdxBlockCatOffset
`coded_block_flag`	8
`significant_coeff_flag`	29
`last_significant_coeff_flag`	29
`coeff_abs_level_minus1`	20

Tabla 2.11: Valor de `ctxIdxBlockCatOffset` según el tipo de SE.

2.3.5. Codificación de Elementos Sintácticos

Habiendo concretado todos los aspectos necesarios acerca de cada SE de manera individual, finalmente queda exponer su orden de codificación dentro de cada MB. Cabe resaltar que todos aquellos SE cuyo *binstring* cuenta con más de un *bin* se comienzan a codificar por el de índice 0.

Retomando el orden de codificación, el primer SE a ser codificado es `mb_type`. Posteriormente, se codifica la información de predicción asociada a cada uno de los dieciséis B4 que forman el MB. Esta información la componen `prev_intra4x4_pred_mode_flag`

y `rem_intra4x4_pred_mode`. El segundo SE únicamente se codifica para cada B4 en el caso de que el modo de predicción empleado para el mismo no sea el más probable, es decir si `prev_intra4x4_pred_mode_flag` es '0'.

Después, se continúa con `coded_block_pattern`. En caso de que su valor sea 0, esto indicará que no hay coeficientes significativos en todo el MB, por lo que se dará el proceso por terminado.

Si por el contrario `coded_block_pattern` es distinto de 0, se prosigue con `mb_qp_delta` y con los SE residuales de cada B4, siempre que dicho B4 pertenezca a un B8 en el que existan coeficientes significativos. Cabe recordar que cada uno de los cuatro bits de `coded_block_pattern` son los que señalan qué B8 cumplen esta condición.

Por otro lado, atendiendo a los SE residuales, el primer paso a realizar se corresponde con la codificación de `coded_block_flag`. Si este es distinto de '0' habrá coeficientes significativos que codificar. En caso contrario, se considera el B4 como terminado.

Cuando existen coeficientes, primeramente se codifica el llamado "mapa de significancia", formado por `significant_coeff_flag` y `last_significant_coeff_flag`. Nótese que el segundo sólo se codifica para aquellos coeficientes significativos, es decir, sólo cuando el `significant_coeff_flag` ligado al coeficiente i es '1'.

Finalmente, se concluye con la codificación de los coeficientes significativos. Primero se codifica su nivel en valor absoluto menos una unidad, `coeff_abs_level_minus1`, y posteriormente su signo `coeff_sign_flag`. Tras esto, se codifica el `end_of_slice_flag` cuyo valor será '1' si el MB codificado es el último del *slice*. En caso contrario será '0'.

En las Figuras 2.26 y 2.27 se incluyen los diagramas de flujo que resumen los procesos descritos. Es preciso señalar que los SE que aparecen marcados con " * " son aquellos cuyo *bin* con `binIdx = 0` es el LSB.

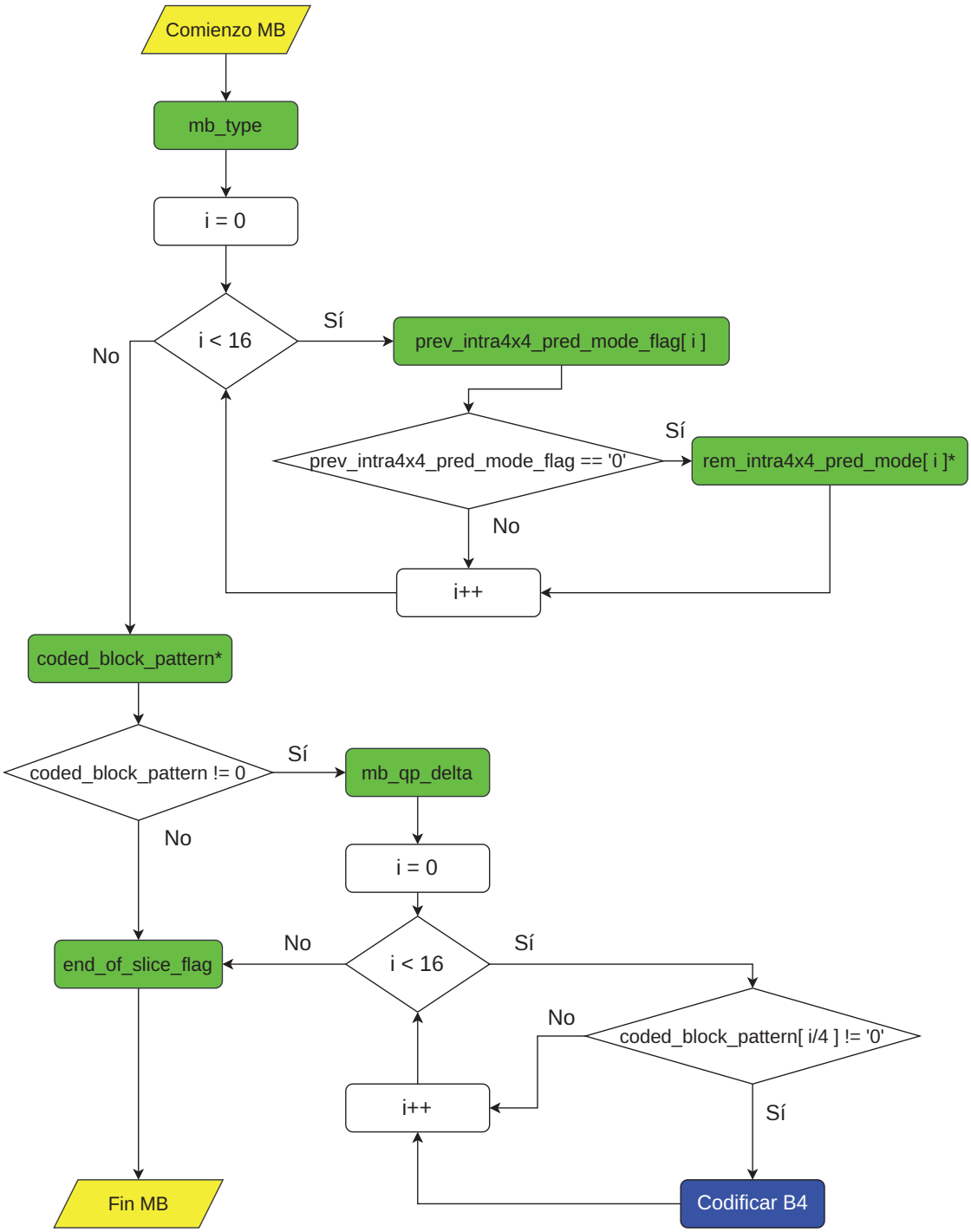

Figura 2.26: Orden de codificación de los SE en cada MB.

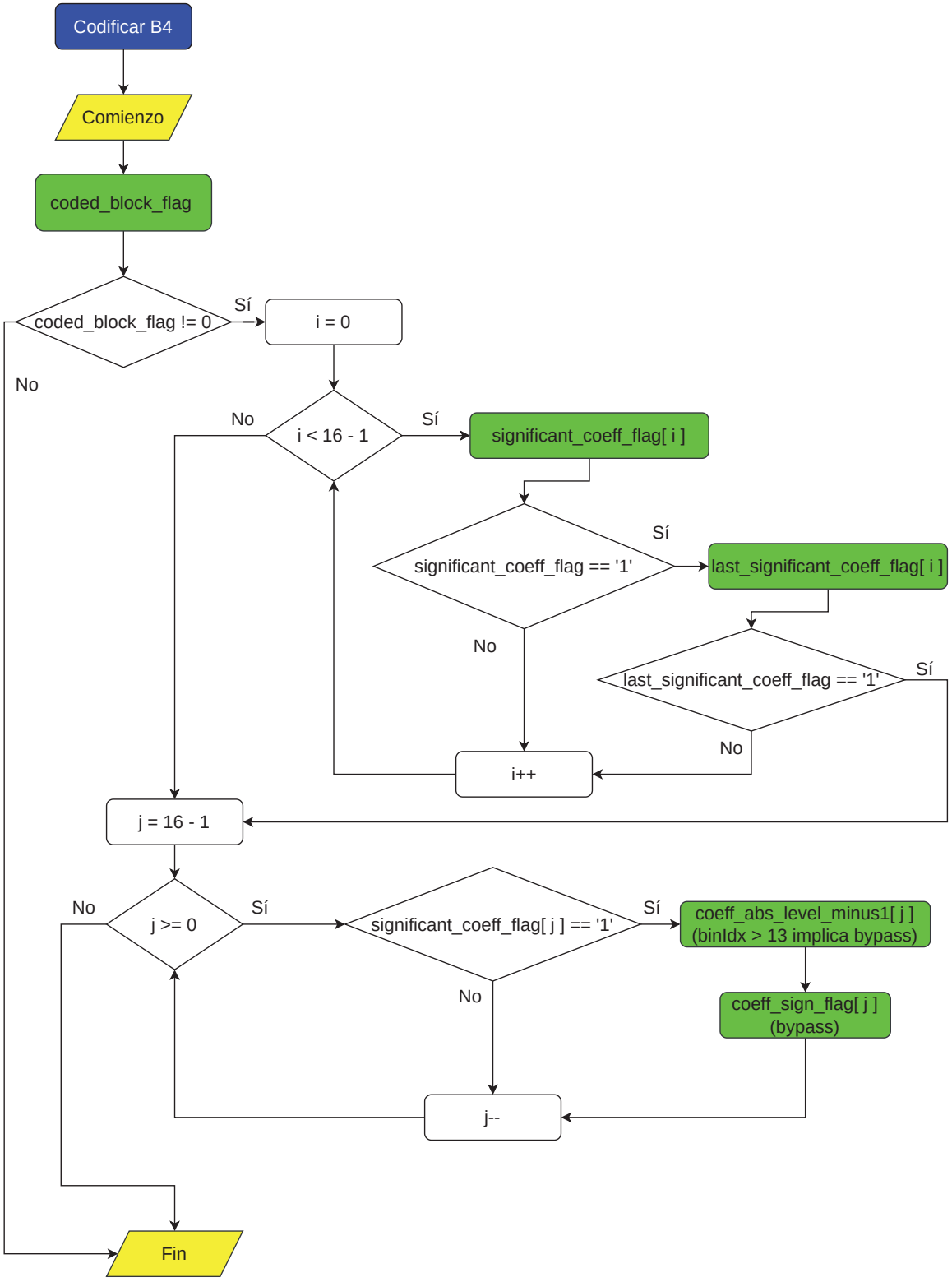

Figura 2.27: Orden de codificación de los SE en cada B4.

2.4. Conclusiones

En este capítulo se han introducido algunos conceptos clave relativos a la compresión de vídeo. Asimismo, se ha expuesto la necesidad de uso de estas técnicas, así como su importancia en la actualidad. Se ha resumido el flujo de datos que emplea H.264, estándar al que pertenece el codificador entrópico CABAC desarrollado en el presente TFG.

Acto seguido, se ha profundizado en el propio codificador describiendo su diagrama de bloques y cada uno de los subsistemas que lo componen: binarizador, modelador de contextos, y codificador aritmético, estando este último subdividido en el *regular coding engine* y el *bypass coding engine*.

Tras haber detallado el funcionamiento del codificador, se han abarcado todos los aspectos de necesario conocimiento acerca de los SE relacionados con CABAC. Primeramente, estos se han expuesto junto con sus significados. A continuación, se ha indicado el conjunto de SE realmente utilizado en este proyecto dadas las restricciones de la Subsección 2.3.2. Después, para este subconjunto de SE, se han descrito los procesos de binarización y de cálculo de índices de contexto y vecinos (si procede). Finalmente, se ha concluido con el orden de codificación de estos elementos sintácticos, tanto a nivel de MB como a nivel de B4.

Capítulo 3

Diseño e implementación de la arquitectura

Tras haber realizado el análisis del estado del arte en el que, además de la introducción a la codificación de vídeo y al estándar H.264/AVC, se ha explorado el funcionamiento, las restricciones y los elementos sintácticos utilizados en el presente proyecto, ahora se introduce la solución implementada. En primer lugar, se ofrece una vista global de la arquitectura escogida, explicando el flujo de datos de la misma. Luego, se analiza detalladamente cada uno de los submódulos que la componen.

Habiendo expuesto el diseño desarrollado, se procede a describir el proceso de verificación utilizado para comprobar su correcto funcionamiento. Dicho proceso se basa principalmente en el uso del *software* de referencia JM 19.0 en conjunto con Vivado™ 2020.2. Asimismo, se exponen las funciones de interés dentro de JM 19.0 y las modificaciones realizadas para obtener la información necesaria para llevar a cabo la verificación.

3.1. Diseño de CABAC

Para el diseño de la arquitectura se han estudiado soluciones como las presentadas en [17], [18] y [19], entre otras. Si bien aportan optimizaciones importantes, los objetivos del proyecto de hacer una arquitectura sencilla y con una utilización baja en recursos, así como fácilmente ampliable en el futuro, han llevado a adoptar en este Trabajo de Fin de Grado (TFG) soluciones arquitecturales más sencillas. Esto redundará a que la extensión de funcionalidad en el futuro sea menos difícil.

3.1.1. Vista global de la arquitectura utilizada

Atendiendo a la arquitectura propuesta (véase la Figura 3.1), cabe recalcar que las entradas de datos del se corresponden con los coeficientes residuales cuantizados de transformadas (de ahora en adelante coeficientes) y con los Elementos Sintácticos (del inglés *Syntax Elements*) (SE) `prev_intra4x4_pred_mode_flag` y `rem_intra4x4_pred_mode`. Dicha información es proporcionada directamente por el módulo previo a *Context Adaptative Binary Arithmetic Coding* (CABAC) en la implementación de H.264 para el proyecto del cual forma parte este diseño.

A diferencia de los SE mencionados en el párrafo anterior, el resto de ellos han de ser deducidos de los propios coeficientes. Para la obtención de los elementos sintácticos residuales se ha diseñado el *Intérprete de SE residuales*. Este tiene como salida los `significant_coeff_flag`, `last_significant_coeff_flag`, `coeff_abs_level_minus1`, `coeff_sign_flag` y `coded_block_flag` correspondientes a cada Bloque 4x4 (B4). Nótese que los coeficientes introducidos al intérprete provienen del *Buffer de coeficientes* y del *Serializador*[1]. Ello es debido a que los coeficientes se reciben por B4, siendo necesario, por un lado, aplanar los datos de entrada y, por otro, colocarlos en orden de escaneo inverso para su codificación como ya se comentó en la Subsección 2.3.4. De la primera tarea, además de calcular el `coded_block_pattern`, se encarga el *Buffer de coeficientes*,

[1]Tanto el *Buffer de coeficientes* como el *Serializador* son subsistemas reutilizados del diseño de *Context Adaptive Variable Length Coding* (CAVLC) para H.264, perteneciente al proyecto del cual forma parte este TFG. Por lo tanto, su desarrollo ha sido completamente externo a este trabajo, razón por la cual se destacan en naranja en la Figura 3.1.

mientras que de la segunda es responsable el *Serializador*.

El valor de `coded_block_pattern` calculado por el *Buffer de Coeficientes* podría ser introducido directamente tanto al *Intérprete de SE residuales* así como al *Modelador de Contextos*. Sin embargo, para futuras ampliaciones del diseño se considera que dicho valor ha de ser comprobado previamente por la *Unidad de control de salida*. Esta unidad es responsable de gestionar las señales de control del *Serializador* y del *Buffer de coeficientes*. También agrupa todos los `prev_intra4x4_pred_mode_flag` y `rem_intra4x4_pred_mode` de cada Macrobloque (MB) y los envía a la *First In First Out* (FIFO) correspondiente.

En cuanto al *Modelador de Contextos*, conviene hacer hincapié en que este módulo únicamente calcula los incrementos de índices de contexto, `ctxIdxInc`, para aquellos SE que requieren datos de vecinos: `coded_block_flag` y `coded_block_pattern`. Los `ctxIdxInc` de los `coeff_abs_level_minus1` se determinan en el *Intérprete de SE residuales*, a la vez que se calcula el valor del SE en sí. En el caso de `end_of_slice_flag`, `prev_intra4x4_pred_mode_flag`, `rem_intra4x4_pred_mode`, `mb_type` y `mb_qp_delta`[2], sus incrementos de índice de contexto son fijos, no requiriéndose ningún cálculo adicional. Asimismo, los incrementos de índice de contexto de `significant_coeff_flag` y `last_significant_coeff_flag` pueden deducirse fácilmente al tiempo que se binarizan, de modo que en este diseño se calculan en el *Binarizador*, aunque también sería posible llevar a cabo esta operación en el *Intérprete de SE*. Se ha decidido hacerlo de esta manera por no almacenar información redundante en las FIFO.

Una vez se conocen los `ctxIdxInc`, algunos se guardan encapsulados con el SE al que están asociados en la FIFO que corresponda[3]. Seguidamente, el *Binarizador* se encarga de extraer de dichas FIFO los SE pertinentes en cada momento. Además, tiene la función de binarizar cada SE a la vez que actualiza y recupera los contextos[4] requeridos por el codificador binario aritmético, *BAC*. Este último genera las órdenes de escritura de nuevos bits, las cuales son llevadas a cabo por el *Escritor de bits*.

[2] `mb_type` y `mb_qp_delta` no son entradas para la arquitectura propuesto dado que siempre son cero.

[3] El bloque *FIFO* de la Figura 3.2 contiene varias FIFO (tipo *Look-Ahead*). Algunas admiten una única clase de SE, mientras que otras, por conveniencia, almacenan dos tipos de SE diferentes.

[4] Los contextos se almacenan en el módulo *Tablas de Contextos*, que además implementa los procesos necesarios para la inicialización de los mismos.

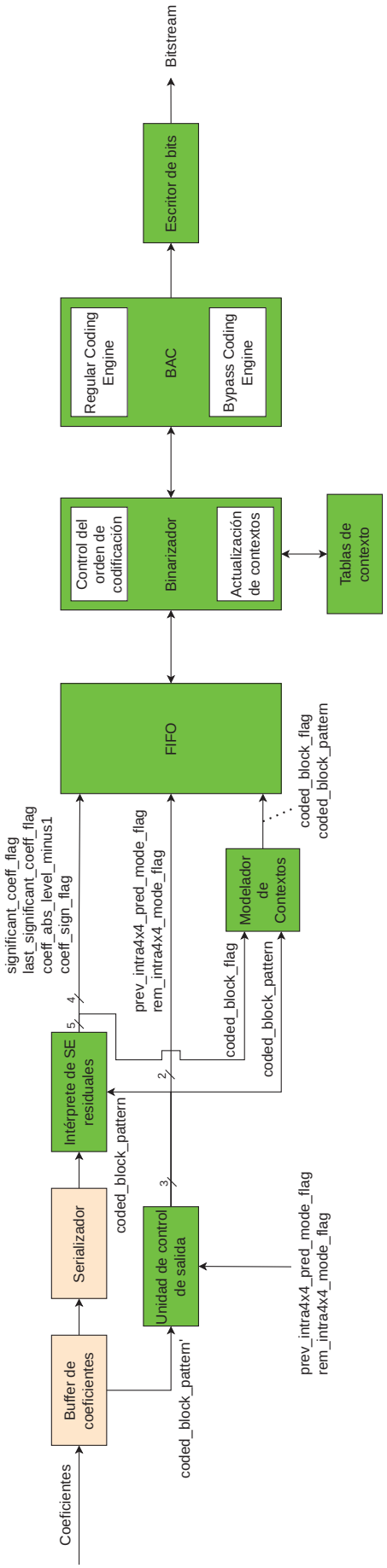

Figura 3.1: Arquitectura de CABAC con módulos auxiliares.

3.1.2. Intérprete de Elementos Sintácticos residuales

Como bien se indicó en el apartado anterior, este subsistema tiene como fin obtener los SE residuales y sus `ctxIdxInc`, salvo en el caso de `coeff_sign_flag` y de `coded_block_flag`. Es preciso recordar que el primero será codificado por el *Bypass Coding Engine*, por lo que no requiere contextos, mientras que el incremento de índice de contexto del segundo se determina en el *Modelador de Contextos*.

Por otro lado, para el diseño de este bloque es conveniente tener en cuenta que a su entrada se tiene un nuevo coeficiente por cada ciclo de reloj. Dichos coeficientes aparecen en orden inverso de escaneo. Todos los coeficientes generados en una secuencia de vídeo pasan por este módulo independientemente de su valor, pues es tarea del *Intérprete de SE residuales* descartar aquellos que no sean imprescindibles.

En el diseño realizado, todos los SE residuales se deducen simultáneamente. Sin embargo, cabe señalar que estos sólo se envían cuando son necesarios. Es decir, si `coded_block_pattern` indica que para el Bloque 8x8 (B8) al que pertenece el B4 que se está procesando no aparece ningún coeficiente significativo, no se genera ninguno de los SE residuales. Por otro lado, si existen coeficientes significativos en el Bloque 8x8, pero no en el B4, se envía únicamente el `coded_block_flag` asociado a este último bloque. Por contra, si existen coeficientes significativos en el B4, solo se codificarán estos, descartando los que sean originalmente cero.

A continuación se explica cómo el *Intérprete de SE residuales* gestiona cada SE, siempre y cuando proceda, mientras que la obtención de los mismos se realiza de acuerdo a sus propias definiciones (Subsección 2.3.1). Retomando los ejemplos de las Figuras 2.24 y 2.25 se ha confeccionado la Figura 3.2. En ella se observa que por cada B4 se envía un vector de `significant_coeff_flag` y otro de `last_significant_coeff_flag` agrupados en un `record`[5] una vez han sido procesados todos los coeficientes.

En lo que respecta a los coeficientes, cada ciclo de reloj se introduce en la FIFO un `record` que contiene un `coeff_abs_level_minus1` con sus incrementos de índice de

[5]Un `record` es un tipo de dato estructurado en *Very high speed integrated circuit Hardware Description Language* (VHDL), equivalente al `struct` de C.

contexto referidos los *bins* de índices 0 y 13, `bin0_ctxIdxInc` y `bin1-13_ctxIdxInc`, respectivamente, y con su `coeff_sign_flag` asociado. Dichos `ctxIdxInc` se calculan mediante las expresiones relativas a `coeff_abs_level_minus1` expuestas en la Subsección 2.3.4.

Finalmente, el `coded_block_flag` (no representado en la Figura 3.2) asignado a cada B4 se envía tras haber procesado todos los SE de dicho bloque.

Nótese que cada `record` se representa por medio de los rectángulos de colores que agrupan los valores que lo componen. Los que aparecen en vertical se envían al registro de salida en el mismo ciclo que se crean, mientras que el que aparece en horizontal se envía al leer todos los coeficientes. Es decir, por cada B4 se genera un `record` para el mapa de significancia y varios para los coeficientes según sea necesario.

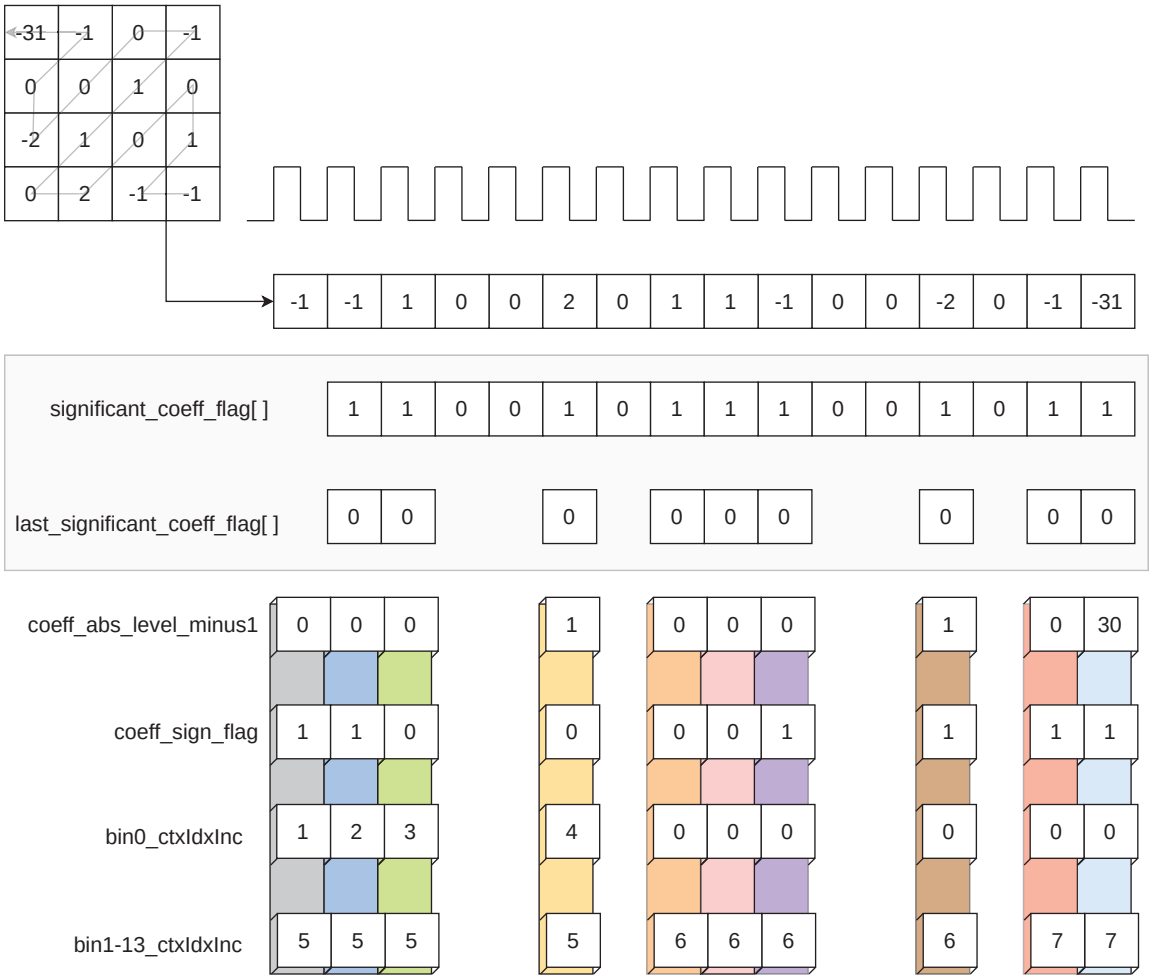

Figura 3.2: Ejemplo de obtención de SE residuales.

3.1.3. Modelador de Contextos

Según se indicó anteriormente, el *Modelador de Contextos* diseñado se encarga de calcular los `ctxIdxInc` de `coded_block_flag` y `coded_block_pattern`. Estos cálculos se realizan de acuerdo a lo ya explicado en la Subsección 2.3.4 acerca de estos SE. En cuanto a la utilización de datos de vecinos, en las Subsecciones 3.1.3.1 y 3.1.3.2 se describen los mecanismos empleados para su gestión.

3.1.3.1. Gestión de los contextos de `coded_block_pattern`

Los datos de vecinos de este elemento sintáctico se almacenan en un *array* registrado de `coded_block_pattern`. El número de elementos que ha de almacenar dicho *array* se corresponde con el número máximo de MB en la dimensión horizontal de la resolución utilizada. En la Figura 3.3 se representa gráficamente la subdivisión de un *slice* en MB. En azul claro se colorean los MB cuyos datos asociados ya no son necesarios. En azul oscuro se hallan los que se necesitan para futuros `coded_block_pattern` y para el actual. En gris se colorea el MB cuyo `coded_block_pattern` se está codificando, `MB actual`.

Cabe recalcar que para cada `coded_block_pattern` únicamente son necesarios los valores de los `coded_block_pattern` asociados al MB vecino que se encuentra inmediatamente a la izquierda, `A`, y al inmediatamente superior, `B`.

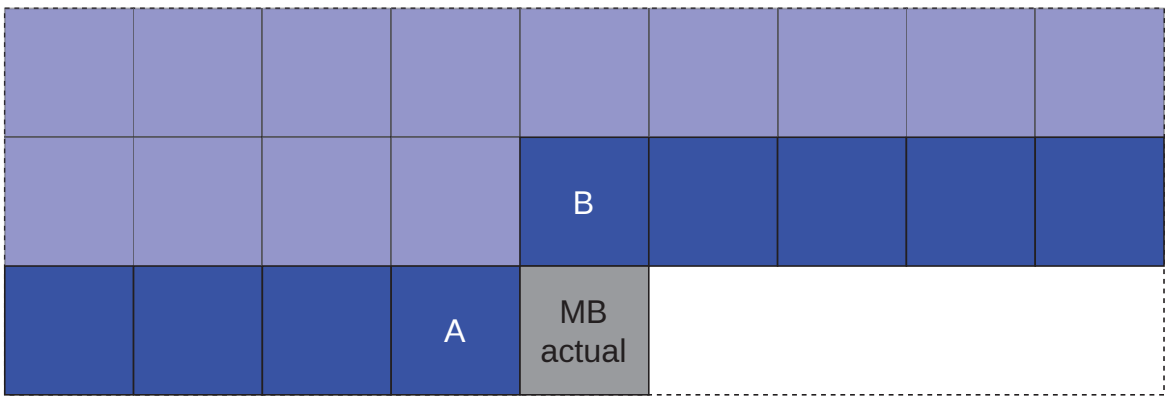

Figura 3.3: Datos de vecinos almacenados.

Por otro lado, coded_block_flag también necesita valores de coded_block_pattern vecinos. Esto añade cierta complejidad al diseño, pues debido a su construcción, cada coded_block_pattern llega al *Modelador de Contextos* 17 ciclos de reloj antes que el primer coded_block_flag del MB que se está codificando.

Esta diferencia en ciclos de reloj se debe a que coded_block_flag se envía una vez han pasado todos los coeficientes por el *Intérprete de SE residuales*, que son 16. A esa cantidad ha de sumarse el ciclo que tarda coded_block_flag en propagarse por el registro de salida de dicho módulo. Además, en caso de que alguno de los tres primeros B8 del MB que se está codificando no contenga coeficientes significativos, no se enviarían los coded_block_flag asociados a estos bloques, siendo mayor el número de ciclos entre el coded_block_pattern y el primer coded_block_flag.

En las Figuras 3.4 y 3.5 se muestra la estrategia llevada a cabo para paliar este inconveniente. Básicamente se utiliza un registro aparte donde siempre se introduce el nuevo valor de coded_block_pattern. Con la llegada de otro coded_block_pattern, se copia el valor de dicho registro al *array* en la posición correspondiente, a la vez que se almacena el nuevo valor en este registro.

En los ejemplos de las Figuras mencionadas se observa que cuando MB actual se corresponde con $\texttt{MB}_{\texttt{(X,Y)}}$, representado en azul claro, se almacena el valor de su coded_block_pattern asociado en un registro aparte, también en azul claro. Al mismo tiempo, el *array* de registrado de coded_block_pattern permanece inalterado hasta que se pasa a codificar otro coded_block_pattern de modo que este pueda ser utilizado correctamente por los coded_block_flag correspondientes. Acto seguido, cuando MB actual pasa a ser $\texttt{MB}_{\texttt{(X+1,Y)}}$ se tiene un nuevo valor de coded_block_pattern. Por tanto, ahora se inserta el valor previo coded_block_pattern al *array* y en el registro se introduce el nuevo, representado en gris.

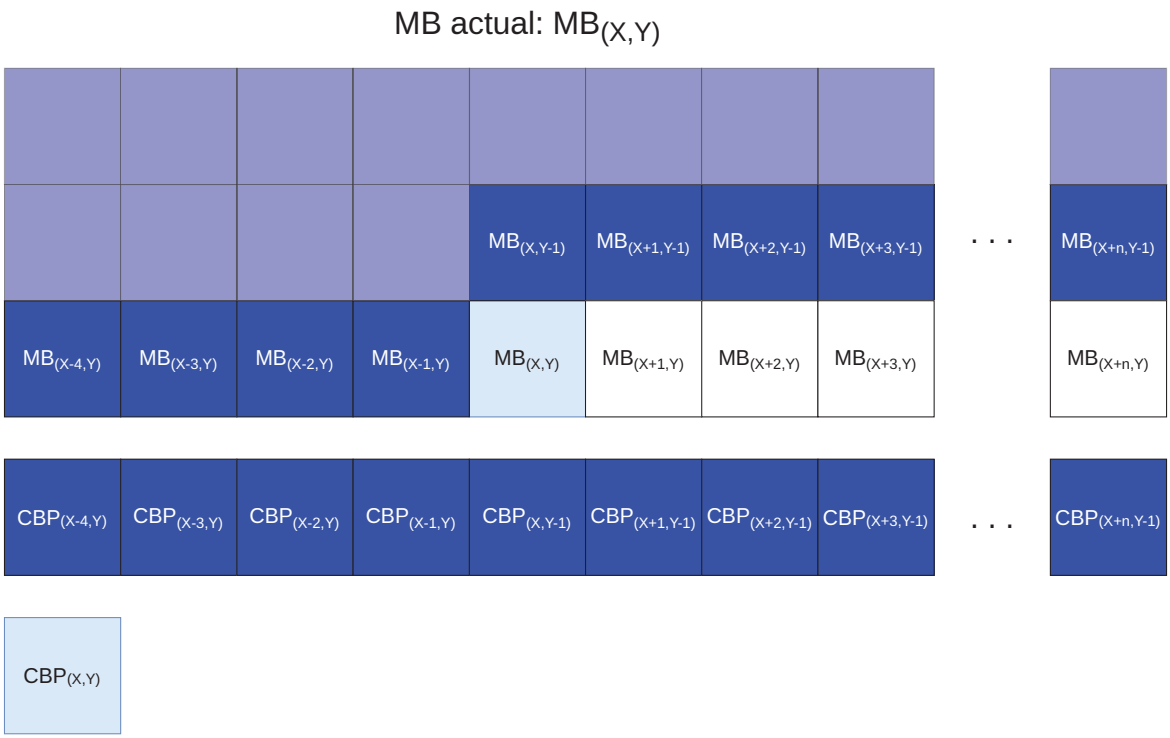

Figura 3.4: Gestión de vecinos de `coded_block_pattern` (I).

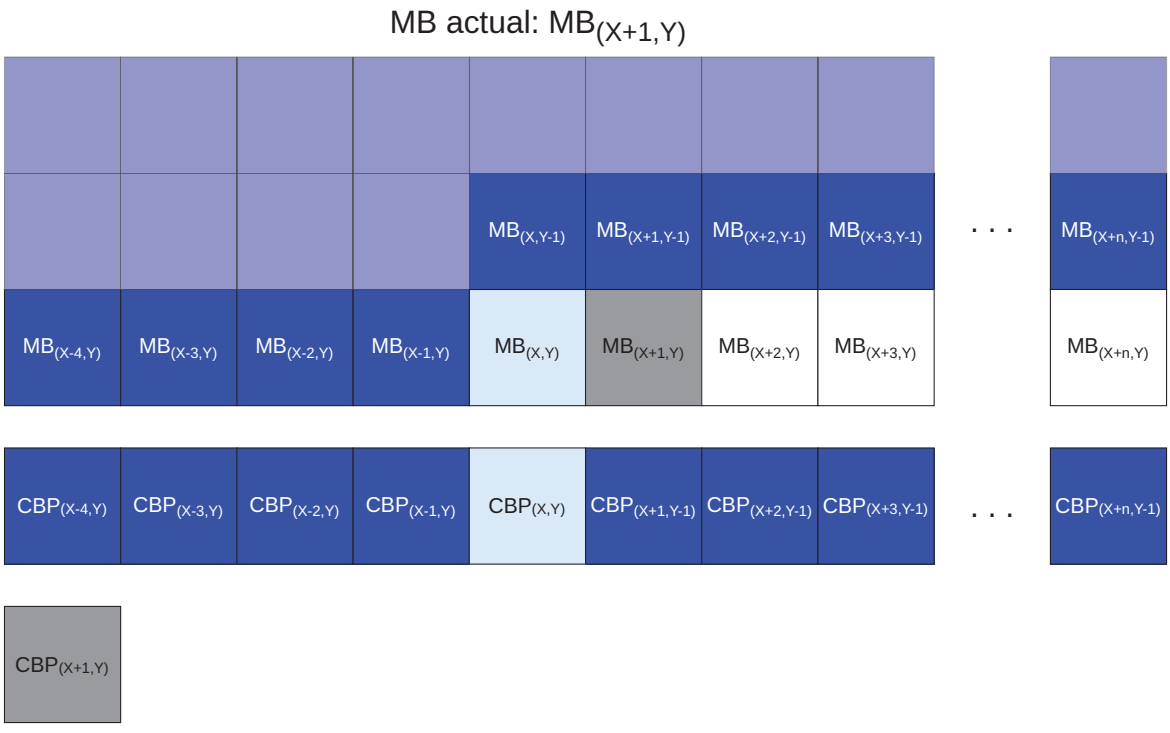

Figura 3.5: Gestión de vecinos de `coded_block_pattern` (II).

3.1.3.2. Gestión de los contextos de `coded_block_flag`

Atendiendo a la gestión de los vecinos para `coded_block_flag`, la solución implementada es similar a la expuesta en el apartado anterior: los `coded_block_flag` pertenecientes al MB actual se guardan en un vector registrado específico para ello. Es decir, este vector recoge los datos de vecinos que se hallan en el MB que se está codificando.

Por otro lado, los vecinos que pertenecen a otros MB se guardan en un *array* de vectores y un vector, ambos registrados. El *array* de vectores almacena vectores de cuatro `coded_block_flag` por cada MB (los asociados a los B4 de índices 10, 11, 14 y 15). Estos valores son los que utilizarán los MB de la fila siguiente como "datos del vecino `B`" según corresponda.

En cuanto al vector único mencionado previamente, este se utiliza para los vecinos `A`, siendo necesarios solamente los del MB anterior. Para este segundo caso, los `coded_block_flag` almacenados son los relativos a los B4 de índices 5, 7, 13 y 15. En la Figura 3.6 se destacan en azul oscuro los datos registrados, mientras que en azul claro se representan los descartados atendiendo al estado actual de la codificación. Además, en gris se indica el B4 asociado al `coded_block_flag` que se está procesando en ese momento.

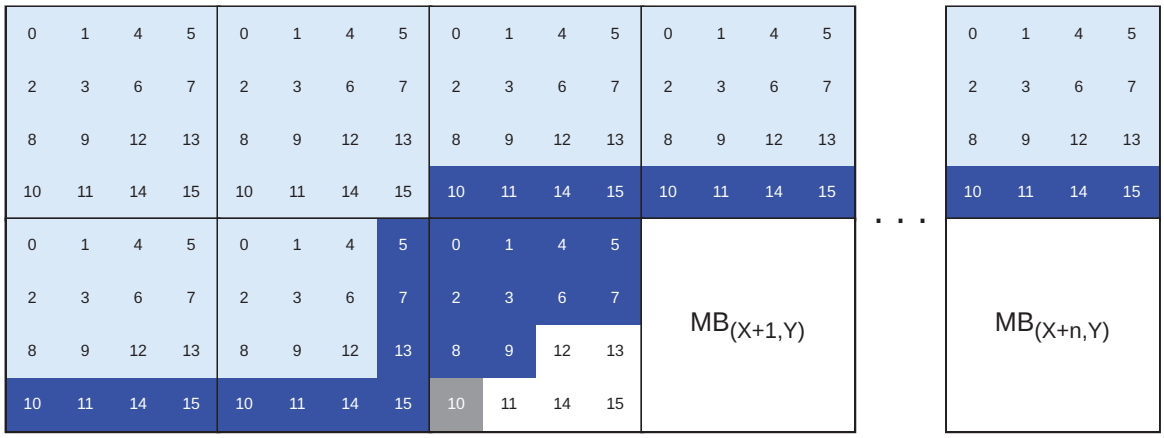

Figura 3.6: Gestión de vecinos de `coded_block_flag`.

3.1.4. Binarizador

Este submódulo asume la binarización de los SE conforme a lo indicado en la Subsección 2.2.1. Adicionalmente, el binarizador realiza el control del orden de codificación además de encargarse de obtener los datos relativos a los contextos requeridos por el codificador binario aritmético (del inglés *Binary Arithmetic Coder*) (BAC) así como de la actualización de los propios contextos.

3.1.4.1. Control del orden de codificación

El control de orden de codificación resulta un proceso necesario dada la arquitectura escogida para el sistema. En ella no se almacena información acerca de la relación de cada SE con un B4, B8 o MB en particular. Este proceso básicamente analiza los propios valores de los SE deduciendo cuántos de cada tipo han de ser codificados por cada bloque. Para ello son de gran ayuda las FIFO utilizadas, pues el orden de llegada de los SE a las mismas es crucial.

En cuanto a la implementación de este proceso en sí, se ha diseñado una Máquina de Estados Finitos (MEF) que controla la secuencia de SE a codificar en cada instante. En la Figura 3.7 se incluye una simplificación de su grafo de estados. Para facilitar la comprensión de dicho diagrama, en la Tabla 3.1 se recogen las abreviaciones empleadas para referirse a los SE utilizados. Además, en las Tablas 3.2 y 3.3 se halla la definición de cada una de las señales relativas a esta MEF. Nótese que las señales con sufijo `_i` se corresponden con entradas (*inputs*), las de sufijo `_s` con señales internas y las que tienen sufijo `_o` son salidas (*outputs*). Del mismo modo, en el grafo de estados estas señales aparecen coloreadas en naranja, fucsia y azul, respectivamente.

Elemento sintáctico	Abreviación
mb_type	mbt
prev_intra4x4_pred_mode_flag	pmf
rem_intra4x4_pred_mode	rem
coded_block_pattern	cbp
mb_qp_delta	qpd
coded_block_flag	cbf
significant_coeff_flag	scf
last_significant_coeff_flag	lscf
coeff_abs_level_minus1	calm1
coeff_sign_flag	csf
end_of_slice_flag	eosf

Tabla 3.1: Abreviaciones de SE utilizadas en la Figura 3.7.

Señal	SE asociado	Descripción
mbt_fifo_read_o	mb_type	Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.
predinfo_done_s	prev_intra4x4_pred_mode_flag rem_intra4x4_pred_mode	'1' si se ha codificado la información de predicción de los 16 B4 que conforman el MB, '0' en caso contrario.
predinfo_fifo_read_o		Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.
pmf_i	prev_intra4x4_pred_mode_flag	Valor del pmf que se está codificando.
rem_done_s	rem_intra4x4_pred_mode	'1' si se ha terminado de codificar el rem actual, '0' en caso contrario.
cbp_i	coded_block_pattern	Valor del cbp que se está codificando.
cbp_done_s		'1' si se ha terminado de codificar el cbp actual, '0' en caso contrario.
cbp_fifo_read_o		Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.
qpd_fifo_read_o	mb_qp_delta	
cbf_all_coded_s	coded_block_flag	'1' cuando todos los cbp de un MB han sido codificados, '0' en caso contrario.
cbf_i		Valor del cbf que se está codificando.
cbf_fifo_read_o		Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.

Tabla 3.2: Señales de la MEF del binarizador (I).

Señal	SE asociado	Descripción
sigmap_done_s	significant_coeff_flag last_significant_coeff_flag	'1' si se ha terminado de codificar el los scf y lscf del B4 actual, '0' en caso contrario.
sigmap_fifo_read_o		Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.
scf_i	significant_coeff_flag	Valor del scf que se está codificando.
scf_get_next_s		'1' para incrementar el índice del vector de scf del B4 actual, '0' en caso contrario.
lscf_i	last_significant_coeff_flag	Valor del lscf que se está codificando.
lscf_get_next_s		'1' para incrementar el índice del vector de lcsf del B4 actual, '0' en caso contrario.
levelinfo_fifo_read_o	coeff_abs_level_minus1 coeff_sign_flag	Confirmación de lectura de la salida de la FIFO. Esto implica la actualización de su salida.
levelinfo_b4_all_coded_s		'1' si se han codificado todos los coeficientes del B4 actual, '0' en caso contrario.
levelinfo_mb_all_coded_s		'1' si se han codificado todos los coeficientes del MB actual, '0' en caso contrario.
calm1_done_s	coeff_abs_level_minus1	'1' si se ha terminado de codificar el rem actual, '0' en caso contrario.

Tabla 3.3: Señales de la MEF del binarizador (II).

Atendiendo al diseño de la MEF, en la Figura 3.7 aparecen un total de once estados. Cada uno de ellos está asociado a la codificación del SE homónimo. Cabe destacar que la MEF implementada realmente cuenta con un mayor número de señales de las mostradas en esta figura con el fin de facilitar su entendimiento. Asimismo, las señales de actualización de la tabla de contexto asociada a cada SE siempre están a '1' durante la codificación del mismo (considerando que el *BAC* tiene su señal de `ready` activa, que la FIFO de ese SE tiene datos válidos en su salida, etc.) por lo que no se ha considerado su inclusión en el diagrama. El cálculo de los nuevos `pStateIdx` se realiza según lo explicado en las Subsecciones 2.2.3.1 y 2.2.3.3 partiendo de que se introduce un nuevo *bin* al *BAC* cada ciclo de reloj.

La MEF deriva de los diagramas de flujo en alto nivel de las Figuras 2.26 y 2.27, los cuales han sido adaptados para la implementación *hardware*. El primer SE que se codifica tanto en un *slice* como en un MB es `mb_type`. La MEF no permanece más de un ciclo de reloj en este estado, ya que, como se explicó en la Subsección 2.3.3, `mb_type` siempre es 0, de modo que su *binstring* está constituido únicamente por un *bin*. Igual ocurre con `mb_qp_delta`. Por tanto, en estos estados únicamente resulta de interés la señal de confirmación de lectura a la FIFO correspondiente para que esta actualice su salida.

Tras codificar `mb_type` siempre se procesa toda la información de predicción del MB completo. En el caso de que no se haya utilizado el Modo Más Probable (MPM) (`pmf_i = '0'`) para un B4 es necesario indicar cuál se ha empleado por medio de la codificación del `rem_intra4x4_pred_mode` asociado. A diferencia del SE anterior, este requiere de varios ciclos para su codificación, pues su *binstring* lo conforman un total de tres *bins*, de ahí que sea necesaria la señal `rem_done_s`. Mientras no se haya terminado de codificar este SE no se prosigue con la información de predicción (`predinfo`) del siguiente B4.

Una vez se concluye con la información de predicción de todo el MB se continúa con `coded_block_pattern`. En este punto debe tenerse en cuenta que por cada *bin* de este SE cuyo valor sea '1' será necesario codificar cuatro `coded_block_flag`. Ello se debe a que cada *bin* de `coded_block_pattern` está asociado a un B8, como ya se ha mencionado en

el Capítulo 2. A su vez, cada B8 engloba cuatro B4. Entonces, cada `coded_block_flag` finalmente indicará qué B4 contienen coeficientes significativos. Es decir, a partir del valor de `coded_block_pattern` es posible determinar el número de `coded_block_flag` a codificar, y a partir del número de `coded_block_flag` no nulos será posible calcular cuántos mapas de significancia (`sigmap`) deberán procesarse. Si `coded_block_pattern` es nulo, esto implica que no hay coeficientes distintos de cero en el MB concluyéndose la codificación del mismo con el `end_of_slice_flag`[6]. Si hay coeficientes significativos se envía `mb_qp_delta` y seguidamente el primer `coded_block_flag` que corresponda.

Si `coded_block_flag` no es nulo, se procesan los SE que conforman el llamado "mapa de significancia": `significant_coeff_flag` y `last_significant_coeff_flag`. Por el contrario, si `coded_block_flag` fuese no sería necesario enviar información adicional acerca del B4, ya que no hay coeficientes que codificar y por tanto dicho mapa es innecesario.

Los SE del mapa de significancia tienen la particularidad, respecto al resto, de que se almacenan en un vector por cada B4. En otras palabras, no es necesario leer de la FIFO asociada a `sigmap` cada ciclo de reloj, sino una vez por B4. Por cada `significant_coeff_flag` no nulo ha de enviarse un `last_significant_coeff_flag` y codificarse un coeficiente con su respectivo signo una vez haya terminado el análisis de `sigmap`. Si `last_significant_coeff_flag` es '1', se determina que no hay más coeficientes significativos en el B4. Entonces, se codificarán los N coeficientes no nulos según se haya deducido de estos dos SE. Sin embargo, también puede procederse a codificar los coeficientes pese a no existir un `last_significant_coeff_flag` = '1', pues como se mencionó en la Subsección 2.3.1 esta circunstancia implica que el último coeficiente del Bloque 4x4 en orden de escaneo no es nulo.

Finalmente, se codifica cada `coeff_abs_level_minus1` con su signo, `coeff_sign_flag`, hasta haber leído de la FIFO los N coeficientes no nulos. Tras ello, se concluye la codificación del MB con `end_of_slice_flag`, pasando al siguiente.

[6]El `end_of_slice_flag` se codifica al final de cada MB con la particularidad de que si su valor es '1' se entiende que el MB al que pertenece es el último del *slice*. Por tanto, el proceso `EncodeTerminate` determina que es necesario llevar a cabo `EncodeFlush` (Apartado 2.2.3.4). Además, se activa la señal de reinicialización de las tablas de contextos.

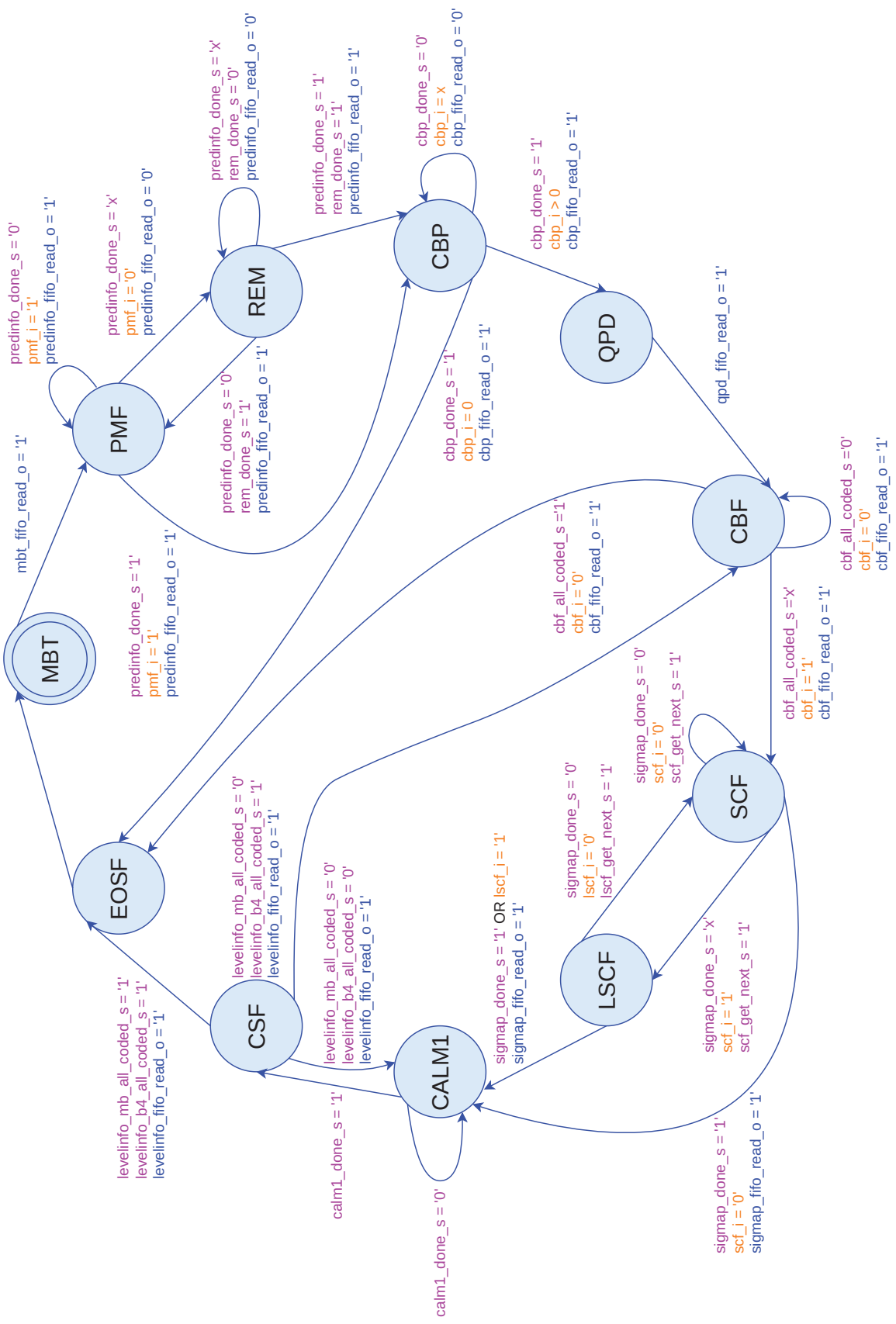

Figura 3.7: Grafo de estados de la MEF de contol de orden de codificación.

3.1.4.2. Binarización

En las Secciones 2.2.1 y 2.3.3 se expusieron, respectivamente, los diferentes tipos de binarización existentes y el empleado para cada SE. De la Tabla 2.7, contenida en la segunda sección mencionada, se deduce que los esquemas de binarización utilizados son:

- *Fixed-Length* (FL).
- *Unary*.
- Códigos predefinidos en tablas.
- *Concatenated Unary/k-th order Exp-Golomb* (UEGk).

Como también se mencionó en la Subsección 2.3.3, los *binstrings* de `mb_type` y `mb_qp_delta` son siempre 0. Además, estos SE son los únicos entre los utilizados que emplean los códigos predefinidos en tablas y *Unary*, respectivamente. Por consiguiente, no ha sido necesario, por un lado, almacenar los *binstrings* relativos a todos los posibles valores de `mb_type`, y por otro lado, tampoco se ha precisado de un codificador específico para *Unary*.

Atendiendo a los esquemas de binarización restantes, cabe señalar que FL se corresponde directamente con el formato en el que se introducen los datos a las FIFO, por lo que no se requiere ninguna operación adicional sobre los SE que son binarizados de esta manera. Sin embargo, para los `coeff_abs_level_minus1` sí ha sido necesario diseñar un módulo UEGk. Para ello, se ha tomado como base lo explicado en el Apartado 2.2.1.3. Partiendo de esto se ha implementado la MEF cuyo grafo de estados simplificado se muestra en la Figura 3.8.

Al igual que para la MEF expuesta en la subsección anterior, las señales con sufijo `_i` representan entradas (*inputs*), las que tienen sufijo `_s` se corresponden con señales internas y las de sufijo `_o` con salidas (*outputs*). Igualmente, estas señales aparecen en el grafo de estados coloreadas en naranja, fucsia y azul, respectivamente. También se incluye la Tabla 3.4 con las definiciones de las señales para facilitar su entendimiento.

Señal	Definición
`calm1_i`	Valor del `coeff_abs_level_minus1` a binarizar.
`pfx_lb_s`	'1' si el *bin* que se encuentra a la salida del módulo se corresponde con el último *bin* del prefijo, '0' en caso contrario.
`pfx_done_s`	'1' cuando ya se han enviado todos los *bins* del prefijo, '0' en caso contrario.
`sffx_lb_s`	'1' si el *bin* que se encuentra a la salida del módulo se corresponde con el último *bin* del sufijo, '0' en caso contrario.
`last_inc_k_s`	'1' si se está realizando el último incremento necesario de `k`, '0' en caso contrario.
`last_dec_k_s`	'1' si se está realizando el último decremento necesario de `k`, '0' en caso contrario.
`bin_o`	*Bin* obtenido. Se actualiza cada ciclo de reloj.
`done_o`	'1' cuando se ha terminado de binarizar el coeficiente en su totalidad, '0' en caso contrario. Nótese que cuando su valor es '1' coincide con el último *bin* del *binstring*.
sffx_o	'1' si el *bin* que se encuentra a la salida del módulo pertenece al sufijo, '0' en caso contrario. Esta señal se utiliza para enviar los *bins* del sufijo a través del *bypass coding engine*.

Tabla 3.4: Señales de la MEF de UEGk.

En cuanto a la MEF de esta sección, su estado inicial es `REPOSO`. En él se evalúa el valor del coeficiente `coeff_abs_level_minus1` a binarizar. Si es cero[7] su *binstring* únicamente está compuesto por un *bin*, por lo que no es necesario cambiar de estado. Por contra, si es distinto de cero puede darse la situación de que su valor sea inferior al valor de corte establecido, `uCoff`, o que sea mayor o igual al mismo. En el primer caso el *binstring* para el SE introducido consta tan solo del prefijo *Truncated Unary* (TU) tal y como se expuso en el Apartado 2.2.1.3. Para el segundo caso, sí son necesarios el mencionado prefijo y también el sufijo construido mediante la adaptación de Exp-Golomb de orden k (EGk) que se propone en el estándar.

En ambas situaciones es necesario cambiar de estado a `SOLO_PFJ` y a `PFJ_SJF`, respectivamente. Sendos estados son necesarios dado que en el primero se implementa la parte de *Unary* y en el segundo la de TU. Es decir, el último *bin* relativo al prefijo

[7]Los `coeff_abs_level_minus1` pueden ser 0 ya que su valor se corresponde con el absoluto del coeficiente original menos una unidad.

que se envía desde el primer estado siempre es '0', mientras que desde el segundo es '1'. Ello puede apreciarse en la transición de `SOLO_PFJ` a `REPOSO` y, por ejemplo, en la de `PFJ_SFJ` a `INC_K`.

Una vez concluido con el prefijo se procede a la binarización del sufijo mediante la implementación del Algoritmo 1 incluido en el Apartado 2.2.1.3. La codificación del sufijo que se describe en dicho algoritmo puede subdividirse en dos partes: incremento y decremento del orden, `k`. Por tanto, se han diseñado dos estados, `INC_K` y `DEC_K`, asociados, respectivamente, a cada una de estas subdivisiones. Cabe hacer hincapié en que en la parte de incremento de `k` siempre se generan *bins* con polaridad '1'. Sin embargo, en la otra parte el primer *bin* siempre se corresponde con un '0' mientras que los que le siguen pueden ser indistintamente '1' ó '0' dependiendo de `k` y del propio coeficiente, de ahí que en el grafo de estados aparezca con valor 'X' para este caso. Finalmente, cuando se concluye con el decremento de `k`, la MEF retorna a su estado inicial.

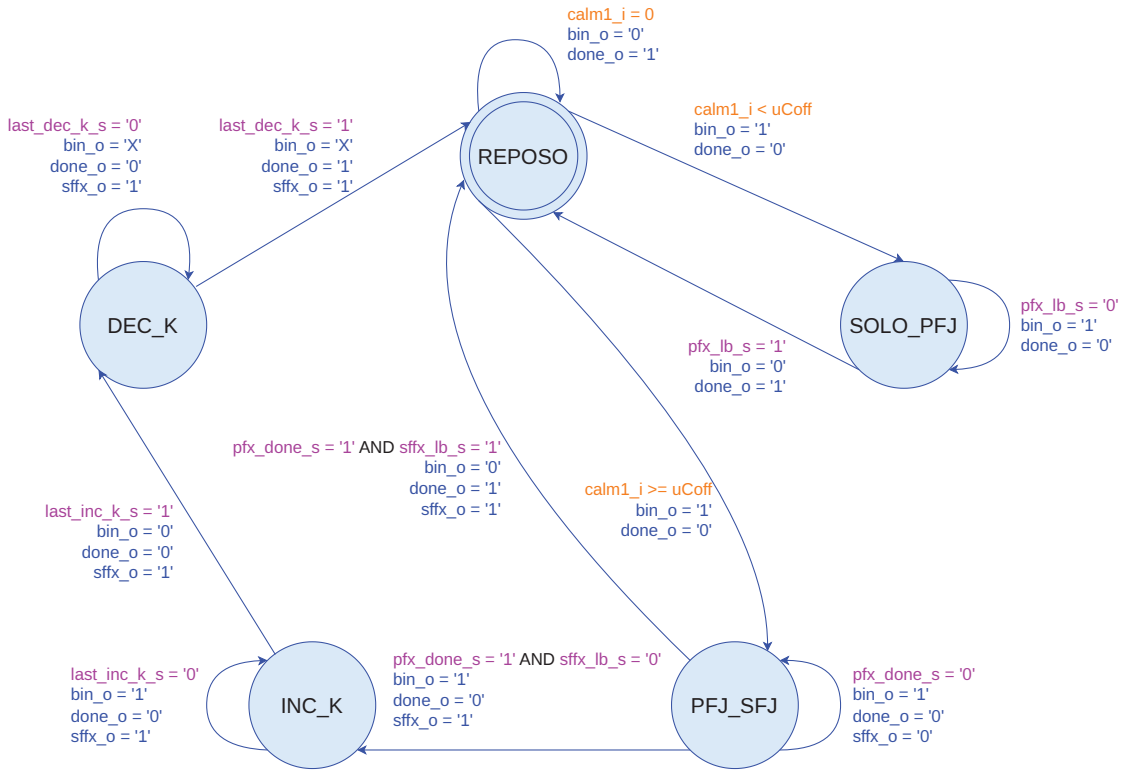

Figura 3.8: Grafo de estados de la MEF de UEGk.

3.1.5. Codificador Binario Aritmético

Como puede deducirse en la Subsección 2.2.3, el módulo *BAC* implementa los siguientes procesos ya descritos en ella: `EncodeDecision(ctxIdx,binVal)`, `RenormE`, `EncodeBypass(binVal)`, `EncodeTerminate(binVal)`, `EncodeFlush` y genera las órdenes de escritura que posteriormente realiza el escritor de bits. No obstante, cabe señalar que el primero de estos procesos sufre una ligera modificación, pues la actualización de contextos es realizada por el binarizador como ya se mencionó en el apartado anterior.

Por consiguiente, la interfaz del BAC está compuesta por las señales de la Tabla 3.5. En ella se refleja que el codificador aritmético cuenta desde el principio con la información necesaria para realizar la codificación. Es decir, este no debe de determinar si el *bin* es el símbolo más probable (del inglés *Most Probable Symbol*) (MPS), o si debe de codificarse con el *bypass coding engine*. Tampoco debe de determinar el valor de `pStateIdx` necesario para la obtención de la fila correcta de la tabla `rangeTabLPS`. Además, no ha de deducir si el *bin* que tiene a su entrada está relacionado con `EncodeTerminate`.

Por otro lado, en cuanto a la propia implementación de los procesos mencionados previamente, estos se han realizado de la siguiente manera:

- `EncodeDecision(ctxIdx,binVal)`: mediante una MEF.
- `EncodeBypass(binVal)`: íntegramente en lógica combinacional ya que no necesita iteraciones y por tanto puede realizarse en un ciclo de reloj.
- `EncodeTerminate(binVal)`: mediante una MEF.
- `RenormE`: está integrado en las MEF de `EncodeDecision(ctxIdx,binVal)` y `EncodeTerminate(binVal)` pues son los procesos que lo utilizan.
- `EncodeFlush`: también forma parte de la MEF de `EncodeTerminate(binVal)`.

Señal	Tipo	Descripción
clk_i	Entrada	Reloj del sistema.
rst_n_i		Reset activo a nivel bajo. Puede ser síncrono o asíncrono según se configure el módulo.
valid_i		'1' cuando los datos son válidos, '0' en caso contrario.
bin_i		*Bin* a codificar.
mps_i		'1' si el *bin* introducido se corresponde con el MPS para su contexto, '0' en caso contrario.
bp_i		'1' si el *bin* introducido ha de codificarse mediante el *bypass coding engine*, '0' en caso contrario.
rlps_row_aii		Fila de la tabla `rangeTabLPS` asociada al `pStateIdx` relativo al contexto del *bin* introducido.
ett_i		'1' si el *bin* introducido está relacionado con el proceso `EncodeTerminate`, '0' en caso contrario.
ready_o	Salida	'1' cuando el BAC puede recibir datos nuevos, '0' en caso contrario.
valid_o		'1' cuando los datos a la salida del BAC son válidos, '0' en caso contrario.
write_bits_ro		Orden de escritura a ser ejecutada por el escritor de bits.
ett_done_o		'1' cuando se ha concluido la codificación del *bin* asociado a `EncodeTerminate`, '0' en caso contrario.

Tabla 3.5: Descripción de la interfaz del BAC.

Las MEF asociadas a `EncodeDecision` y `EncodeBypass` resultan bastante sencillas, pues las operaciones descritas en las Figuras 2.13 y 2.15 son fáciles de implementar. Por consiguiente, no es necesario detenerese a describirlas. Sin embargo, la tercera sí cuenta con un cierto grado de complejidad. Por tanto, en la Figura 3.9 se incluye el grafo de estados de esta MEF. Además, en la Tabla 3.6 se describen las señales utilizadas por dicha MEF. Nótese que las señales con sufijo `_i` se corresponden con entradas (*inputs*), las de sufijo `_s` con señales internas y las de sufijo `_o` son salidas (*outputs*). Nuevamente, estas señales aparecen coloreadas en el grafo de estados en naranja, fucsia y azul, respectivamente.

Señal	Descripción
bin_i	*Bin* a codificar.
more_iterations_s	'1' si se requieren más iteraciones de RenormE, '0' en caso contario.
flush_rn_done_s	'1' si el RenormE asociado a EncodeFlush ha terminado, '0' en caso contrario.
flush_s	'1' si la orden de escritura generada está asociada a EncodeFlush, '0' en caso contrario.
ett_done_o	'1' cuando se ha concluido la codificación del *bin* asociado a EncodeTerminate, '0' en caso contrario.

Tabla 3.6: Señales de la MEF de EncodeTerminate(binVal).

La MEF en cuestión consta de un total de cinco estados: ETT, ETT_RN, FLUSH, FLUSH_RN y FLUSH_WR. Los estados con sufijo _RN hacen referencia a RenormE. Ello se debe a que en cada estado de esta MEF, salvo en el último, se realiza una primera iteración de RenormE y, en caso de que se precisen más iteraciones para ajustar las variables internas del BAC, codILow y codIRange, se transiciona a estos estados según corresponda. Cabe recordar que, como ya se ha mencionado en varias ocasiones, la mayoría de las órdenes de escritura se generan durante los procesos de renormalización. Por tanto, al no conocerse la salida exacta en cada estado, salvo en dos transiciones concretas, se omite en el diagrama la señal asociada a estas órdenes.

El estado inicial es ETT. En este primer estado se evalúa si ha de desempeñarse el proceso de EncodeFlush. Ello depende únicamente del valor del *bin* introducido, pues si esta MEF es la que toma el control de la salida se sabe que dicho *bin* indica el fin de cada MB. Ahora, de ser '1' su valor, se sabe que además es el último del *slice*. En caso de no ser así, simplemente se realiza el proceso de renormalización, el cual puede requerir de una o varias iteraciones. Dependiendo de ello puede ser necesario transicionar a ETT_RN.

Con FLUSH y FLUSH_RN se tiene una situación idéntica. Sin embargo, una vez es completada la renormalización relativa a FLUSH, se crea en primer lugar la orden de escritura asociada a PutBit(codILow $\gg$ 9) & 1 y se transiciona a FLUSH_WR. A continuación, se genera otra orden, en esta ocasión asociada a WriteBits(((codILow

≫ 7) & 3) | 1 , 2) indicándose que es la última orden de escritura creada durante una operación de `EncodeFlush` (`flush_s = '1'`). Finalmente se regresa a `ETT`, pues al ser únicamente una orden de escritura conocida de antemano no es necesario que la MEF permanezca en `FLUSH_WR` por más de un ciclo.

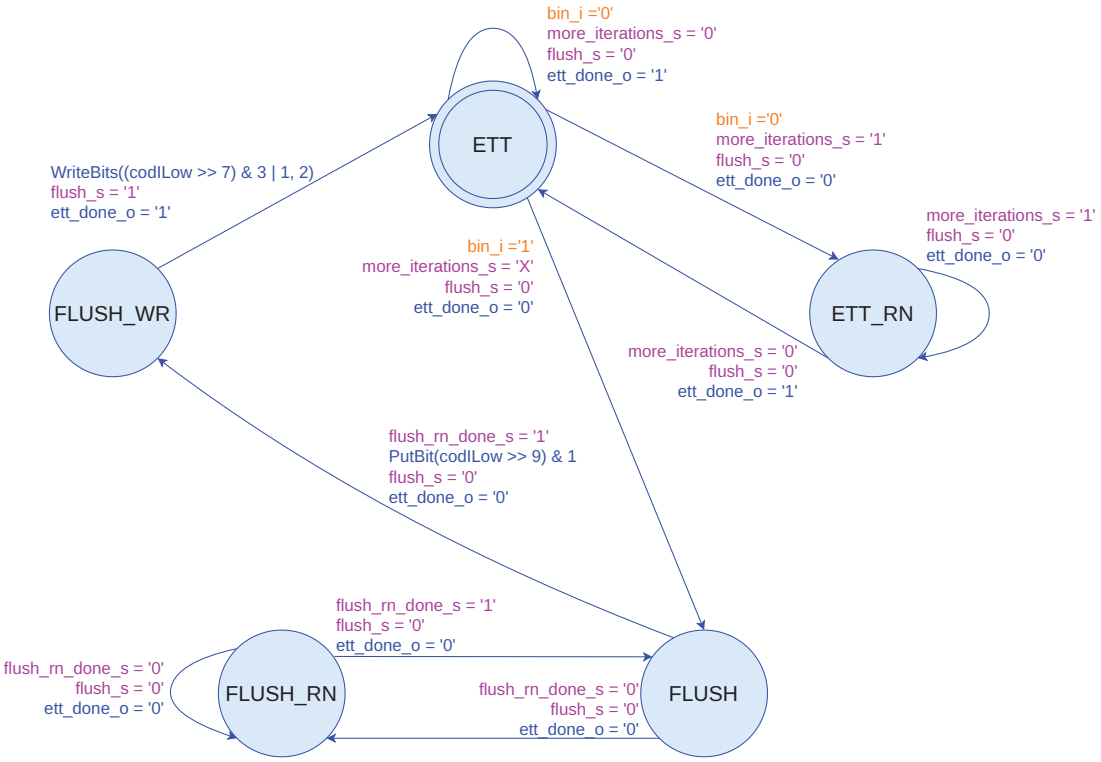

Figura 3.9: Grafo de estados de la MEF de `EncodeTerminate(binVal)`.

3.1.6. Escritor de bits

El escritor de bits se corresponde con las implementaciones de los procesos `PutBit(B)` y `WriteBits(B,N)` mencionados en el Apartado 2.2.3.4. Recordando lo explicado en las secciones citadas, `WriteBits(B,N)` es el único proceso que escribe directamente sobre el *bitstream*. Por consiguiente, `PutBit(B)` lo invoca cuando es necesario realizar escrituras en el modo de funcionamiento normal. Sin embargo, `WriteBits(B,N)` también puede ser invocado directamente por `EncodeFlush` (ver Figura 2.18).

Por tanto, además de conocer la polaridad de los bits a escribir en el *bitstream* (`B`) y el número de ellos (`N`), también es preciso señalar si están asociados a `EncodeFlush`. En caso de que efectivamente lo estén, `WriteBits(B,N)` escribirá los dos bits que se le

indiquen directamente en el *bitstream*.

Para ello, este módulo cuenta con una entrada de datos tipo `record`, `write_bits_ri`, que contiene la siguiente información:

- `nr`: número de bits que han de ser escritos al bitstream (equivalente a `N`).
- `flush`: su valor es '1' si la instrucción de escritura a la que pertenece está relacionada con `EncodeFlush`, '0' en caso contrario.
- `val`: vector de dos bits que ha de interpretarse de la siguiente manera según corresponda:
  - Modo de funcionamiento normal (equivalente a `PutBit(B)`): en este modo, el bit más significativo (del inglés *Most Significant Bit*) (MSB) se corresponde con el inverso de `firstBitFlag`. En caso de que dicho MSB sea '1', se realizarán dos escrituras al *bitstream*. La primera de ellas consiste en escribir un único bit con la polaridad indicada por bit menos significativo (del inglés *Least Significant Bit*) (LSB), mientras que en la segunda se escriben `nr` bits con la polaridad opuesta a la indicada.

    Sea el siguiente ejemplo en el que para una orden de escritura dada se genera una serie de bits, `resultado`, a escribir en el *bitstream*:

    ```
    nr    = 10   ┐
    flush = '0'  ├→ resultado = '100 0000 0000'
    val   = '11' ┘
    ```

  - Operación de escritura relacionada con `EncodeFlush`: como se comentaba anteriormente, en este modo simplemente se añaden los bits indicados por `val` al *bitstream*. Nótese que ahora el valor de `nr` resulta indiferente:

    ```
    nr    = 10   ┐
    flush = '1'  ├→ resultado = '11'
    val   = '11' ┘
    ```

En cuanto a la salida del escritor de bits, para agilizarla se utilizan palabras de 8 bits en lugar de realizar una escritura bit a bit. Asimismo, cabe señalar que cuando se procesa una orden de escritura relativa a `EncodeFlush`, ha de introducirse un *padding* o relleno en los bits restantes para completar aquella palabra en la que se han escrito los últimos bits. Dicho *padding* es equivalente a asignarle el valor '0' a dichos bits. Retomando el ejemplo anterior y suponiendo que `resultado` se ha escrito sobre una palabra nueva, `palabra`, se obtiene `palabra_final` con su respctivo *padding*:

$$\left.\begin{array}{l}\texttt{resultado} = \text{`}11\text{'} \\ \texttt{palabra} = \text{`}11UU\ UUUU\text{'}\end{array}\right\} \rightarrow \texttt{palabra\_final} = \text{`}1100\ 0000\text{'}$$

La funcionalidad de este módulo se ha implementado por medio de la MEF cuyo grafo de estados simplificado es el de la Figura 3.10. En ella aparecen varios tipos de señales, siendo las de sufijo `_i` entradas (*inputs*), las de sufijo `_s` señales internas y las de sufijo `_o` son salidas (*outputs*). Al igual que para los grafos de estados anteriores, dichas señales aparecen coloreadas en naranja, fucsia y azul, respectivamente.

La MEF utiliza un total de tres estados: `PRIMERA`, `RESTO` y `FLUSH`. El primer estado se utiliza para la primera escritura resultante de procesar una nueva orden de escritura. En caso de que la orden de escritura mencionada pueda realizarse completamente sobre la palabra de 8 bits en la cual se está escribiendo (`done_s = '1'`), el estado seguirá siendo el mismo. Además, con `read_o = '1'` se recupera la siguiente orden de escritura pendiente en la FIFO de entrada de este subsistema.

Si no es posible completar la orden sobre la palabra actual (`done_s = '0'`), se transicionará a `FLUSH` o a `RESTO` según el valor de `flush_i`. Cabe señalar que no existe ninguna condición por la que la MEF deba permanecer más de un ciclo en el estado `FLUSH`. Sin embargo, esto sí ocurre con `RESTO`, pues en él se realizan el resto, valga la redundancia, de escrituras para terminar de cumplir con la orden. Este estado resulta necesario, ya que a diferencia de en `PRIMERA`, en `RESTO` simplemente se escriben los bits requeridos hasta llegar a `nr`. Es decir, no se comprueba el MSB de `val`, pues esto siempre es previamente verificado en `PRIMERA`.

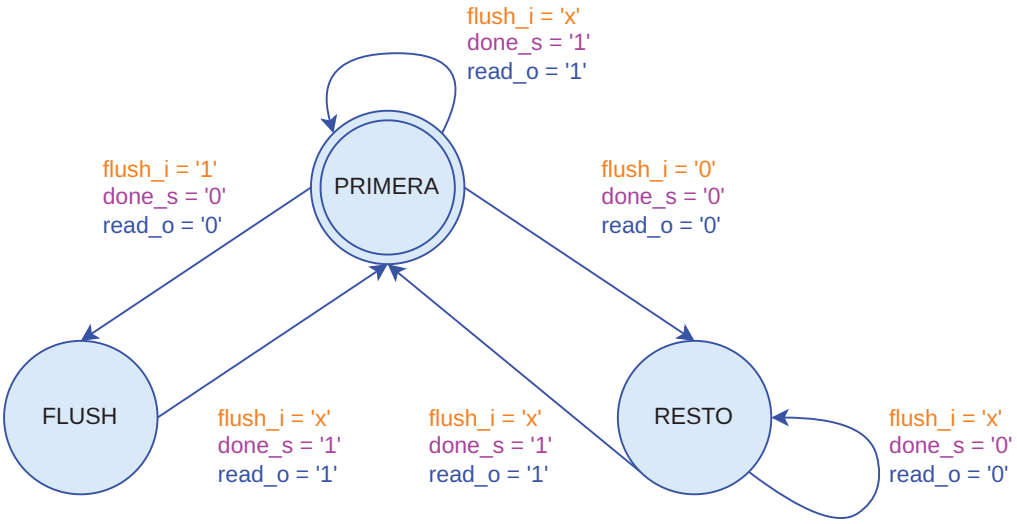

Figura 3.10: Grafo de estados de la MEF del escritor de bits.

3.2. Verificación de CABAC

Dada la complejidad y extensión del diseño, la estrategia de verificación seguida consistió, primeramente, en la verificación funcional con Vivado™ de cada módulo por separado mediante *testbenches*. Estos *testbenches* no fueron exhaustivos en general, pues para algunos módulos como *BAC* o el *Binarizador* no tenía sentido realizar una verificación funcional por separado. Es decir, continuando con los ejemplos mencionados, resulta más eficiente llevar a cabo una simulación de varios submódulos donde no sea necesario generar los datos de entrada manualmente. Obviamente, para ello es necesario cerciorarse de que los módulos *Intérprete de SE residuales* y *Unidad de control de salida* funcionan correctamente. En el caso de estos dos últimos, sí se pudo realizar una verificación funcional con un mayor grado de detalle.

En resumen, puede decirse que la estrategia de verificación consistió en una primera serie de *testbenches* individuales para cada módulo, y, con la creación de distintos módulos, se fueron realizando simulaciones en conjunto. En las siguientes secciones se desarrolla el procedimiento seguido así como los programas utilizados para poder realizar simulaciones que incluyesen el diseño en su totalidad.

3.2.1. JM 19.0

JM 19.0 es el *software* de referencia para la codificación y decodificación de vídeo según el estándar H.264/14496-10. Este programa fue desarrollado por los creadores de dicha norma: *Joint Video Team* (JVT), una alianza estratégica entre el grupo de expertos en imágenes en movimiento (del inglés *Moving Picture Experts Group*) (MPEG) de la ISO/IEC, y el grupo de expertos en codificación de vídeo (del inglés *Video Coding Experts Group*) (VCEG) de la ITU-T.

El código fuente está escrito en C y puede hallarse tanto en GitHub como en GitLab. Este *software* facilita significativamente la comprensión del estándar, especialmente si se combina con herramientas como gdbgui, cuyo uso se comenta más adelante.

En el manual de usuario del programa [20] se explican detenidamente todos y cada uno de los parámetros de configuración que contiene este *software*. Esto resulta crucial de cara a la generación de archivos de configuración del programa para su uso. No obstante, cabe recalcar que los archivos de configuración utilizados en este proyecto han sido generados por una herramienta en Python desarrollada internamente por la división de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA), lo cual facilita y agiliza este proceso considerablemente.

3.2.1.1. Funciones de interés

El código fuente de JM 19.0 es bastante extenso y cuenta con un gran número de funciones, estructuras de datos, variables globales, constantes, etc. Del total de funciones existentes únicamente están relacionadas con el desarrollo de este proyecto un reducido conjunto de ellas (ver Tabla 3.7).

Además de conocer las funciones a partir de las cuales puede obtenerse la información deseada es preciso saber cómo se relacionan entre sí, especialmente para facilitar la comprensión del algoritmo. Para ello ha sido fundamental el uso de Doxygen con Graphviz para generar diagramas de llamadas como el que se muestra en la Figura 3.11 para `writeMB_I_typeInfo_CABAC`. No obstante, ha de recalcarse que estos grafos fueron complementarios al uso de otras herramientas como gdbgui.

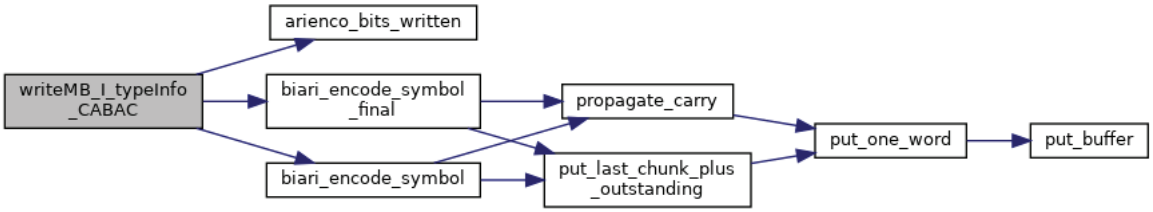

Figura 3.11: Grafo de llamadas de `write_significant_coefficients`.

Archivo	Función	Información extraída
cabac.c	writeMB_I_typeInfo_CABAC	mb_type.
	writeIntraPredMode_CABAC	prev_intra4x4_pred_mode_flag y rem_intra4x4_pred_mode.
	writeCBP_CABAC	coded_block_pattern.
	writeCBP_BIT_CABAC	condTermFlagA y condTermFlagB de coded_block_pattern.
	writeDquant_CABAC	mb_qp_delta.
	writeRunLevel_CABAC	coded_block_flag.
	write_and_store_CBP_block_bit	condTermFlagA y condTermFlagB de coded_block_flag.
	write_significance_map	significant_coeff_flag y last_significant_coeff_flag.
	write_significant_coefficients	coeff_abs_level_minus1 y coeff_sign_flag.
macroblock.c	write_terminating_bit	end_of_slice_flag.
biariencode.c	biari_encode_symbol	*Bins* codificados mediante el *regular coding engine* y estado interno del BAC.
	biari_encode_symbol_eq_prob	*Bins* codificados mediante el *bypass coding engine* y estado interno del BAC.
	biari_encode_symbol_final	*Bins* asociados a EncodeTerminate y a EncodeFlush y estado interno del BAC.
	put_buffer	Bits a escribir directamente en el *bitstream*.
	put_one_bit_final	

Tabla 3.7: Funciones de interés de JM 19.0. Adaptada de [8].

3.2.1.2. Uso de gdbgui

El programa gdbgui[8] no es más que una interfaz gráfica para el depurador de GNU, gdb. En las primeras fases de la verificación y de desarrollo gdbgui supuso una gran ayuda, pues en ocasiones los grafos de llamadas generados por Doxygen y Graphviz aparecen incompletos o son demasiado enrevesados. Dicha ausencia de elementos es debida en parte a que en JM 19.0 se utilizan subconjuntos de funciones dependiendo de la configuración, por lo que es común ver estructuras de datos con punteros a las funciones correspondientes. Esto parece ocasionar problemas a las herramientas mencionadas previamente.

Por consiguiente, este programa fue empleado con el fin de obtener un mayor grado de profundización y entendimiento del código del *software* de referencia, pues debido a la extensión y complejidad del mismo es habitual perderse. En la Figura 3.12 se muestra un fotograma del vídeo tomado como referencia para el análisis de la función `writeMB_I_typeInfo_CABAC` expuesto en la Figura 3.13. En la parte superior de esta segunda Figura aparece el nombre del archivo de configuración empleado. De este se deduce que el vídeo `crop_vid_moon_flyby_close` cuenta con resolución 96x80.

Por otro lado, en dicha Figura se aprecia una modificación del código original que abarca las líneas 781-785. Esta modificación en concreto es utilizada para obtener el estado interno del codificador aritmético tras la codificación de, en este caso, el SE `mb_type`. La extracción de estados internos de dicho módulo se aborda más adelante en el Apartado 3.2.1.4.

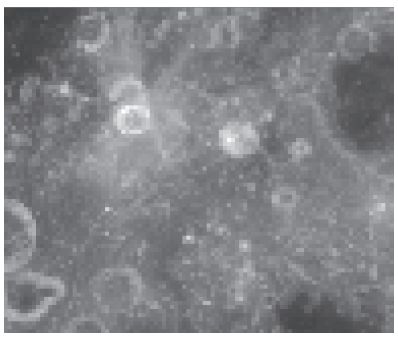

Figura 3.12: Fotograma de `crop_vid_moon_flyby_close`.

[8]Enlace a la página de gdbgui: https://www.gdbgui.com/

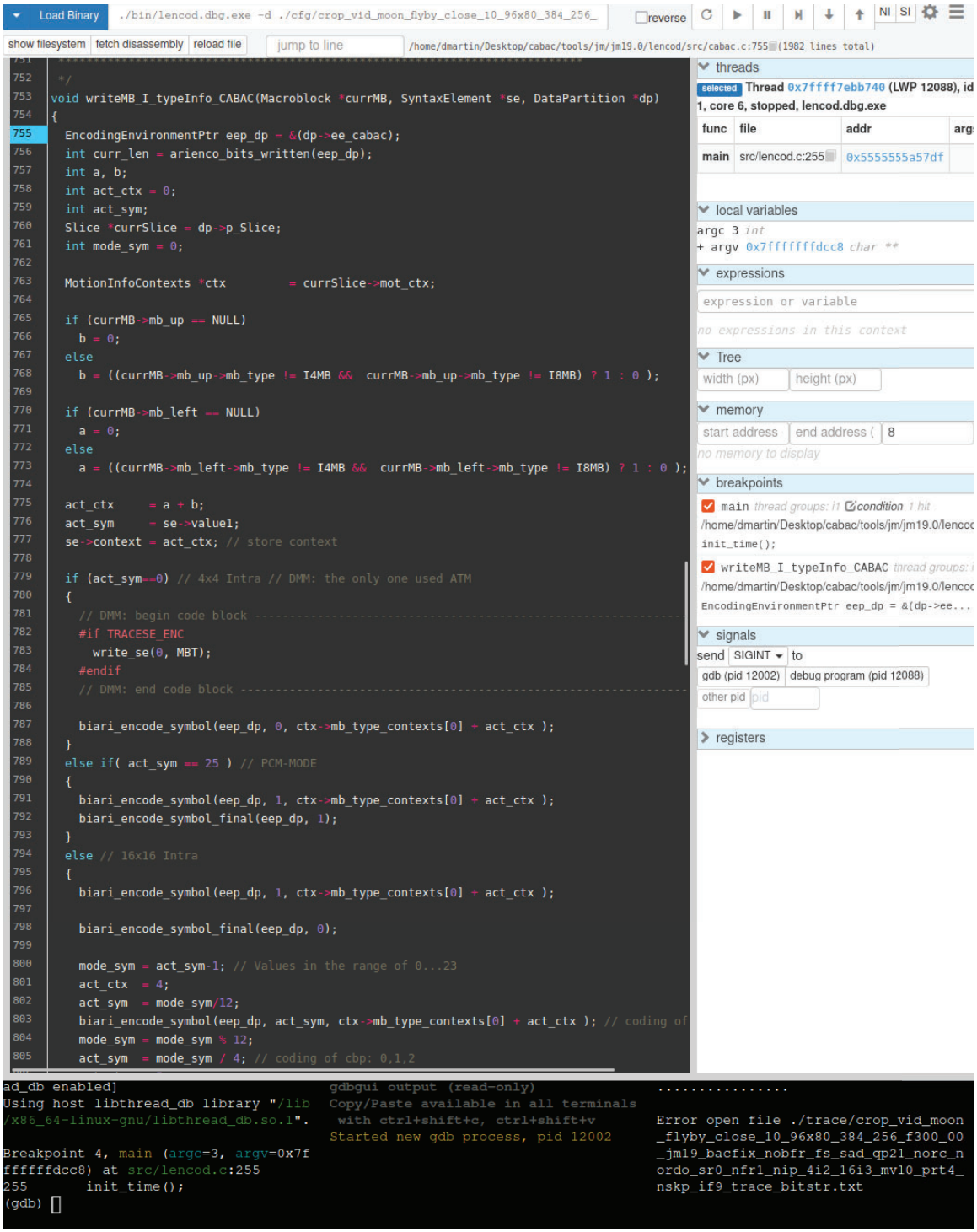

Figura 3.13: Ejemplo de uso de gdbgui para el análisis de JM 19.0.

3.2.1.3. Obtención de trazas de referencia

Además del tipo de modificación del código fuente de JM 19.0 mencionado en el apartado anterior, también se realizaron otras modificaciones para obtener las trazas de referencia de CABAC. Primeramente, cabe destacar que este *software* cuenta con diversas macros de configuración entre las cuales se halla `TRACE`. Esta macro habilita la generación de un fichero `trace_bitstr.txt` en el que se incluye información superficial relativa a la codificación de CABAC. En la Figura 3.14 se muestra un fragmento de uno de estos archivos.

En dicho fragmento se observa que JM 19.0 indica los números de fotograma, MB y *slice*. Además, los números precedidos de '@' se corresponden con el número de bits que se han escrito hasta el momento[9], aunque como se aprecia, no se incluyen los bits en sí. También aparece entre paréntesis el valor del símbolo que se está codificando, al igual que se menciona el SE al que pertenece o su tipo. Concretamente, se puede ver que `mb_type`, `coded_block_pattern` y `mb_qp_delta` sí aparecen explícitamente, mientras que a `prev_intra4x4_pred_mode_flag` y `rem_intra4x4_pred_mode` se les hace alusión mediante "Intra 4x4 mode". Por otro lado, los SE residuales son referenciados mediante "Luma4x4 sgn".

En resumen, este tipo de archivo no resulta de gran ayuda tanto para el desarrollo como para la verificación del diseño, ya que no provee ni las trazas de referencia ni los estados internos del codificador aritmético. No obstante, cabe señalar que JM 19.0 sí proporciona el *bitstream* en otro tipo de archivo: `.264`. Sin embargo, los `.264` incluyen, además de los bits generados por CABAC, aquellos asociados a las cabeceras. En otras palabras, estos archivos no se corresponden con el resultado puro de la codificación de CABAC, sino que contienen toda la información necesaria para poder decodificar el vídeo.

[9]El número de bits escritos es distinto de 0 al momento de empezar a codificar el vídeo en sí. Esto se debe a que en estos archivos también se incluye información acerca de la configuración del propio vídeo, por lo que en este punto ya se han generado algunos bits.

```
 69  ********** Pic: 0 (I/P) MB: 0 Slice: 0 **********
 70
 71  @97    mb_type (I_SLICE) ( 0, 0) =   9 (  0)
 72  @97    Intra 4x4 mode  = predicted (context: 0) ( -1)
 73  @98    Intra 4x4 mode  = predicted (context: 1) ( -1)
 74  @99    Intra 4x4 mode  =   0 (context: 2) (  0)
 75  @103   Intra 4x4 mode  =   0 (context: 3) (  0)
 76  @107   Intra 4x4 mode  = predicted (context: 4) ( -1)
 77  @108   Intra 4x4 mode  = predicted (context: 5) ( -1)
 78  @109   Intra 4x4 mode  =   1 (context: 6) (  1)
 79  @113   Intra 4x4 mode  = predicted (context: 7) ( -1)
 80  @114   Intra 4x4 mode  = predicted (context: 8) ( -1)
 81  @115   Intra 4x4 mode  = predicted (context: 9) ( -1)
 82  @116   Intra 4x4 mode  = predicted (context: 10) ( -1)
 83  @117   Intra 4x4 mode  = predicted (context: 11) ( -1)
 84  @118   Intra 4x4 mode  =   1 (context: 12) (  1)
 85  @122   Intra 4x4 mode  =   0 (context: 13) (  0)
 86  @126   Intra 4x4 mode  =   0 (context: 14) (  0)
 87  @129   Intra 4x4 mode  = predicted (context: 15) ( -1)
 88  @130   CBP ( 0, 0) =  15 ( 15)
 89  @130   Delta QP ( 0, 0) =   0 (  0)
 90  @130   Luma4x4 sng( 0) level =-31 run = 0 (-31)
 91  @130   Luma4x4 sng( 1) level = -1 run = 0 ( -1)
 92  @130   Luma4x4 sng( 2) level = -2 run = 1 ( -2)
 93  @130   Luma4x4 sng( 3) level = -1 run = 2 ( -1)
 94  @130   Luma4x4 sng( 4) level =  1 run = 0 (  1)
 95  @130   Luma4x4 sng( 5) level =  1 run = 0 (  1)
 96  @130   Luma4x4 sng( 6) level =  2 run = 1 (  2)
 97  @130   Luma4x4 sng( 7) level =  1 run = 2 (  1)
 98  @130   Luma4x4 sng( 8) level = -1 run = 0 ( -1)
 99  @130   Luma4x4 sng( 9) level = -1 run = 0 ( -1)
100  @130   Luma4x4 sng(10) level =  0 run = 0 (  0)
101  @186   Luma4x4 sng( 0) level = -2 run = 0 ( -2)
102  @186   Luma4x4 sng( 1) level = -1 run =10 ( -1)
103  @186   Luma4x4 sng( 2) level =  0 run = 0 (  0)
```

Figura 3.14: Ejemplo de archivo `trace_bitstr.txt`.

La solución adoptada para generar las trazas de referencia consistió en aprovechar la ya mencionada macro de configuración `TRACE`. Para ello, fue necesario un detenido estudio del código fuente de JM 19.0 para, primeramente, determinar qué parte del mismo se encargaba de esta escritura, y seguidamente averiguar cómo referenciar al archivo en el que se volcarían los datos. Por simplicidad, se decidió aprovechar el propio `trace_bitstr.txt`, siendo el código necesario para ello el de la Figura 3.15 y el resultado el de la Figura 3.16.

En cuanto a la Figura 3.15, cabe hacer hincapié que las funciones modificadas ya habían sido previamente mencionadas en la Tabla 3.7. Atendiendo a la Figura 3.16, en ella se pueden ver los bits obtenidos a causa de la modificación introducida. Sin embargo, al estar mezclados con el resto de información del archivo, no resulta un

buen fichero para posteriormente comparar las salidas generadas por el diseño y la referencia. Por consiguiente, fue necesario crear un *script* en Python (`cabac_trace.py`) que formatea los archivos `trace_bit_str.txt` con trazas de CABAC con el fin de facilitar la comparación entre las salidas obtenidas en verificación. En la Figura 3.17 se expone un ejemplo del resultado final.

```
70  static inline void put_one_byte_final(EncodingEnvironmentPtr eep,
71                                        unsigned int b)
72  {
73    eep->Ecodestrm[(*eep->Ecodestrm_len)++] = (byte) b;
74      // DMM: begin code block ----------------------------------------------------
75      #if TRACE
76        byte byte_written = 0;
77        // note that byte type is actually an unsigned char
78        byte bit = 0;
79        byte_written = eep->Ecodestrm[(*eep->Ecodestrm_len)-1];
80        for(int i=7; i>=0; i--){
81          bit = (byte_written>>i) & 0x1;
82          // +48 to get ASCII representation of its value
83          putc(bit + 48, p_Enc->p_trace);
84        }
85        putc('\n',p_Enc->p_trace);
86      #endif
87    // DMM: end code block ------------------------------------------------------
88  }
89
90  static forceinline void put_buffer(EncodingEnvironmentPtr eep)
91  {
92    // DMM
93    byte byte_written = 0;
94    // DMM: note that byte type is actually an unsigned char
95    byte bit = 0;
96
97    while(eep->Epbuf>=0)
98    {
99      eep->Ecodestrm[(*eep->Ecodestrm_len)++] = (byte)(
100       (eep->Ebuffer>>((eep->Epbuf--)<<3))&0xFF
101     );
102     // DMM: begin code block ----------------------------------------------------
103     #if TRACE
104       byte_written = eep->Ecodestrm[(*eep->Ecodestrm_len)-1];
105       for(int i=7; i>=0; i--){
106         bit = (byte_written>>i) & 0x1;
107         // +48 to get ASCII representation of its value
108         putc(bit + 48, p_Enc->p_trace);
109       }
110       putc('\n',p_Enc->p_trace);
111     #endif
112     // DMM: end code block ------------------------------------------------------
113   }
114   while(eep->C > 7)
115   {
116     eep->C-=8;
117     ++(eep->E);
118   }
119   eep->Ebuffer = 0;
120 }
```

Figura 3.15: Modificación del código fuente de JM 19.0 para la obtención de trazas de referencia.

```
 98  @130   Luma4x4 sng( 8) level = -1 run = 0 ( -1)
 99  @130   Luma4x4 sng( 9) level = -1 run = 0 ( -1)
100  11100011
101  00100101
102  11110100
103  11000100
104  10110011
105  01010100
106  00001001
107  01111011
108  @130   Luma4x4 sng(10) level =  0 run = 0 (  0)
109  @186   Luma4x4 sng( 0) level = -2 run = 0 ( -2)
110  @186   Luma4x4 sng( 1) level = -1 run =10 ( -1)
```

Figura 3.16: Archivo `trace_bitstr.txt` con trazas de referencia.

```
 1  # This file was automatically generated by cabac_trace.py from
 2  # crop_vid_moon_flyby_close_10_96x80_384_256_f300_00_jm19_bacfix
 3  # _intra_sad_qp21_norc_nordo_nfr300_4i2_16i3_uv00_trace_bitstr.txt
 4
 5  11100011 00100101 11110100 11000100 10110011 01010100
 6  00001001 01111011 10111011 01010010 01100101 10100001
 7  11111111 11110001 10011101 01010100 11111001 10000000
 8  10001101 10011111 01011111 00100001 11010111 11001010
 9  00001010 00101001 10111100 10110010 11110101 01000110
10  00110111 10000100 00110100 10000010 00001100 00110110
11  11101100 10111101 01110111 01011011 10100110 11111100
12  10100010 10110110 11001101 11000001 01110000 01111100
13  01011010 01010111 11010001 00000101 10100111 11111111
14  00001111 00100011 10110010 11110010 00110000 10011100
15  11000110 11001001 10010001 11010111 00100000 10101111
16  11000101 01110111 11110010 11001110 11110001 10011011
17  01111101 01001100 10000001 10110110 11110000 10011100
18  00001100 11010011 10100001 00111111 01000010 10001110
19  01010001 01101011 00010011 01100110 00100000 00111011
20  01110010 00111001 11100100 00110001 10010010 11110110
21  11010111 10010000 10010110 00011110 01000001 00100011
22  01110010 00011001 01100100 10010011 10000011 10111001
23  00000111 10011010 11000010 01101100 10101101 01001110
24  00011110 00010111 00101100 11000101 00011000 11000101
25  01101010 01010101 00110010 01000001 01001000 00001110
26  10100011 00111000 10111101 11010100 11011001 10100101
27  10001001 00101000 01110100 01100111 01101100 00111010
```

Figura 3.17: Archivo `cabac_bitstr.txt` con trazas de referencia formateadas.

3.2.1.4. Obtención de estados internos del Codificador Binario Aritmético

Al igual que en la subsección anterior, en esta también fue preciso un detenido análisis del código fuente del *software* de referencia. Tras haber ubicado las funciones de interés, ya recogidas en la Tabla 3.7, se creó un nuevo fichero de funciones, `trace_se.c`, a partir de uno ya existente desarrollado por la división DSI del IUMA, `trace_jm.c`. Este segundo archivo, del cual se comentan más detalles en la siguiente subsección, es imprescindible para la simulación con Vivado™. De `trace_jm.c` se aprovecharon las funciones de abrir y cerrar un nuevo fichero `.txt` en el cual se escriben los datos de interés.

En cuanto a `trace_se.c`, las funciones desarrolladas son las siguientes:

- `start_trace_se`: crea el fichero de salida y habilita la escritura en el mismo (adaptado de `trace_jm.c`).
- `end_trace_se`: cierra el fichero de salida una vez se ha terminado de escribir toda la información (adaptado de `trace_jm.c`).
- `write_se`: escribe en el fichero de salida el valor que se está codificando y el SE al que pertenece.
- `write_ctf`: escribe en el fichero de salida los valores de los `condTermFlagN` de `coded_block_pattern` y `coded_block_flag`.
- `write_bac_state`: escribe en el fichero de salida el estado interno del BAC tras la codificación de cada símbolo por el *regular coding engine*.
- `write_bac_state_eqp`: escribe en el fichero de salida el estado interno del BAC tras la codificación de cada símbolo por el *bypass coding engine*.
- `write_mb_header`: escribe en el fichero de salida los delimitadores de MB donde también se indica el índice de MB, *slice* y fotograma.

Invocando a estas funciones desde las de la Tabla 3.7 según corresponda, los archivos `trace_se.txt` tienen el aspecto mostrado en la Figura 3.18. Cabe señalar que, como se

menciona en dicha figura, se ha mantenido la filosofía de JM 19.0, pues la generación de este tipo de archivo se habilita por medio de una nueva macro de configuración, `TRACE_SE`.

Asimismo, en esta imagen se indica el orden en el que se muestran los datos, apareciendo en primer lugar el número de fotograma, coordenadas del MB y *slice* al que estos pertenecen. A continuación, para cada SE existe una abreviación, mostrándose en este caso `mbt` para `mb_type`, `pred` para `prev_intra4x4_pred_mode_flag` y `rem` para `rem_intra4x4_pred_mode`. Al lado del nombre de cada SE aparece el valor que se está codificando y en la línea siguiente, de izquierda a derecha, se tiene: valor del *bin* que se codificó y `codIRange`, `pStateIdx` y `valMPS` tras la codificación de cicho *bin*. Además, como se comentó anteriormente, para `coded_block_flag` y `coded_block_pattern` también se incluyen los valores de los `condTermFlagN`.

```
1  # This file has been automatically generated by enabling TRACE_SE.
2  # Here you will be able to find the state of the Binary Arithmetic
3  # Coder after each bin has been coded. Information is displayed as
4  # follows:
5  # syntax_element: symbol_val
6  # coded_bin codIRange pStateIdx valMPS
7  # Any doubt contact dmartin@iuma.ulpgc.es fmachado@iuma.ulpgc.es
8
9
10 ********** Pic: 0 (I/P) MB (y, x): (0, 0) Slice: 0 **********
11 mbt  : 0
12 0 494 53 0
13
14 pred : 1
15 1 370  4 0
16
17 pred : 1
18 1 284  2 0
19
20 pred : 0
21 0 312  3 0
22
23 rem  : 0
24 0 256  0 1
25 0 256  0 0
26 0 256  1 0
```

Figura 3.18: Archivo `trace_se.txt` con estados internos del BAC.

Este tipo de archivo supuso una significativa mejora a la hora de depurar el diseño, pues resulta más ágil que utilizar gdbgui con puntos de ruptura. Ello se debe a que, cuando las trazas del módulo diseñado no coinciden con las de referencia, puede detectarse rápidamente en qué parte se halla el error con simplemente comparar la

secuencia de valores de `codIRange` en las formas de onda de Vivado™ con las obtenidas en el `trace_se.txt` correspondiente. Una vez detectado el subsistema que origina el fallo, puede afinarse aún más la ubicación de la parte defectuosa considerando el resto de información incluida en este fichero.

3.2.2. Simulación con Vivado™

Como ya se mencionó en la Sección 3.2 se realizaron pruebas de verificación funcional a cada módulo por separado aunque con distintos grados de exhaustividad. Ello se debe a que posteriormente se verificaron en conjunto todos los subsistemas que componen el diseño. Para llevar a cabo la verificación general fueron necesarios dos tipos de archivo adicionales a los abordados en la subsección anterior: `trace_enc.txt` y `stimuli.dat`.

El primero de ellos es generado por las funciones incluidas en el ya mencionado `trace_jm.c`. El fichero `trace_jm.c` se utiliza para obtener estados intermedios del codificador H.264. De la gran cantidad de información que se extrae con dicho archivo, para la verificación del diseño descrito en el presente documento únicamente se requieren los coeficientes de cada B4, la información de predicción y el parámetro de cuantización (del inglés *Quantization Parameter*) (QP). Por tanto, `trace_jm.c` ha sido modificado, deshabilitando aquellas zonas del código destinadas a la obtención de información no relevante para CABAC, con el fin de crear archivos `trace_enc.txt` de menor tamaño. En la Figura 3.19 se halla un ejemplo de este tipo de archivo.

Sin embargo, este formato presenta algunas dificultades para ser leído directamente por el banco de pruebas en VHDL mediante el cual se realiza la verificación del diseño. Por consiguiente, a partir de los `trace_enc.txt`, con `cabac_trace.py` también se crean los `stimuli.dat`, los cuales facilitan la simulación, ya que resulta más sencillo realizar la adaptación en Python que leer los `trace_enc.txt` desde el *testbench*. En la Figura 3.20 se muestra la apariencia de los ficheros `stimuli.dat`.

```
3  frame_nr = 0;
4  macroblock_nr = 0;
5  MB_row = 0;
6  MB_col = 0;
7  b8 = 0;
8  b4 = 0;
9  b4_row = 0;
10 b4_col = 0;
11
12 // Mode of intra 4x4 prediction [MBrow][MBcol][b4row][b4col]
13 best_i4_pred[0][0][0][0][0] = 2; // DC
14
15 // Intra 4x4 prediction codification:
16 code_i4_pred[0][0][0][0] = -1; // -1 is the most probable mode: code=-1
17
18 // Residual transformed quantized coefficients [MBrow][MBcol][b4row][b4col]
19 resid_quant_transf_coeff_i4[0][0][0][0] = {
20     -31 , -1 ,  0 , -1 ,
21       0 ,  0 ,  1 ,  0 ,
22      -2 ,  1 ,  0 ,  1 ,
23       0 ,  2 , -1 , -1
24 };
```

Figura 3.19: Archivo `trace_enc.txt` con coeficientes e información de predicción.

```
1  # Automatically generated by cabac_trace.py
2  # Any doubt contact dmartin@iuma.ulpgc.es
3  # This file's name: crop_vid_moon_flyby_close_10_96x80_384_256_f300_00
4  # _jm19_bacfix_intra_sad_qp21_norc_nordo_nfr30_4i2_16i3_uv00_stimuli.dat
5
6  21
7  0 0 0 0 0
8  2
9  -1
10 -31  -1   0  -1
11   0   0   1   0
12  -2   1   0   1
13   0   2  -1  -1
14 21
15 0 0 0 1 0
16 2
17 -1
18 -2  0  0  0
19  0  0  0  0
20  0  0 -1  0
21  0  0  0  0
```

Figura 3.20: Archivo `stimuli.dat` con coeficientes e información de predicción.

Una vez comentados todos los tipos de archivo, puede explicarse la estructura de verificación utilizada (véase la Figura 3.21). En primer lugar, se utiliza el *software* de referencia para obtener los archivos `trace_enc.txt` y `trace_bitstr.txt` al codificar el vídeo indicado por el archivo de configuración, `.cfg`. Los ficheros `trace_enc.txt` y `trace_bitstr.txt` contienen, respectivamente, los coeficientes e información de predicción y las trazas de referencia sin formatear. A continuación, sendos archivos son introducidos a `cabac_trace.py` el cual los formatea generando un fichero `stimuli.dat` y un `cabac_bitstr.txt`. Ambos ficheros son leídos por el banco de pruebas. Los datos del primero se introducen al diseño bajo prueba (del inglés *Design Under Test*) (DUT),

mientras que los del segundo se comparan con la salida obtenida por dicho diseño. En caso de que las salidas no coincidan se detiene la simulación automáticamente con el fin de facilitar la detección del error. Adicionalmente se cuenta con los archivos `trace_se.txt` (no representados en la imagen) para la identificación y corrección de estos errores. Además, cabe señalar que la salida del DUT siempre se almacena en archivos `vhdl.txt`. De no producirse errores durante la simulación, los `vhdl.txt` son idénticos a los `cabac_bitstr.txt`.

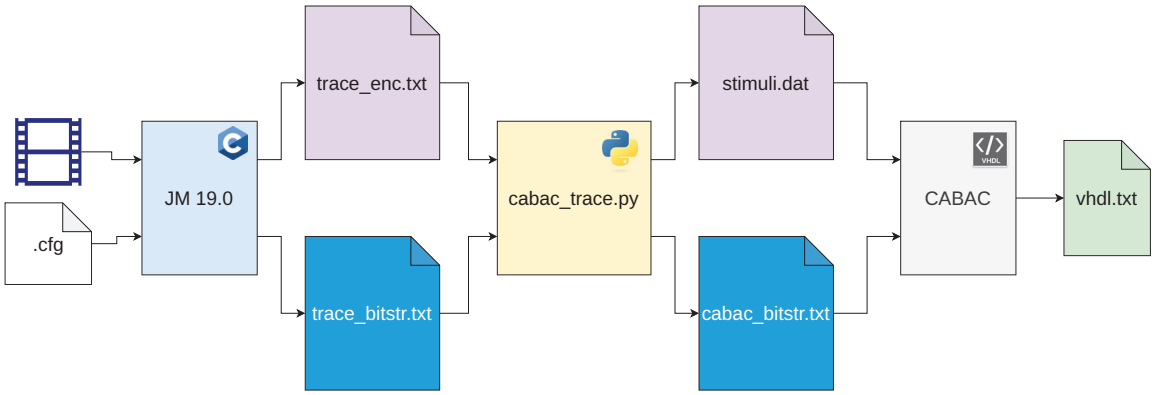

Figura 3.21: Estructura de la verificación del diseño.

3.3. Conclusiones

Como se mencionaba al comienzo de este capítulo, se ha realizado una explicación simplificada de la implementación en VHDL del algoritmo CABAC. Pese a la complejidad inherente tanto al algoritmo como al diseño nivel de transferencia de registros (del inglés *Register Transfer Level*) (RTL) además de las limitaciones de tiempo, se ha logrado desarrollar una arquitectura comprensible y modular que se ajusta a los requisitos del estándar H.264.

Durante el proceso de diseño, se ha optado por una solución más accesible en lugar de seguir arquitecturas excesivamente optimizadas y complejas, como las propuestas en la literatura, lo cual ha resultado clave para cumplir con los objetivos del proyecto. Este enfoque ha permitido centrarse en la correcta comprensión, funcionalidad y flujo de datos del sistema, logrando una implementación que, aunque no optimizada a nivel de frecuencia, es completamente operativa.

Por otro lado, cabe señalar que el análisis del *software* de referencia JM 19.0 combinado con herramientas como gdbgui, Doxygen y Grpahviz contribuyó significativamente al desarrollo del proyecto. Además, su utilización en el proceso de verificación fue fundamental para garantizar el correcto funcionamiento del diseño.

La necesidad de analizar y modificar el código fuente del *software* de referencia en `C` para integrarlo en el flujo de la verificación, así como la creación de utilidades específicas en Python refleja el alto nivel de detalle y precisión requerido para asegurar la fiabilidad del sistema. Además, la estructura de verificación automatizada, mediante la comparación entre los archivos de salida del diseño (`vhdl.txt`) y los generados a partir de la referencia (`cabac_bitstr.txt`), ha proporcionado un entorno robusto que asegura la fiabilidad del sistema. Esta metodología no solo ha permitido verificar el comportamiento del diseño, sino también agilizar su depuración y refinamiento.

Capítulo 4

Resultados y conclusiones

Como su propio nombre indica, en este último capítulo del se recogen los resultados obtenidos del diseño desarrollado y las conclusiones de este Trabajo de Fin de Grado (TFG). Dichos resultados se corresponden con la frecuencia de operación alcanzada, tasas de macrobloques por segundo y su equivalente en Fotogramas Por Segundo (del inglés *Frames Per Second*) (FPS) a la resolución objetivo: *Ultra High Definition* (UHD) (3840x2160).

En cuanto a las conclusiones, se incluye una valoración del trabajo realizado así como las lecciones aprendidas en el transcurso. Asimismo, se procede a comprobar el cumplimiento de los objetivos establecidos en el anteproyecto. Finalmente, se cierra el capítulo con las líneas futuras de mejora del trabajo descrito en este documento.

4.1. Prestaciones del diseño

Habiendo comentado el desarrollo del diseño y el proceso seguido para la verificación del mismo, en las subsecciones siguientes se detallan las prestaciones obtenidas en términos de frecuencia de reloj, tasas de procesamiento, *throughput* y finalmente los recursos ocupados.

4.1.1. Frecuencia

En cuanto a la frecuencia de operación, el diseño funciona correctamente con un reloj de 100 MHz, como puede verse en la Figura 4.1. Atendiendo al *worst negative slack* obtenido, se deduce que el periodo puede decrementarse un total de 0,225 ns. Por tanto, la frecuencia máxima teórica se puede determinar a partir de las siguientes expresiones donde *wns* es el *worst negative slack*, T_{100MHz} el periodo para la frecuencia de 100 MHz, T_{max} el periodo para la máxima frecuencia teórica y f_{max} la máxima frecuencia teórica en sí:

$$T_{100MHz} = \frac{1}{100MHz} = 10ns$$

$$T_{max} = T_{100MHz} - wns = 10 - 0,225 = 9,775ns$$

$$f_{max} = \frac{1}{T_{max}} = \frac{1}{9,775ns} \approx 102,30MHz$$

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0,225 ns	Worst Hold Slack (WHS):	0,072 ns	Worst Pulse Width Slack (WPWS):	3,750 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	22296	Total Number of Endpoints:	22296	Total Number of Endpoints:	9745

All user specified timing constraints are met.

Figura 4.1: *Slack* obtenido para un reloj de 100MHz tras la implementación.

Para garantizar la estabilidad del diseño debe considerarse un margen de guarda en cuanto al *worst negative slack*. Por tanto, teniendo en cuenta que la Nexys 4 DDR tiene un único oscilador de cristal a 100 MHz del cual pueden obtenerse varios relojes

mediante el uso de bucles de enganche de fase (del inglés *Phase-Locked Loops*) (PLL) o *Mixed-Mode Clock Managers* (MMCM) [21], por no añadir complejidad al diseño se fija la máxima frecuencia de funcionamiento a 100 MHz. De esta manera, con el valor de la máxima frecuencia teórica calculado previamente y la frecuencia configurada, el cálculo del margen del *worst negative slack* es el siguiente:

$$\left.\begin{array}{l} T_{max} = 9,775ns \Rightarrow 100\,\% \\ wns = T_{100MHz} - T_{max} = 0,225ns \Rightarrow wns_{\%}\,\% \end{array}\right\} \rightarrow wns_{\%} = \frac{0,225 \cdot 100}{9,775} \approx 2,30\,\%$$

Nótese que, al igual que para el cálculo de la máxima frecuencia teórica, T_{max} es el periodo a dicha frecuencia, *wns* es el *worst negative slack* y $wns_{\%}$ representa lo mismo expresado en porcentaje respecto de la máxima frecuencia teórica. Por consiguiente, el margen de seguridad correspondiente a una frecuencia máxima de funcionamiento de 100 MHz es del 2,30 %.

La ruta crítica que fija esta máxima frecuencia de reloj parte desde el *Binarizador* hasta las *Tablas de contexto*. Las señales pertenecientes a esta ruta están relacionadas con la actualización de los contextos, concretamente los relativos a `last_significant_coeff_flag`. Para solucionar este inconveniente conviene segmentar este camino así como los asociados a otros Elementos Sintácticos (del inglés *Syntax Elements*) (SE) como `coeff_abs_level_minus1`, pues estas rutas son las que mayor retraso tienen. Ello se debe a que parten de una Máquina de Estados Finitos (MEF) muy compleja como es la de control de orden de codificación, ya descrita en el Apartado 3.1.4.1. Dicha MEF así como el módulo *Tablas de contexto* contienen una cantidad significativa de lógica combinacional, de ahí que determinen la ruta crítica. Además, otro módulo en el cual es preciso sustituir lógica combinacional por lógica secuencial es el *Escritor de bits*, dado que algunas de sus señales se encuentran en las que menor *worst negative slack* tienen junto con las mencionadas previamente.

4.1.2. Tasa de codificación

Dada la naturaleza de *Context Adaptative Binary Arithmetic Coding* (CABAC), la tasa de FPS así como el *throughput* son variables y dependen del vídeo a codificar. De igual manera, también tienen influencia la resolución y el parámetro de cuantización (del inglés *Quantization Parameter*) (QP), pues una resolución mayor implica un mayor número de coeficientes por fotograma, mientras que a mayor valor del QP el tamaño de los coeficientes disminuye. En otras palabras, el valor de los coeficientes a codificar (y por tanto la longitud de su *binstring*) es inversamente proporcional al QP, ya que según incrementa el valor del segundo se aplica una compresión mayor. Por tanto, se ha simulado la codificación de varios vídeos de referencia variando el valor del QP. En la Tabla 4.1 se incluye la media de las prestaciones obtenidas por vídeo y de manera global, mientras que en las Tablas 4.2 y 4.3 se recogen los resultados obtenidos para cada vídeo y QP. Cabe señalar que los resultados expuestos en dichas tablas se han calculado descartando aquellos ciclos de reloj que se emplean en cargar el *Buffer de coeficientes* para poder comenzar a procesar la información, pues se trata de un módulo auxiliar que no participa directamente en el proceso de codificación. Asimismo, en dichas tablas también se han calculado los FPS equivalentes en UHD para cada caso a partir de la tasa obtenida de macrobloques por segundo.

En lo que respecta a la Tabla 4.1, los resultados son satisfactorios especialmente si se considera la complejidad y envergadura del proyecto. Las tasas de FPS obtenidas superan los 10 FPS establecidos en los objetivos del proyecto en algunos vídeos, siendo la media de 10,54 FPS.

Atendiendo a los resultados de las Tablas 4.2 y 4.3, se observa que el *throughput* disminuye conforme aumenta el QP. Ello se debe a que, como se comentó previamente, aplicar una mayor cuantización genera un mayor número de coeficientes nulos además de acortar las longitudes de los *binstrings* de los que no lo sean. Por consiguiente, se reduce significativamente la cantidad de datos que el codificador CABAC debe procesar y por tanto el número de bits que se generan es menor.

En cuanto al número de macrobloques procesados por segundo y la tasa de FPS en UHD, estos mejoran según se incrementa el QP. En lo que a estas métricas respecta, al disminuir la cantidad de datos a procesar se requiere menos tiempo para codificar cada fotograma como bien reflejan los resultados.

Vídeo	**Throughput medio (Mbps)**	**Media de macrobloques por segundo**	**Media de FPS en UHD**
crop_vid_moon_flyby	40,57	309129,57	9,54
mov_squares1px	13,18	414302,23	12,79
syntrawvid_2mb	2,60	438655,28	13,54
syntrawvid_b4cmb	5,11	411599,80	12,70
syntrawvid_px1	38,67	257704,62	7,95
syntrawvid_px2	43,31	218159,59	6,73
Resultado global	**23,91**	**341591,85**	**10,54**

Tabla 4.1: Prestaciones medias obtenidas por el diseño.

Vídeo	Resolución (píxeles)	QP	Throughput (Mbps)	Macrobloques por segundo	FPS en UHD
crop_vid_moon_flyby	96x80	21	71,28	119082,40	3,68
		27	70,31	213083,72	6,58
		33	49,18	362834,76	11,20
		46	8,31	421005,75	12,99
		51	3,76	429641,24	13,26
mov_squares1px	112x96	21	18,39	386724,50	11,94
		27	17,17	406355,49	12,54
		33	14,48	418067,60	12,90
		46	9,15	428994,59	13,24
		51	6,72	431368,97	13,31
syntrawvid_2mb	96x96	21	4,78	438655,28	13,54
		27	3,30	438655,28	13,54
		33	2,69	438655,28	13,54
		46	1,17	438655,28	13,54
		51	1,07	438655,28	13,54

Tabla 4.2: Prestaciones obtenidas por el diseño (I).

Vídeo	Resolución (píxeles)	QP	Throughput (Mbps)	Macrobloques por segundo	FPS en UHD
syntrawvid_b4cmb	96x80	21	6,89	396322,13	12,23
		27	7,17	392747,27	12,12
		33	7,10	404858,30	12,50
		46	2,63	430582,87	13,29
		51	1,77	433488,43	13,38
syntrawvid_px1	96x80	21	46,80	137674,67	4,25
		27	50,02	185930,05	5,74
		33	48,00	229779,41	7,09
		46	28,30	356083,09	10,99
		51	20,26	379055,90	11,70
syntrawvid_px2	96x80	21	51,07	67101,63	2,07
		27	49,34	101397,25	3,13
		33	54,60	196948,61	6,08
		46	37,54	350827,95	10,83
		51	23,99	374522,48	11,56

Tabla 4.3: Prestaciones obtenidas por el diseño (II).

4.1.3. Recursos

Con relación a los recursos, en la Tabla 4.4 se incluye un resumen de su utilización sobre la AMD Artix™ 7 que incluye la Nexys 4 DDR. En general, la ocupación de recursos es baja, pues todos los valores se encuentran por debajo del 10 %. Sin embargo, el número de Look-Up Tables (LUT) como lógica resulta moderado al igual que los multiplexores F8 y F7. Ello se debe a la gran cantidad de lógica combinacional utilizada a lo largo del diseño, especialmente en las MEF. Si bien estos valores son aceptables, a futuro resulta interesante añadir segmentación con la cual también se podrá obtener una mayor frecuencia de trabajo. A grandes rasgos, la ocupación de recursos obtenida es adecuada para un diseño de estas características, ya que tiene margen para escabilidad, pudiendo añadir más módulos de los que conforman H.264 al sistema.

Recurso	Total utilizado	Total Utilizado (%)
LUT como lógica	5561	8,77
LUT como memoria	40	0,21
Slice flip-flops	9662	7,62
Multiplexores F7	1040	3,28
Multiplexores F8	448	2,83

Tabla 4.4: Utilización de recursos en la Artix™ 7.

4.2. Entorno de desarrollo del Trabajo de Fin de Grado

El trabajo descrito a lo largo de esta memoria ha consistido en el diseño e implementación en *Field Programmable Gate Array* (FPGA) del sistema de codificación entrópica CABAC conforme al estándar H.264/AVC. Las condiciones de contorno y la motivación principal para la realización de este trabajo se encuadran en que debe ser posible su uso con el resto de unidades del compresor de vídeo siguiendo el estándar H.264 previamente desarrollado por el Instituto Universitario de Microelectrónica Aplicada (IUMA). Dado que la implementación se realiza sobre una FPGA y que la aplicación final será la adquisición, codificación, transmisión y almacenamiento de vídeo en satélites de observación de la Tierra tanto los recursos de procesamiento como el ancho de banda de conexión son limitados.

El algoritmo CABAC es capaz de obtener una alta eficiencia en compresión aunque es mucho más complejo que el *Context Adaptative Variable Length Coding* (CAVLC). Su realización presenta grandes retos tanto a nivel de implementación como de diseño, como indica la gran cantidad de aportaciones que se han realizado en el ámbito científico (mencionadas algunad de ellas en el análisis del estado del arte). Para reducir el nivel de complejidad, la implementación llevada a cabo en este trabajo se ha realizado bajo una serie de restricciones, enunciadas en el Capítulo 2. Aun así, la implementación ha supuesto más de siete mil líneas de código VHDL, sin contar los submódulos previamente diseñados y reutilizados.

Asimismo, como se ha mencionado, se ha utilizado como referencia en el desarrollo de CABAC el *software* JM (en la versión 19.0). No obstante, el análisis del JM 19.0 no es una tarea sencilla, pues es bastante complejo y extenso, además de contar con escasos comentarios y explicaciones. De cualquier manera, supuso una gran ayuda a los problemas derivados de la complejidad de la documentación del estándar (hay que recordar que el estándar solo especifica el decodificador, no el codificador). Como dificultad añadida, entre el *software* de referencia JM y el estándar no hay concordancia entre los nombres de las variables.

Las pruebas de síntesis e implementación se desarrollaron finalmente sobre la tarjeta Nexys 4 DDR. Aunque se había planificado su realización sobre la tarjeta AMD ZC706, esta no estaba disponible en el momento de hacer la validación por lo que se optó por utilizar la Nexys. Esto de ninguna manera modifica los objetivos iniciales, puesto que ambas tarjetas contienen dispositivos de Xilinx y el diseño se ha realizado tecnológicamente agnóstico por lo que puede ser trasladado a cualquier FPGA en la cual haya los recursos necesarios.

Finalmente, el desarrollo del TFG ha supuesto un gran aprendizaje tanto en diseño nivel de transferencia de registros (del inglés *Register Transfer Level*) (RTL), en la utilización de herramientas específicas (como el caso de AMD Vivado™), así como en el uso de otras herramientas apenas vistas en el grado como git o LaTeX. También se ha profundizado en la utilización del lenguaje C a causa del análisis del código fuente de JM 19.0. Igualmente, se han adquirido ciertos conocimientos de Python debido tanto a

los *scripts* necesarios para la verificación del diseño, como para la primera aproximación a CABAC en este lenguaje desarrollada durante las prácticas en empresa del grado, lo que constituyó una gran ayuda a nivel conceptual.

4.2.1. Cumplimiento de los objetivos

El objetivo general del presente proyecto se corresponde con describir en VHDL, verificar e implementar el algoritmo CABAC, del estándar de codificación H.264, en una FPGA para su empleo a bordo de satélites. La resolución de vídeo máxima será UHD (3840x2160) a una tasa de 10 FPS. Dicho objetivo general se desglosa en los siguientes objetivos específicos:

O1. Estudiar el algoritmo CABAC y algunas arquitecturas usadas.

O2. Realizar la descripción *hardware* en *Very high speed integrated circuit Hardware Description Language* (VHDL) del algoritmo CABAC.

O3. Verificar que el sistema cumple con las especificaciones proporcionadas.

O4. Implementar el algoritmo en una FPGA conjuntamente con el resto de etapas que conforman el estándar H.264.

O5. Generar la documentación del proyecto realizado.

Del presente documento puede concluir que se han cumplido todos los objetivos. En primer lugar, en esta memoria se recoge toda la información recabada acerca de CABAC. Se estudiaron algunas de las arquitecturas de las existentes en el estado del arte. Dada la complejidad de CABAC y de las condiciones de contorno en la que se realizó el proyecto (ver la sección anterior), la solución adoptada fue la más sencilla posible.

Teniendo en cuenta lo anterior se realizó una descripción en VHDL de la arquitectura con el objetivo de alcanzar las prestaciones propuestas (resolución UHD a 10 FPS). El diseño se ha realizado de tal manera que la ocupación hardware en el FPGA es menor al 9 % y que puede ser transferible sin ninguna modificación entre distintas FPGA.

La interfaz se diseñó de tal manera que la conexión con el resto de bloques del H.264 diseñado en el IUMA sea directa.

Se ha realizado la verificación del bloque CABAC completo. Para ello fue necesario extraer los estímulos y resultados del *software* de referencia JM. A tal efecto se utilizaron vídeos específicamente pensados para la aplicación final a bordo de satélites. Normalmente estos vídeos cuentan con pocas variaciones entre *frames*, siendo el objetivo final obtener altas tasas de compresión debido a las restricciones tanto en los recursos de almacenamiento disponibles en los satélites como en el ancho de banda disponible en las conexiones con las estaciones terrestres.

Se realizó una validación inicial sobre una tarjeta Nexys 4 DDR, lo cual indica que la descripción y la arquitectura realizadas implementan correctamente la función CABAC del H.264. Aunque, dado el escaso tiempo para desarrollar el TFG no se han podido comprobar las prestaciones finales del sistema, sí se puede concluir que el sistema funciona correctamente.

El último objetivo se cumple con la existencia de esta memoria que se ha desarrollado describiendo tanto el análisis del estado del arte, el diseño y la implementación realizada de CABAC.

4.2.2. Líneas futuras

El diseño realizado en este TFG da lugar a diversas mejoras que pueden ser objeto de nuevos TFG. Algunas de ellas están orientadas a la mejora del propio diseño, mientras que otras a mejoras de los procesos. Asimismo, también se incluyen líneas de trabajo enfocadas en ampliaciones del diseño con el fin de hacerlo más completo:

- Rediseñar los módulos que se han reutilizado de otros proyectos con el fin de integrarlos con el *Intérprete de SE residuales*. De esta forma se podría reducir la latencia del sistema al no almacenar coeficientes nulos.
- Añadir mayor segmentación para incrementar la frecuencia de operación. Dicha segmentación es especialmente importante en el *Modelador de Contextos*. Con-

siderando lo mencionado en la Subsección 4.1.1, también es interesante añadir segmentación en el *Escritor de bits*.

- Mejorar la MEF encargada de gestionar el control del orden de codificación. Analizar y mejorar esta MEF tendría un impacto positivo en las prestaciones.
- Añadir el proceso de las crominancias y aumentar los coeficientes de codificación incluyendo `Intra` 16x16.
- La verificación es uno de los procesos más críticos. En el desarrollo actual, se propone automatizar el sistema de generación de trazas y archivos de coeficientes, de modo que no sean necesarios tantos *scripts* y ficheros.
- Para mejorar la verificación se propone utilizar alguna metodología de verificación específica, por ejemplo *Universal VHDL Verification Methodology* (UVVM), con el objetivo de facilitar la realización de los bancos de pruebas y mejorar la cobertura del diseño.

Anexo A

Presupuesto

Habiendo completado satisfactoriamente el diseño propuesto, ahora se realiza un análisis de los costes económicos del presente Trabajo de Fin de Grado (TFG). Dichos costes se desglosan en los siguientes apartados:

- Recursos humanos.
- Recursos materiales.
  - Recursos *hardware*.
  - Recursos *software*.
- Redacción del documento.
- Derechos de visado del Colegio Oficial de Ingenieros Técnicos de Telecomunicaciones (COITT).
- Gastos de tramitación y envío.
- Material fungible.
- Presupuesto final (con impuestos).

A.1. Recursos humanos

El coste de los recursos humanos se corresponde con los honorarios que recibe el ingeniero a cargo del desarrollo del proyecto considerando el salario correspondiente a un graduado en ingeniería en tecnologías de la telecomunicación. Contemplando 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 [22]. El resultado del coste de los recursos humanos se halla en la Tabla A.1.

Personal	Coste total mensual (€)	Tiempo	Total (€)
Ingeniero técnico	760,05	4 meses	3040,20

Tabla A.1: Trabajo tarifado por tiempo empleado.

El coste final del trabajo tarifado por tiempo empleado es de TRES MIL CUARENTA EUROS CON VEINTE 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, V_{ad} el valor de adquisición, V_{res} el valor residual y N el número de años considerados para la amortización de cada producto. Nótese que el valor residual se supone 0 € para todos los casos.

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

A.2.1. Recursos *software*

Dado que todos los recursos *software* empleados en el desarrollo del trabajo (listados más abajo) son directamente gratuitos, ofrecen versiones gratuitas para estudiantes o

se encuentran disponibles para su uso en los laboratorios de la ULPGC, el coste total asociado a los recursos *software* es de CERO EUROS.

- Paquete ofimático Microsoft-Office 365.
- Microsoft Project 2013.
- Entorno LaTeX.
- Vivado™ Design Suite 202X.
- VSCodium.
- Doxygen.
- Zotero.

A.2.2. Recursos *hardware*

Como el trabajo se ha elaborado en un periodo inferior al estipulado para la amortización, se realiza una amortización equiparable al periodo de duración del mismo. En la Tabla A.2 aparece el coste total de los recursos *hardware*.

Descripción	**Uds.**	**Tiempo de uso**	V_{ad} **(€)**	**Coste anual (€)**	**Coste final (€)**
PC personal	1	4 meses	1.700,00	566,67	188,89
Estación de trabajo del laboratorio	1	4 meses	1.400,00	466,67	155,56
Nexys 4 DDR	1	1 semana	306,84	102,28	2,13
Total					**346,58**

Tabla A.2: Amortización de los recursos *hardware*.

El coste total asociado a los recursos *hardware* es de TRESCIENTOS CUARENTA Y SEIS EUROS CON CINCUENTA Y OCHO CÉNTIMOS.

A.3. Redacción del documento

El coste de la redacción del documento se calcula según la siguiente expresión donde R representa dichos costes, P es el presupuesto y C_n es el coeficiente de ponderación del presupuesto. Como el coste total del proyecto no supera los 30.050,00 €, C_n se supone de valor unitario.

$$R = 0,07 \cdot P \cdot C_n$$

Por otro lado, el presupuesto se calcula como la suma del coste de las amortizaciones y los costes del trabajo tarifado por tiempo empleado. En la Tabla A.3 se halla el resultado de esta operación.

Descripción	Coste (€)
Coste tarifado por tiempo empleado	3.040,20
Amortización de los recursos *software*	0,00
Amortización de los recursos *hardware*	346,58
Total	**3386,78**

Tabla A.3: Total de las amortizaciones y trabajo tarifado por tiempo empleado.

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

$$R = 0,07 \cdot P \cdot C_n = 0,07 \cdot 3386,78 \cdot 1 \approx 237,07 \text{ €}$$

El coste derivado de la redacción del documento es de DOSCIENTOS TREINTA Y SIETE EUROS CON SIETE 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 [23]:

$$V = 0,007 \cdot P \cdot C_r$$

Nótese que V hace alusión a los costes de los derechos de visado, P al presupuesto de ejecución sin impuestos[1] y C_r al coeficiente reductor. Por ser $P < 6000$ €, el valor de C_r es 1. Asimismo, en [23] se menciona que el mínimo importe para los derechos de visado de un documento de estas características es de 40 €. Finalmente, los costes de los derechos de visado son:

$$V = 0,007 \cdot P \cdot C_r = 0,007 \cdot (3386,78 + 237,07) \cdot 1 = 25,37 \text{ €} \rightarrow V = 40 \text{ €}$$

El coste de los derechos de visado es de CUARENTA EUROS.

A.5. Gastos de administración

De acuerdo con [23], 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

Además de los costes expuestos previamente, en esta sección también se incluyen los asociados al material de papelería empleado durante el transcurso del proyecto, así como los derivados de la impresión de la memoria y su encuadernación. En la Tabla A.4

[1] En el presupuesto de ejecución sin impuestos deben incluirse los costes de redacción del documento.

se muestran estos gastos.

Descripción	Coste (€)
Material de papelería	5,00
Impresión de la memoria	30,00
Encuadernación	5,00
Total	**40,00**

Tabla A.4: Total del material fungible.

El coste total del material fungible es de CUARENTA EUROS.

A.7. Presupuesto final del proyecto

Al desarrollo del presente TFG se le aplica el Impuesto General Indirecto Canario (IGIC), el cual representa un siete por ciento del valor del presupuesto. En la Tabla A.5 se recoge el presupuesto final.

Descripción	Coste (€)
Coste tarifado por tiempo empleado	3.040,20
Amortización de los recursos *software*	0,00
Amortización de los recursos *hardware*	346,58
Redacción del documento	237,07
Derechos de visado del COITT	40,00
Gastos de administración	9,00
Material fungible	40,00
Subtotal	3712,85
Total (+7 % IGIC)	**3972,75**

Tabla A.5: Presupuesto final.

El valor del presupuesto final para el presente TFG es de TRES MIL NOVECIENTOS SETENTA Y DOS EUROS CON SETENTA Y CINCO CÉNTIMOS.

MARTIN MARTIN DAVID - 42418968E

Firmado digitalmente por MARTIN MARTIN DAVID - 42418968E
Fecha: 2025.05.26 19:31:46 +01'00'

Bibliografía

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

[2] X. Tian, T. M. Le e Y. Lian, "Design of a CABAC Encoder," en *Entropy Coders of the H.264/AVC Standard*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, Series Title: Signals and Communication Technology, ISBN: 978-3-642-14702-9 978-3-642-14703-6. DOI: 10.1007/978-3-642-14703-6_4.

[3] Q. Zhao, L. Yu, Z. Du et al., "An overview of the applications of earth observation satellite data: Impacts and future trends," *Remote Sensing*, vol. 14, n.º 8, ene. de 2022, Number: 8 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2072-4292. DOI: 10.3390/rs14081863.

[4] P. Naik, H. G. Virani y N. Guinde, "Hardware Implementation of CABAC Decoder in HEVC Using HLS," en *2023 3rd Asian Conference on Innovation in Technology (ASIANCON)*, Ravet IN, India: IEEE, 25 de ago. de 2023, ISBN: 9798350302288. DOI: 10.1109/ASIANCON58793.2023.10270117.

[5] P. Pérez Carballo, *Tema 1: Introducción al diseño de sistemas electrónicos basados en hardware programable*, Hardware Programable, Universidad de Las Palmas de Gran Canaria, feb. de 2023.

[6] P. Pérez Carballo, *Tema 2: Arquitecturas de las FPGAS: Aspectos Tecnológicos y Dispositivos programación*, Hardware Programable, Universidad de Las Palmas de Gran Canaria, feb. de 2023.

[7] S. Z. Ahmed, G. Sassatelli, L. Torres y L. Rougé, "Survey of New Trends in Industry for Programmable Hardware: FPGAs, MPPAs, MPSoCs, Structured

ASICs, eFPGAs and New Wave of Innovation in FPGAs," en *2010 International Conference on Field Programmable Logic and Applications*, 2010. DOI: 10.1109/FPL.2010.66.

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

[9] Y.-L. S. Lin, C.-Y. Kao, H.-C. Kuo y J.-W. Chen, *VLSI Design for Video Coding: H. 264/AVC Encoding from Standard Specification to Chip*. Springer Science & Business Media, 2009.

[10] G. Sullivan y T. Wiegand, "Video Compression - From Concepts to the H.264/AVC Standard," *Proceedings of the IEEE*, vol. 93, n.º 1, ene. de 2005, ISSN: 0018-9219. DOI: 10.1109/JPROC.2004.839617.

[11] T. Wiegand, G. Sullivan, G. Bjontegaard y A. Luthra, "Overview of the H.264/AVC video coding standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, n.º 7, jul. de 2003, Conference Name: IEEE Transactions on Circuits and Systems for Video Technology, ISSN: 1558-2205. DOI: 10.1109/TCSVT.2003.815165.

[12] ITU-T, *H.264: Advanced video coding for generic audiovisual services*, ago. de 2021.

[13] A. Tamhankar y K. Rao, "An overview of H.264/MPEG-4 Part 10," en *Proceedings EC-VIP-MC 2003. 4th EURASIP Conference focused on Video/Image Processing and Multimedia Communications (IEEE Cat. No.03EX667)*, vol. 1, jul. de 2003. DOI: 10.1109/VIPMC.2003.1220437.

[14] J. G. Viera Santana, *Tema 7: Codificación de la señal de vídeo. Formatos básicos: JPEG, MPEG-2, MPEG-4*, Sistemas Audiovisuales y Multimedia, Universidad de Las Palmas de Gran Canaria, 2022.

[15] A. Mehta, "VLSI implementation of fully hardwired high performance h.264 CABAC encoder," Tesis doct., Vellore Institute of Technology, mayo de 2014.

[16] 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*, vol. 13, n.º 7, jul. de 2003, ISSN: 1051-8215, 1558-2205. DOI: 10.1109/TCSVT.2003.815173.

[17] R. Osorio y J. Bruguera, "High-Throughput Architecture for H.264/AVC CABAC Compression System," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, n.º 11, nov. de 2006, ISSN: 1051-8215, 1558-2205. DOI: 10.1109/TCSVT.2006.883508.

[18] R. Osorio y J. Bruguera, "Arithmetic coding architecture for H.264/AVC CABAC compression system," en *Euromicro Symposium on Digital System Design, 2004. DSD 2004.*, 2004. DOI: 10.1109/DSD.2004.1333259.

[19] J. Zhou, D. Zhou, W. Fei y S. Goto, "A high-performance CABAC encoder architecture for HEVC and H.264/AVC," en *2013 IEEE International Conference on Image Processing*, 2013. DOI: 10.1109/ICIP.2013.6738323.

[20] A. M. Tourapis, A. Leontaris, K. Suhring y G. Sullivan, "H. 264/14496-10 AVC reference software manual," *Doc. JVT-AE010*, 2009.

[21] *Nexys4 DDR™ FPGA Board Reference Manual*, 11 de abr. de 2016.

[22] Universidad de Las Palmas de Gran Canaria, *Boletín Oficial de la Universidad de Las Palmas de Gran Canaria, Nº 13*, 2 de ago. de 2024.

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