UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

IMPLEMENTACIÓN DE UN COMPRESOR – DESCOMPRESOR DE IMÁGENES CON JPEG200 EN UN DSP

T E S I S

Que para obtener el título de

INGENIERO ELÉCTRICO ELECTRÓNICO

P R E S E N T A :

P E D R O J I M É N E Z R O S A S

DIRECTOR DE TESIS: M.I. Larry H. Escobar Salguero

CIUDAD UNIVERSITARIA 2009

A mis padres, Pedro y Josefina,
Gracias por los buenos ejemplos, gracias por confiar en mí

A mi hermana, Jazmín Yaritza,
Que el camino que elijas esté lleno de éxito

Agradecimientos

Me gustaría expresar mi gratitud al M. I. Larry Escobar Salguero por haberme guiado con su conocimiento y experiencia a lo largo de la realización de esta tesis. También quiero agradecer al Ing. Edgar Alonso Trueba por la valiosa ayuda que me prestó en numerosas ocasiones.

INDICE GENERAL

Capítulo 1
Introducción

La mayor parte de la información que los seres humanos recibimos del medio ambiente la percibimos a través de nuestros ojos, es decir, en forma de imágenes, las cuales son un medio bastante efectivo para comunicar y almacenar información, ya que nuestro cerebro es particularmente apto para el procesamiento de datos visuales [1]; debido a esto y al creciente desarrollo de las computadoras digitales en los últimos años, cada día resulta más necesaria y atractiva la incorporación del procesamiento de imágenes en aplicaciones de la ciencia y la ingeniería tales como la telemedicina, la telefonía móvil, la visión de máquinas y los cada vez más numerosos servicios de Internet [6].

Desde los años sesenta hasta la fecha, el campo del procesamiento de imágenes ha crecido considerablemente, además de las aplicaciones en programas espaciales, las técnicas de procesamiento digital de imágenes ahora son usadas en una gran cantidad de aplicaciones, como en la representación con colores de niveles de intensidad de rayos X y otras imágenes usadas en la industria, la medicina y las ciencias biológicas; en el estudio de patrones de contaminación con imágenes generadas desde el aire o con satélites; y en la restauración de imágenes degradadas de objetos no recuperables o de resultados de experimentos que son demasiado costosos como para ser realizados nuevamente. Ejemplos similares de aplicaciones exitosas del procesamiento digital de imágenes se pueden encontrar en astronomía, biología, medicina nuclear, defensa e industria [2].

Sin embargo, uno de los problemas que tienen que solucionarse cuando se trabaja con imágenes digitales es que requieren de un gran poder de procesamiento, de mucho espacio en memoria para ser almacenadas y de mucho ancho de banda para ser transmitidas. Estas son las razones por las cuales se han desarrollado *técnicas de compresión de imágenes digitales* [3].

Las técnicas de compresión de imágenes son de gran importancia ya que permiten que la información sea almacenada, reproducida y transmitida usando velocidades de transferencia y cantidades de datos menores, lo cual se logra aprovechando la redundancia que existe dentro de los datos de una imagen (valores parecidos de píxeles cercanos entre sí) para encontrar una representación que ocupe menos espacio [20].

La Transformada Coseno Discreta (TCD) y la Transformada Wavelet Discreta (TWD) permiten describir imágenes en el dominio de la frecuencia y debido a que una parte importante de la energía de las imágenes se concentra en las componentes de más baja frecuencia, los coeficientes generados por la TCD y la TWD se pueden representar con un número menor de bits y se pueden descartar los coeficientes con los valores más pequeños mediante un proceso conocido como Cuantización, lo que da como resultado una reducción en la cantidad de datos requeridos para representar una imagen sin que haya una pérdida apreciable en su calidad.

Uno de los estándares de compresión que ha llegado a ser de los más utilizados para la compresión de imágenes a color y en escala de grises es el estándar JPEG (Joint

Photographic Experts Group), debido a que logra mayores tasas de compresión en imágenes fotográficas que cualquier formato de archivos gráficos de uso común. Por ejemplo, a una fotografía que para ser almacenada en un archivo BMP de Windows requiere de 1 Mbyte usualmente la puede comprimir de modo que ocupe únicamente 50 kbytes, es decir, reduce la cantidad de memoria requerida para almacenarla al 5% [3].

La técnica que usa JPEG para la compresión de imágenes fijas está basada en la TCD, sin embargo, el comité JPEG decidió desarrollar un nuevo estándar de compresión de imágenes, el JPEG2000, basado en la TWD [11].

Una de las principales razones por las cuales el comité JPEG decidió usar la TWD en el estándar JPEG2000 es que con la TCD se tiene un error al reconstruir la imagen conocido como artefactos de bloque. Esto se debe a que la TCD divide la imagen en bloques que procesa de forma independiente y al cuantizar los coeficientes resultantes con una cantidad pequeña de bits existe un error en la reconstrucción que es más visible en las orillas de los bloques, provocando discontinuidades en la imagen. La TWD, en cambio, procesa las imágenes en su totalidad y no en bloques artificiales, por lo cual no provoca la aparición de artefactos de bloque [11].

El estándar JPEG no usó la TWD desde un principio porque cuando estaba siendo desarrollado, a mediados de la década de los 80's, aunque esta transformada ya era conocida, todavía no había un método eficiente para codificar sus coeficientes de modo que fuera comparable con la TCD; sin embargo, el estado actual de las técnicas de compresión de imágenes basadas en la TWD ha mejorado significativamente desde la introducción del JPEG original [11].

Por otro lado, actualmente existen microcontroladores de propósito específico llamados *procesadores digitales de señales* (DSPs por sus siglas en inglés), cuyo objetivo es el procesamiento de señales. En particular, el diseño de la familia de DSPs C6000 de la marca Texas Instruments está orientado hacia el procesamiento digital de imágenes [24], lo cual la hace apropiada para la programación eficiente de procesos de compresión como el que especifica el estándar JPEG2000.

En la figura 1.1 se muestra un diagrama de bloques de un sistema compresor de imágenes basado en el estándar JPEG2000, de los cuales el bloque principal es la TWD.

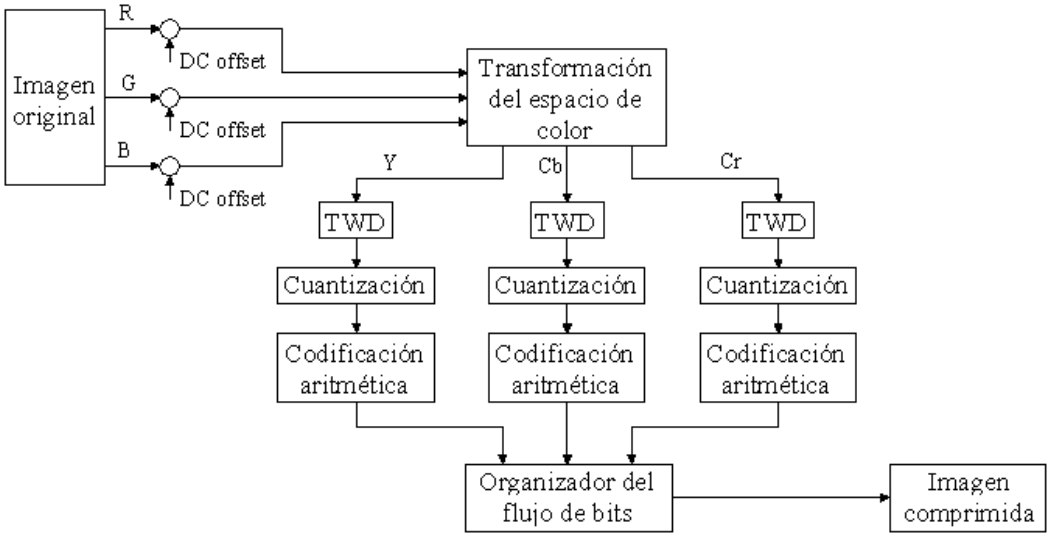


Figura 1.1 Diagrama de bloques de un compresor del estándar JPEG2000.

El *objetivo de esta tesis* es implementar un proceso de compresión y descompresión de imágenes basado en el estándar JPEG2000 en la tarjeta de desarrollo del DSP TMS320C6416 (DSKC6416) que genere porcentajes de compresión menores al tres por ciento con una muy buena calidad de la imagen recuperada y que a diferencia de los compresores basados en la TCD no presente el problema de artefactos de bloque, para lo cual se programará cada uno de los bloques de la figura 1.1 en el DSP TMS320C6416.

A continuación se hace una descripción breve de los bloques que componen el sistema de compresión.

Acondicionamiento de la imagen.
Se parte de una imagen digital almacenada en un archivo, el cual se acondiciona utilizando Matlab para que pueda ser almacenado en la memoria del DSKC6416 a través del Code Composer Studio (CCS). El CCS es el software que se utiliza para programar al DSP.

Suma de una componente de DC.
A los valores de los píxeles que están guardados en memoria de forma binaria se les hace un corrimiento para asegurarse de que tengan un intervalo dinámico que esté centrado aproximadamente en cero.

Transformación del espacio de color.
La imagen original está representada en el espacio de color RGB (Red, Green, Blue) y antes de ser comprimida se pasa al espacio de color YCbCr (Luminancia, Crominancia Azul, Crominancia Roja).

Transformada Wavelet Discreta.
Para representar a los píxeles de la imagen en el dominio de la frecuencia y así remover la redundancia que existe entre ellos, se aplica la TWD a cada componente del modelo YCrCb por separado. Lo que hace posible la compresión es que la energía de la mayoría de las imágenes está concentrada en pocos coeficientes.

Cuantización.
La Cuantización transforma los coeficientes resultantes de la TWD en un conjunto finito de valores, lo cual provoca que los coeficientes con valores pequeños, es decir, los que cooperan menos con la calidad de la imagen recuperada, se hagan cero.

Codificación aritmética.
A los coeficientes cuantizados se les aplica un método de compresión conocido como Codificación Aritmética Binaria, el cual aprovecha las estadísticas de que se dispone sobre el comportamiento de los coeficientes para representar una secuencia completa de bits mediante un intervalo numérico que se va reduciendo conforme los coeficientes son procesados.

Organizador del flujo de bits.
Esta es la etapa final del sistema compresor y en ella se integran y se organizan los datos resultantes de los procedimientos anteriores para formar un flujo de bits listo para su almacenamiento o transmisión.

El proceso de descompresión se implementará siguiendo un procedimiento similar al mostrado en la figura 1.1, pero en orden inverso, con la finalidad de reconstruir la imagen a partir del flujo de bits, verificar que el proceso de compresión se haya llevado a cabo correctamente, evaluar la calidad visual de las imágenes recuperadas, medir los tiempos de compresión y descompresión y calcular los porcentajes de error.

Se espera que este proyecto proporcione un bloque básico que se pueda utilizar en diversas aplicaciones que utilicen imágenes fijas, así como en aplicaciones que involucren imágenes en movimiento, tales como sistemas de almacenamiento de video en tiempo real y sistemas de vigilancia.

En el *capítulo 2* se describe la forma en que se generan las imágenes digitales y la forma en que se representan sus colores y se exponen brevemente los inicios y algunas de las aplicaciones del Procesamiento Digital de Imágenes.

En el *capítulo 3* se hace una descripción de diferentes técnicas que son utilizadas para la compresión de imágenes y la forma en que los distintos formatos de archivo y estándares de compresión las utilizan.

En el *capítulo 4* se describen los procedimientos que se deben llevar a cabo para implementar un sistema básico de compresión y descompresión de imágenes a color basado en el estándar JPEG2000.

En el *capítulo 5* se describe la arquitectura del DSP TMS320C6416 y la forma en que se implementaron los procesos de compresión y descompresión de imágenes en el módulo de desarrollo DSKC6416.

En el *capítulo 6* se presentan y analizan los resultados obtenidos y en el *capítulo 7* se establecen las conclusiones generales que se obtuvieron con la realización de este trabajo.

Capítulo 2
Fundamentos de Imágenes Digitales

Actualmente las imágenes digitales se han convertido en parte de la vida cotidiana, pudiendo encontrarlas, por ejemplo, en las imágenes generadas por computadora, en los efectos espaciales que vemos en el cine y en la televisión, en las cámaras digitales, en las impresoras con calidad fotográfica y en los programas de computadora usados para manipular imágenes [1].

Además, la utilización del procesamiento de imágenes digitales en diferentes áreas de la ciencia y la ingeniería se ha venido incrementando constantemente, con aplicaciones que van desde la generación de imágenes médicas hasta la exploración planetaria [1].

Un requisito indispensable para lograr que una técnica de procesamiento sea eficiente es contar con una representación adecuada de la imagen [5]. En este capítulo se expone la forma en que se generan las imágenes digitales, la forma en que se representan las imágenes digitales a color y se mencionan las clases fundamentales de operaciones que existen para procesar imágenes y algunas de sus aplicaciones.

2.1 Generación de una imagen digital

Una imagen natural es un patrón de formas y colores que se combinan sin interrupciones para formar una reproducción fiel de un área de la escena original y se conoce como *imagen de tono continuo* [4].

Una imagen se puede definir dentro de un sistema de coordenadas con el origen colocado por convención en la esquina superior izquierda, como se muestra en la figura 2.1, y si se trata de una imagen en escala de grises se puede representar mediante una función *f(x,y)*, donde la amplitud *f* para cualquier par de coordenadas *x* y *y* es proporcional a la intensidad de la luz detectada en ese punto [1].

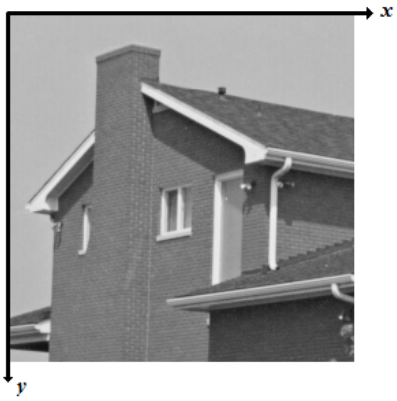


Figura 2.1 Imagen de tono continuo en escala de grises

Para procesar una imagen de tono continuo usando una computadora, primero se debe convertir a una forma digital, es decir, traducirla a un conjunto de datos numéricos. Esta representación digital es únicamente una aproximación de la imagen original, sin embargo, ese es el precio que debemos pagar por la conveniencia de manipular la imagen utilizando una computadora [2]. La traducción de una imagen de tono continuo a una forma digital se consigue a través de los procesos de *Muestreo* y C*uantización*.

2.1.1 Muestreo

Muestreo es el proceso de tomar el valor de la función *f(x,y)* en intervalos discretos de espacio, es decir, tomar muestras de la intensidad de luz en la imagen de tono continuo en lugares específicos formando un arreglo rectangular, como se muestra en la figura 2.2. Cada muestra corresponde a una pequeña área cuadrada de la imagen conocida como *píxel* y se le asigna unas coordenadas *(x,y)*, donde *x* indica la columna en la que se encuentra y *y* indica la línea [1].

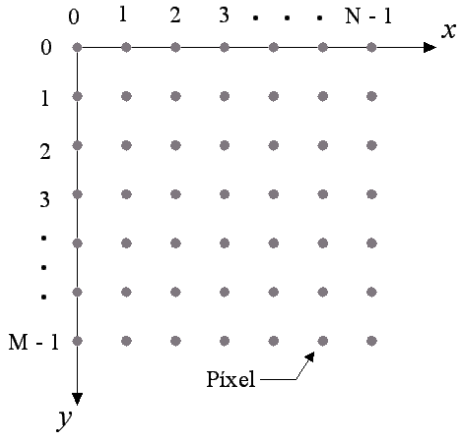


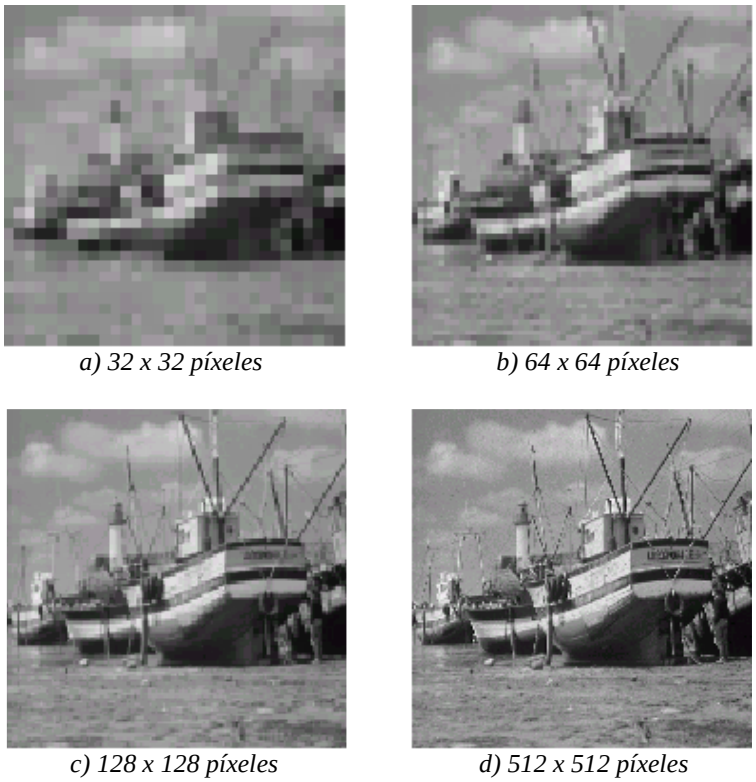
Figura 2.2 Esquema de una imagen muestreada

Un método comúnmente utilizado para realizar este proceso es el que utilizan las cámaras de televisión, las cuales recorren la imagen línea por línea y luego muestrean cada una. Por otro lado, el dispositivo de acoplamiento de carga (CCD) y el sensor de imágenes CMOS (Complementary Metal-Oxide Semiconductor) son muy utilizados en cámaras digitales y escáners, estos dispositivos están basados en la tecnología de los semiconductores y consisten en arreglos discretos de fotodetectores, por lo cual realmente toman muestras en dos dimensiones [1].

2.1.2 Resolución espacial

La resolución espacial de una imagen es el tamaño físico de los píxeles en la imagen, es decir, el área en la escena que se representa con un solo píxel. Un Muestreo muy denso da como resultado una imagen de alta resolución en la cual se dispone de muchos píxeles para representar una parte pequeña de la escena, en cambio, un Muestreo poco denso produce

una imagen de baja resolución en la que hay pocos píxeles, cada uno de ellos representando un área relativamente grande [1]. En la figura 2.3 se puede apreciar el efecto que tiene en la calidad de una imagen el incremento de la resolución espacial.



a) 32 x 32 píxeles *b) 64 x 64 píxeles*

c) 128 x 128 píxeles *d) 512 x 512 píxeles*

Figura 2.3 Imagen con cuatro resoluciones espaciales diferentes

Los cambios graduales en una imagen pueden representarse adecuadamente con un Muestreo poco denso, mientras que cambios rápidos únicamente se pueden representar apropiadamente con un Muestreo muy denso [1].

Un conjunto de resoluciones muy usado en aplicaciones de video digital es el de los *formatos intermedios* del formato CIF (Common Intermediate Format) [9]. En la tabla 2.1 se muestran las resoluciones del formato CIF y de sus formatos intermedios y en la figura 2.4 se muestran las dimensiones relativas de estos formatos.

Formato	Resolución
4CIF	704 x 576 píxeles
CIF	352 x 288 píxeles
Quarter CIF (QCIF)	176 x 144 píxeles
Sub-QCIF (SQCIF)	128 x 96 píxeles

Tabla 2.1 Resoluciones de los formatos intermedios CIF

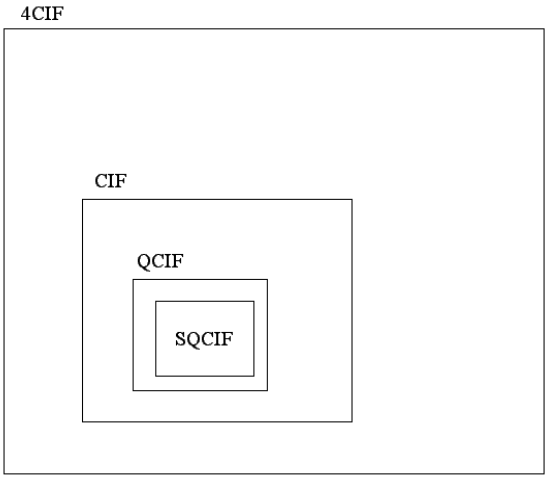


Figura 2.4 Resoluciones basadas en el formato CIF

2.1.3 Cuantización

Una vez que la imagen fue muestreada y ordenada en un arreglo de píxeles, cada uno de ellos debe ser descrito con un valor numérico discreto de acuerdo con la cantidad de luz que posea. Este proceso se conoce como *Cuantización* [4].

El proceso de Cuantización realiza el mapeo de la variable continua *f*, la intensidad de luz de cada píxel, en una variable discreta *u*, la cual toma valores de un conjunto finito de números llamados *niveles de cuantización*. Generalmente, este mapeo es una función escalera como la que se muestra en la figura 2.5 [5].

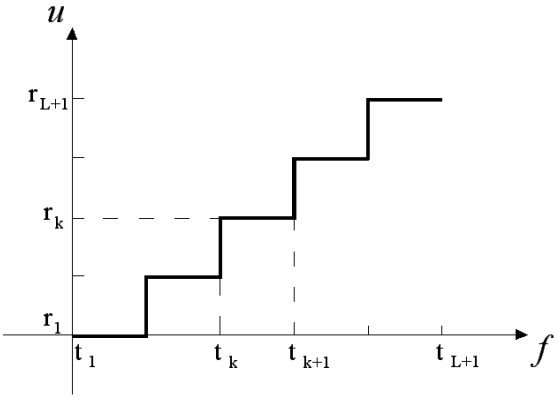


Figura 2.5 Mapeo de la variable continua f en la variable discreta u

La Cuantización se realiza de la siguiente forma: se define un conjunto de niveles de decisión con t_1 y t_{L+1} como los valores mínimo y máximo de la variable *f* y si $f(x_0, y_0)$ está en el intervalo $[t_k, t_{k+1})$, a $u(x_0, y_0)$ se le asigna el nivel de cuantización r_k, como se muestra en la figura 2.5 [5].

Por convención, un conjunto de *n* niveles de cuantización está compuesto por los enteros *0, 1, 2,…, n-1*, donde el cero representa al negro, *n-1* al blanco y los valores intermedios representan varios tonos de gris [1].

2.1.4 Resolución en amplitud

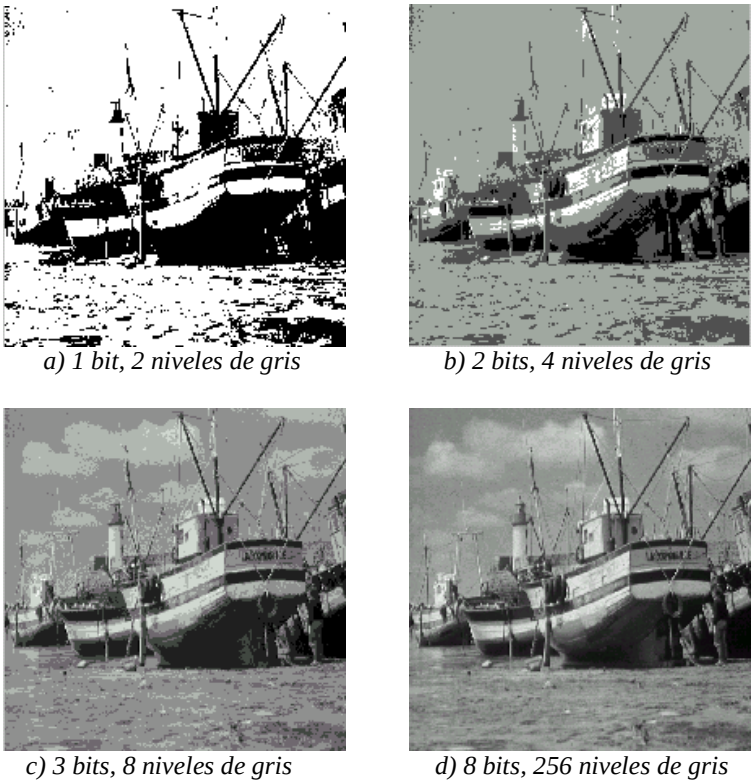
Los niveles de cuantización son llamados comúnmente *niveles de gris* y el término que los agrupa a todos desde el negro hasta el blanco es el de *escala de grises*.

Por conveniencia y por eficiencia en el procesamiento de la imagen digital, el número de niveles de gris *n* usualmente es una potencia entera de dos, como se muestra en la ecuación (2.1)

$$n = 2^b \tag{2.1}$$

donde b es el número de bits usado para representar a la variable *u*.

El valor de b es típicamente ocho, lo que implica imágenes con 256 posibles niveles de gris, desde el 0 (negro) hasta el 255 (blanco). La figura 2.6 muestra que mientras mayor es el número de niveles de gris, mayor es la semejanza de la imagen con la escena original [1].



a) 1 bit, 2 niveles de gris *b) 2 bits, 4 niveles de gris*

c) 3 bits, 8 niveles de gris *d) 8 bits, 256 niveles de gris*

Figura 2.6 Imagen cuantizada usando 1, 2, 3 y 8 bits por píxel

2.2 Modelos de Color

Las imágenes digitales a color se pueden representar mediante una función vectorial *f(x,y)*, donde las componentes del vector *f* para cualquier par de coordenadas *x* y *y* representan el color del píxel localizado en ese punto [1].

Existen varias formas de representar colores utilizando datos numéricos y cada una de ellas recibe el nombre de *Modelo de Color*. La elección del Modelo de Color para una

aplicación determinada usualmente depende del dispositivo que se utilice para desplegar la imagen [3].

Los Modelos de Color más utilizados son: RGB, usado en aplicaciones de computadora, CMYK, usado en impresión a color [3], y YCbCr, usado en sistemas de video [10]; sin embargo, ninguno de estos modelos está relacionado directamente con las nociones más intuitivas de matiz, saturación e intensidad, lo que ha llevado a la creación de otros modelos de color como el HSI [8].

2.2.1 RGB

El Modelo de Color más comúnmente usado en aplicaciones de computadora es conocido como RGB (rojo, verde, azul) y está basado en el hecho de que todos los colores del espectro se pueden generar combinando luz roja, verde y azul en diferentes proporciones, por lo cual estos tres colores son conocidos como *colores aditivos*. Por ejemplo, en la mayoría de los monitores a color hay tres emisores de luz por cada píxel, uno rojo, uno verde y uno azul, y ajustando individualmente la intensidad de cada emisor se puede controlar el color del píxel [3].

El conjunto de colores que puede ser representado por un Modelo de Color se conoce como *espacio de color,* el cual, para el modelo RGB, como puede apreciarse en la figura 2.7 es un cubo.

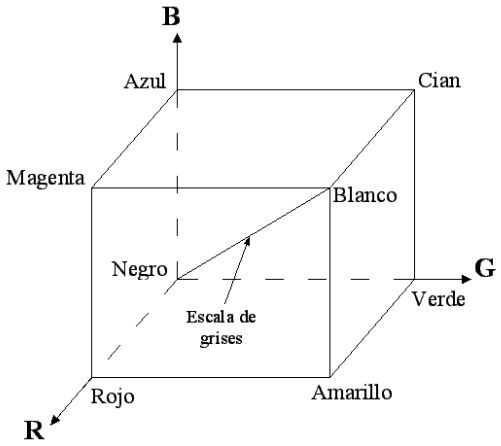


Figura 2.7 Espacio de Color RGB

Los ejes de este espacio de color representan a cada uno de los colores que componen el modelo. La proyección de un punto P contenido en el cubo sobre un eje indica la intensidad de luz con la que coopera el color de ese eje en la formación del color en ese punto. El origen del espacio de color es la ausencia de los tres colores, es decir, el negro, y la esquina opuesta del cubo es la combinación de los tres colores con el máximo de intensidad que pueden tomar, es decir, el blanco [4].

Un color contenido en una cara del cubo RGB se forma combinando los colores de los ejes que forman esa cara, como se puede observar en la figura 2.8. El volumen dentro del cubo contiene todos los colores del espectro que se pueden generar mezclando los tres colores del modelo.

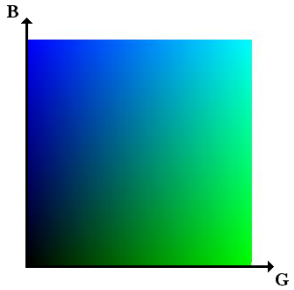


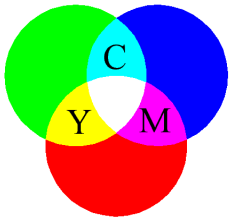
Figura 2.8 Colores contenidos en la cara posterior del cubo RGB

Los valores que puede tomar cada componente del modelo RGB están determinados por el número de bits usado para representarlo, el cual, para imágenes fotográficas, es usualmente 8 [3].

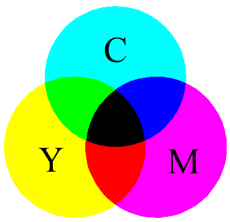
Todos los modelos de color se pueden obtener a partir de la información que proporcionan dispositivos como escáners y cámaras, los cuales representan los colores utilizando el modelo RGB [3].

2.2.2 CMYK

El Modelo de Color CMYK (cian, magenta, amarillo, negro) está basado en los colores sustractivos, los cuales, a diferencia de los colores aditivos del modelo RGB que emiten luz, absorben una parte de la luz que incide sobre ellos y la otra la reflejan. Los colores sustractivos son cian, magenta y amarillo (CMY) y resultan de la combinación de luz roja, verde y azul como se muestra en la figura 2.9a [4].



a) Formación de los colores sustractivos



b) Combinación de los colores sustractivos

Figura 2.9 Colores sustractivos

Debido a que el cian se obtiene combinando luz verde con luz azul, cuando se imprime sobre papel blanco y se ilumina con luz blanca, refleja estos dos colores y absorbe la luz roja. Del mismo modo, el magenta absorbe la luz verde y el amarillo absorbe la luz azul.

Cuando se combinan cian y magenta, el cian absorbe la luz roja y el magenta absorbe la luz verde, lo que da como resultado que sólo se refleje la luz azul, como se puede observar en la figura 2.9b. Cuando se combinan amarillo y magenta sólo se refleja luz roja y cuando se mezclan el amarillo y el cian sólo se refleja luz verde. Imprimiendo los tres colores sustractivos en diferentes proporciones sobre papel blanco, el proceso de impresión puede crear todos los colores del espectro [4].

Las componentes del modelo CMYK representan las cuatro tintas usadas comúnmente en la impresión a color. En teoría, la combinación del cian, magenta y amarillo absorbe toda la luz, dando como resultado el color negro (figura 2.9b), sin embargo, en la práctica dicha combinación no resulta en un negro puro y, aunque lo hiciera, se necesitaría tres veces más tinta que si simplemente se usara tinta negra. Es por eso que este Modelo de Color tiene cuatro componentes: cian, magenta, amarillo y negro [3].

Las impresoras a color y copiadoras requieren una entrada CMY o realizan internamente una conversión de RGB a CMY, la cual se lleva a cabo mediante la ecuación (2.2) [8].

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(2.2)}$$

2.2.3 YCbCr

El Modelo de Color YCbCr (Luminancia, Crominancia azul, Crominancia roja) se utiliza en sistemas de video digital y está basado en una señal denominada *luminancia*, la cual se utiliza en los sistemas de televisión en blanco y negro, y en dos señales llamadas *diferencia de color*, las cuales se utilizaron para hacer compatibles los primeros sistemas de televisión a color con los sistemas en blanco y negro [10].

La luminancia, cuyo nivel es proporcional a la intensidad de luz presente en la escena original, se puede obtener a partir del modelo RGB considerando el peso que tiene cada una de sus componentes en la sensación de brillo de nuestro sistema visual de acuerdo con la ecuación (2.3).

$$Y = 0.299R + 0.587G + 0.114B \quad \textbf{(2.3)}$$

Los sistemas de televisión a color compatibles con los sistemas en blanco y negro transmiten la información de luminancia, ya que ésta es la señal que puede ser utilizada por los receptores para decodificar la imagen en blanco y negro, y transmiten la información que requiere un receptor a color para reconstruir las componentes RGB. Transmitir la luminancia junto con las componentes RGB requiere de mucho ancho de banda, por lo tanto, es más conveniente transmitir la luminancia y dos señales diferencia de color, las cuales se obtienen restando la luminancia a cualquiera de las componentes RGB. Las posibles señales diferencia de color son: *R-Y*, *G-Y* y *B-Y*.

Cuando un sistema transmite las señales diferencia de color *R-Y* y *B-Y* junto con la luminancia, los receptores en blanco y negro ignoran la información de color y reproducen únicamente la luminancia; mientras que los receptores a color utilizan las tres señales y recuperaran las componentes RGB aplicando la transformación que se muestra en la ecuación (2.4).

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -0.509 & -0.194 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} Y \\ R-Y \\ B-Y \end{bmatrix} \quad \textbf{(2.4)}$$

Cuando se transmite una señal en blanco y negro y el receptor es a color, éste último aplica la misma transformación, con la salvedad de que las señales diferencia de color no se

transmiten, y las tres componentes de color son iguales a la luminancia, como se muestra en la ecuación (2.5), lo que da como resultado una imagen en blanco y negro en un receptor a color [10].

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & -0.509 & -0.194 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} Y \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} Y \\ Y \\ Y \end{bmatrix} \tag{2.5}$$

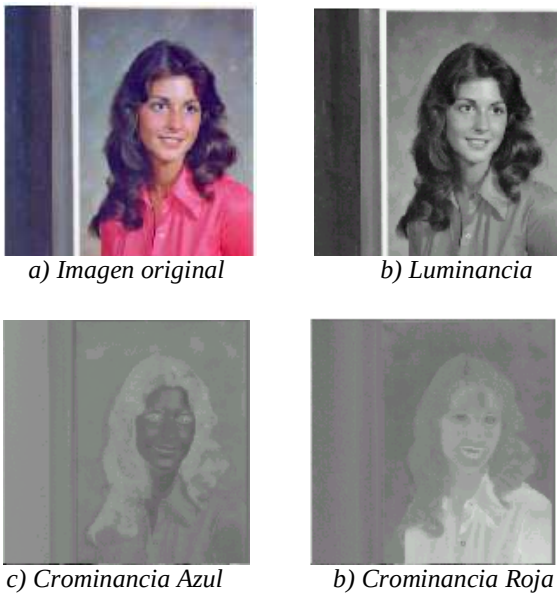
Las componentes Cb y Cr del modelo YCbCr se obtienen a partir de las señales diferencia de color *B-Y* y *R-Y* como se observa en la ecuación (2.6)

$$\begin{aligned} Cb &= 0.564(B-Y) \\ Cr &= 0.713(R-Y) \end{aligned} \tag{2.6}$$

donde Y se calcula con la ecuación (2.3).

Aun cuando no busquen ningún tipo de compatibilidad con los sistemas de televisión en blanco y negro, los sistemas de video digital y de alta definición utilizan el Modelo YCbCr debido a que el ancho de banda asignado a las componentes Cb y Cr puede ser considerablemente inferior al ancho de banda asignado a la componente Y sin que se observe una pérdida en la calidad de la imagen, lo cual es posible debido a que *el sistema visual humano tiene mayor agudeza para distinguir distintos niveles de iluminación que para diferenciar colores* [10].

En la figura 2.10 se muestra una imagen a color, que ha sido separada en sus componentes YCbCr, donde se puede apreciar que el modelo YCbCr, a diferencia del modelo RGB que tiene casi la misma información en cada una de sus componentes, concentra la mayor parte de la información de la imagen en una sola componente, la luminancia [3].



a) Imagen original *b) Luminancia*

c) Crominancia Azul *b) Crominancia Roja*

Figura 2.10 Imagen separada en sus componentes YCbCr

Dado que el sistema visual humano es menos sensible al color que a la luminancia, es posible reducir la resolución de las crominancias sin que haya un efecto obvio en la calidad de la imagen. En la figura 2.11 se muestran tres formatos muy utilizados para representar la resolución de las crominancias. En el formato 4:4:4 las tres componentes tienen la misma resolución, en el formato 4:2:2 las crominancias tienen la misma resolución que la luminancia en sentido vertical, pero tienen la mitad en sentido horizontal y en el formato 4:2:0 las crominancias tienen la mitad de la resolución tanto horizontal como verticalmente; por lo tanto, cuando se representa una imagen en el formato 4:2:0, se requiere la mitad de datos para representar una imagen que cuando se representa en el formato 4:4:4 [9].

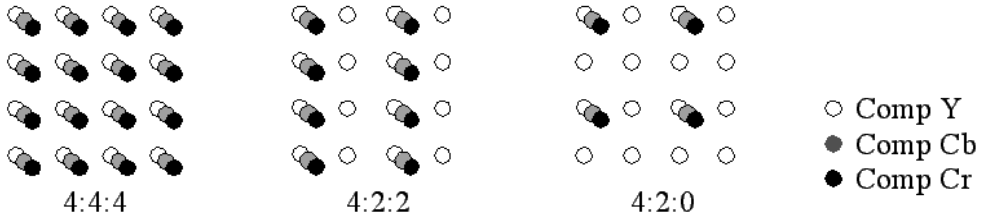


Figura 2.11 Formatos de resoluciones de crominancias

2.2.4 HSI

Las tres componentes del modelo HSI son matiz (H), saturación (S) e intensidad (I). Los dos primeros especifican el color. El matiz determina el color dominante que percibe un observador (como azul o amarillo) y la saturación indica que tanto ese color ha sido diluido por la luz blanca [4].

El modelo HSI es más conveniente que el RGB para ciertas tareas de procesamiento digital de imágenes debido a que en este modelo las componentes intensidad y color son independientes entre sí, lo que permite manipular a una sin afectar a la otra y a que las componentes de saturación y matiz están directamente relacionadas con el modo en que los seres humanos percibimos el color [8].

Algunos ejemplos de aplicación de este modelo son el diseño de sistemas para verificar el grado de madurez de las frutas y la inspección del acabado de color de determinados productos. La idea consiste en utilizar las propiedades del color del mismo modo en que lo haría una persona encargada de esta tarea.

En la figura 2.12 se observa el diagrama de cromaticidad del modelo HSI, en el cual todos los colores que se obtienen con la combinación de los tres colores que se encuentran en los vértices (rojo, verde y azul) caen dentro del triángulo. La saturación indica que tan diluido con luz blanca se encuentra el color en el punto P y es proporcional a la distancia del punto P al centro del triángulo. A medida que P se aleja del centro del triángulo y se aproxima a cualquiera de los lados, el grado de saturación se incrementa. El matiz es el ángulo que existe entre la línea que une al centro del triángulo con el vértice que representa al color rojo y la línea que une al centro del triángulo con el punto P, por lo tanto, cuando H = 0°, el color es rojo y cuando H = 60°, el color es amarillo.

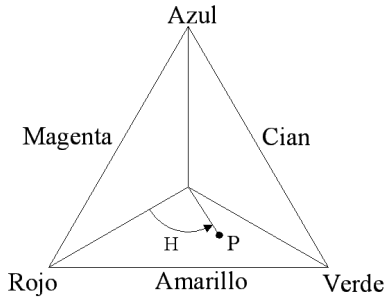


Figura 2.12 Diagrama de cromaticidad del modelo HSI

La intensidad en el modelo HSI se mide con respecto a una línea que es perpendicular al plano del triángulo de la figura 2.12 y pasa a través de su centro. Combinando el matiz, la saturación y la intensidad en un espacio de color en tres dimensiones se obtiene la representación piramidal de la figura 2.13, donde cada triángulo marcado con líneas más gruesas corresponde a un nivel de intensidad [8]. Las intensidades a lo largo de esta línea, por debajo del triángulo central, progresan hacia el negro y, por encima, progresan hacia el blanco.

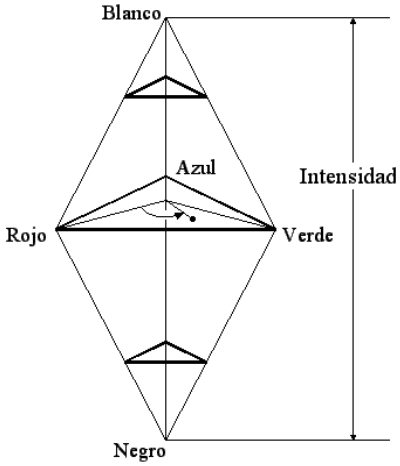


Figura 2.13 Espacio de color del modelo HSI

Para cualesquiera tres componentes de color RGB normalizadas en el intervalo [0,1], las componentes de intensidad, matiz y saturación se obtienen a través de las ecuaciones (2.7), (2.8) y (2.9), respectivamente.

$$I = \frac{1}{3}(R+G+B) \tag{2.7}$$

$$H = \cos^{-1}\left[\frac{[(R-G)+(R-B)]/2}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right] \tag{2.8}$$

$$S = 1 - \frac{3}{R+G+B}\min(R,G,B) \tag{2.9}$$

Existen modelos de color similares al HSI, entre los cuales se encuentran matiz, saturación y luminosidad (HSL), matiz, saturación y valor (HSV) y matiz, saturación y brillo (HSB). Aunque los términos intensidad, luminosidad, valor y brillo son usualmente utilizados como sinónimos, de hecho se refieren a diferentes formas de representar la brillantez de un color [1].

2.3 Procesamiento Digital de Imágenes

Procesamiento de Imágenes es un término general que se aplica a una amplia variedad de técnicas utilizadas para la manipulación y modificación de imágenes. Los fotógrafos y los físicos pueden realizar cierto Procesamiento de Imágenes usando sustancias químicas y equipo óptico, sin embargo, en este trabajo únicamente se hablará sobre el procesamiento que se realiza sobre imágenes digitales usando computadoras, es decir, sobre el *Procesamiento Digital de Imágenes* (PDI) [1].

Durante los últimos cuarenta años se han desarrollado numerosos y muy variados tipos de operaciones de PDI, como técnicas para mejorar la calidad de una imagen y técnicas para extraer información de ella de forma automática. Cualquiera que sea la operación, los pasos a seguir son los mismos: una técnica digital es aplicada sobre una imagen para formar un resultado digital, como una nueva imagen o una lista de datos [4].

2.3.1 Inicios

En la década de los años veinte algunas fotografías fueron digitalizadas para ser transmitidas a través del océano atlántico vía cable submarino con el propósito de utilizarlas en periódicos, sin embargo, esta aplicación no es considerada como resultado del PDI de acuerdo con la definición actual, ya que en su creación no se utilizaron computadoras, por lo tanto, la historia del PDI comienza con el desarrollo de las computadoras digitales y de la tecnología para almacenar, visualizar y transmitir datos [2].

Las primeras computadoras que fueron lo suficientemente poderosas para llevar a cabo tareas de PDI aparecieron a principios de los años sesentas. La disponibilidad de esas computadoras y el nacimiento del programa espacial fueron la primera motivación para el desarrollo de técnicas de PDI. La primera vez que se utilizó una computadora para el mejoramiento de imágenes fue en el Laboratorio de Propulsión a Chorro (Pasadena, California) de la NASA (National Aeronautics and Space Administration) en 1964, donde se procesaron fotografías de la superficie lunar transmitidas por la nave espacial *Ranger 7* para corregir distorsiones geométricas inherentes a la cámara de televisión que la nave llevaba a bordo [2].

Luego de procesar digitalmente aquellas fotografías con éxito, la NASA continuó financiando la investigación y el desarrollo en el campo del PDI para apoyar sus demás programas espaciales hasta que la formación de imágenes digitales y el PDI se convirtieron en parte integral de sus programas de exploración planetaria [7].

En paralelo con las aplicaciones espaciales, las técnicas de PDI comenzaron a ser usadas a finales de los años sesentas y principios de los setentas en la generación de

imágenes médicas, en observaciones remotas de los recursos de la Tierra y en astronomía [2].

2.3.2 Clases fundamentales de PDI

Las operaciones de PDI se pueden agrupar en cinco clases fundamentales: mejoramiento, restauración, análisis, compresión y síntesis. Una aplicación determinada puede hacer uso de una o varias de estas clases [4].

El *mejoramiento de imágenes* se encuentra entre las más simples y las más usadas de las áreas del PDI. Básicamente, la idea detrás del mejoramiento de imágenes es resaltar detalles que están oscurecidos o resaltar características de interés de la imagen. Las operaciones de mejoramiento se pueden utilizar para mejorar el contraste, las características de brillo o reducir el contenido de ruido de la imagen. Es importante resaltar que el mejoramiento de imágenes es un área más bien subjetiva, ya que la evaluación de los resultados depende de la apreciación del observador [2].

La *restauración de imágenes* es un área que también está relacionada con el mejoramiento de la apariencia de las imágenes, sin embargo, a diferencia del mejoramiento que está basado en apreciaciones humanas subjetivas, la restauración de imágenes se basa en modelos matemáticos y probabilísticos de los tipos de degradación que podría alterarlas [2]. Usualmente, las imágenes que son apropiadas para la restauración son aquellas que han sufrido de alguna forma de distorsión inherente al sistema de adquisición, como son distorsión geométrica, enfoque inapropiado, ruido electrónico o movimiento de la cámara durante la captura [4].

Las operaciones de *análisis de imágenes* no producen resultados en forma de imágenes, en vez de eso producen información numérica o gráfica basada en las características de la imagen original. Primero, separa la imagen en objetos individuales y luego los clasifica utilizando algún proceso de medición. Las operaciones de *análisis* incluyen la clasificación de objetos, mediciones automatizadas y la descripción y extracción de características de una imagen en su conjunto [4].

El objetivo de la *compresión de imágenes* es reducir la cantidad de memoria necesaria para guardar una imagen o el ancho de banda requerido para transmitirla [2]. La compresión de imágenes es posible debido a que la mayoría de las imágenes contienen una gran cantidad de información redundante, las operaciones de compresión eliminan esa redundancia. Una vez que una imagen ha sido comprimida, ésta puede ser descomprimida con una operación inversa para restaurarla a su forma original [4].

Las operaciones de *síntesis* crean imágenes a partir de otras o de datos numéricos que no provienen de imágenes. Estas operaciones se utilizan cuando se necesita una imagen que es impráctica de adquirir o que simplemente no existe en una forma física en absoluto [4].

2.3.3 Aplicaciones

Actualmente, existe un gran número de aplicaciones de PDI utilizadas en un amplio espectro de actividades humanas, desde el análisis de imágenes satelitales de la superficie

terrestre hasta la generación de imágenes biomédicas [7]. En esta sección únicamente se mencionan algunas de estas aplicaciones.

En la industria manufacturera los sistemas automáticos de inspección visual mejoran la productividad y la calidad de los productos al analizar características predeterminadas para buscar defectos y variaciones en el proceso [4].

Se aplican técnicas de PDI en imágenes tomadas con sensores instalados en satélites artificiales y en aviones para obtener información relacionada con los recursos agrícolas, hidráulicos, forestales, geológicos y minerales de la Tierra. Estas técnicas también se aplican en imágenes satelitales para la planeación de ciudades, movilización de recursos, control de inundaciones y monitoreo de la producción agrícola [7].

Varios tipos de dispositivos formadores de imágenes usados para auxiliar en el diagnóstico médico, como rayos-X y tomografías, hacen un amplio uso de las técnicas de PDI [4]. Uno de los eventos más importantes en la aplicación del PDI en el diagnóstico médico fue la invención a principios de los años setentas de la Tomografía Axial Computarizada (CAT), la cual consiste en la generación de una imagen tridimensional del interior de un objeto, usualmente un cuerpo humano, a partir de radiografías individuales [2].

Las agencias de inteligencia de Estados Unidos y de otros países utilizan las técnicas de PDI para interpretar automáticamente imágenes satelitales en busca de posibles amenazas militares y el ejército las utiliza para rastrear objetivos en tiempo real en sistemas de guía de misiles [4].

En la exploración espacial y la astronomía se utilizan técnicas de PDI para corregir defectos en las imágenes originales, tales como respuestas no lineales de los sensores, distorsiones geométricas y ruido, para detectar características que cambian a lo largo del tiempo como actividad solar y otros eventos cósmicos, y para la creación de representaciones tridimensionales del terreno de otros planetas a partir de datos de elevación recogidos por satélites exploradores o vehículos espaciales [4].

2.4 Resumen

En este capítulo se describió la forma en que se transforma una imagen de tono continuo en un arreglo de datos numéricos para procesarla con una computadora, se describieron los Modelos de Color utilizados más frecuentemente para representar con datos numéricos los colores de una imagen digital y se mencionaron las clases fundamentales de operaciones que existen para procesar imágenes, así como los orígenes del procesamiento digital de imágenes y algunas de sus aplicaciones más importantes.

Capítulo 3
Compresión de Imágenes

Una de las tecnologías que ha hecho posible la revolución multimedia de la que hemos sido testigos en la última década y que incluye al Internet, la telefonía móvil, los reproductores de MP3, la televisión digital y la videocomunicación es la compresión de datos.

La compresión de datos es necesaria debido a que una gran cantidad de información que usamos y generamos todos los días se encuentra en forma digital y el número de bytes requerido para representarla puede ser muy grande. Por ejemplo, para almacenar dos minutos de música digital con calidad de CD sin compresión se necesitarían más de diez megabytes y para almacenar únicamente un segundo de video digital más de veinte [12].

Aunque la capacidad de almacenamiento y transmisión de datos se incrementa continuamente con las nuevas innovaciones tecnológicas, el ritmo al que se genera nueva información que requiere de ser transmitida y almacenada es de al menos el doble, lo que ha provocado que la compresión de datos se haya convertido en parte integral de casi toda la tecnología de la información [12].

Una de las aplicaciones más importantes de la compresión de datos es la compresión de imágenes. En este capítulo se describen algunas de las técnicas más utilizadas y eficientes de compresión de imágenes y la forma en que los formatos de archivos de imágenes y los estándares de compresión las utilizan.

3.1 Compresión sin pérdidas

La compresión de imágenes se puede describir como un método que toma una imagen inicial *D* y genera una representación de ella que ocupa menos espacio *c(D)*. El proceso inverso, llamado descompresión, toma la imagen comprimida *c(D)* y reconstruye la imagen *D'*, estos procesos se ilustran en la figura 3.1.

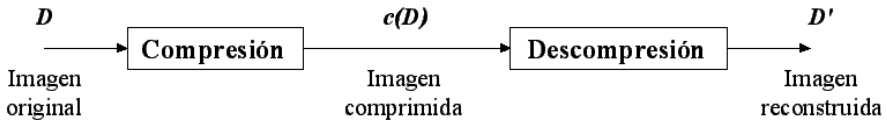


Figura 3.1 Procesos de compresión y descompresión

Si la imagen *D'* es una replica exacta de la imagen original *D*, el método aplicado para la compresión es llamado *sin pérdidas*, en cambio, si *D'* no es una copia exacta de *D*, sino simplemente una aproximación, entonces el método de compresión utilizado es llamado *con pérdidas*. En algunas aplicaciones de compresión de imágenes no se puede permitir la utilización de métodos de compresión con pérdidas, como en la generación de imágenes médicas, donde la utilización de un método de compresión con pérdidas podría ocasionar

una falta de precisión en el diagnóstico [7], o en las imágenes satelitales, donde el costo de su obtención y su utilización hace indeseable cualquier tipo de pérdida [2].

Algunas de las técnicas de compresión sin pérdidas actualmente en uso son la Codificación por Longitud de Series, las codificaciones de diccionario, la Codificación Huffman y la Codificación Aritmética. Estas técnicas normalmente proveen tasas de compresión del 50 al 10% [2]. Las dos primeras se basan en el hecho de que los píxeles en la mayoría de las imágenes no son independientes entre sí, sino que existe una correlación muy grande entre ellos, y las dos últimas buscan una forma distinta de representar a los píxeles que disminuya la redundancia que existe en su representación original [13].

3.1.1 Codificación por Longitud de Series

La Codificación por Longitud de Series aprovecha el hecho de que en la mayoría de las imágenes, sobre todo en las regiones donde no hay bordes, los valores de píxeles cercanos entre sí son idénticos o la variación entre ellos es muy pequeña [7]. Esta técnica consiste en representar una serie de píxeles idénticos únicamente con su longitud y el valor de los píxeles, por ejemplo, si en una imagen nos encontráramos con la secuencia de píxeles d:

d = 71 71 71 71 71 71 71 71 60 60 60 60 60 60 60 60 60 60 9 9 9 9 9 9 9 9 9 9 9 9 32 32 32 32

observaríamos que contiene series largas de píxeles con valores 71, 60, 9 y 32, de modo que en vez de almacenar individualmente cada píxel podríamos representar la secuencia de píxeles d únicamente indicando el valor del píxel y la longitud de la serie, obteniéndose la secuencia más compacta d':

d' = (71 8) (60 10) (9 12) (32 4)

donde cada par se colocó entre paréntesis con el único fin de hacer más compresible la representación.

En algunos casos, la aparición de series de píxeles no es apreciable en forma simple, sin embargo, éstos pueden preprocesarse con el propósito de ayudar a la Codificación por Longitud de Series [7]. Por ejemplo, la secuencia de píxeles d:

d = 15 17 19 21 23 26 29 32 35 38 41 44 50 56 62 68 74 85 96 107 118 115 112 109 106 103 100

puede ser preprocesada simplemente obteniendo la diferencia entre ellos de acuerdo con la ecuación (3.1) [7]

$$e(i) = d(i) - d(i-1) \qquad \textbf{(3.1)}$$

para obtener la secuencia e:

e = 15 2 2 2 2 3 3 3 3 3 3 3 6 6 6 6 6 11 11 11 11 -3 -3 -3 -3 -3 -3

la cual puede ser fácilmente codificada por longitud de series con lo que se obtendría la secuencia e':

e' = (15 1) (2 4) (3 7) (6 5) (11 4) (-3 6)

Por otro lado, en las imágenes binarias, como las que se generan al enviar un fax, la aparición de series de valores idénticos es bastante común ya que este tipo de imágenes consisten únicamente de 0's y 1's [7]. Por ejemplo, si la secuencia d representa un segmento de una imagen binaria,

d = 000000001111111000000011111111111111110000000111111000000000000000

la Codificación por Longitud de Series la representaría guardando únicamente las longitudes de las series de 0's y 1's como se muestra en la secuencia d':

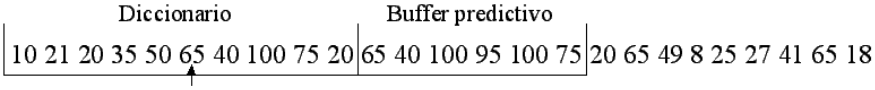
d' = 8 7 7 17 7 6 13

3.1.2 Codificaciones de diccionario

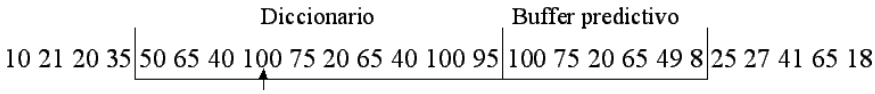
Los algoritmos de codificación de diccionario tienen su origen en los trabajos de Abraham Lempel y Jacob Ziv de 1977 y 1978. Los algoritmos basados en su trabajo de 1977 son conocidos como algoritmos LZ77 y los basados en su trabajo de 1978 como algoritmos LZ78. En 1984, Terry Welch de la compañía Sperry (ahora Unisys) realizó una mejora en el algoritmo LZ78 haciendo más práctica y eficiente su implementación y el algoritmo resultante se conoce como LZW [13].

En las codificaciones de diccionario, los datos comprimidos son índices que apuntan a algún elemento de un buffer o lista que contiene secuencias de valores de píxeles [3].

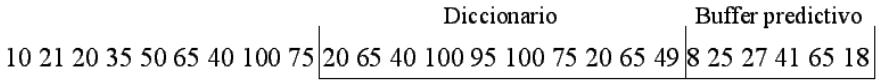
En la figura 3.2 se muestra la Codificación LZ77 con un ejemplo que contiene los tres casos que se pueden presentar durante la compresión, utilizando una secuencia de valores de píxeles y una ventana deslizante que se compone de dos partes: un buffer con diez píxeles que ya han sido comprimidos, llamado *diccionario*, y un buffer con seis píxeles que van a ser comprimidos, llamado *buffer predictivo*.



a) Los datos (5,3,95) representan la secuencia 65 40 100 95



b) Los datos (7, 4, 49) representan la secuencia 100 75 20 65 49



c) Los datos (0, 0, 8) representan al píxel con valor 8

Figura 3.2 Codificación LZ77

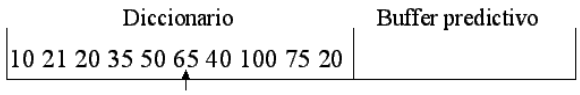
El proceso comienza con un apuntador que busca en el diccionario un píxel que tenga el mismo valor que el primer píxel del buffer predictivo, en la figura 3.2a sería el 65. Al

encontrarlo, el compresor observa que la secuencia 65 40 100 que está en el buffer predictivo también se encuentra en el diccionario, entonces desplaza la ventana deslizante cuatro lugares a la izquierda (figura 3.2b) y la secuencia 65 40 100 95 se agrega a los datos comprimidos representándola con tres valores, la distancia que existe entre el primer píxel del buffer predictivo y el apuntador (5), la longitud de la secuencia que se encontró tanto en el diccionario como en el buffer predictivo (3) y el valor del siguiente píxel en el buffer predictivo (95), por lo tanto, la representación de la secuencia 65 40 100 95 sería: (5, 3, 95).

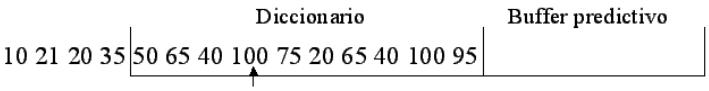
El segundo caso que se puede presentar es que el apuntador encuentre en el diccionario más de un píxel con el mismo valor que el primer píxel del buffer predictivo, como se muestra en la figura 3.2b; entonces, si ambos píxeles dan inicio a una secuencia que también se encuentre en el buffer predictivo, el compresor escoge aquel que dé inicio a la secuencia más larga, en este caso la secuencia más larga es la que comienza siete lugares a la izquierda del buffer predictivo y que es 100 75 20 65 y su representación sería (7, 4, 49).

En la figura 3.2c se muestra el tercer caso, que es cuando el apuntador no encuentra en el diccionario ningún píxel con el mismo valor que el primer píxel del buffer predictivo, entonces la ventana deslizante se desplaza un lugar a la derecha y el primer píxel del buffer predictivo se representa de esta forma: (0,0,8), donde los dos ceros indican al descompresor que no se encontró ninguna coincidencia en el diccionario.

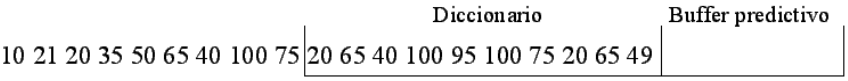
El proceso de descompresión se muestra en la figura 3.3, en la cual los píxeles en el diccionario ya han sido descomprimidos y el buffer predictivo está vacío.



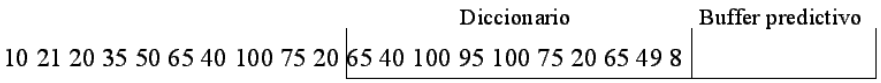
a) Los píxeles en el diccionario ya han sido descomprimidos



b) Después de descomprimir los datos (5, 3, 95)



c) Después de descomprimir los datos (7, 4, 49)



d) Después de descomprimir los datos (0, 0, 8)

Figura 3.3 Descompresión LZ77

En este punto, el descompresor lee los datos comprimidos (5,3,95), entonces el apuntador de descompresión se mueve 5 lugares a la izquierda a partir del buffer predictivo donde queda apuntando al píxel con valor 65 (figura 3.3a), copia tres píxeles, es decir, la secuencia 65 40 100, en el buffer predictivo, copia el píxel con valor 95 y recorre la ventana deslizante cuatro posiciones a la derecha como se muestra en la figura 3.3b.

Luego, el descompresor lee los datos (7,4,49), entonces el apuntador se mueve siete lugares a la izquierda a partir del buffer predictivo (figura 3.3b), copia cuatro píxeles en el buffer predictivo, agrega el píxel con valor 49 y desplaza la ventana deslizante cinco lugares a la derecha, como se muestra en la figura 3.3c.

Al leer los datos (0,0,8) el descompresor sabe que no hubo coincidencias en el diccionario, así que agrega el píxel con valor 8 y desplaza la ventana deslizante un lugar (figura 3.3d). El proceso continúa de esta forma hasta reconstruir la secuencia original.

En la Codificación LZW, el diccionario es una lista que se construye en un lugar separado de los píxeles a comprimir. Para mostrar cómo funciona la codificación LZW supongamos una imagen que contiene únicamente cuatro niveles de gris, 0, 1, 2 y 3, y que su primera línea está formada por la secuencia de píxeles: 2 3 1 2 1 3 1 0 3 1 2 1 3 1 0 2. Al comenzar la compresión el diccionario únicamente contiene los valores de píxeles 0, 1, 2 y 3, como se muestra en los primeros cuatro renglones de la tabla 3.1.

	Índice	Valores de píxel y secuencias	Datos comprimidos
Diccionario inicial	1	0	
	2	1	
	3	2	
	4	3	
	5	2 3	3
	6	3 1	4
	7	1 2	2
	8	2 1	3
	9	1 3	2
	10	3 1 0	6
	11	0 3	1
	12	3 1 2	6
	13	2 1 3	8
	14	3 1 0 2	10
	15	2	3

Tabla 3.1 Codificación LZW

Al tomar el valor del primer píxel, 2, el compresor observa que aparece en el diccionario, entonces toma el valor del siguiente píxel y forma la secuencia 2 3, la cual no está en el diccionario, por lo tanto, la agrega con el índice 5 y el primer dato comprimido es el índice del píxel con valor 2, es decir, el índice 3, como se muestra en la tercera columna de la tabla 3.1. Al tomar el valor del siguiente píxel, 3, observa que está en el diccionario y toma el valor del siguiente píxel, 1, para formar la secuencia 3 1 que, al no aparecer en el diccionario, agrega con el índice 6 y el nuevo dato comprimido es el índice 4.

Ahora veamos lo que pasa en el renglón 10 de la tabla 3.1. El valor del nuevo píxel es 3 y aparece anteriormente en el diccionario, así que toma el valor del siguiente píxel, 1, y dado que la secuencia 3 1 aparece con el índice 6 en el diccionario, toma el valor del siguiente píxel, 0, y forma la secuencia 3 1 0 que, al no aparecer antes en el diccionario, agrega con el índice 10 y el nuevo dato comprimido es el índice 6.

La cadena final de datos comprimidos es 3 4 2 3 2 6 1 6 8 10 3. El proceso de descompresión comienza con el mismo diccionario inicial, es decir, sólo con los primeros

cuatro renglones de la tabla 3.1. Cuando el primer dato comprimido, el índice 3, es leído, el descompresor identifica el valor de píxel 2. El segundo índice es 4, lo cual indica que el siguiente valor de píxel es 3, entonces el descompresor, siguiendo la misma regla que el compresor, agrega al diccionario la secuencia 2 3 con el índice 5. El siguiente dato comprimido es el 2, de modo que el siguiente valor de píxel es 1, y la nueva secuencia que se agrega al diccionario es 3 1. De este modo el descompresor construye el mismo diccionario que el compresor al mismo tiempo que descomprime los píxeles de la imagen.

En los dos ejemplos anteriores se puede observar que en las codificaciones de diccionario mientras más grande sea el diccionario y más largas las secuencias que contenga, mayor será la compresión.

3.1.3 Codificación Huffman

En 1952, D. A. Huffman desarrolló un método de compresión conocido como Codificación Huffman en el cual no se usa un número fijo de bits para representar símbolos, sino códigos de longitud variable conocidos como códigos Huffman [3]. A diferencia de las dos técnicas anteriores, la Codificación Huffman no aprovecha la redundancia que existe entre los píxeles de la imagen, sino la redundancia que existe en la forma de representarlos [13].

La codificación Huffman representa a los píxeles que aparecen con mayor frecuencia en la imagen con códigos pequeños y a los píxeles que aparecen con poca frecuencia con códigos más largos provocando una disminución en el promedio de bits utilizado para representar a cada píxel. A los dos píxeles menos frecuentes les asigna códigos de la misma longitud y que sólo son diferentes en el bit menos significativo [7].

En la tabla 3.2 se puede observar con un ejemplo el primer paso de la generación de códigos Huffman [2]. En la primera columna se colocan los valores que pueden tomar los píxeles de la imagen que se desea comprimir ordenándolos de acuerdo con su probabilidad de aparecer en la imagen, de la mayor a la menor. Luego, se realiza una serie de reducciones en el número de píxeles combinando los dos que tengan la probabilidad menor en un solo *símbolo compuesto* que los represente en la siguiente columna y que tiene una probabilidad igual a la suma de los dos píxeles que representa.

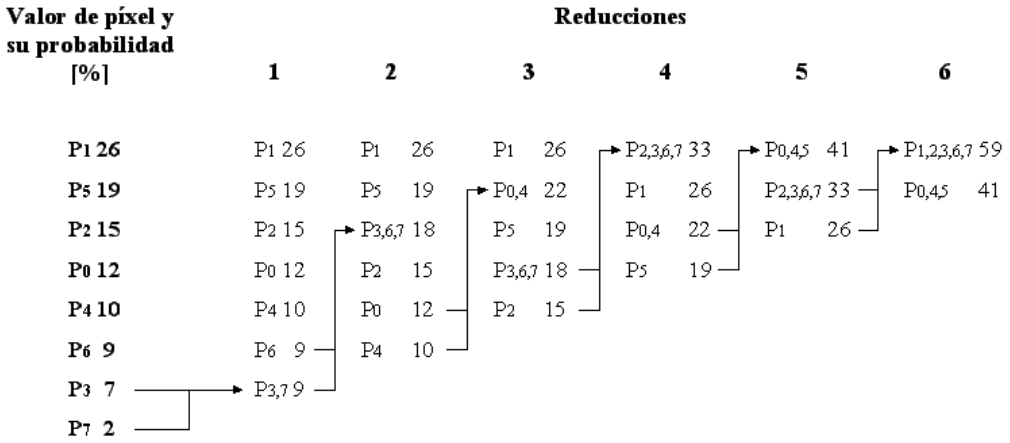


Tabla 3.2 Primer paso de la Codificación Huffman

Por ejemplo, para formar la primera reducción los dos valores de píxel con las probabilidades más pequeñas, P_3 y P_7 con probabilidades de 2% y 7%, respectivamente, se combinan para formar el símbolo compuesto $P_{3,7}$ con probabilidad de 9%, entonces éste nuevo símbolo se coloca en la primera columna de reducciones de modo que las probabilidades también queden ordenadas de la más alta a la más baja. Este proceso se repite hasta que únicamente quedan dos símbolos compuestos, en este caso $P_{1,2,3,6,7}$ y $P_{0,4,5}$.

El segundo paso de la codificación Huffman es asignar un código binario a cada símbolo, comenzando con la última columna hasta llegar al conjunto de valores de píxeles original. Los valores binarios 0 y 1 se asignan arbitrariamente a los dos símbolos de la última columna, como se muestra en la tabla 3.3. Como el símbolo compuesto $P_{1,2,3,6,7}$ se generó combinando el símbolo $P_{2,3,6,7}$ y el píxel P_1 de la columna anterior, el 0 utilizado para codificar a $P_{1,2,3,6,7}$ se asigna a $P_{2,3,6,7}$ y a P_1 y para distinguirlos se le agrega un 0 a $P_{2,3,6,7}$ del lado derecho y un 1 a P_1. Esta operación se repite siguiendo la misma ruta que en la tabla 3.3, pero en sentido inverso, hasta llegar al conjunto original de píxeles.

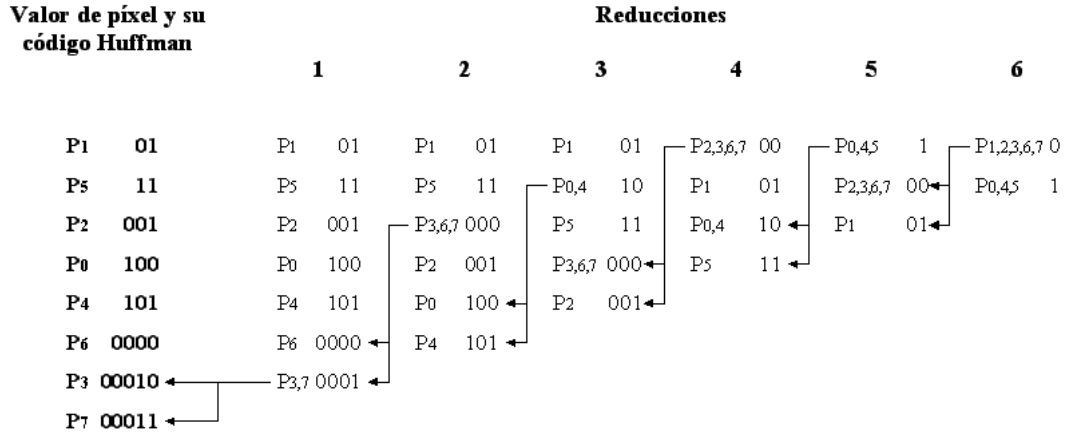


Tabla 3.3 Segundo paso de la Codificación Huffman

Para comprimir una secuencia de píxeles, se sustituye cada uno con el código Huffman que le corresponde de acuerdo con la primera columna de la tabla 3.3. Por ejemplo, la secuencia P_4 P_1 P_4 P_3 P_5 quedaría representada por 101 01 101 00010 11.

La descompresión de una cadena de códigos Huffman se realiza buscando de izquierda a derecha códigos válidos, es decir, códigos que se encuentren en la primera columna de la tabla 3.3. Por ejemplo, se puede observar que en los datos comprimidos 101011010001011 el primer código válido es el correspondiente al píxel P_4, 101, porque los códigos 1 y 10 no se encuentran en la primera columna de la tabla 3.3. El siguiente código válido es 01 y corresponde al píxel P_1. Al descomprimir la cadena completa se llega a la conclusión de que los píxeles originales eran P_4, P_1, P_4, P_3 y P_5.

3.1.4 Codificación Aritmética

La Codificación Aritmética es el resultado de la generalización práctica de varios algoritmos que tienen su origen en un trabajo no publicado de Peter Elias [2]. Este tipo de codificación, a diferencia de la Codificación Huffman, no convierte cada píxel de la imagen

en un código específico, sino que una secuencia completa de píxeles se representa con un intervalo de números reales contenido entre 0.0 y 1.0, el cual se vuelve más pequeño conforme los píxeles son procesados, los píxeles con las probabilidades más altas de aparecer en la imagen provocan una reducción menor que aquellos que tienen las probabilidades más bajas [7].

El primer paso en la codificación aritmética es asignar a cada valor de píxel un intervalo entre 0.0 y 1.0 de acuerdo con su probabilidad de aparecer en la imagen. En la tabla 3.4 se muestra un ejemplo de esta asignación en una imagen con cuatro niveles de gris.

Valor de píxel	Probabilidad [%]	Intervalo
0	35	[0.0, 0.35)
1	20	[0.35, 0.55)
2	30	[0.55, 0.85)
3	15	[0.85, 1.0)

Tabla 3.4 Asignación de intervalos en la Codificación Aritmética

Si se deseara comprimir la secuencia de píxeles *d* = 2, 0, 2, 3, 1 utilizando la tabla 3.4 se procedería de la siguiente forma: como el valor del primer píxel es 2, el intervalo que le corresponde se divide exactamente en las mismas proporciones que el intervalo original, con lo cual se obtendrían los nuevos intervalos [0.55 0.655), [0.655, 0.715), [0.715, 0.805) y [0.805, 0.85) correspondientes a los valores de píxel 0, 1, 2 y 3, respectivamente, como se muestra en la figura 3.4.

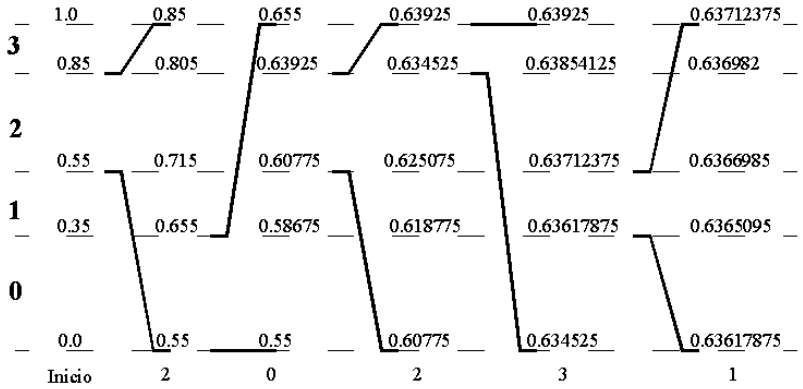


Figura 3.4 Codificación Aritmética

El valor del siguiente píxel es 0, por lo cual ahora el intervalo de interés es [0.55, 0.655) y se divide en las mismas proporciones que el intervalo original, como se muestra en la figura 3.4. Este proceso continua hasta procesar el último píxel y llegar a un intervalo final, en este caso el valor del último píxel es 1 y el intervalo final es [0.63617875, 0.63712375). Teóricamente, cualquier número dentro del intervalo final puede representar la secuencia de píxeles *d*, sin embargo, usualmente se utiliza su límite inferior [13].

La descompresión de la secuencia *d* se lleva a cabo de forma similar, es decir, partiendo del intervalo inicial [0.0, 1.0) dividido de la misma forma que en la tabla 3.4. Cuando el

descompresor recibe el número a = 0.63617875, éste sabe que el valor del primer píxel comprimido fue 2 porque a está dentro del intervalo [0.55 0.85), luego en este intervalo se realiza la misma división que en el proceso de compresión (figura 3.4) y el descompresor reconoce que el valor del segundo píxel es 0 porque a está en el intervalo [0.55, 0.655) y de ese mismo modo después reconocerá los valores de píxel 2, 3 y 1.

Si el descompresor conoce la longitud de la secuencia puede detenerse una vez que haya descomprimido el número correcto de píxeles, o podría detenerse al encontrar un símbolo especial que indicara el final de la secuencia y que fuera conocido tanto por el compresor como por el descompresor.

La implementación práctica de este método presenta el problema de que conforme el número de píxeles en la secuencia se incrementa, el intervalo correspondiente se vuelve más pequeño y el intervalo final podría reducirse tanto que ninguna computadora tendría precisión suficiente para representarlo [11]. Una solución es identificar y separar los decimales de los límites de los intervalos que después de procesar cierto número de píxeles ya no sufrirán cambios, por ejemplo, en la figura 3.4 puede observarse que a partir de que el píxel con valor 3 es comprimido, los límites de los nuevos intervalos siempre comenzarán con 0.63, entonces los decimales 6 y 3 se separan y se almacenan y los decimales restantes se recorren a la izquierda de modo que el nuevo intervalo de interés es [0.4525, 0.925) [9].

3.2 Compresión con pérdidas

Los métodos de compresión de imágenes con pérdidas proporcionan tasas de compresión hasta 15 veces mayores que los métodos de compresión sin pérdidas, sin embargo, la información que se pierde durante el proceso de compresión provoca que la imagen reconstruida no sea una copia exacta de la original [2].

La compresión con pérdidas usualmente se utiliza en aplicaciones donde el objetivo primordial es obtener las tasas de compresión más altas posibles sin que las pérdidas afecten de forma apreciable la calidad visual de la imagen. Esto último es posible debido a que el sistema visual humano tiene limitaciones que le impiden notar cierta cantidad de diferencias entre la imagen original y la imagen reconstruida, por ejemplo, a nuestros ojos les resulta sumamente difícil notar cambios pequeños en los píxeles de una región sin bordes [7], distinguir entre colores casi idénticos o variaciones en la imagen demasiado rápidas [3].

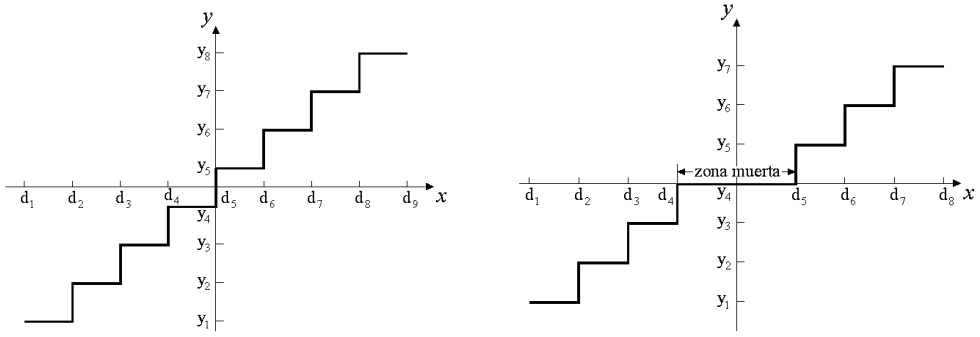
Un componente necesario de los métodos de compresión con pérdidas y que tiene un impacto directo en la tasa de compresión y en la distorsión de la imagen reconstruida es un proceso llamado Cuantización [13], el cual da mejores resultados si se aplica no a la imagen en sí misma sino a una representación de ésta que se obtiene al aplicarle alguno de los procedimientos conocidos como transformadas, las cuales permiten describir imágenes como la suma de un conjunto de funciones base multiplicadas por el conjunto de coeficientes que generan dichas transformadas [7].

Aunque las transformadas no dan como resultado ninguna compresión por sí mismas ya que al ser aplicadas a una imagen con una determinada cantidad de píxeles generan un número igual de coeficientes, tienen el efecto de concentrar en pocos coeficientes la mayor parte de la información importante para la calidad visual de la imagen [9] y la compresión se consigue descartando mediante la Cuantización aquellos coeficientes que tienen los valores más pequeños y, por lo tanto, cooperan muy poco con la calidad de la imagen [11].

3.2.1 Cuantización

La Cuantización consiste en transformar datos de entrada en un conjunto finito de valores [13] y aplicada a la compresión de imágenes tiene el propósito de remover de la imagen la información que no es importante para su calidad visual y conservar aquella que sí lo es [9]. En general, la Cuantización es un proceso irreversible debido a que no existe forma de recuperar los datos originales a partir de los valores cuantizados.

En la figura 3.5 se muestran dos gráficas que ilustran la relación que existe entre los datos de entrada, representados en el eje *x*, y los valores cuantizados, representados en el eje *y*.



a) Cuantizador uniforme sin zona muerta *b) Cuantizador uniforme con zona muerta*

Figura 3.5 Cuantización Uniforme

Los límites de los intervalos en que se divide el eje horizontal de las gráficas de la figura 3.5 se conocen como *niveles de decisión* (d_i), la longitud de estos intervalos se conoce como *tamaño de paso* y los valores cuantizados son conocidos como *niveles de reconstrucción* (y_i). A los datos de entrada se les asigna el valor cuantizado que les corresponda de acuerdo con el intervalo en el que se encuentren, como se muestra en las gráficas de la figura 3.5 y de acuerdo con la ecuación (3.2) [13].

$$y_i = Q(x) \quad si \quad x \in (d_i, d_{i+1}) \tag{3.2}$$

El diseño de un cuantizador consiste en escoger el número de niveles de reconstrucción y en seleccionar los niveles de decisión, mientras menor sea el número de niveles de reconstrucción, mayores serán la compresión y la diferencia entre la imagen original y la reconstruida. El tipo de Cuantización más sencilla y más utilizada es la *Cuantización Uniforme*, en la cual, excepto posiblemente por los extremos y la zona central, todos los niveles de reconstrucción están espaciados uniformemente y todos los intervalos tienen el mismo tamaño de paso [13].

En la figura 3.5 se muestran los dos tipos de Cuantización Uniforme que existen. En la figura 3.5b se muestra un *cuantizador uniforme con zona muerta*, el cual se distingue del *cuantizador uniforme sin zona muerta* de la figura 3.5a en que su gráfica cuenta con una región central en la cual los datos de entrada son cuantizados a cero [11].

Existen otros dos tipos de Cuantización, la *No Uniforme* y la *Adaptativa*, las cuales aprovechan la probabilidad que tienen los diferentes datos de entrada de aparecer en la

imagen para asignar a los más probables un mayor número de niveles de reconstrucción y, de ese modo, disminuir la distorsión en la mayoría de los píxeles y aumentarla en una cantidad pequeña de ellos dando como resultado una disminución en la distorsión total de la imagen.

La Cuantización No Uniforme aprovecha las estadísticas disponibles para construir su esquema de Cuantización, por ejemplo, la figura 3.6 muestra una Cuantización en la que la mayoría de los datos están centrados alrededor de 0.

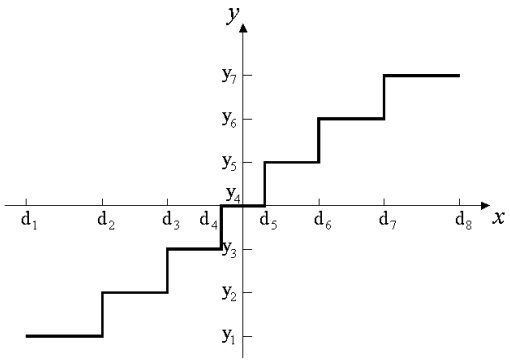


Figura 3.6 Gráfica de un cuantizador No Uniforme

La Cuantización Adaptativa, a diferencia de las Cuantizaciones Uniforme y No Uniforme, diseña el cuantizador durante el proceso, de modo que éste se adapta a las características probabilísticas de cada conjunto de datos de entrada [13].

3.2.2 El espacio vectorial $\mathbf{C^n}$ y la Transformada Discreta de Fourier

En esta sección veremos que la aplicación de uno de los procedimientos conocidos como transformadas a la secuencia de píxeles que forman una línea o una columna de una imagen es posible debido a que dicha secuencia se puede considerar como un vector del espacio vectorial $\mathbf{C^n}$ y en las secciones siguientes veremos la forma en que las transformadas son utilizadas en la compresión de imágenes, para ello comenzaremos definiendo qué es un espacio vectorial y analizando algunas de las propiedades del espacio vectorial $\mathbf{C^n}$.

El espacio vectorial $\mathbf{C^n}$

Un espacio vectorial sobre el campo de los números complejos es un conjunto de objetos, llamados vectores, para los cuales están definidas dos operaciones, la adición de vectores y la multiplicación de vectores por un escalar, en este caso dicho escalar es un número complejo [16]. Un ejemplo de espacio vectorial es el espacio $\mathbf{C^n}$, cuyos vectores tienen la forma de la ecuación (3.3)

$$z = (z_0, \quad z_1, \quad \ldots, \quad z_{n-1}) \qquad \textbf{(3.3)}$$

donde $z_0, z_1, \ldots, z_{n-1}$ son números complejos.

Una *combinación lineal* es un vector *z* que se genera al sumar un conjunto de vectores $v_0, v_1, \ldots, v_{n-1}$ multiplicados por un conjunto de escalares $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ como se muestra en la ecuación (3.4)

$$z = \sum_{i=0}^{n-1} \alpha_i v_i = \alpha_0 v_0 + \alpha_1 v_1 + \ldots + \alpha_{n-1} v_{n-1} \tag{3.4}$$

Una *base del espacio vectorial* $\mathbf{C^n}$ es un conjunto de vectores $v_0, v_1, \ldots, v_{n-1}$ que pertenecen a este espacio y que mediante una combinación lineal pueden generar cualquiera de los vectores de $\mathbf{C^n}$. El ejemplo más sencillo de una base del espacio vectorial $\mathbf{C^n}$ es la *base euclidiana de* C^n [16], la cual se define como

$$E = \{e_0, \quad e_1, \quad \ldots, \quad e_{n-1},\} \tag{3.5}$$

donde

$$e_0 = (1, \ 0, \ 0, \ \ldots, \ 0), \ e_1 = (0, \ 1, \ 0, \ \ldots, \ 0), \ \ldots, \ e_{n-1} = (0, \ 0, \ 0, \ \ldots, \ 1) \tag{3.6}$$

Por ejemplo, el vector

$$z = (1, \quad 3, \quad 2, \quad -1) \tag{3.7}$$

se puede generar con la combinación lineal de la *base euclidiana de* $\boldsymbol{C^4}$ como se muestra en la ecuación (3.8).

$$z = 1(1, \ 0, \ 0, \ 0) + 3(0, \ 1, \ 0, \ 0) + 2(0, \ 0, \ 1, \ 0) + (-1)(0, \ 0, \ 0, \ 1) \tag{3.8}$$

El espacio vectorial $\mathbf{C^n}$ posee una operación entre vectores denominada *producto interno,* la cual se define de acuerdo con la ecuación (3.9) [16]

$$\langle z, w \rangle = \sum_{j=0}^{n-1} z_j \overline{w}_j \tag{3.9}$$

donde $\overline{w}_j$ es el complejo conjugado del elemento *j* del vector *w*.

Dos vectores *z* y *w* son *ortogonales* si el producto interno entre ellos es igual a cero y una base es ortogonal si todos sus vectores son ortogonales entre sí [16]. Con un análisis simple de la ecuación (3.6) puede observarse que la base euclidiana para $\mathbf{C^n}$ es ortogonal.

La ecuación (3.10) muestra cómo generar un vector *z* mediante una combinación lineal de la base ortogonal $B=\{v_0, v_1,\ldots,v_k,\ldots,v_{n-1}\}$.

$$z = \sum_{i=0}^{n-1} \alpha_i v_i = \alpha_0 v_0 + \alpha_1 v_1 + \ldots + \alpha_k v_k + \ldots + \alpha_{n-1} v_{n-1} \tag{3.10}$$

Para saber el valor del coeficiente α_k de la ecuación (3.10) podemos aplicar en ambos lados de la ecuación la operación producto interno con el vector v_k, como se muestra en la ecuación (3.11),

$$\langle z, v_k \rangle = \alpha_0 \langle v_0, v_k \rangle + \alpha_1 \langle v_1, v_k \rangle + \ldots + \alpha_k \langle v_k, v_k \rangle + \ldots + \alpha_{n-1} \langle v_{n-1}, v_k \rangle \tag{3.11}$$

dado que *B* es una base ortogonal, todos los productos internos de la ecuación (3.11) son cero, excepto por $\langle z, v_k \rangle$ y $\langle v_k, v_k \rangle$, y la ecuación (3.11) se reduce a

$$\langle z, v_k \rangle = \alpha_k \langle v_k, v_k \rangle$$
$$\alpha_k = \frac{\langle z, v_k \rangle}{\langle v_k, v_k \rangle} \quad \textbf{(3.12)}$$

El producto interno de un vector *v* consigo mismo se denomina *la norma del vector v*, y se representa como se muestra en la ecuación (3.13). Si la norma de todos los vectores de una base ortogonal *B* es igual a uno, se dice que *B* es una *base ortonormal*.

$$\|v\| = \langle v, v \rangle \quad \textbf{(3.13)}$$

Si la base *B* es ortonormal, la ecuación (3.12) se reduce a

$$\alpha_k = \langle z, v_k \rangle \quad \textbf{(3.14)}$$

y la ecuación (3.10) queda de esta forma:

$$z = \sum_{i=0}^{n-1} \langle z, v_i \rangle v_i \quad \textbf{(3.15)}$$

Un vector *z* se puede representar con respecto a una base $B=\{v_0, v_1,\ldots,v_{n-1}\}$ con los coeficientes $\alpha_0, \alpha_1, \ldots, \alpha_{n-1}$ que se utilizan para generarlo mediante la combinación lineal de los vectores de dicha base [16].

La Transformada Discreta de Fourier

La Transformada Discreta de Fourier (TDF), la cual tiene su origen en el trabajo del matemático francés Jean Baptiste Joseph Fourier *La Teoría Analítica del Calor* publicado en 1822 [2], es un caso particular de la ecuación (3.14), ya que consiste en la obtención de los coeficientes que representan a un vector *z* del espacio $\mathbf{C^n}$ con respecto a la base ortonormal *F* que se muestra en la ecuación (3.16) [16]

$$F = \{E_0, \quad E_1, \quad \ldots, \quad E_{N-1}\} \quad \textbf{(3.16)}$$

donde

$$E_i(n) = \frac{1}{\sqrt{N}} \left[\cos\left(2\pi \frac{in}{N}\right) + j\,\mathrm{sen}\left(2\pi \frac{in}{N}\right) \right] \quad para \quad 0 \le n \le N-1 \quad \textbf{(3.17)}$$

y *N* es la longitud de los vectores del espacio $\mathbf{C^n}$. En la ecuación (3.18) se muestra la forma de obtener la TDF a partir de la ecuación (3.14).

$$\alpha_i = \langle z, E_i \rangle = \sum_{n=0}^{N-1} z(n)\overline{E}_i(n) = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} z(n) \left[\cos\left(2\pi \frac{in}{N}\right) - j\,\mathrm{sen}\left(2\pi \frac{in}{N}\right) \right]$$
$$= \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} z(n) e^{-j2\pi\, n\, i/N} \quad \textbf{(3.18)}$$

Por ejemplo, la base ortonormal *F* del espacio vectorial $\mathbf{C}^4$ sería

$$F = \{E_0, \quad E_1, \quad E_2, \quad E_3\} \tag{3.19}$$

donde

$$E_0 = (E_0(0), \quad E_0(1), \quad E_0(2), \quad E_0(3)) = \frac{1}{\sqrt{4}}(1, \quad 1, \quad 1, \quad 1) \tag{3.20a}$$

$$E_1 = \frac{1}{2}(1, \quad j1, \quad -1, \quad -j1) \tag{3.20b}$$

$$E_2 = \frac{1}{2}(1, \quad -1, \quad 1, \quad -1) \tag{3.20c}$$

$$E_3 = \frac{1}{2}(1, \quad -j1, \quad -1, \quad j1) \tag{3.20d}$$

y los coeficientes que representan al vector *z=(3, 5, -1, 2)* con respecto a *F* se pueden obtener utilizando la ecuación (3.18) como se muestra en la ecuación (3.21),

$$\alpha_0 = \langle z, E_0 \rangle = \frac{1}{\sqrt{4}}[3(1) + 5(1) + (-1)(1) + 2(1)] = 4.5 \tag{3.21a}$$

$$\alpha_1 = \langle z, E_1 \rangle = \frac{1}{2}[3(1) + 5(-j) + (-1)(-1) + 2(j)] = 2 - j1.5 \tag{3.21b}$$

$$\alpha_2 = \langle z, E_2 \rangle = \frac{1}{2}[3(1) + 5(-1) + (-1)(1) + 2(-1)] = -2.5 \tag{3.21c}$$

$$\alpha_3 = \langle z, E_3 \rangle = \frac{1}{2}[3(1) + 5(j) + (-1)(-1) + 2(-j)] = 2 + j1.5 \tag{3.21d}$$

La combinación lineal con la cual se genera al vector *z*, utilizando los coeficientes obtenidos en la TDF, se denomina Transformada Discreta de Fourier Inversa (TDFI) y se muestra en la ecuación (3.22).

$$\begin{aligned} z = \sum_{i=0}^{N-1} \alpha_i E_i(n) &= \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \alpha_i \left[\cos\left(2\pi \frac{in}{N}\right) + j\text{sen}\left(2\pi \frac{in}{N}\right) \right] \\ &= \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} \alpha_i e^{j2\pi\, n\, i/N} \end{aligned} \tag{3.22}$$

La secuencia de píxeles que forman una línea o una columna de una imagen tiene la estructura de la ecuación (3.3), por lo tanto, se considera un vector del espacio vectorial $\mathbf{C}^n$ y se puede representar con los coeficientes que se generan al aplicarle la TDF.

La Transformada Coseno Discreta (TCD) y la Transformada Wavelet Discreta (TWD) también permiten obtener los coeficientes que representan a una secuencia de píxeles con respecto a una base del espacio vectorial $\mathbf{C}^n$ y una de sus principales aplicaciones es la compresión de imágenes con pérdidas debido a que la mayoría de los coeficientes que estas transformadas generan tienen valores muy pequeños que se pueden descartar mediante Cuantización y se puede representar a la imagen con un número pequeño de coeficientes sin afectar de forma notable la calidad de la imagen reconstruida [11].

3.2.3 Transformada Coseno Discreta

La TCD fue propuesta en 1974 por Ahmed, Natarajan y Rao y desde entonces se ha convertido en una de las transformadas más utilizadas para comprimir imágenes, así como voz y audio, debido a que la forma en que las transforma facilita la compresión y a que puede ser implementada eficientemente tanto en software como en hardware [9].

La TCD es un caso especial de la TDF en el cual se ha conseguido hacer más sencilla y eficiente la aplicación de la transformada al eliminar la necesidad de realizar operaciones con números complejos. Una forma de obtener la TCD a partir de la TDF es extender de forma simétrica el vector *z* al que se aplicará la TCD [20], como se muestra en la figura 3.7, y formar la base ortonormal de la TCD a partir de las funciones base de la TDF desplazándolas $\frac{1}{2}$ como se muestra en la ecuación (3.23)

$$C = \{E'_0, \quad E'_1, \quad \ldots, \quad E'_{N-1},\} \tag{3.23}$$

donde

$$E'_i(n) = E_i\left(n+\frac{1}{2}\right) = \frac{1}{\sqrt{N}}\left[\cos\left(2\pi\frac{i\left(n+1/2\right)}{N}\right) + j\,\mathrm{sen}\left(2\pi\frac{i\left(n+1/2\right)}{N}\right)\right]$$

$$E'_i(n) = \frac{1}{\sqrt{N}}\left[\cos\left(\pi\frac{i(2n+1)}{N}\right) + j\,\mathrm{sen}\left(\pi\frac{i(2n+1)}{N}\right)\right] \tag{3.24}$$

y *N* es la longitud del vector *z*.

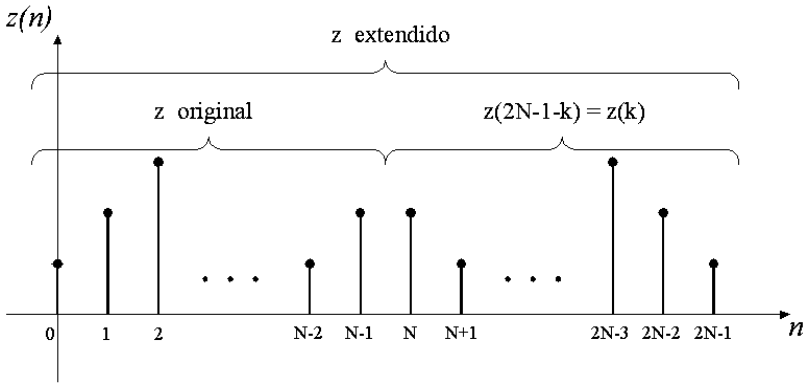


Figura 3.7 Extensión simétrica del vector z

La ecuación (3.25) muestra la forma en que se obtendrían los coeficientes del vector extendido *z* con respecto a la base ortonormal *C*,

$$\alpha_i = \langle z, E'_i \rangle = \frac{1}{\sqrt{2N}}\sum_{n=0}^{2N-1} z(n)\left[\cos\left(\pi\frac{i(2n+1)}{2N}\right) - j\,\mathrm{sen}\left(\pi\frac{i(2n+1)}{2N}\right)\right] \tag{3.25}$$

donde *N*, la longitud original de *z*, se ha sustituido por *2N* debido a que el vector extendido *z* tiene el doble de longitud que el vector *z* original.

Ahora veamos lo que sucede con los términos *k* y *2N - 1 - k* (con *k* menor a *N*) de la sumatoria de la ecuación (3.25),

$$\alpha'_i = z(k)\left[\cos\left(\pi\frac{i(2k+1)}{2N}\right) - j\,\text{sen}\left(\pi\frac{i(2k+1)}{2N}\right)\right] +$$

$$+ z(2N-1-k)\left[\cos\left(\pi\frac{i(2(2N-1-k)+1)}{2N}\right) - j\,\text{sen}\left(\pi\frac{i(2(2N-1-k)+1)}{2N}\right)\right] \quad \textbf{(3.26)}$$

en la figura 3.7 podemos observar que

$$z(2N-1-k) = z(k) \quad \textbf{(3.27)}$$

por lo tanto, sustituyendo la ecuación (3.27) en la ecuación (3.26) obtenemos

$$\alpha'_i = z(k)\left[\cos\left(\pi\frac{i(2k+1)}{2N}\right) + \cos\left(\pi\frac{i(4N-2-2k+1)}{2N}\right) - \right.$$
$$\left. - j\,\text{sen}\left(\pi\frac{i(2k+1)}{2N}\right) - j\,\text{sen}\left(\pi\frac{i(4N-2-2k+1)}{2N}\right)\right]$$

$$\alpha'_i = z(k)\left\{\cos\left(\pi\frac{i(2k+1)}{2N}\right) + \cos\left(2\pi i - \pi\frac{i(2k+1)}{2N}\right) - \right.$$
$$\left. - j\left[\text{sen}\left(\pi\frac{i(2k+1)}{2N}\right) + \text{sen}\left(2\pi i - \pi\frac{i(2k+1)}{2N}\right)\right]\right\} \quad \textbf{(3.28)}$$

Las funciones seno y coseno tienen las propiedades que se muestran en las ecuaciones (3.29) y (3.30),

$$\text{sen}(2\pi i - \beta) = -\text{sen}(\beta)\ ,\ i = 1,2,3,\ldots \quad \textbf{(3.29)}$$

$$\cos(2\pi i - \beta) = \cos(\beta)\ ,\ i = 1,2,3,\ldots \quad \textbf{(3.30)}$$

con la aplicación de las propiedades de las ecuaciones (3.29) y (3.30) en la ecuación (3.28) se obtiene

$$\alpha'_i = z(k)\left\{\cos\left(\pi\frac{i(2k+1)}{2N}\right) + \cos\left(\pi\frac{i(2k+1)}{2N}\right) - j\left[\text{sen}\left(\pi\frac{i(2k+1)}{2N}\right) - \text{sen}\left(\pi\frac{i(2k+1)}{2N}\right)\right]\right\}$$

$$\alpha'_i = 2z(k)\cos\left(\pi\frac{i(2k+1)}{2N}\right) \quad \textbf{(3.31)}$$

La ecuación (3.31) implica que las funciones seno de los primeros *N* términos de la ecuación (3.25) se eliminan con las funciones seno de los últimos *N* términos, por lo tanto, en la práctica los cálculos relacionados con las funciones seno no se realizan, únicamente se

calculan los primeros *N* coeficientes de la ecuación (3.25) y la TCD se aplica utilizando la ecuación (3.32)

$$\alpha_i = d_i \sum_{n=0}^{N-1} z(n) \cos\left(\pi \frac{i(2n+1)}{2N} \right) \tag{3.32}$$

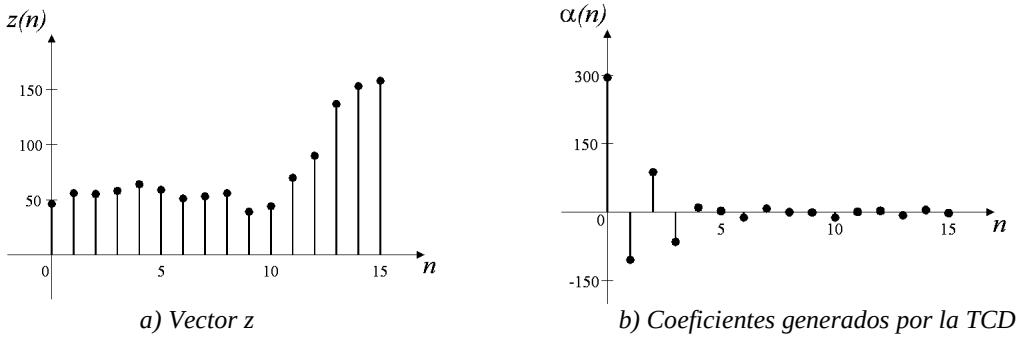
donde

$$d_i = \begin{cases} \frac{1}{\sqrt{N}} & si \quad i = 0 \\ \sqrt{\frac{2}{N}} & si \quad i = 1,\ldots,N-1 \end{cases} \tag{3.33}$$

La recuperación de la versión extendida del vector *z* podría realizarse mediante la combinación lineal de la base *C* usando los coeficientes obtenidos con la ecuación (3.32), sin embargo, en este caso las operaciones con las funciones seno también son innecesarias y sólo hace falta recuperar al vector *z* original, para lo cual se puede utilizar la ecuación (3.34), la cual recibe el nombre de Transformada Coseno Discreta Inversa (TCDI).

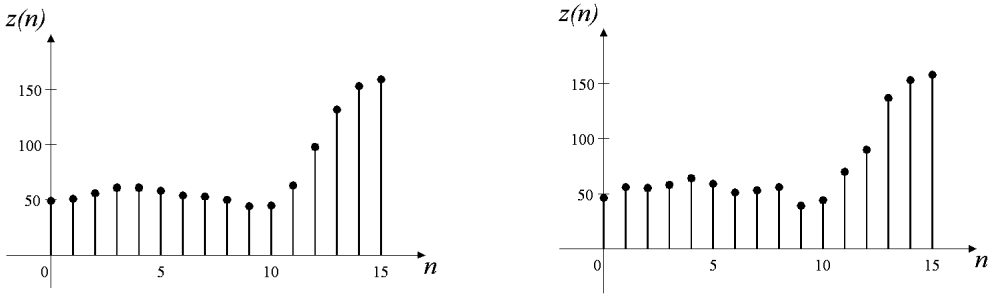
$$z(n) = d_i \sum_{i=0}^{N-1} \alpha_i \cos\left(\pi \frac{i(2n+1)}{2N} \right) \tag{3.34}$$

En la ecuación (3.24) se puede observar que los vectores de la base ortonormal *C*, que de ahora en adelante llamaremos *funciones base*, son funciones senoidales discretas de diferentes frecuencias, cada una de estas funciones representa las variaciones que existen en el vector al que se ha aplicado la TCD, las funciones con frecuencias bajas representan las variaciones graduales en el vector y las funciones con frecuencias altas representan las variaciones rápidas, por ejemplo, en la figura 3.8b se puede observar que de los coeficientes que se generan al aplicar la TCD al vector *z* que se muestra en la figura 3.8a los que tienen los valores mayores son los de las funciones base con frecuencias bajas (los más cercanos al extremo izquierdo), lo cual indica que en el vector *z* las variaciones que predominan en su formación son las variaciones lentas.



a) Vector z *b) Coeficientes generados por la TCD*

Figura 3.8 Aplicación de la TCD a un vector



c) TCDI con 8 coeficientes *d) TCDI con los 16 coeficientes*

Figura 3.8 Aplicación de la TCD a un vector (Continuación)

En la figura 3.8c se muestra la reconstrucción del vector *z* a través de la TCDI utilizando únicamente la primera mitad de los coeficientes generados y en la figura 3.8d se muestra la reconstrucción del mismo vector usando todos los coeficientes. Debido a que las variaciones lentas son las que predominan en el vector *z*, la gráfica de la figura 3.8c es casi idéntica a la del vector original a pesar de que únicamente se utilizó la mitad de los coeficientes.

En la mayoría de las imágenes, al igual que en el vector *z* del ejemplo anterior, las variaciones que predominan son las variaciones lentas, lo cual quiere decir que los coeficientes de las funciones base con frecuencias altas son pequeños y se pueden descartar sin afectar demasiado la calidad de la imagen reconstruida [3].

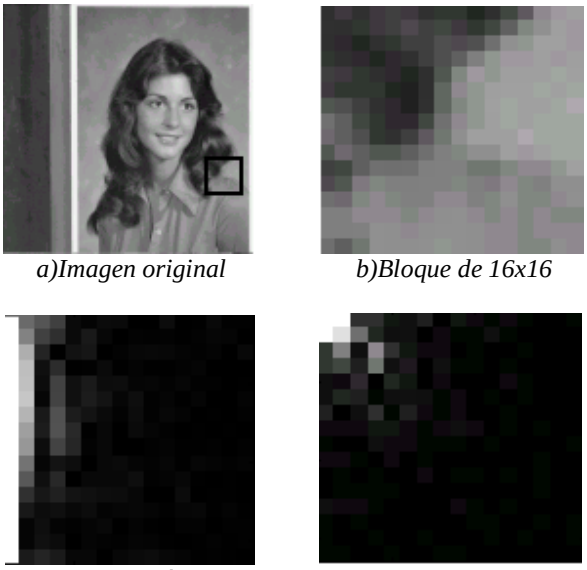
Para aplicar la TCD a una imagen mediante la ecuación (3.32) se debe sustituir cada una de sus líneas con los coeficientes que se generan al aplicarles la TCD individualmente, con lo cual se obtiene un arreglo de coeficientes que tiene las mismas dimensiones que la imagen original, y luego aplicar la TCD a las columnas de coeficientes y sustituirlas con los coeficientes generados.

El número de operaciones que se realizan al aplicar la TCD mediante la ecuación (3.32) a una imagen de N x N píxeles, suponiendo que ya se cuenta con los valores de las funciones coseno en una tabla, es el siguiente: en la ecuación (3.32) podemos observar que para calcular cada coeficiente se requieren N+1 multiplicaciones y N-1 sumas, lo que da 2N operaciones, cada línea genera N coeficientes, por lo tanto, requiere de $N(2N) = 2N^2$ operaciones, dado que la imagen contiene N líneas, para aplicar la TCD a todas ellas se requieren $N(2N^2) = 2N^3$ operaciones y para aplicar la TCD a las columnas de coeficientes se requiere el mismo número de operaciones, por lo tanto, para aplicar la TCD a una imagen de N x N píxeles se necesita realizar $4N^3$ operaciones. Por ejemplo, transformar una imagen de 128x128 píxeles requeriría de $4(128^3) = 8{,}388{,}608$ operaciones.

Con el propósito de reducir este número tan grande de operaciones, las imágenes se dividen en bloques y cada uno de ellos se transforma mediante la TCD de forma independiente [10]. Por ejemplo, si la imagen de 128x128 píxeles del ejemplo anterior se divide en 256 bloques de 8x8 píxeles, el número de operaciones para aplicar la TCD a cada bloque sería $4(8^3) = 2{,}048$ y para aplicar la TCD a todos los bloques únicamente se necesitarían $256(2{,}048) = 524{,}288$ operaciones. En la mayoría de las aplicaciones los bloques se eligen de 8x8 ó 16x16 píxeles [15].

En la figura 3.9a se muestra una imagen de la cual se ha tomado un bloque de 16 x 16 píxeles para aplicarle la TCD, el bloque se muestra amplificado en la figura 3.9b, los coeficientes resultantes de la aplicación de la TCD a las líneas del bloque se han

representado en escala de grises en la figura 3.9c, con los coeficientes mayores en color blanco y los coeficientes nulos en negro, y en la figura 3.9d se muestran los coeficientes resultantes de la aplicación de la TCD a las columnas de coeficientes de la figura 3.9c.



a)Imagen original *b)Bloque de 16x16*

c) TCD a las líneas *d) TCD a las columnas*

Figura 3.9 TCD a un bloque de 16x16 píxeles

En la figura 3.9d puede observarse que los coeficientes con mayor peso se concentran en la esquina superior izquierda, por lo tanto, los coeficientes de las funciones coseno con las frecuencias más bajas son los que más cooperan en la calidad visual de la imagen. Para recuperar la imagen a partir de los coeficientes de la figura 3.9d se aplica la TCDI primero a las columnas y a luego a las líneas.

3.2.4 Transformada Wavelet Discreta

En las secciones anteriores hemos visto que las funciones base de la TDF y la TCD pueden usarse para representar las variaciones de diferentes frecuencias que componen una imagen y que los coeficientes que generan estas transformadas nos dicen cuanto coopera cada una de estas variaciones en su formación, sin embargo, no se puede saber a través de estos coeficientes en qué parte de la imagen existen variaciones rápidas o en qué parte existen variaciones lentas, esto se debe a que las funciones base de la TDF y TCD no están localizadas en el espacio, es decir, son diferentes de cero en toda sus extensión [14].

La TWD se diferencia de la TDF y la TCD por utilizar funciones base localizadas en el espacio, es decir, son diferentes de cero únicamente en un intervalo pequeño. Este tipo de funciones base tienen la ventaja de permitir un análisis local de los vectores, por ejemplo, al obtener el coeficiente α_k que representa a un vector z con respecto a una base Wavelet ortonormal $B=\{v_0, v_1,\ldots,v_{n-1}\}$ mediante la ecuación (3.14), la mayoría de los términos de la sumatoria serían iguales a cero, por lo tanto, el coeficiente α_k únicamente nos daría información acerca del vector z en el intervalo donde la función base v_k es diferente de cero

y si el valor del coeficiente α_k es mayor que los demás o tiene alguna otra característica de interés, sería posible identificar el lugar del vector con el que está asociado ese coeficiente y analizarlo con más detalle [16].

El término Wavelet fue usado por primera vez en el campo de la sismología para describir las perturbaciones que emanan de un impulso sísmico, sin embargo, los orígenes de la teoría Wavelet se encuentran en diversas técnicas que se desarrollaron de forma independiente en diversas aplicaciones del procesamiento de señales, como el filtrado subbanda, usado en compresión de imágenes y audio; el análisis multirresolución, usado en visión por computadora; y la expansión en series Wavelet, usada en matemáticas aplicadas, y que recientemente han sido reconocidas como diferentes puntos de vista de una misma teoría [15].

Los fundamentos matemáticos de la teoría Wavelet en señales continuas fueron construidos a mediados de los años ochenta por un grupo de investigadores del Centro de Física Teórica de Marsella encabezado por el geofísico Jean Morlet, el físico teórico Alex Grossman y el matemático Y. Meyer. Las conexiones con los resultados que hasta entonces se habían obtenido en las señales discretas fueron establecidas por Ingrid Daubechies y Stephane Mallat [15].

Una base Wavelet del espacio vectorial $\mathbf{C^n}$ se genera a partir del desplazamiento de dos funciones base localizadas en el espacio, *u* y *v*, conocidas como *Wavelet padre* y *Wavelet madre*, respectivamente [16]. La base Wavelet más sencilla que se conoce es la base Wavelet *Haar*, la cual es ortonormal y se genera a partir de la Wavelet padre y la Wavelet madre que se muestran en las ecuaciones (3.35) y (3.36) [19],

$$u = \frac{1}{\sqrt{2}}(1,\ 1,\ 0,\ 0,\ \ldots,\ 0) \tag{3.35}$$

$$v = \frac{1}{\sqrt{2}}(1,\ -1,\ 0,\ 0,\ \ldots,\ 0) \tag{3.36}$$

por ejemplo, la base Wavelet Haar del espacio vectorial $\mathbf{C^6}$ se formaría como se muestra en la ecuación (3.37)

$$W = \{u,\ R_2u,\ R_4u,\ v,\ R_2v,\ R_4v,\} \tag{3.37}$$

donde

$$\begin{aligned} u &= \frac{1}{\sqrt{2}}(1,\ 1,\ 0,\ 0,\ 0,\ 0) \\ R_2u &= \frac{1}{\sqrt{2}}(0,\ 0,\ 1,\ 1,\ 0,\ 0) \\ R_4u &\quad \frac{1}{\sqrt{2}}(0,\ 0,\ 0,\ 0,\ 1,\ 1) \\ v &= \frac{1}{\sqrt{2}}(1,\ -1,\ 0,\ 0,\ 0,\ 0) \\ R_2v &= \frac{1}{\sqrt{2}}(0,\ 0,\ 1,\ -1,\ 0,\ 0) \\ R_4v &= \frac{1}{\sqrt{2}}(0,\ 0,\ 0,\ 0,\ 1,\ -1) \end{aligned} \tag{3.38}$$

y R_k es un operador que realiza un desplazamiento circular de *k* unidades en el vector en que se aplica, como se muestra en la ecuación (3.39).

$$R_k u(n) = u(n-k) \tag{3.39}$$

En general, una base Wavelet del espacio vectorial $\mathbf{C^n}$ se puede expresar a través de la ecuación (3.40) [16]

$$W = \{\varphi_k\}_{k=0}^{M-1} \cup \{\psi_k\}_{k=0}^{M-1} = \{\varphi_0,\ \varphi_1,\ \ldots,\ \varphi_{M-1},\ \psi_0,\ \psi_1,\ \ldots,\ \psi_{M-1}\} \tag{3.40}$$

donde $M = \frac{N}{2}$, *N* es la longitud de los vectores del espacio vectorial $\mathbf{C^n}$ y φ_k y ψ_k son funciones que se obtienen al realizar desplazamientos circulares en la Wavelet padre y la Wavelet madre como se muestra en las ecuaciones (3.41) y (3.42).

$$\varphi_k = R_{2k} u = u(n-2k) \tag{3.41}$$

$$\psi_k = R_{2k} v = v(n-2k) \tag{3.42}$$

En la ecuación (3.38) puede observarse que el producto interno entre cualquier par de funciones base Haar es igual a cero y que la norma de todas las funciones base es igual a uno, por lo tanto, la base Wavelet Haar es ortonormal y la TWD utilizando la base Haar se puede obtener a partir de la ecuación (3.14) como se muestra en la ecuación (3.43)

$$\alpha_k = \begin{cases} \langle z, \varphi_k \rangle = \sum_{n=0}^{N-1} z(n)\varphi_k(n) & si\ \ 0 \le k \le M-1 \\ \langle z, \psi_{k-M} \rangle = \sum_{n=0}^{N-1} z(n)\psi_{k-M}(n) & si\ \ M \le k \le N-1 \end{cases} \tag{3.43}$$

La recuperación del vector al que se aplicó la TWD a partir de los coeficientes generados se denomina Transformada Wavelet Discreta Inversa (TWDI) y si la base utilizada es ortonormal se realiza a través de la ecuación (3.44).

$$z(n) = \sum_{i=0}^{M-1} \alpha_i \varphi_i + \sum_{i=M}^{N-1} \alpha_i \psi_{i-M}, \qquad 0 \le n \le N-1 \tag{3.44}$$

Además de la base Wavelet Haar existen otras bases Wavelet ortonormales, con las cuales la aplicación de la TWD y la TWDI también se realiza mediante las ecuaciones (3.43) y (3.44), respectivamente. En el anexo A se muestra una lista que contiene las funciones Wavelet padre y Wavelet madre de algunas de las bases ortonormales Wavelet más comunes actualmente en uso [14]. Por ejemplo, en las ecuaciones (3.45) y (3.46) se muestran la Wavelet padre y la Wavelet madre de la base ortonormal *Daubechies 6*, la cual fue inventada por Ingrid Daubechies.

$$u = (0.3327,\ 0.8069,\ 0.4599,\ -0.1350,\ -0.0854,\ 0.0352,\ 0,\ 0,\ \ldots,\ 0) \quad \textbf{(3.45)}$$

$$v = (0.8069,\ -0.3327\ \ 0,\ \ 0,\ \ldots,\ \ 0,\ \ -0.0352,\ \ -0.0854,\ \ 0.1350,\ \ 0.4599) \quad \textbf{(3.46)}$$

En la ecuación (3.43) puede observarse que la TWD genera dos grupos de coeficientes, los coeficientes de las funciones φ_k y los de las funciones ψ_k. El primer grupo brinda información acerca de las variaciones con frecuencias bajas que existen en diferentes lugares del vector z y el segundo proporciona información acerca de las variaciones con frecuencias altas; sin embargo, es posible que nos interese realizar un análisis más detallado acerca de las frecuencias presentes en el vector *z*, lo cual no es posible con sólo dos grupos de coeficientes. Una forma de solucionarlo es obteniendo a partir de la Wavelet padre y la Wavelet madre un nuevo par de funciones u_2 y v_2, conocidos como Wavelet padre y Wavelet madre de la segunda etapa, y formar una nueva base, en la cual en lugar de las funciones base de la Wavelet padre *u* se utilizan las funciones base que se generan con los vectores u_2 y v_2 [16] como se muestra en la ecuación (3.47)

$$W = \{\varphi_{-2,k}\}_{k=0}^{(N/4)-1} \cup \{\psi_{-2,k}\}_{k=0}^{(N/4)-1} \cup \{\psi_{-1,k}\}_{k=0}^{(N/2)-1} \quad \textbf{(3.47)}$$

donde

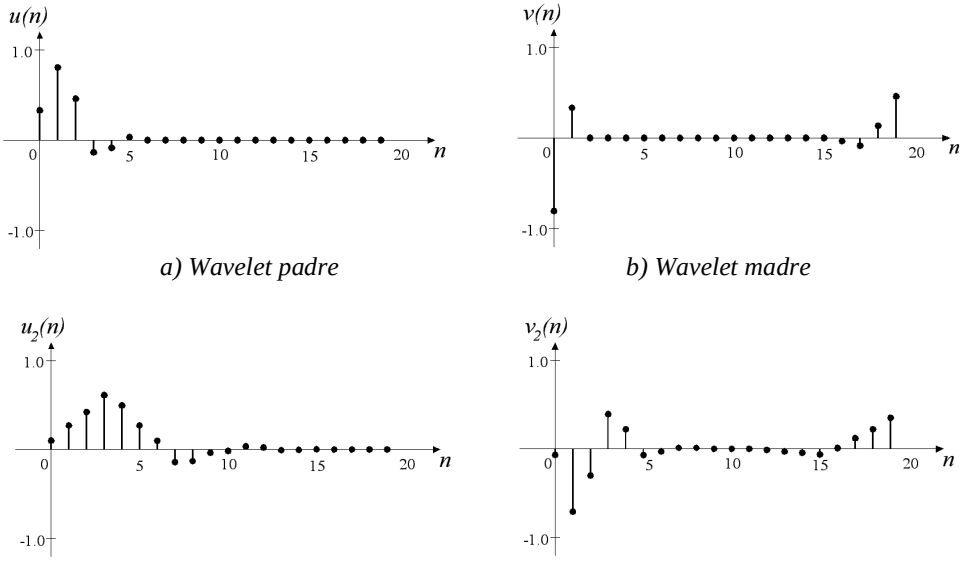
$$\varphi_{-2,k} = R_{4k} u_2 \quad \textbf{(3.48)}$$

$$\psi_{-2,k} = R_{4k} v_2 \quad \textbf{(3.49)}$$

$$\psi_{-1,k} = R_{2k} v \quad \textbf{(3.50)}$$

Ahora la información de frecuencias bajas que aportaban las funciones φ_k está dividida en frecuencias bajas-altas y frecuencias bajas-bajas y se cuenta con tres grupos de coeficientes, los correspondientes a las funciones $\psi_{-1,k}$ que representan las frecuencias altas, los correspondientes a las funciones $\psi_{-2,k}$ que representan las frecuencias bajas-altas y los correspondientes a las funciones $\varphi_{-2,k}$ que representan las frecuencias bajas-bajas. Una base Wavelet puede tener más de dos etapas, con la única condición de que la longitud del vector al que se aplica la TWD sea divisible entre 2^p, donde *p* es el número de etapas.

En la figura 3.10c se puede observar que la Wavelet padre de la segunda etapa de la base Daubechies 6 es similar a la Wavelet padre de la primera etapa, la cual se muestra en la figura 3.10a sólo que extendida y lo mismo sucede con la Wavelet madre y la Wavelet madre de la segunda etapa que se muestran en la figura 3.10b y 3.10d, respectivamente [16]. La descripción del procedimiento con el cual se obtienen la Wavelet padre y la Wavelet madre de la segunda etapa y de las etapas siguientes no está dentro de los objetivos de este trabajo, ya que en compresión de imágenes usualmente sólo se utiliza una etapa.

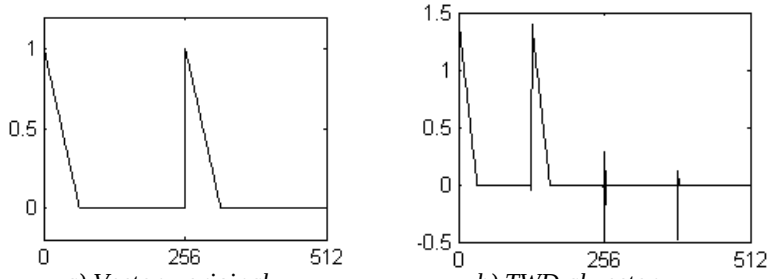


a) Wavelet padre *b) Wavelet madre*

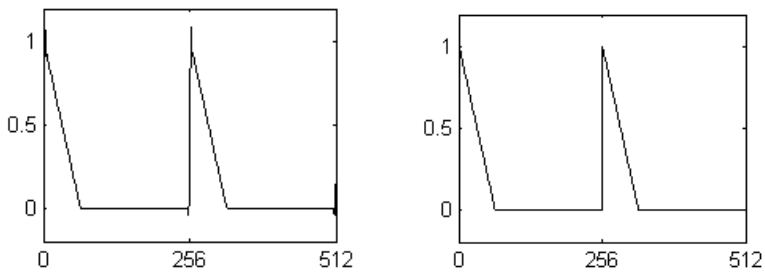
c) Wavelet padre segunda etapa *d) Wavelet madre segunda etapa*

Figura 3.10 Dos etapas de la base Wavelet Daubechies 6

En la figura 3.11b se muestran los coeficientes que se obtienen al aplicar la TWD con la base Daubechies 6 al vector *z* que se muestra en la figura 3.11a (aunque se trata de funciones discretas, se muestran en forma continua con el propósito de hacer más compresible la gráfica), los primeros 256 coeficientes corresponden a las funciones φ_k y los últimos 256 a las funciones ψ_k, en la figura 3.11c se muestra la reconstrucción del vector *z* a través de la TWDI utilizando sólo los coeficientes de las funciones φ_k y en la figura 3.11d se muestra la reconstrucción del vector *z* utilizando todos los coeficientes.



a) Vector z original *b) TWD al vector z*



c) TWDI con 256 coeficientes *d) TWDI con 512 coeficientes*

Figura 3.11 Aplicación de la TWD a un vector

En la figura 3.11b podemos observar que los primeros 256 coeficientes proporcionan información acerca de los lugares del vector *z* en que tiene variaciones con frecuencias bajas, que parecen una réplica contraída del vector original y que la mayoría de los coeficientes que dan información sobre las frecuencias altas, los últimos 256, son cero. En la figura 3.11c se observa que con los coeficientes de frecuencias bajas el vector *z* está casi completamente reconstruido y en la figura 3.11d vemos que los últimos 256 coeficientes únicamente aportan información en los lugares en que el vector tiene variaciones rápidas.

La TWD se aplica a una imagen de la misma forma que la TCD, primero se aplica a las líneas para obtener un arreglo de coeficientes del mismo tamaño que la imagen original y después a las columnas de coeficientes. En la figura 3.12b se muestran los coeficientes que se generan al aplicar la TWD con la base Daubechies 6 a las líneas de la imagen de la figura 3.12a y en la figura 3.12c se muestran los coeficientes generados al aplicar la TWD a las columnas, los coeficientes han sido normalizados para que se encuentren en el intervalo [0, 255] y los coeficientes de las funciones ψ_k han sido multiplicados por 5 y se les ha agregado un offset de +128 para observar sus característica con mayor facilidad.

a) Imagen original *b) TWD a las líneas* *c) TWD a las columnas*

Figura 3.12 TWD a una imagen

Como puede observarse en la figura 3.12b los coeficientes de las funciones φ_k quedan en la mitad izquierda del arreglo, mostrando información de baja frecuencia de la imagen y formando una réplica contraída de la imagen original, y los coeficientes de las funciones ψ_k quedan del lado derecho, mostrando la información de frecuencias altas, por lo cual son mayores en las partes en que la imagen tiene bordes. En la figura 3.12c puede observarse que la aplicación de la TWD a las columnas también divide los coeficientes en dos grupos, pero esta vez con los coeficientes que tienen información de frecuencias bajas en la mitad superior y los que tienen información de frecuencias altas en la mitad inferior, por lo tanto, ahora se tienen cuatro grupos de coeficientes. A los diferentes grupos de coeficientes que resultan de la aplicación de la TWD se les denomina *sub-bandas* [17], en la figura 3.13 se muestra un esquema de las sub-bandas en que se divide una imagen al aplicársele la TWD.

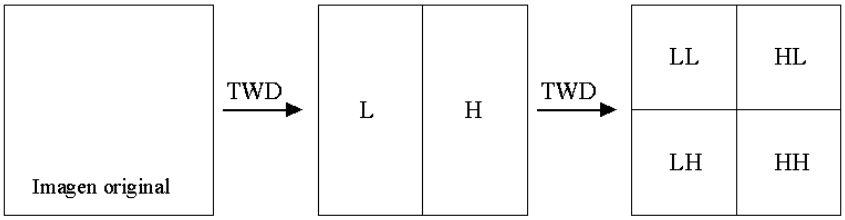


Figura 3.13 Esquema de la división en sub-badas que realiza la TWD

La sub-banda LL contiene los coeficientes que representan a las frecuencias bajas horizontal y verticalmente, la sub-banda HL contiene los coeficientes que representan a las frecuencias altas horizontalmente y a las frecuencias bajas verticalmente, la sub-banda LH contiene los coeficientes que representan a las frecuencias bajas horizontalmente y a las frecuencias altas verticalmente y la sub-banda HH contiene los coeficientes que representan a las frecuencias altas tanto horizontal como verticalmente [17].

El número de operaciones que se requieren para aplicar la TWD con la base Daubechies 6 a una imagen de N x N píxeles es el siguiente: en la ecuación (3.43) puede observarse que si no se toman en cuenta los términos en los que hay multiplicaciones por cero, para calcular cada coeficiente se requieren 6 multiplicaciones y 5 sumas lo que da 11 operaciones, cada línea genera N coeficientes por lo tanto requiere 11N operaciones, la aplicación de la TWD a las N líneas requiere $N(11N) = 11N^2$ operaciones y se requiere el mismo número de operaciones para aplicar la TWD a las columnas, por lo tanto, en total se necesitan $2(11N^2) = 22N^2$ operaciones, por ejemplo, una imagen de 128 x 128 píxeles requiere $22(128^2) = 360{,}448$ operaciones.

En la sección anterior vimos que para aplicar la TCD a una imagen de 128 x 128 píxeles se requerían 8,388,608 operaciones y que este número se reducía a 524,288 si la imagen se dividía en bloques de 8 x 8 píxeles, la aplicación de la TWD a una imagen completa de las mismas dimensiones, en cambio, requiere de únicamente 360,448 operaciones, por lo cual en la mayoría de las aplicaciones no hace falta dividir la imagen en bloques.

Una forma de aplicar la TWD es considerar a la Wavelet padre, la Wavelet madre y al vector *z* al que se aplicará la transformada como funciones periódicas y utilizar la operación convolución, la cual para funciones periódicas se define como se muestra en la ecuación (3.51) [16].

$$(z * y)(k) = \sum_{n=0}^{N-1} z(n)y(k-n), \quad 0 \le k \le N-1 \tag{3.51}$$

La reflexión conjugada de un vector *w*, $\tilde{w}$, se define como se muestra en la ecuación (3.52) [16]

$$\tilde{w}(n) = \overline{w}(-n) \tag{3.52}$$

Ahora veamos el elemento *k* de la convolución entre el vector z y la reflexión conjugada de la Wavelet padre,

$$(z * \tilde{u})(k) = \sum_{n=0}^{N-1} z(n)\tilde{u}(k-n) = \sum_{n=0}^{N-1} z(n)\overline{u}(n-k) = \langle z, u(n-k)\rangle, \ 0 \le k \le N-1 \tag{3.53}$$

por lo tanto, el vector $z * \tilde{u}$ es igual a

$$z * \tilde{u} = \left(\langle z, u\rangle, \ \langle z, u(n-1)\rangle, \ \langle z, u(n-2)\rangle, \ \ldots, \ \langle z, u(N-1)\rangle\right) \tag{3.54}$$

Si sustituimos la ecuación (3.41) en la ecuación (3.54) y sólo tomamos los elementos pares del vector $z * \tilde{u}$, obtenemos el vector α_φ que se muestra en la ecuación (3.55)

$$\alpha_\varphi = \left(\langle z, \varphi_0 \rangle, \langle z, \varphi_1 \rangle, \langle z, \varphi_2 \rangle, \ldots, \langle z, \varphi_{M-1} \rangle\right) \quad \textbf{(3.55)}$$

Si comparamos la ecuación (3.55) con la ecuación (3.43) podemos observar que es posible obtener todos los coeficientes de las funciones φ_k tomando los elementos pares del vector $z * \tilde{u}$ y de forma similar se pueden obtener los coeficientes de las funciones ψ_k, el proceso se ilustra en la etapa de análisis de la figura 3.14 donde ↓2 indica que sólo se toman los elementos pares de los vectores $z * \tilde{u}$ y $z * \tilde{v}$.

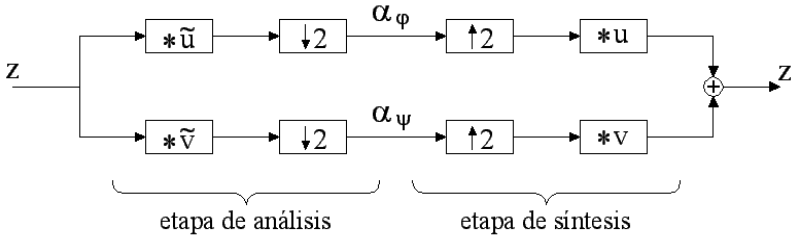


Figura 3.14 Aplicación de la TWD mediante la operación convolución

La recuperación del vector *z* se realiza durante la etapa de síntesis de la figura 3.14 donde ↑2 representa la inserción de un cero entre los elementos de los vectores α_φ y α_ψ. Ahora veamos el elemento *n* de la convolución entre el vector sobremuestreado α_φ y la Wavelet padre, *u*,

$$\begin{aligned}(\alpha_\varphi * u)(n) = \sum_{k=0}^{N-1} \alpha(k) u(n-k) &= \alpha_0 u(n) + 0 \cdot u(n-1) + \alpha_1 u(n-2) + 0 \cdot u(n-3) + \alpha_2 u(n-4) + \ldots + 0u(n-N+1) \\ &= \alpha_0 u(n) + \alpha_1 R_2 u(n) + \ldots + \alpha_{M-1} R_{N-2} u(n)\end{aligned}$$

$$(\alpha_\varphi * u)(n) = \alpha_0 \varphi_0(n) + \alpha_1 \varphi_1(n) + \alpha_2 \varphi_2(n) + \ldots + \alpha_{M-1} \varphi_{M-1}(n) \quad \textbf{(3.56)}$$

por lo tanto, el vector $\alpha_\varphi * u$ es igual a la primera sumatoria de la ecuación (3.44), la segunda sumatoria se obtiene con el vector $\alpha_\psi * v$ y la suma de estos dos vectores permite la recuperación del vector *z* original [16].

Debido a que la aplicación de un filtro a una señal también se realiza mediante la operación convolución, es posible utilizar el esquema de la figura 3.14 para aplicar la TWD como si se tratara de un banco de filtros pasobanda [16].

3.3 Estándares de compresión y formatos de archivo de imagen

En la década de los ochentas del siglo pasado la compresión de imágenes comenzó a convertirse en un área de aplicación comercialmente importante y desde entonces comenzaron a desarrollarse una serie de estándares internacionales de compresión. El objetivo de un estándar de compresión de imágenes es permitir la interoperabilidad entre equipo y sistemas de diferentes fabricantes[9].

Cada estándar describe una estructura para la representación de las imágenes comprimidas, el procedimiento para la descompresión y, posiblemente, un descompresor de

referencia. Con el propósito de dejar espacio para la innovación, los estándares no definen un compresor de imágenes, sino que éste se deja al criterio del diseñador. Sin embargo, en la práctica el número de alternativas de diseño que cumplen con los requerimientos del estándar está bastante limitado por la estructura con la que deben cumplir y por el descompresor de referencia [9], [17].

En esta sección se describen los formatos archivos de imágenes GIF (Graphics Interchange Format) y PNG (Portable Network Graphics) y los estándares de compresión JPEG (Joint Photographic Expert Group) y JPEG2000 (Joint Photographic Expert Group 2000). Aunque GIF y PNG no son estándares de compresión, se mencionan debido a que al igual que los estándares, utilizan algunas de las técnicas de compresión descritas en las secciones anteriores. Mientras que GIF y PNG operan directamente sobre los píxeles y utilizan métodos de compresión sin pérdidas, JPEG y JPEG2000 pueden realizar compresión con pérdidas utilizando transformadas y Cuantización.

3.3.1 Formato GIF

En 1987, la compañía Compuserve Information Services desarrolló un formato de archivos gráficos comprimidos llamado GIF que permitía que varias computadoras pudieran compartir imágenes [14]. La versión original de GIF se conoce como GIF 87a, fue distribuida de forma gratuita y fue adoptada por prácticamente todas las aplicaciones de procesamiento de imágenes [3].

Los píxeles de una imagen GIF se comprimen utilizando la técnica de compresión sin pérdidas LZW. El diccionario se inicializa de modo que contenga todos los valores que pueda tener un píxel de la imagen, por ejemplo, si los píxeles se representan con 8 bits entonces el diccionario inicialmente contiene los valores del 0 al 255.

Al iniciar el proceso de compresión, cada valor se almacena utilizando el número de bits más pequeño posible, usualmente 9. Cuando el número de índices se vuelve demasiado largo para ser representado con nueve bits, se aumenta a 10.

El máximo número de bits permitido por GIF es 12 y cuando el número de índices alcanza 2^{12}-1 tanto el compresor como el descompresor GIF dejan de agregar cadenas al diccionario [3], en este punto el compresor revisa la tasa de compresión y puede decidir que lo más conveniente es borrar el diccionario actual y comenzar uno nuevo [14].

Actualmente, el formato GIF es usado comúnmente por navegadores de Internet, sin embargo, GIF procesa una imagen línea por línea de modo que descubre la correlación entre píxeles en las líneas, pero no entre líneas, se puede decir que la compresión del formato GIF es unidimensional mientras que una imagen es bidimensional [14].

3.3.2 Formato PNG

El formato de archivo PNG fue desarrollado a mediados de los años noventas por un grupo organizado por Thomas Boutell llamado PNG Development Group con el propósito de reemplazar a GIF [14] debido a que la compañía Unisys, quien obtuvo la patente sobre la técnica de compresión LZW al comprar a la compañía Sperry comenzó a pedir a los

usuarios de GIF el pago de una licencia por su uso, lo cual hizo imposible su utilización en diversas aplicaciones, sobre todo las de software libre [3].

Como LZW ya no podía ser utilizado sin el pago de una licencia, en su lugar se utilizó un método de compresión gratuito conocido como Deflate, el cual está basado en la técnica LZ77. Este método utiliza una ventana deslizante de 32 kilobytes o de una potencia de dos más pequeña y los datos comprimidos, píxeles individuales y parejas distancia-longitud, se representan a través de códigos.

Los códigos que Deflate utiliza están contenidos en dos tablas. La primera contiene 286 códigos, de 0 a 255 representan valores de píxeles, 256 es un marcador especial y de 257 a 285 indican la longitud de la secuencia de píxeles del buffer predictivo que se encontró en el diccionario. La segunda tabla contiene 29 códigos que indican la distancia entre el buffer predictivo y el apuntador que apunta al primer píxel de la secuencia que se va a representar.

Dado que el número de códigos no es una potencia de dos, para representarlos eficientemente PNG agrega una etapa más, en la cual los códigos que representan a los píxeles individuales y a las parejas distancia-longitud son representados, a su vez, con códigos Huffman [3].

PNG es muy utilizado por la comunidad de Internet en situaciones donde se requiere de compresión sin pérdidas como cuando se trabaja con imágenes que son editadas continuamente [3].

3.3.3 Estándar JPEG

El estándar de compresión de imágenes a color y en escala de grises JPEG fue desarrollado en 1992 de forma conjunta por las dos principales organizaciones del área de compresión de imágenes, la International Standards Organization (ISO) y la International Telecommunication Union (ITU-T) [17].

Los principales objetivos de JPEG son: la obtención de altas tasas de compresión cuando la calidad deseada de la imagen recuperada va de buena a muy buena, permitir a usuarios avanzados la manipulación de parámetros para lograr el equilibrio deseado entre tasa de compresión y calidad de la imagen y que el método de compresión no sea demasiado complejo para que pueda ser implementado tanto en software como en hardware en diferentes plataformas [14].

JPEG define cuatro modos de operación, el modo *secuencial*, el modo *progresivo*, el modo *jerárquico* y el modo *sin pérdidas* [21], [22]. En el modo secuencial cada componente del espacio de color de la imagen se comprime en un solo barrido de izquierda a derecha y de arriba hacia abajo. En el modo progresivo la imagen se comprime en varios barridos para que al ser descomprimida con cada barrido se logre una imagen de mejor calidad [14]. En el modo jerárquico la imagen se comprime con varias resoluciones para que pueda ser vista en diferentes dispositivos [7]. El modo sin pérdidas siempre conserva una copia exacta de la imagen original, pero la tasa de compresión no es tan buena como en la compresión con pérdidas y no comprime tan bien como otros formatos que ya están disponibles como PNG [3].

De los cuatro modos de operación de JPEG, el único que realmente es usado con amplitud es el modo *secuencial*, en la figura 3.15 se muestran los pasos que deben seguirse para llevar a cabo la compresión en este modo [21], [22].

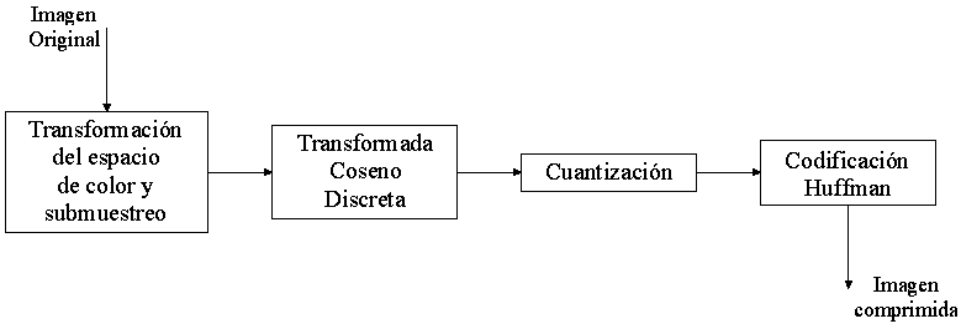


Figura 3.15 Modo Secuencial de JPEG

El primer paso es transformar la imagen del modelo de color RGB al YCbCr y submuestrear las crominancias en formato 4:2:2 ó 4:2:0 con el propósito de reducir el número de datos a procesar en las etapas siguientes, esta reducción afecta poco a la calidad de la imagen recuperada debido a que el sistema visual humano es menos sensible a los cambios de color que a los cambios en la intensidad de luz [11].

Luego se aplica la TCD a cada componente del modelo de color YCbCr de forma independiente tomando bloques de 8x8 y con la Cuantización se eliminan aquellos coeficientes que no son esenciales para la calidad visual de la imagen. La Cuantización es el paso que provoca que la tasa de compresión sea más alta y es al mismo tiempo la responsable por las pérdidas en la calidad de la imagen recuperada. Finalmente, a los coeficientes cuantizados se les aplica la Codificación Huffman, cuyos códigos se encuentran en tablas que proporciona el propio estándar [21], [22].

La descompresión de la imagen se realiza básicamente aplicando el proceso inverso a aquel que se siguió en la figura 3.15 [21], [22].

La principal aplicación de JPEG es el almacenamiento y la transmisión de imágenes comprimidas y es utilizado ampliamente en la generación de imágenes digitales, en imágenes embebidas en páginas de Internet, en cámaras digitales y en muchas otras aplicaciones [9].

3.3.4 Estándar JPEG2000

El estándar JPEG se ha convertido en la principal herramienta de compresión de imágenes fotográficas en Internet y es ampliamente usado en el almacenamiento de imágenes, sin embargo, el hecho de que para aplicar la TCD a la imagen ésta se divida en bloques de 8x8 píxeles tiene la desventaja de que conforme la tasa de compresión se incrementa, en la imagen reconstruida comienzan a aparecer discontinuidades en los límites de los bloques conocidas como artefactos, las cuales dan a la imagen un aspecto cuadriculado [9]. Esa es la razón por la cual el comité JPEG decidió desarrollar un nuevo estándar de compresión, el JPEG2000, el cual está basado en la TWD, tiene un mejor desempeño con tasas de compresión más altas y es la parte fundamental de este trabajo [14].

En el siguiente capítulo se describirá con más detalle la forma en que funciona el estándar JPEG2000, en esta sección únicamente se hace una descripción breve de los pasos que el estándar sigue durante la compresión, los cuales se ilustran en la figura 3.16.

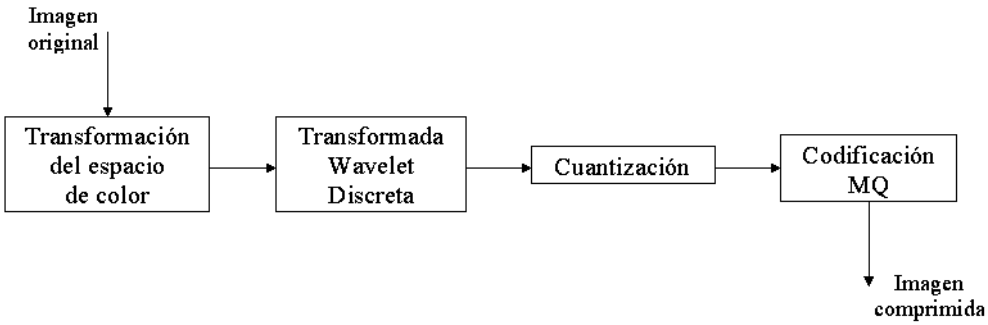


Figura 3.16 Compresión con pérdidas del estándar JPEG2000

En el estándar JPEG2000, al igual que en JPEG, la imagen se transforma del modelo de color RGB al YCbCr, sin embargo, las crominancias no son submuestreadas y la imagen permanece en formato 4:4:4, ya que un efecto similar al submuestreo se logra después de la TWD haciendo cero los coeficientes de las sub-bandas LH, HL y HH [11]. Luego se aplica la TWD sobre cada componente del modelo YCbCr utilizando las funciones base conocidas como *Daubechies 9,7*. La TWD tiene la ventaja de que no se aplica sobre bloques pequeños de la imagen, sino sobre la imagen completa con lo cual se evita la aparición de artefactos al descomprimir la imagen.

A los coeficientes que resultan de la TWD se les puede aplicar una Cuantización diferente de acuerdo con la importancia que tenga en la apariencia de la imagen la sub-banda en la que se encuentren. Por último, a los coeficientes cuantizados se les aplica una variación de la Codificación Aritmética que no utiliza multiplicaciones, sino únicamente sumas, restas y corrimientos de bits, cuyos datos de entrada son bits individuales y es conocida como Codificación MQ [7].

Mientras que en imágenes donde JPEG logra tasas de compresión de 5%, JPEG2000 usualmente puede lograr tasas de compresión menores al 2% con la misma calidad de la imagen recuperada. Con JPEG2000 la degradación que sufre la imagen conforme la tasa de compresión se incrementa es una disminución gradual en la definición de los bordes dentro de la imagen, efecto menos obvio que los artefactos asociados con la DCT, sin embargo, el precio que debe pagarse por estos beneficios es un incremento en la memoria requerida por los procesos de compresión y descompresión, por lo cual almacenar y desplegar una imagen usando JPEG2000 toma más tiempo [9].

3.4 Resumen

En este capítulo se han descrito diferentes técnicas de compresión de imágenes que se han desarrollado debido a la necesidad de disminuir la cantidad de memoria requerida para almacenarlas y de hacer más estrecho el ancho de banda para transmitirlas. Con el propósito de lograr que sistemas y equipo de diferentes fabricantes puedan utilizar estos métodos de compresión sin que haya problemas de compatibilidad, se han desarrollado una serie de formatos de archivo de imagen y estándares internacionales de compresión, los cuales pueden utilizar uno o una combinación de los métodos de compresión. De los formatos de archivo de imagen y estándares actuales, el que ofrece las tasas de compresión más altas en compresión con pérdidas es JPEG2000.

Capítulo 4
El Estándar JPEG2000

A mediados de los años ochenta, miembros de la ITU-T (International Telecommunication Union) y de la ISO (International Standards Organization) formaron un grupo conocido como JPEG (Joint Photographic Experts Group) con el propósito de desarrollar un estándar de compresión de imágenes [11]. El resultado de este esfuerzo conjunto fue el estándar de compresión de imágenes a color y en escala de grises JPEG, el cual ha ganado una amplia aceptación y ha sido muy exitoso en el mercado por más de una década [17].

Sin embargo, debido a que desde la definición de JPEG la tecnología y el mercado han sufrido una notable transformación provocada por el surgimiento y empleo masivo del Internet y un progreso significativo de las tecnologías multimedia y de comunicaciones, el comité JPEG decidió desarrollar un nuevo estándar de compresión, el JPEG2000 [17].

El estándar JPEG2000 fue diseñado con el objetivo de proveer la mejor calidad o desempeño y de tener la capacidad de cumplir con las demandas del mercado actual y se espera que JPEG2000 sea eficaz en áreas de aplicación tales como Internet, fotografía digital, bibliotecas digitales, bases de datos de imágenes, fotocopiado, fax e impresión a color, generación de imágenes médicas, comunicación multimedia móvil, telefonía celular 3G, comercio electrónico, etc. [11].

En este capítulo se enumeran las once partes que componen al estándar JPEG2000 y se describen los pasos que se deben llevar a cabo para implementar un sistema básico de compresión y descompresión tanto en compresión con pérdidas como en compresión sin pérdidas.

4.1 Parte 1 del estándar JPEG2000

El estándar JPEG2000 consta de once partes, la primera describe el *sistema básico de compresión o núcleo* en las modalidades de compresión con pérdidas y compresión sin pérdidas y las otras diez agregan características adicionales a la primera parte. La siguiente es una lista de las partes que componen al estándar, la número siete fue abandonada y de la ocho a la once aún están en desarrollo [17].

Parte 1: Núcleo del sistema de compresión.
Parte 2: Extensiones a la parte 1.
Parte 3: Compresión de secuencias de imágenes, Motion JPEG2000.
Parte 4: Procedimientos para realizar pruebas de conformidad.
Parte 5: Software de referencia para validar sistemas JPEG2000.
Parte 6: Formato de Archivo para almacenar imágenes compuestas.
Parte 7: Esta parte ha sido abandonada.
Parte 8: Seguridad en JPEG2000.

Parte 9: Herramientas interactivas, APIS y protocolos (JPIP).
Parte 10: Datos en 3D y en Punto Flotante.
Parte 11: Aplicaciones inalámbricas.
Parte 12: Formato de archivo multimedia.

En la figura 4.1 se muestra un diagrama de bloques de la parte 1 del estándar, es decir, del Núcleo del Sistema de Compresión de JPEG2000, en la modalidad de compresión con pérdidas, el diagrama de bloques para compresión sin pérdidas es similar pero sin el bloque de Cuantización.

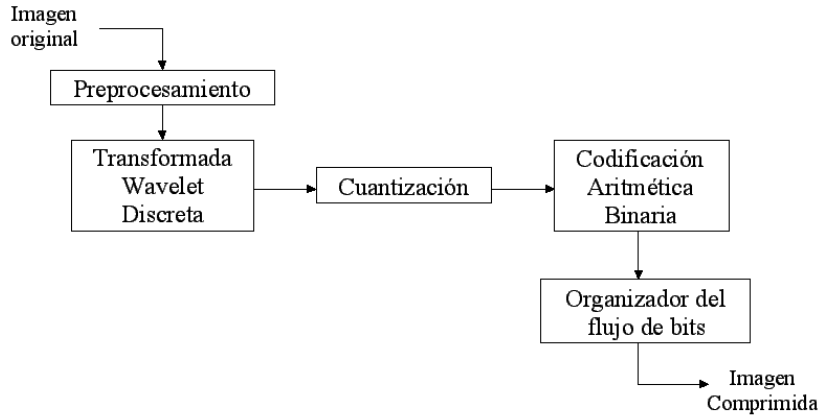


Figura 4.1 Compresor JPEG2000

La descompresión de la imagen se realiza siguiendo el diagrama de la figura 4.1 en sentido inverso, es decir, revirtiendo los efectos que cada bloque provocó en la imagen, comenzando con los datos comprimidos. En las siguientes secciones se describe la forma en que se implementa cada uno de estos bloques durante el proceso de compresión y la forma en que se implementan durante la descompresión.

4.2 Preprocesamiento

Antes de la compresión, los píxeles de la imagen son preprocesados con el propósito de hacer procedimientos posteriores más fáciles de implementar [11]. Esta etapa consta de tres elementos: segmentación, corrimiento de DC y transformación del espacio de color.

Segmentación
La segmentación es opcional y consiste en dividir la imagen en bloques rectangulares que no se solapen, llamados *mosaicos* [17]. Las dimensiones de los mosaicos se pueden escoger arbitrariamente, pero todos ellos deben tener las mismas dimensiones en una misma imagen excepto por los que se encuentren en las orillas si las dimensiones de la imagen no son un múltiplo entero de las dimensiones de los mosaicos, como se muestra en la figura 4.2.

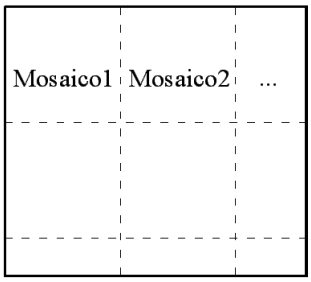


Figura 4.2 Esquema de una imagen segmentada

Cada mosaico se comprime de forma independiente, lo que puede provocar la aparición de artefactos al descomprimir una imagen con una tasa de compresión muy alta, por lo tanto, los mejores resultados se logran cuando la imagen no es segmentada; sin embargo, cuando las dimensiones de los mosaicos son muy grandes, los requerimientos de memoria para la implementación del algoritmo de compresión crecen en la misma proporción. Dimensiones típicas de los mosaicos son 256 x 256 y 512 x 512 píxeles [17].

Corrimiento de DC

Originalmente los píxeles de las imágenes se almacenan como enteros sin signo, sin embargo, con el propósito de hacer más simples algunos de los procesos de la compresión que serán explicados más adelante como la especificación del contexto y la Codificación Aritmética Binaria, a los valores RGB se les hace un corrimiento de DC utilizando la ecuación (4.1) [11]

$$I'(x,y) = I(x,y) - 2^{B-1} \quad \textbf{(4.1)}$$

donde *I(x, y)* representa al componente R, G o B del píxel en la posición *(x, y)* y *B* es el número de bits usado para representar cada componente, por ejemplo, si se utilizan 8 bits, a cada componente se le resta 128.

Transformación del Espacio de Color

Cuando se utiliza compresión con pérdidas, el tercer elemento del preprocesamiento es la transformación de la imagen del espacio de color RGB al YCbCr. Debido a que el sistema visual humano es menos sensible a los cambios en la intensidad de la luz que a los cambios de color, a las crominancias se les puede aplicar una compresión mayor que a la luminancia sin afectar notablemente a la calidad visual de la imagen, lo que da como resultado una compresión mayor en la imagen completa. La transformación del espacio de color en compresión con pérdidas se realiza con la ecuación (4.2) [17].

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(4.2)}$$

Si la imagen se comprime en la modalidad de compresión sin pérdidas, se utiliza una transformación diferente, en la cual se recuperan los valores originales al realizar la transformación inversa, lo cual no es posible al realizar la transformación con la ecuación (4.2) debido al error introducido por la utilización de números no-enteros en la matriz. La transformación en la modalidad sin pérdidas se realiza con de la ecuación (4.3).

$$\begin{bmatrix} Y_r \\ U_r \\ V_r \end{bmatrix} = \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0 & -1 & 1 \\ 1 & -1 & 0 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad \textbf{(4.3)}$$

En la descompresión, la imagen se regresa al espacio de color RGB utilizando la ecuación (4.4), si se utilizó la modalidad con pérdidas, o la ecuación (4.5), si se utilizó la modalidad sin pérdidas, se le suma la componente de DC que le fue restada y, en caso de haberse segmentado la imagen, se unen los mosaicos.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 1.402 \\ 1.0 & -0.344136 & -0.714136 \\ 1.0 & 1.772 & 0.0 \end{bmatrix} \begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} \quad \textbf{(4.4)}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & -0.25 & 0.75 \\ 1 & -0.25 & -0.25 \\ 1 & 0.75 & -0.25 \end{bmatrix} \begin{bmatrix} Y_r \\ U_r \\ V_r \end{bmatrix} \quad \textbf{(4.5)}$$

4.3 Transformada Wavelet Discreta

La Transformada Wavelet Discreta (TWD) se aplica de forma independiente sobre los tres componentes de la imagen que se obtuvieron en la transformación del espacio de color, YCbCr en compresión con pérdidas o $Y_rU_rV_r$ en compresión sin pérdidas. En compresión con pérdidas el estándar JPEG2000 utiliza las funciones base Daubechies 9,7, las cuales fueron inventadas por Ingrid Daubechies y son las que tienen la eficiencia de compresión más alta, y en compresión sin pérdidas utiliza las funciones base LeGall 5,3, las cuales fueron descubiertas por Didier LeGall [11].

En la sección 3.2.4 se mostró con un ejemplo la forma en que se puede aplicar la TWD a una imagen utilizando la base ortonormal Daubechies 6, sin embargo, la TWD no se aplica de la misma forma con las bases Daubechies 9,7 y LeGall 5,3 debido a que el producto interno entre sus funciones base no es cero, es decir, no se trata de bases ortogonales, sino biortogonales, por lo tanto, sus coeficientes no se pueden obtener a través de la ecuación (3.14).

4.3.1 Bases Biortogonales

Un vector *z* del espacio vectorial $\mathbf{C^n}$ se puede generar mediante una combinación lineal de un conjunto de funciones base $\psi_0, \psi_1, \ldots, \psi_{N-1}$, como se muestra en la ecuación (4.6)

$$z = \sum_{i=0}^{N-1} \alpha_i \psi_i \quad \textbf{(4.6)}$$

donde los coeficientes α_i se pueden obtener mediante el producto interno que se muestra en la ecuación (4.7).

$$\alpha_i = \langle z, \tilde{\psi}_i \rangle = \sum_{n=0}^{N-1} z(n)\tilde{\psi}_i(n), \quad 0 \le i \le N-1 \tag{4.7}$$

Un caso particular muy importante es cuando el conjunto ψ_i cumple con la ecuación (4.8),

$$\langle \psi_i, \psi_j \rangle = \begin{cases} 1 & si \quad i = j \\ 0 & si \quad i \neq j \end{cases} \tag{4.8}$$

entonces se dice que el conjunto ψ_i es ortogonal y se cumple que

$$\psi_i = \tilde{\psi}_i \tag{4.9}$$

lo cual implica que se utilizan las mismas funciones ψ_i tanto para la generación del vector *z* mediante la combinación lineal de la ecuación (4.6) como para obtener los coeficientes α_i con la ecuación (4.7) [18].

Si la ecuación (4.8) no se satisface, aún es posible generar al vector *z* mediante una combinación lineal, sin embargo, la ecuación (4.9) no se cumple y las funciones base ψ_i y las funciones $\tilde{\psi}_i$ deben satisfacer la ecuación (4.10) [18].

$$\langle \psi_i, \tilde{\psi}_j \rangle = \begin{cases} 1 & si \quad i = j \\ 0 & si \quad i \neq j \end{cases} \tag{4.10}$$

Si la ecuación (4.10) se cumple, entonces se dice que las funciones base ψ_i son *biortogonales* y las funciones $\tilde{\psi}_i$ son sus *funciones duales* [18]. En la figura 4.3 se representan en un plano dos vectores que forman una base biortogonal del espacio vectorial $\mathbf{C^2}$, ψ_0 y ψ_1, y sus funciones duales, $\tilde{\psi}_0$ y $\tilde{\psi}_1$.

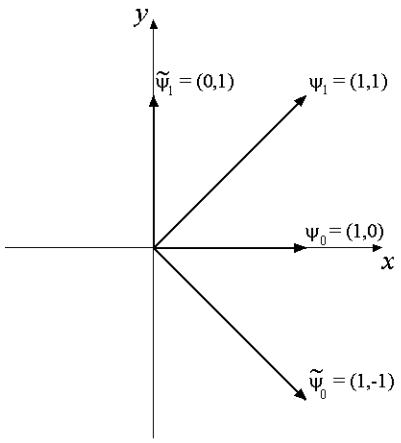


Figura 4.3 Funciones base biortogonales del espacio vectorial $\mathbf{C^2}$

Las funciones ψ_0, ψ_1, $\tilde{\psi}_0$ y $\tilde{\psi}_1$. cumplen con la ecuación (4.10) y cualquier vector *z* del espacio $\mathbf{C}^2$ se puede expresar mediante la combinación lineal que se muestra en la ecuación (4.11).

$$z = \alpha_0\psi_0 + \alpha_1\psi_1 \tag{4.11}$$

Los coeficientes α_0 y α_1 se pueden obtener aplicando la ecuación (4.7) como se muestra en la ecuación (4.12),

$$\begin{aligned}\alpha_0 &= \tilde{\psi}_0(0)\bar{z}(0) + \tilde{\psi}_0(1)\bar{z}(1)\\ \alpha_1 &= \tilde{\psi}_1(0)\bar{z}(0) + \tilde{\psi}_1(1)\bar{z}(1)\end{aligned} \tag{4.12}$$

por ejemplo, los coeficientes del vector $z = (-1,3)$ serían

$$\begin{aligned}\alpha_0 &= 1(-1) - 1(3) = -4\\ \alpha_1 &= 0(-1) + 1(3) = 3\end{aligned} \tag{4.13}$$

y la combinación lineal de la base biortogonal de la figura 4.3 que lo genera sería

$$z = -4\psi_0 + 3\psi_1 = -4(1,\ 0) + 3(1,\ 1) = (-1,\ 3) \tag{4.14}$$

como se ilustra en la figura 4.4

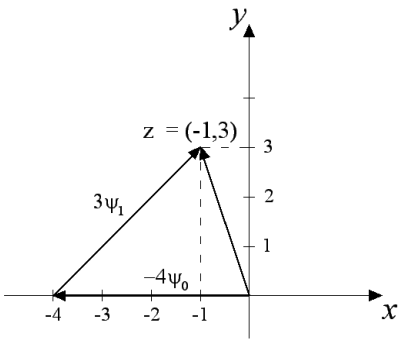


Figura 4.4 Generación de un vector z mediante una base biortogonal

4.3.2 Bases Daubechies 9,7 y LeGall 5,3

Debido a que las funciones base Daubechies 9,7 y LeGall 5,3 son biortogonales, se requieren dos conjuntos de funciones: las funciones duales para la obtención de los coeficientes de la TWD durante la compresión y las funciones base para regresar la imagen a su forma original durante la descompresión.

Las funciones base Daubechies 9,7 se generan a través de la ecuación (3.40) a partir de la Wavelet padre y la Wavelet madre que se muestran en las ecuaciones (4.15) y (4.16) [19]

$$u = (0,\ -\beta_1,\ \beta_2,\ -\beta_3,\ \beta_4,\ -\beta_5,\ \beta_6,\ -\beta_7,\ 0,\ 0,\ \ldots,\ 0) \tag{4.15}$$

$$v = (0\ \ a_1,\ -a_2,\ a_3,\ -a_4,\ a_5,\ -a_6,\ a_7,\ -a_8,\ a_9,\ 0,\ \ldots,\ 0) \tag{4.16}$$

donde los elementos a_i y β_i están dados por [19]:

$$
\begin{aligned}
a_1 &= 0.0378284554956993 & \beta_1 &= 0.0645388826835489 \\
a_2 &= -0.0238494650131592 & \beta_2 &= -0.0406894176455255 \\
a_3 &= -0.110624404085811 & \beta_3 &= -0.418092272881996 \\
a_4 &= 0.377402855512633 & \beta_4 &= 0.788485616984644 \\
a_5 &= 0.852698678836979 & \beta_5 &= -0.418092272881996 \\
a_6 &= 0.377402855512633 & \beta_6 &= -0.0406894176455255 \\
a_7 &= -0.110624404085811 & \beta_7 &= 0.0645388826835489 \\
a_8 &= -0.0238494650131592 & & \\
a_9 &= 0.0378284554956993 & &
\end{aligned}
$$

y N es la longitud del vector al que se aplicará la TWD y de los vectores u y v.

Las funciones duales Daubechies 9,7 se generan de forma similar a las funciones base, sólo que a partir de la Wavelet padre dual y la Wavelet madre dual que se muestran en las ecuaciones (4.20) y (4.21), como se muestra en la ecuación (4.17)

$$\tilde{W} = \{\tilde{\varphi}_k\}_{k=0}^{M-1} \cup \{\tilde{\psi}_k\}_{k=0}^{M-1}, \qquad M = \frac{N}{2} \tag{4.17}$$

donde

$$\tilde{\varphi}_k = \tilde{u}(n-2k) \tag{4.18}$$

$$\tilde{\psi}_k = \tilde{v}(n-2k) \tag{4.19}$$

$$\tilde{u} = (a_1,\ a_2,\ a_3,\ a_4,\ a_5,\ a_6,\ a_7,\ a_8,\ a_9,\ 0,\ \ldots,\ 0) \tag{4.20}$$

$$\tilde{v} = (0,\ 0,\ \beta_1,\ \beta_2,\ \beta_3,\ \beta_4,\ \beta_5,\ \beta_6,\ \beta_7,\ 0,\ \ldots,\ 0) \tag{4.21}$$

Las funciones base LeGall 5,3 también se generan a través de la ecuación (3.40), sus Wavelet padre y Wavelet madre se muestran en las ecuaciones (4.22) y (4.23).

$$u = \left(0,\ \frac{1}{2},\ 1,\ \frac{1}{2},\ 0,\ 0,\ \ldots,\ 0\right) \tag{4.22}$$

$$v = \left(0,\ -\frac{1}{8},\ -\frac{1}{4},\ \frac{3}{4},\ -\frac{1}{4},\ -\frac{1}{8},\ 0,\ \ldots,\ 0\right) \tag{4.23}$$

Las funciones duales LeGall 5,3 se pueden generar a través de las ecuaciones (4.17), (4.18) y (4.19) pero la Wavelet padre dual y la Wavelet madre dual en este caso son las que se muestran en las ecuaciones (4.24) y (4.25)

$$\tilde{u} = \left(-\frac{1}{8},\ \frac{1}{4},\ \frac{3}{4},\ \frac{1}{4},\ -\frac{1}{8},\ 0,\ 0,\ \ldots,\ 0\right) \tag{4.24}$$

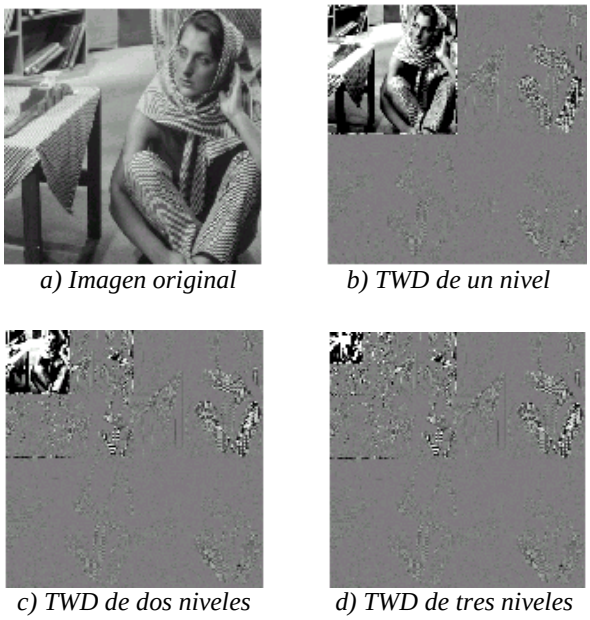
$$\tilde{v} = \left(0,\ 0,\ -\frac{1}{2},\ 1,\ -\frac{1}{2},\ 0,\ \ldots,\ 0\right) \tag{4.25}$$

La aplicación de la TWD con cualquiera de estas dos bases biortogonales a un vector *z* se realiza utilizando la ecuación (4.26), la cual se obtiene a partir de la ecuación (4.7)

$$\alpha_k = \begin{cases} \langle z, \tilde{\varphi}_k \rangle = \sum_{n=0}^{N-1} z(n)\tilde{\varphi}_k(n) & si \quad 0 \le k \le M-1 \\ \langle z, \tilde{\psi}_{k-M} \rangle = \sum_{n=0}^{N-1} z(n)\tilde{\psi}_{k-M}(n) & si \quad M \le k \le N-1 \end{cases} \tag{4.26}$$

donde $M = \frac{N}{2}$ y N es la longitud del vector *z*.

En la figura 4.5a se muestra una imagen a la cual se ha aplicado la TWD con la base Daubechies 9,7 utilizando Matlab, en la figura 4.5b se puede observar que los coeficientes de la sub-banda LL (ver figura 3.13) parecen una réplica en menor escala de la imagen original, lo cual se aprovecha para aplicar la TWD a la sub-banda LL y de ese modo lograr que más coeficientes tengan valores pequeños y puedan ser llevados a cero mediante Cuantización.



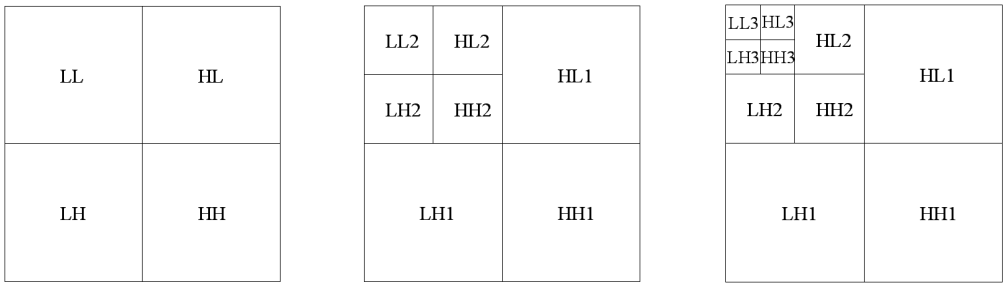
a) Imagen original *b) TWD de un nivel*

c) TWD de dos niveles *d) TWD de tres niveles*

Figura 4.5 Tres niveles de descomposición a una imagen

A cada nueva sub-banda LL generada se le puede aplicar la TWD, con lo cual se obtienen nuevos *niveles de descomposición*. Aunque el número máximo de niveles de descomposición permitido en la parte 1 de JPEG2000 es 32, usualmente no se logra mucha más compresión después de cinco niveles [17]. En la figura 4.5c se muestran los coeficientes resultantes de la TWD con dos niveles de descomposición y en la figura 4.5d se muestran los coeficientes con tres niveles de descomposición.

En la figura 4.6 se muestra un esquema de las sub-bandas que se generan al aplicar la TWD con uno, dos y tres niveles de descomposición.



a) 1 nivel *b) 2 niveles* *c) 3 niveles*

Figura 4.6 Sub-bandas generadas con uno, dos y tres niveles de descomposición

4.3.3 Transformada Wavelet Discreta Inversa

Aunque con una base biortogonal los coeficientes no se obtienen de la misma forma que con una base ortonormal, la forma de generar un vector *z* a partir de los coeficientes que lo representan con respecto a una base es la misma tanto para bases ortonormales como para bases biortogonales, por lo tanto, la TWDI con las bases Daubechies 9,7 y LeGall 5,3 se puede aplicar utilizando la ecuación (3.44).

La aplicación de la TWDI a una imagen con *n* niveles de descomposición consiste en aplicar la TWDI a las columnas y luego a las líneas de las cuatro sub-bandas del último nivel, con lo cual se disminuye en uno el número de niveles de descomposición, y volver a aplicar este procedimiento hasta llegar a la imagen original, en compresión sin pérdidas, o a una aproximación de ésta, en compresión con pérdidas. Por ejemplo, en la figura 4.5d la aplicación de la TWDI se realizaría primero en las sub-bandas LL3, HL3, LH3 y HH3, con lo cual se obtendrían los coeficientes que se muestran en la figura 4.5c, o una aproximación, luego se aplicaría la TWDI a las sub-bandas LL2, HL2, LH2 y HH2 y así sucesivamente hasta llegar a la imagen de la figura 4.5a.

En la figura 4.7a se muestra el resultado de aplicar la TWDI a los coeficientes de la figura 4.5d pero haciendo todos los coeficientes cero excepto por los de la sub-banda LL3, en la figura 4.7b se muestra la TWDI a los coeficientes de la misma figura con todos los coeficientes igual a cero excepto por los de las sub-bandas LL3, HL3, LH3 y HH3 y en la figura 4.7c se muestra el resultado de la TWDI usando todos los coeficientes.

a) Sub-banda LL3 *b) Sub-banda LL2* *c) Todas las sub-bandas*

Figura 4.7 TWDI utilizando la sub-banda LL3, la sub-banda LL2 y todas las sub-bandas

Las sub-bandas LL contienen la información de baja frecuencia de las imágenes, tanto horizontal como verticalmente, y las demás sub-bandas contienen la información de frecuencias altas, esa es la razón por la cual en la figura 4.7a los bordes no están bien definidos pero es posible saber qué son los objetos más grandes presentes en la imagen. En la figura 4.7b se han agregado las sub-bandas HL3, LH3 y HH3, con lo cual ahora la imagen es más parecida a la original debido a que la información de esas sub-bandas aportó una mejor definición a los bordes y a los objetos pequeños. En la figura 4.7c podemos observar que la única información apreciable que han agregado las sub-bandas HL2, LH2, HH2, HL1, LH1 y HH1 es una mayor resolución en los patrones de rayas presentes en la ropa de la mujer y en el mantel de la mesa, por lo cual se pueden llevar a cero más coeficientes de estas sub-bandas que a los de las sub-bandas LL3, HL3, LH3 y HH3 mediante Cuantización sin afectar de forma notable a la calidad de la imagen recuperada.

4.4 Cuantización

En esta etapa, la cual sólo se aplica en compresión con pérdidas, los coeficientes que la TWD genera se cuantizan utilizando un cuantizador uniforme con zona muerta [17]. En la figura 4.8 se muestra la gráfica del cuantizador utilizado por JPEG2000.

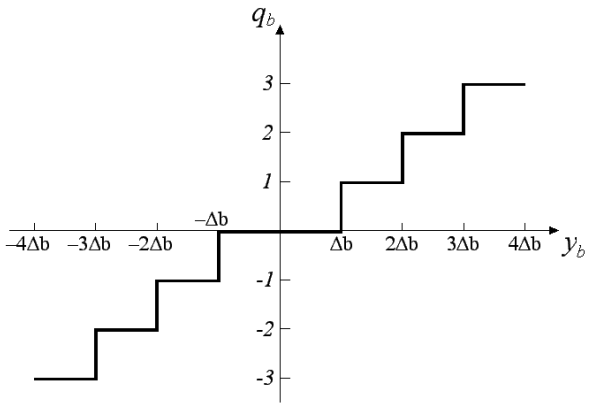


Figura 4.8 Cuantizador Uniforme de JPEG2000

En la figura 4.8 se puede apreciar que la longitud de la zona muerta es del doble que el tamaño de paso, Δ*b*. La aplicación de la Cuantización a los coeficientes se realiza utilizando la ecuación (4.27) [17]

$$q_b(x,y) = sign(y_b(x,y)) \left\lfloor \frac{|y_b(x,y)|}{\Delta b} \right\rfloor \qquad \textbf{(4.27)}$$

donde la función *sign()* regresa el signo, positivo o negativo, de su argumento, y los símbolos ⌊ ⌋ significan que los decimales del número en su interior son truncados.

Por ejemplo, la secuencia *y* al ser cuantizada con los tamaños de paso Δ*b* = 1.5 y Δ*b* = 4 da como resultado las secuencias *q'* y *q''*:

$y =$	−11.6	–9.2	–6.8	–4.4	–2.0	–0.6	0.4	1.6	2.8	5.2	7.6
$q' =$	−7	−6	−4	−2	−1	0	0	1	1	3	5
$q'' =$	−2	−2	−1	−1	0	0	0	0	0	1	1

Durante la descompresión, el proceso inverso a la Cuantización es la Decuantización, en la cual se recupera una aproximación de los valores originales y consiste simplemente en multiplicar los valores cuantizados por el tamaño de paso. Por ejemplo, la Decuantización de las secuencias *q'* y *q''* da como resultado las secuencias *y'* y *y''*:

$y =$	−11.6	–9.2	–6.8	–4.4	–2.0	–0.6	0.4	1.6	2.8	5.2	7.6
$y' =$	−10.5	−9	−6	−3	−1.5	0	0	1.5	1.5	4.5	7.5
$y'' =$	−8	−8	−4	−4	0	0	0	0	0	4	4

La secuencia *y'* es más parecida a la secuencia original que la secuencia *y''* debido a que tiene un tamaño de paso más cercano a uno, sin embargo, los tamaños de paso mayores son los que llevan un mayor número de coeficientes a cero y son los que ofrecen las mayores tasas de compresión, por lo tanto, cuando se escoge el tamaño de paso se tiene el objetivo de lograr un equilibrio entre la calidad deseada de la imagen recuperada y la tasa de compresión.

En el estándar JPEG2000 se permite la aplicación de una Cuantización diferente a cada una de las sub-bandas que genera la TWD, asignando a aquellas que cooperan menos con la calidad visual de la imagen, es decir, las sub-bandas HL, LH y HH, tamaños de paso mayores.

En JPEG se reduce la cantidad de datos a comprimir submuestreando las crominancias en formato 4:2:2 ó 4:2:0, en JPEG2000 esto no está permitido, sin embargo, llevar a cero los coeficientes de las sub-bandas HL1, LH1 y HH1 tiene el mismo efecto que submuestrear en formato 4:2:0 [11].

4.5 Codificación Aritmética Binaria

El siguiente paso en el proceso de compresión es aplicar a los coeficientes, sin importar si fueron cuantizados o no, un método de compresión sin pérdidas conocido como Codificación Aritmética Binaria (CAB) [17], el cual sigue los mismos principios que la Codificación Aritmética. En la CAB los datos de entrada son bits individuales conocidos como *valor de decisión binaria* (D), cada uno de los cuales está asociado a una probabilidad a través de un dato conocido como *contexto* (CX).

Antes de aplicar la CAB, se deben generar los datos D y CX a partir de los coeficientes mediante una codificación conocida como *Codificación de Planos de Bits* (CPB).

4.5.1 Codificación de Planos de Bits

Para realizar la CPB en JPEG2000 se utiliza un algoritmo diseñado por David S. Taubman conocido como EBCOT (Embedded Block Coding with Optimised Truncation), el cual consiste en dividir cada una de las sub-bandas en bloques de coeficientes, usualmente de

32x32 ó 64x64, representarlos en magnitud y signo en vez de complemento a dos, dividir los bloques en *planos de bits* y a partir de ellos generar los datos D y CX que se requieren para aplicar la CAB. Dado que la CAB es un método de compresión sin pérdidas, la división en bloques de coeficientes no genera artefactos [17].

En la figura 4.9 se muestra con un ejemplo la forma en que se divide un bloque de coeficientes en planos de bits. El primer paso es colocar en una matriz *v* únicamente la magnitud de los coeficientes en forma binaria y colocar en una matriz χ los signos, con un 1 representando los signos negativos y con un 0 representando los signos positivos, como se muestra la figura 4.9b. Luego, se generan tantos planos de bits como bits se utilicen para representar la magnitud de los coeficientes, por ejemplo, en la figura 4.9b se puede observar que se están utilizando 3 bits, por lo tanto, se deben generar 3 planos de bits, el primero se representa con v^2 y se forma tomando el bit más significativo de la matriz *v* (figura 4.9c), el segundo se representa con v^1 y se forma tomando los bits centrales (figura 4.9d) y el último plano se representa con v^0 y se forma tomando los bits menos significativos (figura 4.9e). El bit que se encuentra en la posición *(m,n)*, donde *m* es la línea y *n* la columna, del plano de bits v^P se representa de esta forma: $v^P(m,n)$.

6	1	-1	0
0	-2	2	3
2	1	-1	0
5	1	-2	0

a) Bloque de coeficientes

6 110	1 001	1 001	0 000
0 000	2 010	2 010	3 011
2 010	1 001	1 001	0 000
5 101	1 001	2 010	0 000

v

0	0	1	0
0	1	0	0
0	0	1	0
0	0	1	0

χ

b) Representación en magnitud y signo

1	0	0	0
0	0	0	0
0	0	0	0
1	0	0	0

c) v^2

1	0	0	0
0	1	1	1
1	0	0	0
0	0	1	0

d) v^1

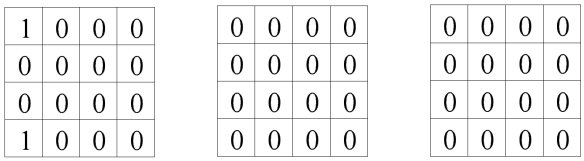
0	1	1	0
0	0	0	1
0	1	1	0
1	1	0	0

e) v^0

Figura 4.9 Formación de planos de bits

El algoritmo EBCOT traduce cada plano de bits en datos D y CX comenzando con el plano de bits más significativos, pero ignora los primeros planos de bits si éstos contienen sólo 0's y empieza la codificación con el primer plano de bits que contenga por lo menos un 1, en el caso de la figura 4.9 el primer plano a codificar es el plano v^2.

El algoritmo EBCOT utiliza tres matrices de banderas durante la codificación, σ, η y σ', las cuales tienen las mismas dimensiones que la matriz de coeficientes, ya que cada bandera corresponde a un coeficiente, inicialmente todas las banderas están puestas en cero. En la figura 4.10 se muestra el estado de estas banderas después de haber sido codificado el primer plano de bits, es decir, el plano v^2.



a) σ *b) η* *c) σ'*

Figura 4.10 Banderas utilizadas por EBCOT

Cuando el algoritmo EBCOT codifica el primer bit igual a 1 de un coeficiente en cualquiera de los planos de bits, se dice que ese coeficiente se vuelve *significativo* y se marca poniendo en uno la bandera correspondiente en la matriz σ, y cuando un coeficiente tiene por lo menos un vecino significativo, se dice que está en un *vecindario preferente*, por ejemplo, en la figura 4.10a podemos observar que después de haber sido codificado el primer plano de bits, los coeficientes que se encuentran en las posiciones *(0,0)* y *(3,0)* se han vuelto significativos y que sus vecinos, es decir, los coeficientes que se encuentran en las posiciones *(1,0), (1,1), (0,1), (2,0), (2,1)* y *(3,1),* ahora están en un vecindario preferente.

En cada plano de bits el algoritmo EBCOT utiliza tres procedimientos: Significant Propagation Pass (SPP), el cual codifica los bits de los coeficientes que todavía no se hayan vuelto significativos y que se encuentren en un vecindario preferente, Magnitud Refinement Pass (MRP), el cual codifica los bits de los coeficientes que se volvieron significativos en planos de bits anteriores, y Clean Up Pass (CUP), el cual codifica los bits que no fueron codificados por ninguno de los dos procedimientos anteriores.

Cada procedimiento recorre los planos de bits en el orden que se muestra en la figura 4.11 y en cada bit debe decidir, utilizando las banderas σ y η, si debe codificar ese bit o si debe dejarlo para los procedimientos siguientes. Cuatro bits en sentido vertical forman una *tira*. El recorrido comienza desde la esquina superior izquierda, se recorre una tira por columna hasta que se llega a la última y luego se recorren las tiras siguientes, se ha decidido que cada tira sea de cuatro bits con el propósito de facilitar la implementación del EBCOT tanto en software como en hardware [11]. Si las columnas no tienen un número de bits que sea múltiplo de 4, las últimas tiras pueden tener menos de cuatro bits, como puede observarse en la parte inferior de la figura 4.11 [17].

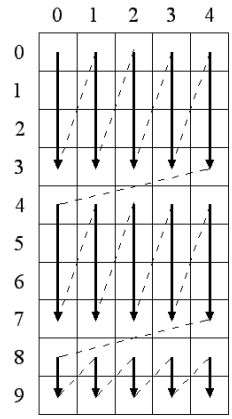


Figura 4.11 Orden en que se codifican los bits en un plano de bits

Significant Propagation Pass

El SSP codifica los bits que todavía no se han vuelto significativos y que están en un vecindario preferente, D es igual al bit que se está codificando y CX puede tener un valor entre cero y ocho. En el SSP CX se utiliza para indicar a la CAB la probabilidad de que el coeficiente cuyo bit se va a codificar se vuelva significativo en el plano de bits actual y depende del estado de sus ocho vecinos, en la figura 4.12 se muestra un esquema de los ocho vecinos que tiene un elemento X en la matriz σ (los vecinos que caen fuera de la matriz se consideran no significativos), por ejemplo, si todos los vecinos de X están puestos en uno, existe una alta probabilidad de que el coeficiente que le corresponde a X se vuelva significativo en el plano de bits actual, en cambio, si sólo uno de sus vecinos está puesto en uno, la probabilidad es menor.

D_0	V_0	D_1
H_0	X	H_1
D_3	V_1	D_2

Figura 4.12 Esquema de los vecinos de un elemento X en la matriz σ

Para seleccionar un CX, el SPP toma en cuenta la sub-banda en que se encuentra el bloque de coeficientes que se está codificando. Debido a que las sub-bandas LL y LH tienen información de frecuencias bajas en sentido horizontal, los vecinos H_0 y H_1 influyen más en la probabilidad que tiene un coeficiente de volverse significativo que los vecinos V_i o D_i, por lo cual para codificar un bloque de coeficientes de la sub-banda LL o de la sub-banda LH, se utiliza la tabla 4.1, en cambio, si el bloque se encuentra en la sub-banda HL los vecinos que influyen más son V_0 y V_1 y se utiliza la tabla 4.2, y si el bloque está en la sub-banda HH los vecinos que más influyen son los D_i y se utiliza la tabla 4.3.

ΣH_i	ΣV_i	ΣD_i	CX
2	X	X	8
1	≥1	X	7
1	0	≥1	6
1	0	0	5
0	2	X	4
0	1	X	3
0	0	≥2	2
0	0	1	1
0	0	0	0

Tabla 4.1 CX en las sub-bandas LL y LH

ΣH_i	ΣV_i	ΣD_i	CX
X	2	X	8
≥1	1	X	7
0	1	≥1	6
0	1	0	5
2	0	X	4
1	0	X	3
0	0	≥2	2
0	0	1	1
0	0	0	0

Tabla 4.2 CX en la sub-banda HL

$\Sigma H_i+\Sigma V_i$	ΣD_i	CX
X	≥3	8
≥1	2	7
0	2	6
≥2	1	5
1	1	4
0	1	3
≥2	0	2
1	0	1
0	0	0

Tabla 4.3 CX en la sub-banda HH

Si D es igual a 1, el bit codificado es el primero del coeficiente al que pertenece que no es cero, por lo tanto, el coeficiente se vuelve significativo y se marca poniendo en uno la bandera correspondiente en la matriz σ. Las banderas η se utilizan para evitar que el bit de un coeficiente que se acaba de volver significativo se vuelva a codificar en alguno de los dos procedimientos siguientes, por lo tanto, cuando un coeficiente se vuelve significativo se pone en uno su bandera η y cuando ya se han aplicado los tres procedimientos a un plano de bits todas las banderas η se apagan.

Cada vez que un coeficiente se vuelve significativo, tanto en el SPP como en el CUP, su signo se traduce en un nuevo par D, CX. Debido a que los coeficientes que están relacionados con lugares en que la imagen tiene bordes horizontales y verticales usualmente tienen el mismo signo y los que están relacionados con lugares que se encuentran en lados opuestos de un borde usualmente tienen signos diferentes, el algoritmo EBCOT toma en cuenta los signos de los vecinos en sentido horizontal y vertical que ya son significativos para escoger uno de los cinco contextos que se encuentran en la tabla 4.4, la cual utiliza dos valores de referencia, H y V, que sirven para indicar la situación en que se encuentran los vecinos en sentido horizontal (H) y los vecinos en sentido vertical (V). Los valores de referencia indican una de las tres situaciones que se enlistan a continuación:

- 0: ningún vecino es significativo o ambos son significativos pero tienen signos opuestos.
- 1: un vecino o los dos son significativos y tienen signos positivos.
- -1: un vecino o los dos son significativos y tienen signos negativos.

H	V	$\hat{\chi}$	CX
1	1	0	13
1	0	0	12
1	-1	0	11
0	1	0	10
0	0	0	9
0	-1	1	10
-1	1	1	11
-1	0	1	12
-1	-1	1	13

Tabla 4.4 CX en la codificación del signo de un coeficiente

Los valores de referencia se calculan utilizando las ecuaciones (4.28) y (4.29).

$$H = \min\{1,\ \max\{-1,\ \sigma(m,n-1)[1-2\chi(m,n-1)]+\sigma(m,n+1)[1-2\chi(m,n+1)]\}\} \quad \textbf{(4.28)}$$

$$V = \min\{1,\ \max\{-1,\ \sigma(m-1,n)[1-2\chi(m-1,n)]+\sigma(m+1,n)[1-2\chi(m+1,n)]\}\} \quad \textbf{(4.29)}$$

donde *m* es la línea en que se encuentra el coeficiente cuyo signo se está codificando y *n* es la columna.

El valor de D se determina a través de la tabla 4.5. $\hat{\chi}$ se obtiene de la tabla 4.4.

$\hat{\chi}$	$\chi(m,n)$	D
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 4.5 D en la codificación del signo de un coeficiente

Por ejemplo, para aplicar el SPP al plano de bits de la figura 4.9d comenzamos el recorrido con el bit $v^1(0,0)$, de acuerdo con la figura 4.11, sin embargo, el coeficiente al que pertenece ya es significativo ($\sigma(0,0) = 1$, figura 4.10a), por lo tanto, lo dejamos para los procedimientos siguientes y pasamos al bit $v^1(1,0)$, este bit pertenece a un coeficiente no significativo y que está en un vecindario preferente, por lo tanto, $D = v^1(1,0) = 0$ y CX se obtiene de la tabla 4.1, suponiendo que el bloque de coeficientes se encuentra en la sub-banda LL. A través del esquema de la figura 4.12 y de la figura 4.10a, observamos que el único vecino en la matriz σ que está puesto en uno es V_0, por lo tanto, $\Sigma H_i=0$, $\Sigma V_i=1$, $\Sigma D_i=0$ y, de acuerdo con la tabla 4.1, CX = 3. Debido a que D no es igual a 1, el coeficiente al que pertenece el bit $v^1(1,0)$ no se ha vuelto significativo todavía, la codificación de su signo no se realiza en este SPP y $\sigma(1,0)$ y $\eta(1,0)$ continúan en cero. De forma similar se codifican los demás bits del plano v^1.

Magnitud Refinement Pass

El MRP codifica los bits de los coeficientes que se volvieron significativos en un plano de bits anterior, es decir, aquellos cuya $\sigma(m,n)$ está en uno y $\eta(m,n)$ está en cero, ya que si $\eta(m,n)$ está prendida, el coeficiente correspondiente no se volvió significativo en un plano anterior, sino en el SPP del plano actual. D es igual al bit que se está codificando y CX toma el valor que le corresponde de acuerdo con la tabla 4.6. Como se puede observar en la segunda columna, el MRP, al igual que el SPP, toma en cuenta si los vecinos del coeficiente cuyo bit se está codificando ya se han vuelto significativos.

$\sigma'(m,n)$	$\Sigma H+\Sigma V+\Sigma D$	CX
1	X	16
0	≥ 1	15
0	0	14

Tabla 4.6 CX en MRP

Las banderas σ' se utilizan para indicar cuáles coeficientes tienen por lo menos un bit al cual ya se ha aplicado el MRP, por lo tanto, después de aplicar por primera vez el MRP a un bit de un coeficiente, se pone en uno la bandera que corresponde a ese coeficiente en la matriz σ'.

Por ejemplo, para aplicar el MRP al plano de la figura 4.9d comenzamos el recorrido con el bit $v^1(0,0)$, el cual pertenece a un coeficiente que se volvió significativo en un plano de bits anterior ($\sigma(0,0) = 1$, $\eta(0,0) = 0$, figura 4.10), por lo tanto, $D = v^1(0,0) = 1$ y CX se obtiene de la tabla 4.6. La bandera $\sigma'(0,0)$ está en cero debido a que antes de esta ocasión no se había aplicado el MRP a ningún bit del coeficiente que le corresponde y, suponiendo que ningún coeficiente se volvió significativo en el SPP anterior, $\Sigma H+\Sigma V+\Sigma D = 0$, por lo

tanto, CX = 14. Dado que está es la primera vez que se aplica el MRP a un bit del coeficiente que corresponde a la bandera $\sigma'(0,0)$, ésta pone en uno.

Clean Up Pass

El CUP codifica los bits que no fueron codificados por el SSP ($\eta(m,n) = 0$) ni por el MRP ($\sigma(m,n) = 0$). Generalmente, en este punto de la codificación los coeficientes tienen pocas probabilidades de volverse significativos, por lo tanto, para codificarlos eficientemente, al principio de cada tira se verifica que ninguno de sus coeficientes sea significativo y que ninguno esté en un vecindario preferente, si se cumplen estas dos condiciones se pueden presentar dos casos: que todos los bits de la tira sean cero, entonces la tira completa se codifica con D=0 y CX=17, o que haya por lo menos un uno, entonces se codifica el primer 1 y los ceros anteriores en la tira con tres pares D, CX de acuerdo con la tabla 4.7 y los demás bits de la tira se codifican de la misma forma que en el SSP y se pasa a la siguiente tira. Si no se cumplen las dos condiciones anteriores, todos los bits de la tira se codifican de la misma forma que en el SPP.

bits	D	CX	bits	D	CX	bits	D	CX	bits	D	CX
1	1	17	0	1	17	0	1	17	0	1	17
	0	18	1	0	18	0	1	18	0	1	18
	0	18		1	18	1	0	18	0	1	18
									1		

Tabla 4.7 D y CX si la tira no tiene coeficientes significativos ni en un vecindario preferente

Antes de codificar el primer plano de bits ningún coeficiente se ha vuelto significativo todavía, por lo tanto, ninguno está en un vecindario preferente y no se puede aplicar el SPP. El MRP tampoco se puede aplicar porque no hay coeficientes que se hayan vuelto significativos en planos anteriores debido a que éste es el primero, por lo tanto, al primer plano de bits sólo se le aplica el CUP.

Por ejemplo, debido a que el plano de bits de la figura 4.9c es el primero de su bloque de coeficientes, el único procedimiento que se le aplica es el CUP. Dado que al principio de la codificación todas las banderas están en cero, la primera tira cumple las condiciones de tener sólo bits cuyos coeficientes no son significativos y no están en un vecindario preferente, por lo tanto, el primer 1 y los ceros anteriores se codifican con la tabla 4.7. Debido a que en este caso el primer 1 está al principio de la tira, éste se codifica con los pares D, CX: *(1, 17), (0, 18)* y *(0, 18)* y los otros tres bits de la tira se codifican de la misma forma que en el SPP.

En el ejemplo que estamos analizando se acaba de codificar el bit $v^2(0,0)$, el cual es el primero del coeficiente al que pertenece que es igual a 1, por lo tanto, ese coeficiente se acaba de volver significativo, se debe codificar su signo de la misma forma que en el SPP y se tiene que poner en uno la bandera $\sigma(0,0)$. Dado que en este punto ninguno de los vecinos del coeficiente al que pertenece el bit $v^2(0,0)$ se ha vuelto significativo, los valores de referencia H y V son iguales a cero, por lo tanto, de acuerdo con la tabla 4.4, CX = 9. El valor de D se obtiene de la tabla 4.5, en la tabla 4.4 podemos observar que $\hat{\chi} = 0$ y en la figura 4.9b observamos que $\chi(0,0) = 0$, por lo tanto, D = 0.

4.5.2 Codificador MQ

En la sección anterior vimos que el algoritmo EBCOT produce una secuencia de pares D, CX. Los datos D son los datos de entrada del codificador aritmético binario utilizado por JPEG2000, conocido como *Codificador MQ*, y los datos CX sirven para relacionar una probabilidad con cada símbolo D a través de una tabla que provee el estándar [17]. El nombre de Codificador MQ fue acuñado para indicar que sus orígenes se encuentran en un codificador conocido como Codificador Q.

En el Codificador MQ, al igual que en la Codificación Aritmética, se parte de un intervalo inicial que se va reduciendo conforme se codifican los datos de entrada. Dado que los datos D sólo pueden tomar uno de dos valores, 0 ó 1, el intervalo original se divide en dos sub-intervalos, como se muestra en la figura 4.13, sin embargo, estos sub-intervalos no corresponden a los valores 0 y 1 de forma directa, sino al símbolo más probable (MPS) y al símbolo menos probable (LPS). Inicialmente, el 0 es el MPS, sin embargo, durante la codificación las probabilidades del MPS y del LPS se van adaptando a los datos de entrada y el 0 puede dejar de ser el MPS y convertirse en el LPS, esto se debe a que cuando el codificador recibe 0's, incrementa la probabilidad que tiene asignada al MPS y cuando recibe 1's disminuye esa probabilidad hasta que llega a un punto en que decide que el MPS ahora sea el 1 y el LPS sea el 0.

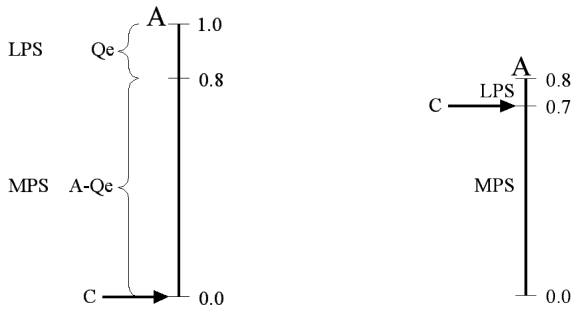


Figura 4.13 Intervalo A y sub-intervalos MPS y LPS en la Codificación MQ

La longitud del intervalo inicial se almacena en una variable llamada A, la longitud del sub-intervalo LPS (LPS_L) depende de la probabilidad de que el siguiente D sea LPS, la cual se representa con Qe y sus posibles valores se encuentran en la tabla B.1 del anexo B, la longitud del intervalo MPS (MPS_L) es igual a la longitud del intervalo A menos la longitud del sub-intervalo LPS (MPS_L = A-LPS_L) y el resultado de la codificación se almacena en una variable llamada C, la cual inicialmente vale 0.

Con el propósito de hacer más eficiente la implementación, las multiplicaciones que se requerían para calcular la longitud de los sub-intervalos en la Codificación Aritmética se han sustituido con aproximaciones de esta forma: el valor de A siempre se mantiene cercano a 1 mediante un procedimiento conocido como *renormalización*, el cual consiste en duplicar el valor de A cada vez que cae debajo de 0.75, entonces en vez de calcular LPS_L con AxQe, se aproxima con $LPS_L \approx Qe$ y MPS_L con MPS_L = A-$LPS_L \approx$ A-Qe.

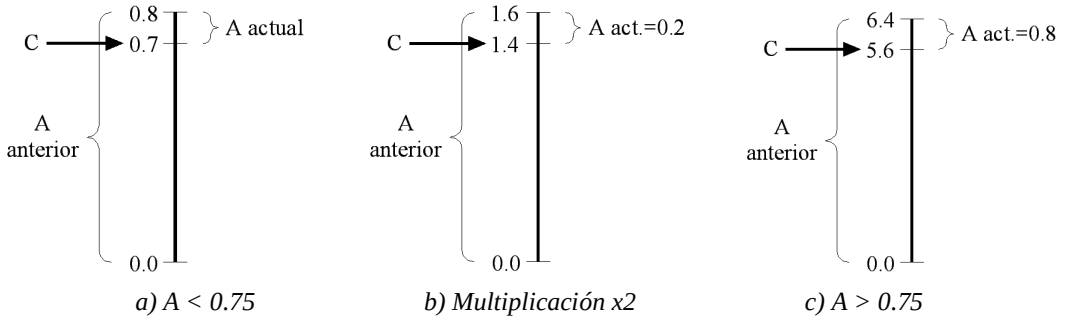
Por ejemplo, el la figura 4.13a se muestra lo que sucedería en caso de que el intervalo A fuera inicializado en 1.0, el primer Qe fuera igual a 0.2 y el primer D fuera MPS, LPS_L

sería igual a Qe = 0.2, MPS_L sería A-Qe = 0.8 y el primer D quedaría representado con C igual a cualquier valor entre 0.0 y 0.8, sin embargo, por simplicidad C siempre apunta al límite inferior del sub-intervalo, por lo tanto, cuando D = MPS el valor de C no se modifica. Cada vez que se codifica un nuevo D, la longitud de A toma el valor de MPS_L o de LPS_L de acuerdo con el valor de D, en este caso D = MPS, por lo tanto, el nuevo valor de A es A = MPS_L = 0.8, como se muestra en la figura 4.13b.

Si el siguiente D es LPS y el nuevo valor de Qe es 0.1, a C se le suma MPS_L utilizando la ecuación (4.30) para que apunte al límite inferior del sub-intervalo LPS, como se muestra en la figura 4.13b, y la longitud de A ahora es A = LPS_L = Qe = 0.1.

$$C = C + MPS_L = C + A - Qe = 0.0 + 0.8 - 0.1 = 0.7 \quad \textbf{(4.30)}$$

En la figura 4.14a se muestra la longitud del intervalo A del ejemplo anterior antes de la codificación del último D, la longitud del intervalo A actual y el lugar al que apunta la variable C, la *renormalización* consiste en multiplicar estos valores por 2 hasta que A actual se vuelva mayor a 0.75, en la figura 4.14b se muestra el resultado que se obtiene al realizar la multiplicación por 2 una vez y en la figura 4.14c al realizar dicha multiplicación dos veces más. En las figuras 4.14b y 4.14c se puede observar que el lugar al que apunta C no cambia con respecto a los límites inferior y superior de A anterior con las multiplicaciones, por lo tanto, la información de los datos D ya codificados no se ha perdido y A actual está cerca de 1, con lo cual ahora la longitud de los nuevos sub-intervalos se pueden obtener mediante las aproximaciones $LPS_L \approx$ Qe y $MPS_L \approx$ A-Qe.

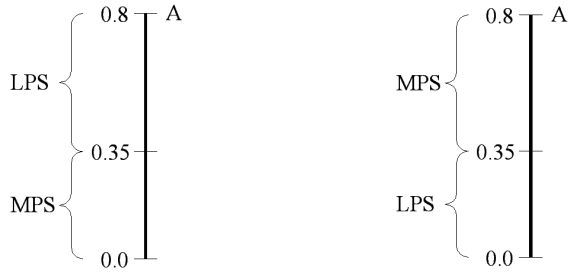


a) A < 0.75 *b) Multiplicación x2* *c) A > 0.75*

Figura 4.14 Renormalización del intervalo A y la variable C

En realidad, conocer la longitud de A anterior no tiene ninguna utilidad, por lo cual durante la renormalización este valor no se toma en cuenta y únicamente se multiplican por 2 A actual y C.

Un problema que se puede presentar con la aproximación es que LPS_L puede hacerse más grande que MPS_L, por ejemplo, si A = 0.8 y Qe = 0.45, LPS_L sería 0.45 y MPS_L 0.35 como se muestra en la figura 4.15a, cuando esto sucede se aplica un procedimiento conocido como *intercambio condicional*, en el cual para codificar el siguiente D, los sub-intervalos LPS y MPS intercambian lugares como se muestra en la figura 4.15b y después retoman sus posiciones originales.



a) $LPS_L > MPS_L$ *b) $MPS_L > LPS_L$*

Figura 4.15 Intercambio condicional

Hasta ahora se han usado valores en punto flotante para representar al intervalo A, la probabilidad del LPS, Qe, y el valor de la variable C con el propósito de hacer más comprensible la forma en que trabaja la Codificación Aritmética Binaria, sin embargo, el Codificador MQ utiliza únicamente valores en hexadecimal y los valores de A y C se almacenan en el registro de 16 bits y en el registro de 32 bits que se muestran en la figura 4.16, respectivamente. El valor inicial de A es 0x0000, el límite inferior que debe pasar para que se lleve a cabo una renormalización es 0x8000 y C se inicializa en cero. La renormalización se realiza desplazando a la izquierda los bits de los registros A y C, lo cual tiene el mismo efecto que la multiplicación por 2.

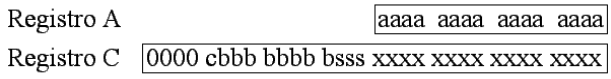


Figura 4.16 Registros A y C del Codificador MQ

Para codificar un dato D, el Codificador MQ primero averigua si D es MPS o LPS y obtiene el valor de Qe utilizando el CX con el que está asociado D, la tabla 4.8 y la tabla B.1. La segunda columna de la tabla 4.8 indica el MPS de cada contexto, si el valor de D coincide con el MPS de su contexto, entonces es MPS, en caso contrario, es LPS. La tercera columna de la misma tabla contiene el índice con el cual se obtiene el valor de Qe en la tabla B.1, por ejemplo, si el primer par D, CX es (0, 17) el Codificador MQ observa en la tabla 4.8 que el MPS de CX = 17 coincide con el valor de D, por lo tanto D es MPS, luego observa que el índice de CX = 17 es 3, al cual le corresponde un Qe igual a 0x0AC1 de acuerdo con la tabla B.1.

Contexto (CX)	MPS	Índice
0	0	4
1	0	0
2	0	0
⋮	⋮	⋮
15	0	0
16	0	0
17	0	3
18	0	46

Tabla 4.8 MPS inicial e índice inicial

Dado que el D del ejemplo que se está analizando es MPS, C no sufre ningún cambio y el nuevo valor de A es A = MPS_L = A-Qe = 0000-0AC1 = F53F. La columna NMPS de la tabla B.1 muestra el nuevo índice que tendrá asignado el CX que se esté utilizando y que se coloca en la línea que le corresponde de la tabla 4.8 en lugar del índice anterior en caso de que D haya sido MPS y la columna NLPS muestra el nuevo índice en caso de que D haya sido LPS, en el ejemplo actual D es MPS, por lo tanto, el nuevo índice de CX = 17 es 4. Cuando se está en un índice que en la columna SWITCH tiene un 1 y D es LPS, el valor del MPS se cambia a 1, si su valor anterior era 0, o a 0, si su valor anterior era 1, en la línea correspondiente de la tabla 4.8. El índice actual, el 3, tiene un SWITCH igual a 0, por lo tanto, el MPS de CX = 17 no cambia.

Cuando al registro C se le asigna un valor por primera vez a través de la ecuación (4.30), los bits x (figura 4.16) son los únicos que se están utilizando y los demás permanecen en cero. El Codificador MQ utiliza un contador CT que se inicializa en 11 y se va decrementando con cada uno de los desplazamientos a la izquierda que se realiza en las renormalizaciones y cuando este contador llega a cero, significa que los bits del registro C se han recorrido a la izquierda 11 veces y que los bits s, b y, posiblemente, el bit de carry c están ocupados, entonces los bits b y c se guardan en un registro temporal T de 32 bits, se borran del registro C y se reinicializa el contador CT pero de ahora en adelante en 8.

En el buffer B que se muestra en la figura 4.17 se almacenan los datos finales de la imagen comprimida a través del registro T. El buffer B cuenta con un apuntador BP que siempre está apuntando al byte anterior a aquel en que se almacenará el siguiente dato.

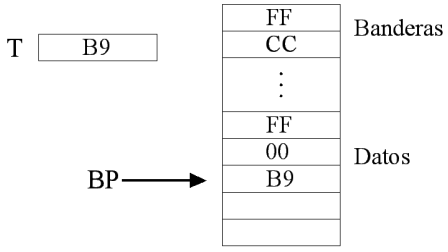


Figura 4.17 Buffer B

JPEG2000 utiliza una serie de banderas en el archivo que contiene la imagen comprimida para comunicar al descompresor los parámetros que son necesarios para descomprimir la imagen, tales como su tamaño, el número de niveles de descomposición, etc., las cuales serán explicadas en la sección 4.6. Para diferenciar estas banderas de los datos comprimidos se coloca un byte FF antes de cada bandera y en caso de que en los datos comprimidos aparezca un byte FF se coloca un byte 00 para evitar que el siguiente dato comprimido se confunda con una bandera, por ejemplo, en la figura 4.17 se puede observar que antes del byte CC se coloca el byte FF para indicar que es una bandera y se puede observar que en los datos comprimidos apareció un byte FF y que para evitar que el byte B9 se confundiera con una bandera se colocó un 00 después del byte FF.

En la figura 4.18 se muestra lo que sucede cuando en el registro T aparece un valor mayor a 0xFF, es decir, cuando el bit de carry c del registro C está en uno (figura 4.16). Debido a que los bytes llegan al buffer B de los más significativos a los menos, el bit c se suma al byte al que está apuntando BP, BP se incrementa y se almacena el dato que había en el registro T pero sin el carry.

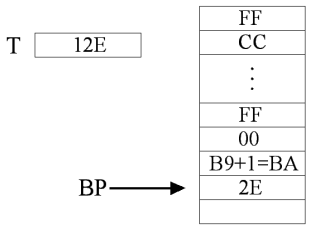
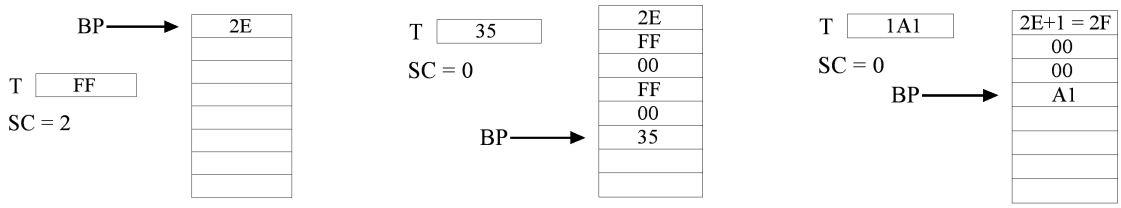


Figura 4.18 T > FF

Cada vez que en el registro T aparece un FF, en vez de ser almacenado inmediatamente en el buffer B, se incrementa un contador llamado Stack Counter (SC), el cual inicialmente está en cero, como se muestra en la figura 4.19a. Se utiliza el contador SC debido a que si en el siguiente T el carry está en uno, al sumarlo a 0xFF éste se convertiría en 0x100 pero sólo se almacenaría el 00 y el 1 se perdería. Si después de un T = FF llega un T < FF, como se muestra en la figura 4.18b, se almacena un FF seguido de un 00, se decrementa SC, se repiten los pasos anteriores hasta que SC es igual a 0 y se almacena el nuevo T.



a) T = FF, SC = SC+1. *b) SC ≠ 0, T < FF.* *c) SC ≠ 0, T > FF.*

Figura 4.19 Forma en que maneja los bytes FF el buffer B

Si SC es diferente de 0 y en vez de llegar un T < FF llega un T > FF, como se muestra en la figura 4.18c, se le suma 1 al byte al que está apuntando BP, se almacenan tantos 00's como indique el SC (figura 4.19a), se reinicializa SC y se almacena el nuevo T pero sin el carry, esto se debe a que el carry convirtió los FF's que estaban pendientes en 00's y se propagó hasta el byte que se almacenó antes de que llegaran los FF's.

Después de que el Codificador MQ codificó todos los pares D, CX que generó el EBCOT los bits que aún se encuentran en el registro C se pasan al buffer B mediante un procedimiento conocido como *flush*, el cual tiene el propósito de colocar en el registro C el mayor número de ceros posibles antes de enviar los dos últimos bytes al buffer B pero sin que C deje de apuntar a un lugar dentro del intervalo A final.

En la figura 4.20 se muestra con un ejemplo los pasos que se llevan a cabo en el procedimiento flush. El primer paso es realizar la suma C+A-1 y asignarla al registro T, con lo cual T queda apuntando justo debajo del límite superior del intervalo A, luego al registro T se le borran los 16 bits menos significativos, si T < C como sucede en la figura 4.20 se le suma 0x8000, luego los bits del registro T se desplazan a la izquierda tantas veces como indique CT y se asigna a C, se envían los bits b del registro C al buffer B a través del registro T, se desplazan a la izquierda los bits del registro C ocho veces y se envían los bits b al bufer B por última vez.

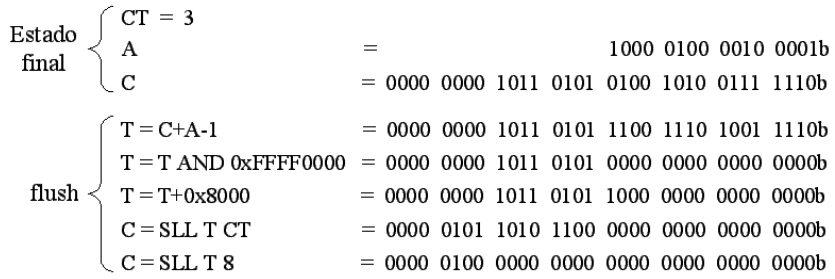


Figura 4.20 Procedimiento flush

4.5.3 Decodificador MQ

La Decodificación MQ es la primera etapa en el proceso de descompresión, en ella se toman los datos comprimidos que se encuentran en el buffer B y se recuperan los pares D, CX y los planos de bits. Una vez que se tienen los planos de bits, se reconstruyen los bloques de coeficientes, se descuantizan si es el caso, se aplica la TWDI y se recupera una imagen en RGB a través de la ecuación (4.4) o la ecuación (4.5).

Durante la compresión la Codificación MQ se realiza después del algoritmo EBCOT, sin embargo, en la descompresión es necesario llevar a cabo la Decodificación MQ y la recuperación de los planos de bits en un mismo proceso.

En la figura 4.21 se muestra el intervalo A utilizado por el Decodificador MQ, el registro en el que se almacena y el registro C, al cual se transfieren los datos desde el buffer B para ser decodificados. El registro C se compone de dos registros, el registro Cx y el registro C-low, al principio de la decodificación el registro Cx se carga con los dos primeros bytes del buffer B, con lo cual queda apuntando a un lugar dentro del intervalo A como se muestra en la figura 4.21. Si Cx es mayor que A-Qe entonces el primer D fue LPS y si es menor fue MPS, sin embargo, para saber el valor de Qe es necesario conocer el CX con el que está asociado para así obtener un índice en la tabla 4.8 y con ese índice obtener el valor de Qe en la tabla B.1. El Decodificador MQ utiliza la tabla 4.8 con los valores iniciales que en ella se muestran, es decir, con los MPS de todos los CX igual a 0 y con los mismos índices que cuando comenzó la codificación.

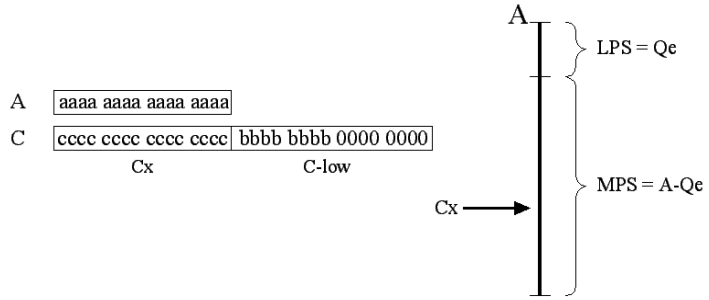


Figura 4.21 Registros A y C usados por el Decodificador MQ

De hecho, la reconstrucción de los planos de bits se realiza volviendo a aplicar el algoritmo EBCOT, es decir, aplicando los procedimientos SPP, MRP y CUP con las banderas de la figura 4.10 a los planos de bits, sólo que está vez se les aplican al mismo

tiempo que se les reconstruye y sin generar datos D, ya que éstos se obtienen con el Decodificador MQ a partir de los CX que genera EBCOT.

Por ejemplo, recordemos que al primer plano de bits únicamente se le aplica el CUP, por lo tanto, se verifica que la primer tira cumpla con las condiciones de que no contenga coeficientes significativos y que ninguno esté en un vecindario preferente, dado que al inicio todas las banderas están en cero, estas condiciones se cumplen y el primer CX es un 17. Ahora que contamos con un CX podemos aplicar la Decodificación MQ para saber si el primer D fue MPS o LPS y verificar en la tabla 4.8 si corresponde a un 0 o a un 1. Si el Decodificador MQ nos entrega un D = 0, automáticamente sabemos que los cuatro bits de la tira son cero y procedemos a obtener los bits de la siguiente tira, en cambio, si nos entrega un D = 1, sabemos que los dos siguientes CX's son 18, obtenemos otros dos D's con la Decodificación MQ, obtenemos los primeros bits de la tira utilizando la tabla 4.6 y, dado que hay un 1 en la tira que vuelve significativo al coeficiente al que pertenece, lo marcamos poniendo en uno la bandera correspondiente en la matriz σ. El siguiente paso en el CUP es codificar el signo del coeficiente que acaba de volverse significativo, entonces a través de las ecuaciones (4.28) y (4.29) y la tabla 4.4 obtenemos un nuevo CX, obtenemos un nuevo D con la Decodificación MQ y con ayuda de la tabla 4.5 recuperamos el signo del coeficiente que se volvió significativo. De forma similar se aplican el SSP, el MRP y el CUP a los demás planos de bits mientras se reconstruyen.

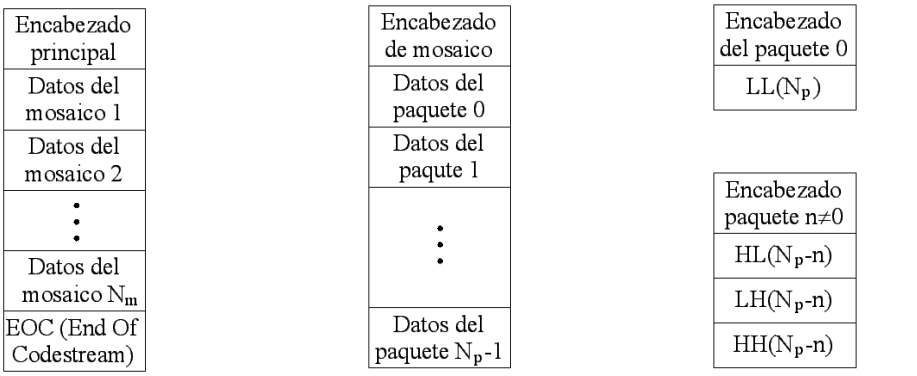
Volviendo al Decodificador MQ, el intervalo A cumple las mismas funciones que durante la codificación, por lo tanto, cada vez que se decodifica un D debe tomar el valor del LPS_L (A = Qe) o del MPS_L (A = A-Qe), según sea el caso, puede sufrir un intercambio condicional y A y C se deben renormalizar cada vez que A tome un valor menor a 0x8000. Las columnas NMPS, NLPS y SWITCH de la tabla B.1 también tienen las mismas funciones que durante la codificación.

Durante las renormalizaciones, el contador CT se decrementa con cada desplazamiento de los bits de los registros A y C y cuando CT es igual a 0, el decodificador toma un nuevo byte del buffer B, lo coloca en los bits b del registro C-low y reinicializa CT en 8. En caso de que ya se hayan utilizado todos los bytes del buffer B y el procedimiento pida cargar otro byte en los bits b, el decodificador simplemente coloca ceros. El valor inicial de CT se establece en 0 con el propósito de cargar un nuevo byte en el registro C la primera vez que ocurra una renormalización.

Si el decodificador encuentra un FF en el buffer B y el siguiente byte es diferente de 00, ha encontrado una bandera y debe realizar la acción que ésta le indique, en caso contrario, la próxima vez que pase un byte al registro C ignorará el 00 que se colocó después del FF y tomará el siguiente byte.

4.6 Organizador del Flujo de Bits

El estándar JPEG2000 define una sintaxis y un conjunto de reglas para organizar los datos de la imagen comprimida (flujo de bits) para que cualquier sistema compatible con el estándar pueda descomprimirla [17]. En la figura 4.22 se muestra un esquema de la forma en que se organizan los datos que se obtienen de las etapas que fueron descritas en las secciones anteriores.



a) Esquema general *b) Datos de mosaico* *c) Paquete 0 y paquete n≠0*

Figura 4.22 Organización del flujo de bits en JPEG2000

Los encabezados se componen de *banderas* y de *segmentos de bandera* y se utilizan para indicar las características de la imagen, tales como su tamaño, el tamaño de los mosaicos, etc. y para definir los parámetros usados durante la compresión, tales como el tamaño de los bloques de coeficientes, los tamaños de paso usados en la Cuantización, el número de niveles de descomposición, etc.

Un segmento de bandera se compone de una bandera y de los parámetros asociados a esa bandera. En la figura 4.23 se muestra un esquema de un segmento de bandera. La bandera se compone de dos bytes, el primero siempre es FF y el segundo puede tomar cualquier valor entre 01 y FE, después de la bandera se colocan dos bytes que se representan con LMAR y que indican la longitud en bytes del segmento de bandera sin tomar en cuenta los dos bytes que componen la bandera. Los parámetros asociados a la bandera pueden tener una longitud de 1, 2 ó 4 bytes o un número variable de bits.

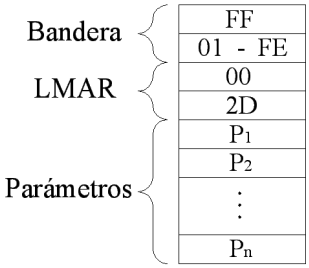


Figura 4.23 Segmento de bandera

Además de las banderas que dan inicio a un segmento de bandera se utilizan otras que sirven para delimitar los encabezados y los datos, como la bandera EOC (End Of Codestream) que se utiliza para marcar el final del flujo de bits, como se muestra en la figura 4.22a.

En la tabla C.1 que se encuentra en el anexo C se muestran los códigos y nombres de las banderas que utiliza JPEG2000, la tabla C.2 indica qué banderas se utilizan en el encabezado principal y de ellas cuáles son obligatorias y cuáles opcionales, y la tabla C.3 indica cuáles se utilizan en el encabezado de cada mosaico.

El número de paquetes es igual al número de niveles de descomposición que se usaron en la TWD más uno, el paquete 0 contiene los datos de la sub-banda LL del último nivel de

descomposición y los demás paquetes contienen los datos de las sub-bandas HL, LH y HH de todos los niveles, por ejemplo, con dos niveles de descomposición, se utilizan tres paquetes, el paquete 0 contiene los datos de la sub-banda LL2, el paquete 1 contiene los datos de las sub-bandas HL2, LH2 y HH2 y el paquete 2 contiene los datos de las sub-bandas HL1, LH1 y HH1. Dentro de un mismo paquete, primero se colocan los datos de las sub-bandas HL, LH y HH de la componente Y y después los de las componentes Cb y Cr.

Los encabezados de paquete se componen de un segmento de bandera, cuya bandera es SOP (Start Of Packet), y de una bandera que separa el encabezado y los datos llamada EPH (End of Packet Header). Los parámetros del segmento de bandera se utilizan para indicar si el paquete tiene una longitud diferente de cero, el número de planos de bits que se codificaron en cada bloque de coeficientes y la longitud en bytes del flujo de bits de cada bloque de coeficientes.

4.7 Resumen

En este capítulo se ha descrito la forma en que se implementan las diferentes etapas que componen un sistema básico de compresión basado en el estándar de compresión de imágenes a color y en escala de grises JPEG2000 tanto en la modalidad de compresión con pérdidas como en la modalidad de compresión sin pérdidas. En este capítulo también se ha descrito la forma en que se revierten los efectos que tuvo la compresión para obtener a partir del flujo de bits una imagen en el modelo de color RGB y se ha mostrado una lista con las once partes que componen al estándar, de las cuales la número 1 comprende el sistema básico de compresión y las otras diez agregan características adicionales.

Capítulo 5
Implementación del Compresor-Descompresor de Imágenes en un DSP

Existen varias razones por las cuales es conveniente digitalizar una señal analógica y procesarla en forma digital, una de las principales es que en el procesamiento digital el mismo hardware puede utilizarse en diferentes aplicaciones simplemente cambiando las instrucciones que tiene programadas [24].

Los DSPs (Digital Signal Processors) son dispositivos cuya finalidad es el procesamiento de señales digitales en tiempo real, se utilizan en aplicaciones como comunicaciones, control y procesamiento de voz, audio y video y se pueden encontrar en dispositivos tales como teléfonos celulares, cámaras digitales, televisiones de alta definición (HDTV), fax, módems, etc [23].

Los DSPs de la familia TMS320C6000 son manufacturados por la compañía Texas Instruments (TI) y están diseñados para ejecutar millones de instrucciones por segundo (MIPS), lo cual los hace apropiados para aplicaciones que requieren de la realización de cálculos numéricos intensivos como telefonía celular 3G, módems DSL (Digital Subscriber Line) y procesamiento de imágenes [24].

En ese capítulo se mencionan las características principales del DSP TMS320C6416 y del módulo de desarrollo DSP Started Kid C6416 (DSKC6416) y se describe la forma en que se implementaron en este DSP un compresor y un descompresor de imágenes a color basados en el estándar JPEG2000 en la modalidad de compresión con pérdidas.

5.1 El módulo de desarrollo DSKC6416

El módulo de desarrollo DSKC6416 es un sistema que contiene las herramientas de hardware y software necesarias para la evaluación y desarrollo de aplicaciones en el DSP TMS320C6416. En la figura 5.1 se muestra un esquema con los principales componentes del DSKC6416, entre los cuales se encuentran el DSP TMS320C6416, dispositivos periféricos y la interfaz con la cual el DSKC6416 se comunica con la PC [25].

5.1.1 El DSP TMS320C6416

El DSP TMS320C6416 es un microcontrolador de propósito específico que se diferencia de los microprocesadores de propósito general por contar con características que lo hacen apropiado para el procesamiento de señales, entre las cuales se encuentran las siguientes [24]:

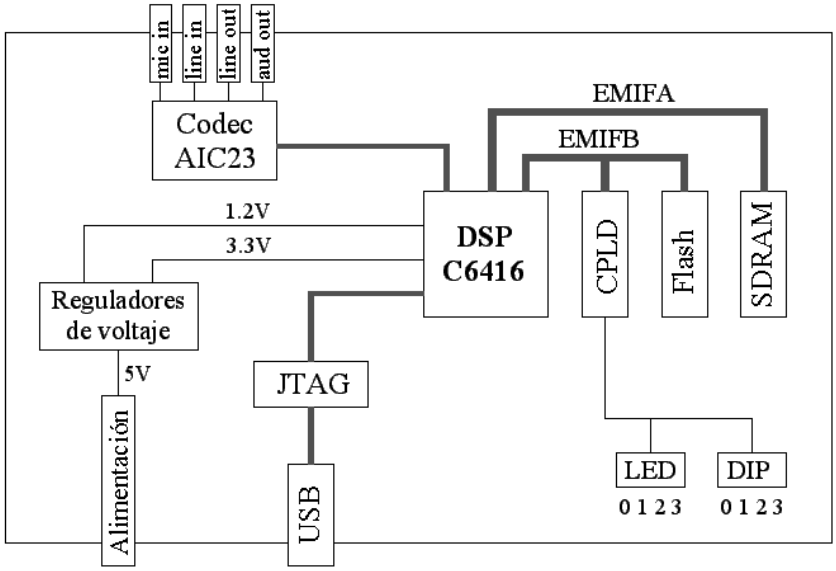


Figura 5.1El módulo de desarrollo DSKC6416

- Está optimizado para manejar ciclos iterativos de operaciones comúnmente presentes en los algoritmos de procesamiento de señales.
- Su conjunto de instrucciones está optimizado para operaciones de procesamiento de señales, por ejemplo, pueden realizar una multiplicación y una acumulación en un solo ciclo de instrucción.
- Cuenta con modos de direccionamiento especializados, como el direccionamiento indirecto y el direccionamiento circular, los cuales son muy eficientes para la implementación de múltiples algoritmos de procesamiento de señales.
- Cuentan con la capacidad de realizar varios accesos a memoria en un mismo ciclo de instrucción.

En la figura 5.2 se muestra un diagrama de bloques de la arquitectura del DSP TMS320C6416.

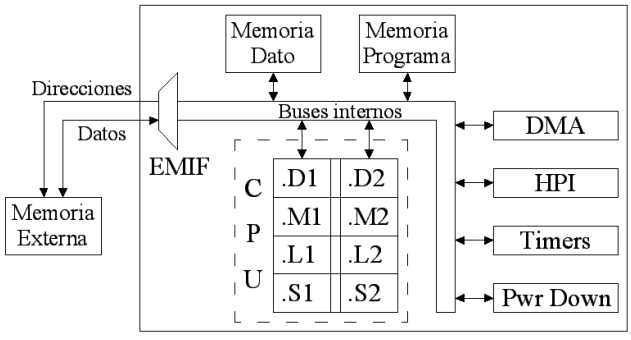


Figura 5.2 Arquitectura del DSP TMS320C6416

El CPU (Central Process Unit) del DSP TMS320C6416 está compuesto por ocho unidades funcionales divididas en dos partes, (A) y (B), cada parte tiene cuatro unidades funcionales, una unidad .M, la cual es utilizada para realizar multiplicaciones, una unidad .L, utilizada para operaciones lógicas y aritméticas, una unidad .S, utilizada para

manipulación de bits y operaciones aritméticas, y una unidad .D, utilizada para carga y almacenamiento de datos y para operaciones aritméticas. Algunas instrucciones como ADD (suma) pueden ser realizadas por más de una unidad.

El bloque DMA (Direct Memory Access) permite mover datos de un lugar de memoria a otro sin la intervención del CPU, el bloque EMIF (External Memory Interface) provee la sincronización necesaria para acceder a memoria externa, el Timer cuenta con contadores de 32 bits, la unidad Power Down (Pwr Down) es usada para ahorrar energía cuando el CPU está inactivo y la interfaz HPI (Host Port Interface) permite el acceso a memoria interna a un procesador o dispositivo huésped [24].

El DSP TMS320C6416 puede trabajar con una frecuencia de reloj 1GHz y realizar hasta 8000 MIPS.

Pipeline

En general, la realización de una instrucción requiere de varios pasos, básicamente estos pasos son *búsqueda* (B), *decodificación* (Dec) y *ejecución* (E). Si estos pasos se realizan en forma secuencial, los recursos con que cuentan los DSPs, tales como varios accesos a memoria en un ciclo de reloj y diferentes unidades funcionales, no se explotan al máximo. Los CPUs de los DSPs de TI están diseñados para realizar los pasos de que consta una instrucción de una forma que se denomina *Pipeline*.

El Pipeline consiste en realizar un paso diferente de varias instrucciones consecutivas al mismo tiempo. En la figura 5.3 se muestra el número de ciclos que les lleva a un CPU secuencial y a un CPU con Pipeline realizar tres instrucciones, en dicha figura podemos observar que a un CPU con Pipeline requiere menos ciclos de reloj para realizar las mismas instrucciones porque mientras ejecuta una instrucción *n*, realiza la decodificación de la instrucción *n+1* y la búsqueda de la instrucción *n+2* [24].

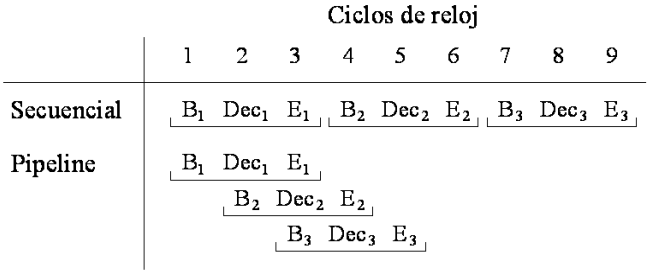


Figura 5.3 Realización de tres instrucciones con y sin Pipeline

5.1.2 Periféricos y comunicación entre el DSKC6416 y la PC

Además del DSP TMS320C6416, el DSKC6416 incluye los siguientes componentes [25]:

- 16 Megabytes de memoria SDRAM (Synchronous Dynamic Random Access Memory).
- 512 kilobytes de memoria flash no volátil.
- Un CPLD (Complex Programable Logic Device) en el cual fueron implementados los registros que se utilizan para configurar el DSKC6416.
- 4 LEDs y 4 DIP switchs que permiten al usuario leer y escribir en los registros del CPLD.

- Un bus de 64 bits llamado EMIFA (External Memory Interface A) a través del cual el DSP se conecta con la SDRAM y un bus de 8 bits llamado EMIFB (External Memory Interface B) con el cual se conecta con el CPLD y la memoria flash.
- Un codificador-decodificador TLV320AIC23 (Codec AIC23) que permite al DSKC6416 recibir y transmitir señales analógicas a través de una entrada para micrófono (mic in), una línea de entrada (line in), una línea de salida (line out) y una salida para audífonos (aud out).
- Reguladores de voltaje de 1.2V y 3.3V para polarizar al DSP.
- Una interfase JTAG (Joint Test Action Group) con la cual el DSKC6416 se comunica con la PC a través de un puerto USB.

El DSKC6416 está diseñado para trabajar con el Ambiente Integral de Desarrollo (IDE) Code Composer Studio (CCS) de TI, el cual incluye las herramientas necesarias para generar el código que se programará en el DSP, tales como un *compilador de lenguaje C*, un *ensamblador* y un *ligador*, y las herramientas para depurar dicho código.

El compilador produce a partir de un archivo con código en lenguaje C, código en lenguaje ensamblador y lo almacena en un archivo con extensión *.asm*, luego el ensamblador toma el archivo asm y genera código máquina que almacena en un archivo con extensión *.obj* [23].

El ligador coloca las secciones de código, constantes y variables en lugares apropiados de la memoria del DSP como se especifica en un *archivo de comandos y descriptor de memoria* con extensión *.cmd* y lo combina con el archivo .obj que generó el ensamblador para generar el archivo con extensión *.out* que se carga en el DSP para su ejecución [24].

Entre las herramientas con que cuenta el CCS para la depuración de código se encuentran la visualización de variables, datos en memoria y registros, uso de breakpoints, graficación de resultados, visualización de imágenes y monitoreo de los procesos en tiempo de ejecución [23].

5.2 Implementación del sistema de compresión

En la figura 5.4 se muestra un esquema general del proceso de compresión de imágenes, desde que se tiene una imagen digital a color guardada en la memoria de la PC hasta que se almacenan en la memoria SDRAM del DSKC6416 los datos de la imagen comprimida.

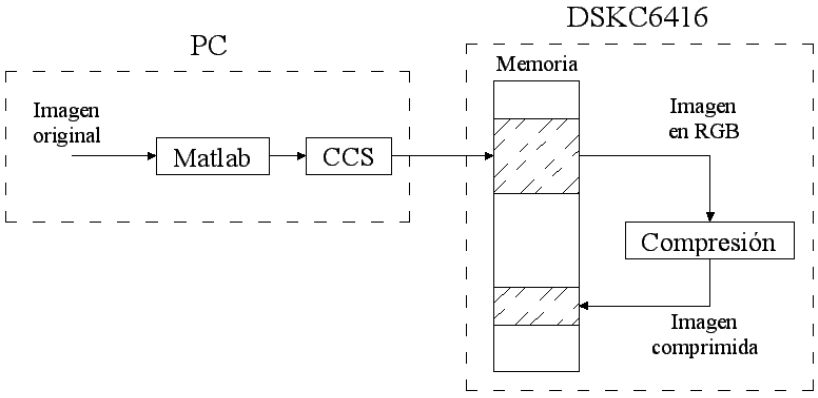


Figura 5.4 Esquema del proceso de compresión

Después de almacenar la imagen original en el modelo de color RGB en la memoria del DSKC6416 a través de Matlab y el CCS, en el DSP se implementó el bloque *compresión*, en el cual se programaron los procedimientos descritos en el capítulo 4 en lenguaje C, en el anexo D se muestra el código fuente y en la figura 5.5 se muestra un diagrama de flujo de la función principal.

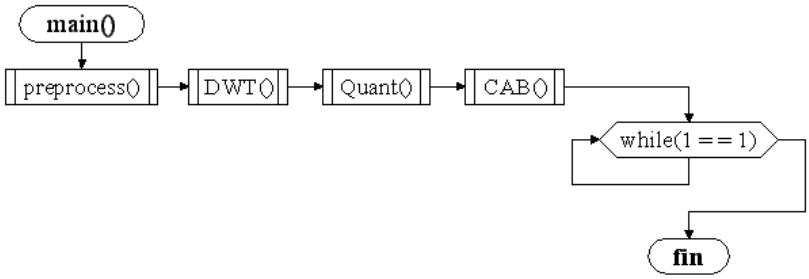


Figura 5.5 Función principal del sistema de compresión

En la función *preprocess()* se realiza el corrimiento de DC y la transformación del espacio de color. En la función *DWT()* se aplicó la Transformada Wavelet Discreta (TWD) a las componentes YCbCr, las cuales fueron almacenadas en los arreglos *Y[][]*, *Cb[][]* y *Cr[][]* en la función *preprocess()*. En la función *Quant()* se implementó la Cuantización y en la función *CAB()* se implementaron el algoritmo EBCOT y la Codificación MQ. El programa principal termina cuando el proceso queda atrapado en el ciclo *while* infinito que se muestra al final del diagrama de flujo de la figura 5.5.

5.2.1 Almacenamiento de la imagen en el DSP y preprocesamiento

Las imágenes utilizadas en este trabajo fueron obtenidas de la página de Internet *http://sipi.usc.edu/database/* en formato TIFF (Tagged Image File Format). Las componentes RGB de las imágenes se obtuvieron a través de la función *imread()* de Matlab y se almacenaron en un archivo llamado *imagen.ent* para cargarlos en la memoria del DSP mediante el CCS. En el archivo *imagen.ent* primero fueron almacenados los datos de la componente R línea por línea comenzando con la parte superior y luego los datos de las componentes G y B. Las imágenes utilizadas son de 512x512 píxeles, por lo tanto, los sistemas compresor y descompresor fueron diseñados para procesar imágenes de estas dimensiones.

Con el propósito de almacenar cada componente RGB en un arreglo de dos dimensiones, se declararon los arreglos *R[][]*, *G[][]* y *B[][]* y se les asignaron las secciones de memoria en que se guardó el archivo *imagen.ent* a través de la directiva *#PRAGMA DATA_SECTION(simb, "mi_seccion")* donde el argumento *simb* indica el nombre de la variable para la cual se ha reservado una sección de memoria y el argumento *mi_seccion* es el nombre de dicha sección, las longitudes y las direcciones de inicio de las secciones de memoria fueron definidas en el archivo de comandos cmd. En la figura 5.6 se muestra un esquema de la forma en fueron almacenadas en memoria las componentes RGB.

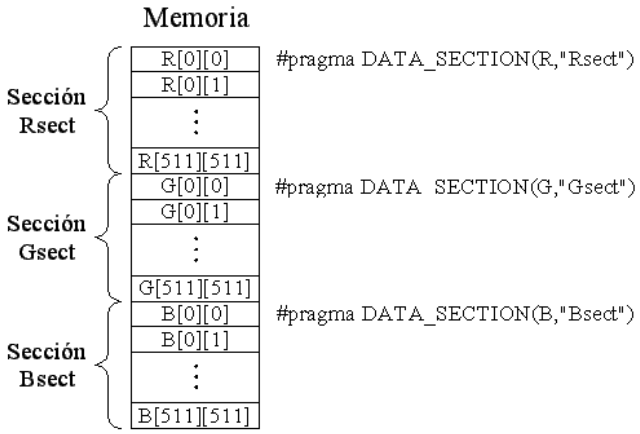


Figura 5.6 Esquema del almacenamiento de las componentes RGB en memoria

Preprocesamiento

En la función *preprocess()* de la figura 5.5 se realiza el corrimiento de DC a cada uno de los elementos de los arreglos *R[][]*, *G[][]* y *B[][]* utilizando la ecuación (4.1) y se realiza la transformación del espacio de color mediante la ecuación (4.2). No se aplicó segmentación. El siguiente segmento de código muestra la forma en que se implementó el corrimiento de DC de la componente R, donde *N* es una constante cuyo valor es igual al número de líneas y columnas que tienen las imágenes, es decir, 512.

```
for(py=0; py<=N-1; py++)
        for(px=0; px<=N-1; px++)
                R[px][py] = R[px][py]-128;
```

El corrimiento de DC de las demás componentes y la aplicación de la ecuación (4.2) se realiza de forma similar.

5.2.2 Transformada Wavelet Discreta

En la función *DWT()* de la figura 5.5 se aplicó la TWD a las componentes YCbCr de la forma en que se explicó en la sección 4.3.2 utilizando cinco niveles de descomposición.

En la figura 5.7 se muestra el diagrama de flujo de la aplicación de la TWD a la componente Y, a las componentes Cb y Cr se les aplicó de la misma forma. En el primer ciclo *for*, la variable *Np* indica el número de elementos que tiene la línea o columna a la que se aplica la TWD en cada nivel de descomposición. Dado que en el primer nivel de descomposición la TWD se aplica a toda la imagen, el valor inicial de *Np* es *N* y dado que en los siguientes niveles sólo se aplica a las sub-bandas LL, *Np* reduce su valor a la mitad cada vez que un nivel de descomposición termina (ver figura 4.6). Como la TWD se aplica con cinco niveles de descomposición, el valor final de Np es 32.

Las líneas del arreglo *Y[][]* se transfieren al arreglo intermedio *Z[]* (tercer ciclo *for*) para aplicarles la TWD en la función *Daub_aZ()*, los coeficientes resultantes se almacenan en el arreglo *coef[]* y en el quinto ciclo *for* son transferidos al arreglo *Y[][]* en el mismo lugar en que se encontraba la línea transformada para que el arreglo *Y[][]*, el cual contenía

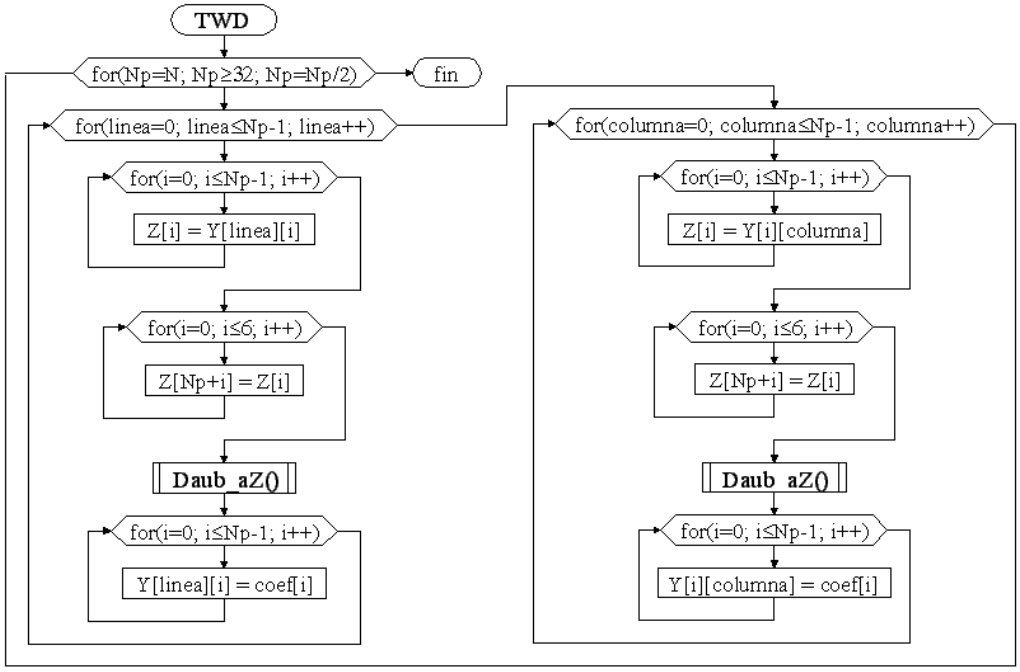


Figura 5.7 Transformada Wavelet Discreta (TWD)

la componente Y, ahora contenga los coeficientes que se obtuvieron al aplicar la TWD a dicha componente.

En la TWD los vectores se consideran periódicos, por lo tanto, para calcular el último coeficiente hace falta copiar los primeros 7 elementos del arreglo *Z[]* después del último elemento de la línea a transformar, lo cual se realiza en el cuarto ciclo *for*.

En la figura 5.8 se muestra el diagrama de flujo de la función *Daub_aZ()*. En esta función se obtuvieron *Np* coeficientes realizando el producto interno entre la línea o columna almacenada en el arreglo *Z[]* y las funciones duales $\tilde{\varphi}_k$ y $\tilde{\psi}_k$ (ver ecuación (4.26)).

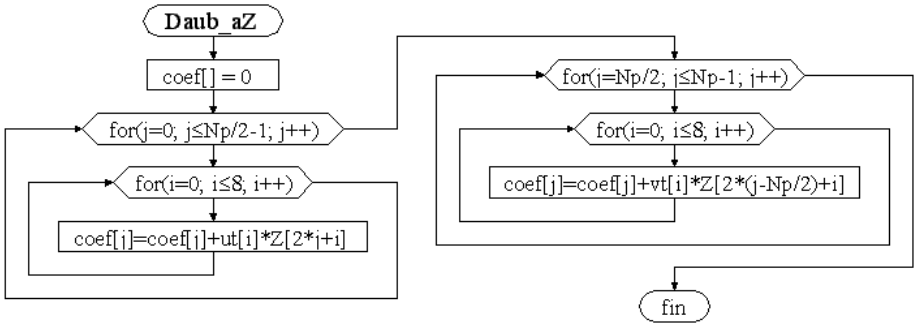


Figura 5.8 Aplicación de la TWD al arreglo intermedio Z[]

Dado que la mayoría de los valores de las funciones duales $\tilde{\varphi}_k$ y $\tilde{\psi}_k$ son cero, en la aplicación de la TWD únicamente se utilizaron los primeros 9 valores de los vectores $\tilde{u}(n)$ y $\tilde{v}(n)$, los cuales se almacenaron en los arreglos *ut[]* y *vt[]* definidos en el archivo Daub9_7.h que se muestra en el anexo F.

Dentro del primer ciclo *for* de la figura 5.8 se calculan los coeficientes de las funciones $\tilde{\varphi}_k$. Debido a que se están ignorando las multiplicaciones por cero, *coef[0]* se calcula con el producto interno entre el arreglo *ut[]* y los primeros 9 elementos del arreglo *Z[]* como se puede observar en la primera línea de la figura 5.9 y, debido a que los vectores $\tilde{\varphi}_k$ se forman desplazando los elementos del vector $\tilde{u}$ de dos en dos, *coef[1]* se calcula con el producto interno entre el arreglo *ut[]* y los nueve elementos del arreglo *Z[]* que comienzan en el índice 2 (segunda línea de la figura 5.9), *coef[2]* se calcula con los elementos que comienzan en el índice 4, y así sucesivamente, esto se consigue con el sumando *2*j* que se encuentra en el índice del arreglo *Z[]* de la figura 5.8.

$$\langle z, \tilde{\varphi}_0 \rangle: \quad coef[0] = ut[0]*z[0] + ut[1]*z[1] + \ldots + ut[8]*z[8]$$
$$\langle z, \tilde{\varphi}_1 \rangle: \quad coef[1] = ut[0]*z[2] + ut[1]*z[3] + \ldots + ut[8]*z[10]$$
$$\langle z, \tilde{\varphi}_2 \rangle: \quad coef[2] = ut[0]*z[4] + ut[1]*z[5] + \ldots + ut[8]*z[12]$$
$$\langle z, \tilde{\varphi}_j \rangle: \quad coef[j] = ut[0]*z[2*j+0] + ut[1]*z[2*j+1] + \ldots + ut[8]*z[2*j+8]$$

Figura 5.9 Producto interno entre el vector z y las funciones $\tilde{\varphi}_k$

Los coeficientes de las funciones $\tilde{\psi}_k$ se calculan dentro del tercer ciclo *for* de forma similar y se almacenan en el arreglo *coef[]* del índice *Np/2* al índice *Np-1* como se puede observar del lado derecho de la figura 5.8.

En la tabla 5.1 se muestran las sub-bandas en que quedaron divididas las componentes YCbCr después de la aplicación de la TWD y sus respectivas dimensiones.

Sub-banda	Dimensiones
LL5	16x16 coeficientes
HL5, LH5, HH5	16x16 coeficientes
HL4, LH4, HH4	32x32 coeficientes
HL3, LH3, HH3	64x64 coeficientes
HL2, LH2, HH2	128x128 coeficientes
HL1, LH1, HH1	256x 256 coeficientes

Tabla 5.1 Sub-bandas generadas con la TWD y sus dimensiones

5.2.3 Cuantización

En la función *Quant()* de la figura 5.5 se aplicó la Cuantización a los coeficientes generados en la función *DWT()*. En el estándar JPEG2000 se puede usar un tamaño de paso diferente para cuantizar cada una de las sub-bandas generadas con la TWD. Debido a que las sub-bandas que cooperan más con la calidad visual de la imagen son las de los últimos niveles de descomposición, se decidió usar el mismo tamaño de paso para las sub-bandas HL, LH y HH dentro de un mismo nivel y usar diferentes tamaños de paso para cada nivel y para la sub-banda LL5 con tamaños de paso mayores en los primeros niveles de descomposición.

Los seis tamaños de paso de la componente Y se almacenaron en el arreglo *YDb[]*, comenzando con el de la sub-banda LL5 y terminando con el de las sub-bandas HL1, LH1 y HH1 como se puede observar en la tabla 5.2.

Tamaño de paso	Sub-banda
YDb[0]	LL5
YDb[1]	HL5, LH5, HH5
YDb[2]	HL4, LH4, HH4
YDb[3]	HL3, LH3, HH3
YDb[4]	HL2, LH2, HH2
YDb[5]	HL1, LH1, HH1

Tabla 5.2 Tamaños de paso de las distintas sub-bandas

La figura 5.10 muestra el diagrama de flujo del proceso de Cuantización de la componente Y. La variable *Nb* indica el número de líneas y columnas que tiene cada sub-banda y *pDb* indica el índice del elemento del arreglo *YDb[]* que se esté utilizando. El proceso comienza con la Cuantización de la sub-banda LL5 y continua con la Cuantización de las sub-bandas de los demás niveles de descomposición, por lo tanto, el valor inicial de *Nb* es 16 (ver tabla 5.1) y el valor inicial de *pDb* es 0 (ver tabla 5.2).

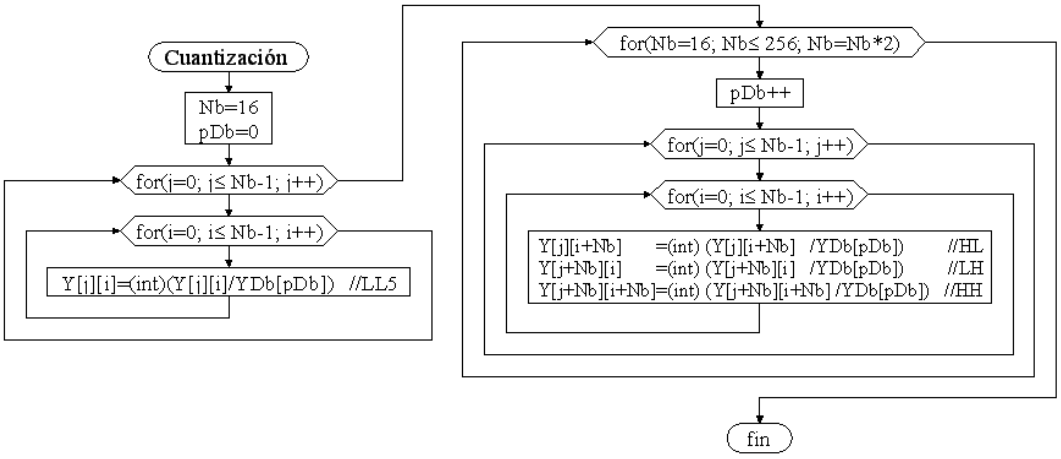


Figura 5.10 Cuantización de la componente Y

Dentro del primer ciclo *for* se cuantiza la sub-banda LL5. La aplicación de la ecuación (4.27) se realiza dividiendo los coeficientes entre el tamaño de paso y convirtiendo el resultado al tipo de variable *entero* (int) a través del operador *cast*, ya que dicha conversión tiene el efecto de eliminar la parte decimal y conservar el signo.

Dentro del tercer ciclo *for* de la figura 5.10 se cuantizan las sub-bandas HL, LH y HH. En este caso, además de indicar las dimensiones de las sub-bandas, la variable *Nb* se utiliza como *offset*, ya que las columnas de las sub-bandas HL comienzan en el índice *Nb* (*Y[0][Nb]*), las líneas de las sub-bandas LH comienzan en el índice Nb (*Y[Nb][0]*) y las líneas y columnas de las sub-bandas HH comienzan en los índices *Nb* (*Y[Nb][Nb]*) como se puede observar en la figura 5.11.

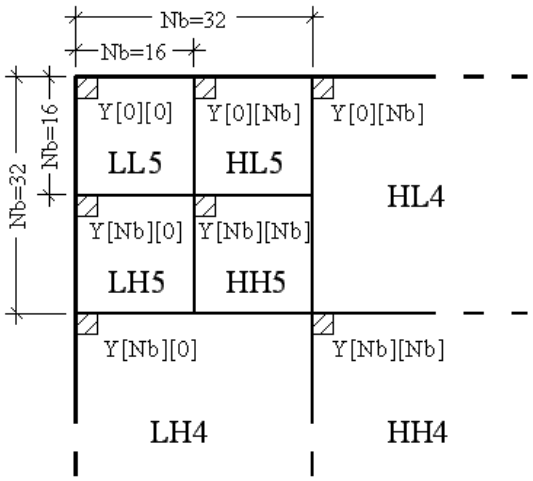
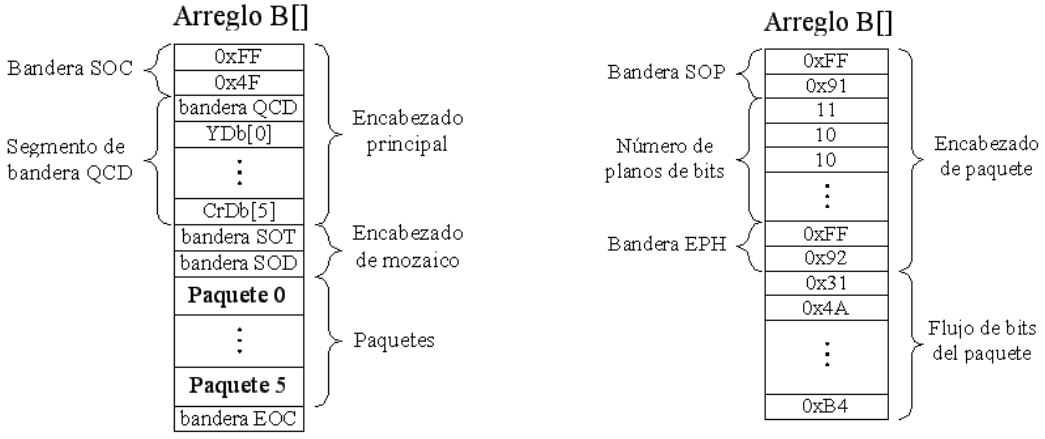


Figura 5.11 Nb como offset de los índices del arreglo Y[][]

La Cuantización de los componentes Cb y Cr se realizó de la misma forma, sus tamaños de paso se almacenaron en los arreglos *CbDb[]* y *CrDb[]*, respectivamente.

5.2.4 Algoritmo EBCOT y Codificación MQ

Después de la Cuantización, se ejecuta la función *CAB()*, en la cual se escribe en el arreglo *B[]* que se muestra en la figura 5.12 el flujo de bits resultante de la compresión junto con las banderas correspondientes, como se explicó en la sección 4.6.



a) Encabezados del flujo de bits *b) Paquete n*

Figura 5.12 Esquema del flujo de bits almacenado en el arreglo B[][]

La forma en que se escriben en el arreglo *B[]* las banderas que componen los encabezados es el siguiente:

```
BP++;    B[BP]=0xFF;
BP++;    B[BP]=0x4F;        // Código de la bandera (ver tabla C.1)
```

donde la variable BP se utiliza como índice del arreglo *B[]* y siempre se incrementa antes de escribir un nuevo dato. En el segmento de bandera QCD que se muestra en la figura 5.12a se escriben los tamaños de paso utilizados en la Cuantización para que el descompresor los reconozca al leer el archivo en que se almacenará el arreglo *B[]*.

Como se explicó en la sección 4.5.1, cada sub-banda se divide en bloques de coeficientes y cada uno se procesa utilizando el método de compresión sin pérdidas Codificación Aritmética Binaria (CAB). Como puede observarse en la figura 5.12a, el flujo de bits que resulta de la aplicación de la CAB es almacenado en seis paquetes. La tabla 5.3 muestra la forma en que está distribuido el flujo de bits de las diferentes sub-bandas en los 6 paquetes, también muestra el número de bloques de coeficientes en que se dividieron las sub-bandas de cada componente YCbCr y sus dimensiones.

Paquete	Sub-banda	Número de bloques de coeficientes	Dimensiones de los bloques de coeficientes
0	LL5	1	16x16
1	HL5, LH5, HH5	1	16x16
2	HL4, LH4, HH4	1	32x32
3	HL3, LH3, HH3	4	32x32
4	HL2, LH2, HH2	16	32x32
5	HL1, LH1, HH1	64	32x32

Tabla 5.3 Distribución del flujo de bits en los paquetes

Como se puede observar en la figura 5.12b, en los encabezados de paquete se escribe el número de planos de bits que contiene cada bloque de coeficientes del paquete, para lo cual la función *CAB()* transfiere cada bloque de coeficientes al arreglo intermedio *Blk[][]* y llama a la función *Nbitps()*.

A continuación se muestra el código fuente de la función *Nbitps()*, en la cual se obtuvo y se almacenó en la variable *max* el coeficiente que tenía el valor absoluto mayor y se determinó el número mínimo de planos de bits que se necesitaban para representarlo tomando en cuenta que con *n* planos se pueden representar coeficientes con valores de hasta 2^n-1. Esta función, además de escribir el número de planos de bits en el encabezado de paquete, lo escribe en el arreglo *Nbp[]*, el cual se utilizará más adelante en una función llamada *Ebcot()*.

```
void Nbitps(/*argumentos/*)
{       for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++)
                        if(abs(Blk[j][i])>max)      max=abs(Blk[j][i]);

        Nbp[pNbp]=0;
        while(max>a-1)
        {       Nbp[pNbp]++;    a=a*2; }

        BP++;   B[BP]=Nbp[pNbp]; }
```

Después de escribir un encabezado de paquete, la función *CAB()* vuelve a transferir cada bloque de coeficientes al arreglo intermedio *Blk[][]* y llama a la función *Ebcot()*, la

cual aplica el algoritmo EBCOT y la Codificación MQ a cada bloque de coeficientes y escribe el flujo de bits resultante en el lugar correspondiente del arreglo *B[]*.

Algoritmo EBCOT

En la figura 5.13 se muestra el diagrama de flujo de la función *Ebcot()*, en la cual cada bloque de coeficientes se procesa para formar pares CX, D como se explicó en la sección 4.5.1. El proceso comienza poniendo en 0 las banderas σ, σ' y η y la matriz de signos χ, las cuales se almacenaron en los arreglos *sigma[][]*, *sigmap[][]*, *eta[][]* y *X[][]*, respectivamente, luego se asignan los valores iniciales a las variables del Codificador MQ, se generan tantos planos de bits como indique el arreglo *Nbp[]*, se les aplica los procedimientos SPP, MRP y CUP, y se llama a la función *flush()*, en la cual se realiza el procedimiento flush descrito en la sección 4.5.2.

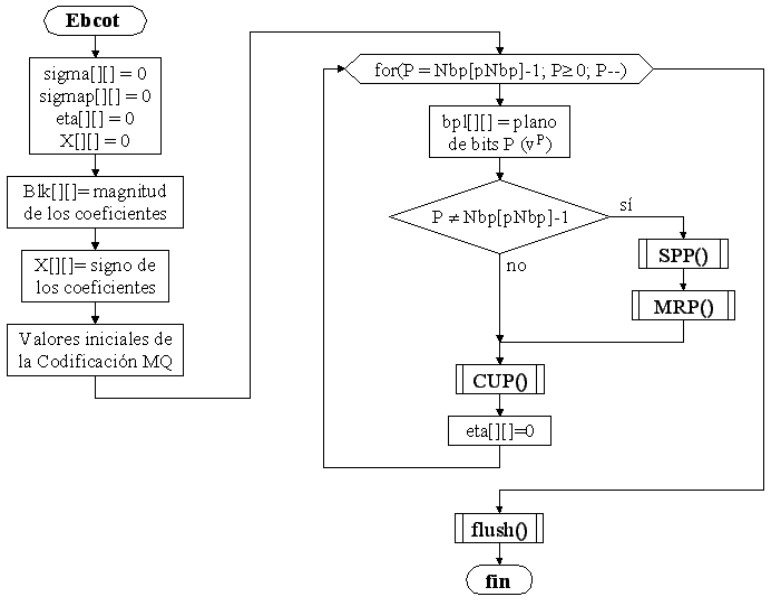


Figura 5.13 Diagrama de flujo de la función Ebcot()

En el siguiente segmento de código se obtiene la representación en magnitud y signo de los bloques de coeficientes multiplicando por menos uno aquellos coeficientes cuyo signo es negativo y poniendo en uno el elemento correspondiente del arreglo *X[][]*. La variable *Nb* se utiliza para indicar las dimensiones del bloque de coeficientes (ver tabla 5.3).

```
for(j=0;j<=Nb-1;j++)
        for(i=0;i<=Nb-1;i++)
                if(Blk[j][i]<0)
                {       X[j][i]=1;          Blk[j][i]=-1*Blk[j][i]; }
```

Como se puede observar en el siguiente segmento de código, el plano de bits v^P se almacenó en el arreglo *bitpl[][]* y se generó realizando la operación binaria AND entre la variable de tipo entero *mascara* y cada elemento del arreglo *Blk[][]*. Si el resultado de la operación AND es mayor que cero, se coloca un 1 en el lugar correspondiente del arreglo *bitpl[][]*, en caso contrario, se coloca un cero. Al generar el plano de bits v^P, *mascara* tiene el valor 2^P debido a que de esta forma el único bit en uno es el que está en la posición *P* y

puede indicar si el bit que se encuentra en la misma posición de un elemento del arreglo *Blk[][]* es 0 ó 1.

```
mascara=1;
if(P!=0)  for(i=1;i<=P;i++)   mascara=mascara*2;
for(j=0;j<=Nb-1;j++)
        for(i=0;i<=Nb-1;i++)
        {       if((v[j][i]&mascara)>0)       bitplane[j][i]=1;
                else bitplane[j][i]=0; }
```

En la figura 5.14 se muestra el diagrama de flujo de la función *MRP()*, en la cual se recorre el arreglo *bitpl[][]* como se muestra en la figura 4.11 en busca de bits que pertenezcan a coeficientes que se hayan vuelto significativos en planos de bits anteriores (*sigma[m][n]=1* y *eta[m][n]=0*) para generar pares CX, D a través de la tabla 4.6 como se explicó en la sección 4.5.1. Las funciones *SPP()* y *CUP()* se desarrollaron de forma similar. Cada vez que se genera un par CX, D en las funciones *SPP()*, *MRP()* y *CUP()*, se le aplica la Codificación MQ en la función *mq_coder()*.

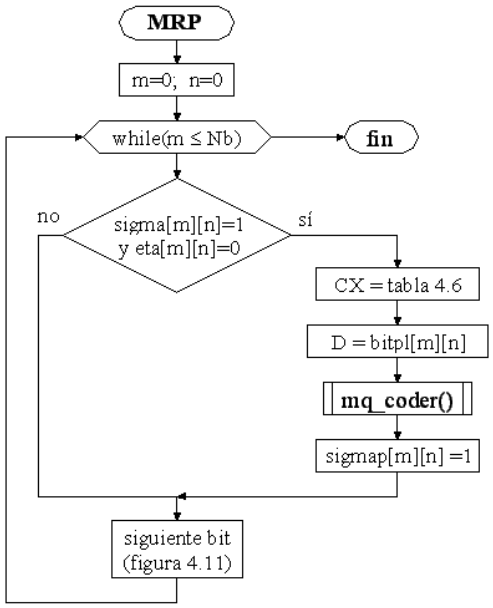


Figura 5.14 Diagrama de flujo de la función MRP()

Codificación MQ

En la Codificación MQ la secuencia completa de datos D generados en el algoritmo EBCOT se representa con el límite inferior del intervalo A, el cual se modifica cada vez que se procesa un dato D como se explicó en la sección 4.5.2. Los datos CX se utilizan para traducir los datos D en Símbolo Más Probable (MPS) o Símbolo Menos Probable (LPS) de acuerdo con la tabla 4.8 y para obtener el índice con el que se accederá a la tabla B.1.

En la figura 5.15 se muestra el diagrama de flujo de la función *mq_coder()*, en la cual se aplica la Codificación MQ a los pares CX, D como se explicó en la sección 4.5.2. Las columnas de la tabla 4.8 se almacenaron en los arreglos *I[]* y *MPS[]*, sus valores iniciales fueron asignados en la función *Ebcot()* y la variable *CX* se utiliza como índice de estos

arreglos. Las columnas de la tabla B.1 se almacenaron en los arreglos *Qe[]*, *NMPS[]*, *NLPS[]* y *SWITCH[]* que fueron definidos en el archivo *tablaB_1.h* que se muestra en el anexo F, la variable *Ii=I[CX]* se utiliza como índice de estos arreglos.

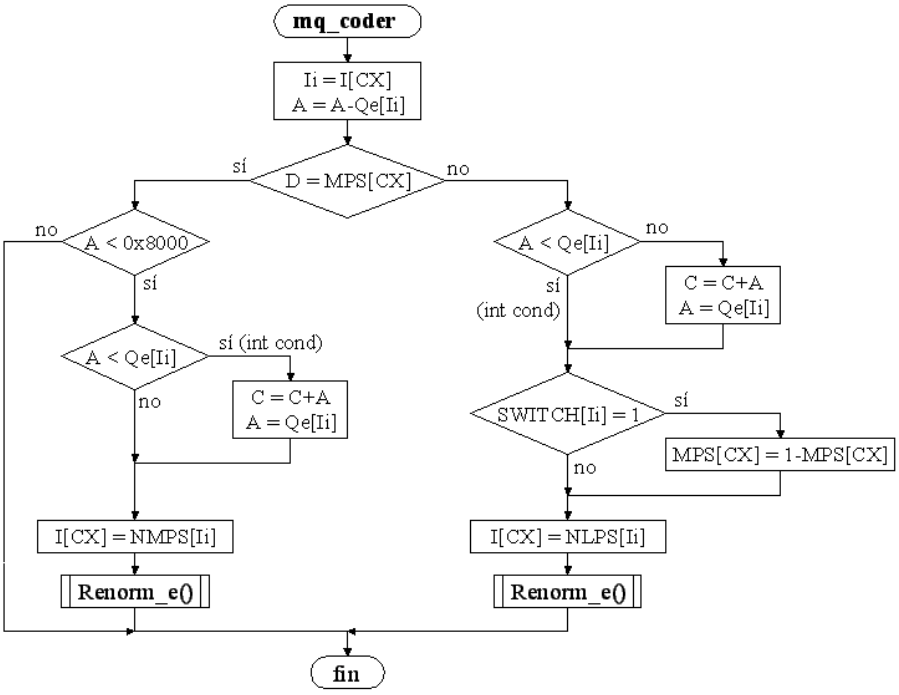


Figura 5.15 Diagrama de flujo de la función mq_coder()

Antes de la pregunta condicional *D = MPS[CX]* de la figura 5.15, a la variable *A* se le asigna el valor del intervalo MPS (*A = A-Qe[Ii]*), si *D* es MPS (rama izquierda) *A* conserva su valor, a menos que ocurra un intercambio condicional, lo cual se verifica con la pregunta condicional *A < Qe[Ii]*, y si *D* es LPS (rama derecha), a *A* se le asigna el valor del intervalo LPS (*Qe[Ii]*) y a *C* se le suma el valor del intervalo MPS como se explicó en la sección 4.5.2, a menos que ocurra un intercambio condicional en cuyo caso *A* y *C* conservan su valor.

Las variables *A* y *C* se renormalizan en la función *Renorm_e()* cuando *D* es MPS y *A* es menor que 0x8000 o cuando *D* es LPS. A continuación se muestra el código de la función *Renorm_e()*, en la cual se desplazan los bits de las variables *A* y *C* a la izquierda y se decrementa el contador *CT* mientras *A* es menor que 0x8000.

```
void Renorm_e(void)
{       while(A<0x8000)
        {       A=A<<1;         C=C<<1;
                CT--;

                if(CT= =0)
                {       Byte_out();
                        CT=8; } } }
```

Cuando el contador *CT* llega a cero, los bits b y posiblemente el bit c de la variable *C* (ver figura 4.16) están ocupados y listos para ser transferidos al arreglo *B[]*, lo cual se realiza en la función *Byte_out()*.

En la variable *C* se almacena el límite inferior del intervalo A, esa es la razón por la cual *C* no se modifica cuando *D* es MPS y se le suma la longitud del intervalo MPS cuando *D* es LPS (ver figura 4.13), sin embargo, el intervalo A se reduce cada vez que la función *mq_coder()* procesa un nuevo dato D y después de procesar un determinado número de datos los bits de la variable *C* dejan de tener la resolución suficiente para representar el límite inferior del intervalo A, lo cual se soluciona renormalizando las variables *A* y *C* y transfiriendo los bits b y c de la variable *C* al arreglo *B[]*, de hecho, cuando se han procesado todos los datos D generados en un bloque de coeficientes, los últimos bits de la variable *C* se transfieren al arreglo *B[]* a través del procedimiento flush como se explicó en la sección 4.5.2, de modo que al final de la función *Ebcot()* en el arreglo *B[]* está representado el límite inferior del intervalo A final, por lo tanto, el flujo de bits resultante de la Codificación MQ y en general de la compresión queda almacenado en el arreglo *B[]*.

En la figura 5.16 se muestra el diagrama de flujo de la función *Byte_out()*, la cual es la encargada de transferir los bits b y c de la variable *C* al arreglo *B[]*.

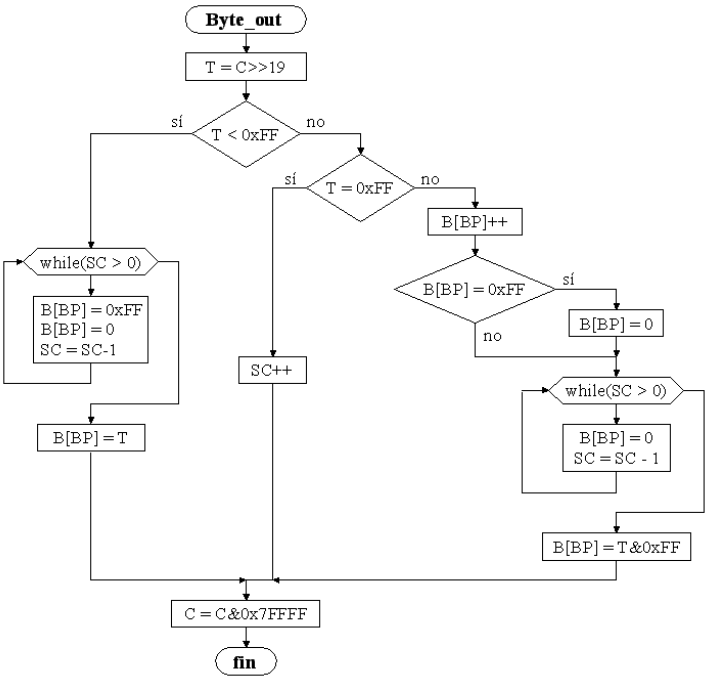


Figura 5.16 Diagrama de flujo de la función Byte_out()

En la función *Byte_out()* los bits b y c de la variable *C* son transferidos a la variable *T* mediante la instrucción *T=C>>19*. Se debe recordar que cuando *T* es igual a 0xFF no se transfiere al arreglo *B[]*, sino que queda pendiente, que el contador *SC* indica cuantos 0xFFs están pendientes y que después de un 0xFF siempre se escribe 0x00 para evitar confundir el siguiente byte con una bandera. La rama izquierda del diagrama de la figura 5.16 se toma cuando *T<0xFF*, si el contador *SC* es mayor que 0, entonces existen tantos 0xFF pendientes como indica SC, por lo tanto, éstos son escritos en el arreglo *B[]* seguidos

de 0's antes de transferir la variable *T* al arreglo *B[]* (ver figura 4.19b). La rama central se toma cuando *T = 0xFF*, en la cual simplemente se incrementa *SC*, ya que la transferencia de *T* al arreglo *B[]* queda pendiente. La rama de la derecha se toma cuando el bit de carry c es uno, es decir, cuando *T > 0xFF*, entonces se suma uno a *B[BP]* (antes de incrementar *BP*), luego se colocan tantos 0's como indique el contador *SC* y se guarda *T* en el arreglo *B[]* pero sin el bit de carry c (ver figura 4.19c). Al final de la función *Byte_out()* se borran los bits b y c de la variable *C*.

Una vez que se procesaron todos los bloques de coeficientes, el arreglo *B[]* queda como se mostró en la figura 5.12.

5.3 Implementación del sistema de descompresión

En la figura 5.17 se muestra un esquema general del proceso con el cual se recupera una imagen en el espacio de color RGB a partir del flujo de bits generado en el proceso de compresión.

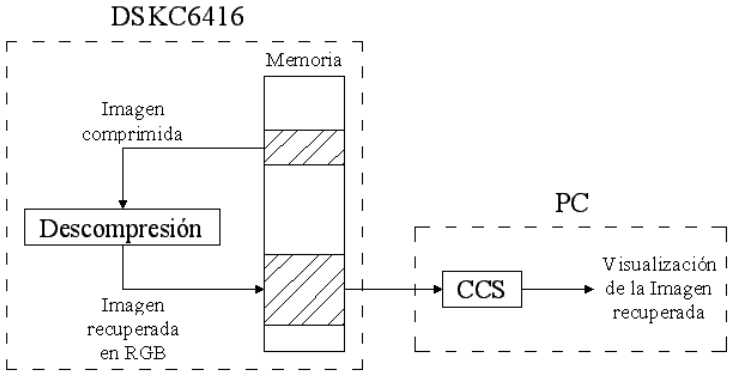


Figura 5.17 Esquema del proceso de descompresión

En el bloque llamado *Descompresión* de la figura 5.17 se programaron los procedimientos descritos en el capítulo 4, el código fuente se encuentra en el anexo E. Una vez que se recupera una imagen en RGB y se almacena en la memoria del DSK6416, ésta se despliega en el monitor de la PC a través del CCS.

En la figura 5.18 se muestra el diagrama de flujo de la función principal del sistema de descompresión implementado en el DSP.

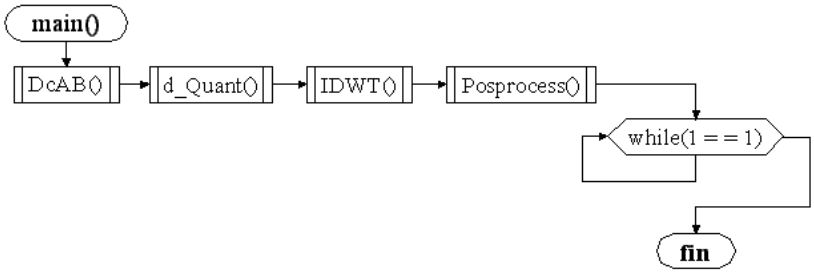


Figura 5.18 Función principal del sistema de descompresión

En la función *DcAB()* se reconstruyen los bloques de coeficientes a partir del flujo de bits generado en el proceso de compresión. En la función *d_Quant()* se realiza la Decuantización de los coeficientes recuperados en la función *DcAB()*. En la función *IDWT()* se aplica la Transformada Wavelet Discreta Inversa (TWDI) a los coeficientes decuantizados para recuperar una imagen en el modelo de color YCbCr y en la función *Posprocess()* se transforma la imagen al modelo de color RGB. El programa principal termina cuando el proceso queda atrapado en el ciclo *while* infinito que se muestra al final del diagrama de flujo.

5.3.1 Decodificación MQ y recuperación de los planos de bits

En las secciones anteriores vimos que el flujo de bits de la imagen comprimida se almacena en el arreglo *B[]*, por lo tanto, la función *DcAB()* de la figura 5.18 lee en este arreglo los datos que necesita para reconstruir los bloques de coeficientes. En la figura 5.19 se muestra un esquema de la forma en que la función *DcAB()* utiliza los datos que lee en el arreglo B[].

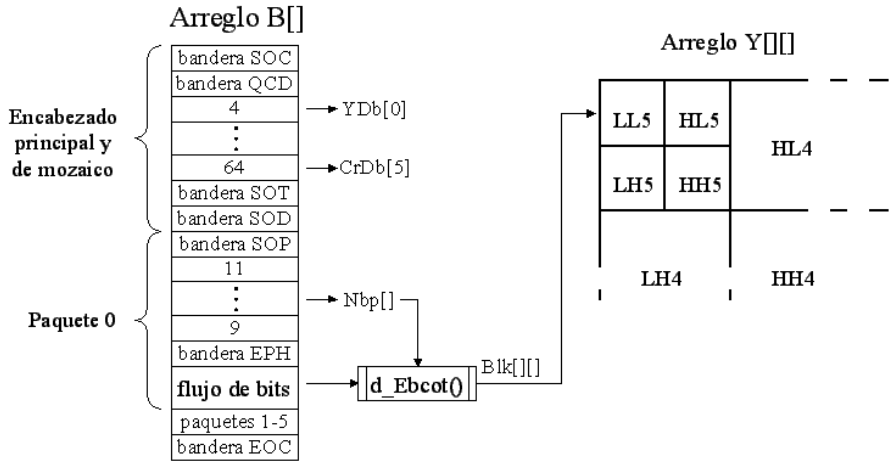


Figura 5.19 Esquema de la extracción de datos del arreglo B[]

Las banderas en el arreglo *B[]* ayudan a la función *DcAB()* a leer los datos en el lugar correcto. En la figura 5.19 se puede observar que cuando la función *DcAB()* encuentra el segmento de bandera QCD almacena los datos que éste contiene debido a que son los tamaños de paso que el descompresor utilizará más adelante en la Decuantización. En siguiente segmento de código se muestra la forma en que los tamaños de paso son leídos, la variable *BP* siempre se incrementa antes de leer un nuevo dato.

```
if(B[BP]==0X5C)                    //QCD (ver tabala C.1)
{       for(pDb=0;pDb<=5;pDb++)
        {       BP++;   YDb[pDb]=B[BP]; }
        for(pDb=0;pDb<=5;pDb++)
        {       BP++;   CbDb[pDb]=B[BP]; }
        for(pDb=0;pDb<=5;pDb++)
        {       BP++;   CrDb[pDb]=B[BP]; } }
```

Cuando la función *DcAB()* encuentra un encabezado de paquete almacena los datos que éste contiene en el arreglo *Nbp[]*, ya que indican el número de planos de bits que componen a los bloques de coeficientes y serán utilizados más adelante en la función *d_Ebcot()*.

Una vez que la función *DcAB()* termina de leer un encabezado de paquete, llama a la función *d_Ebcot()*, la cual lee le flujo de bits contenido en el paquete como se puede observar en la figura 5.19, reconstruye un bloque de coeficientes y lo almacena en el arreglo *Blk[][]*.

En la tabla 5.3 se muestra el número de bloques de coeficientes que contiene cada sub-banda por componente y las sub-bandas a las que corresponden. La función *d_Ebcot()* se llama una vez por cada bloque de coeficientes que se reconstruye, por ejemplo, en la tabla 5.3 se puede observar que en el paquete 0 es llamada tres veces (una por cada componente YCbCr). Cada vez que la función *d_Ebcot()* termina, el bloque de coeficientes reconstruido queda almacenado en el arreglo *Blk[][]* y la función *DcAB()* se encarga de transferir sus datos al lugar correspondiente del arreglo *Y[][]*, *Cb[][]* o *Cr[][]* como se puede observar en la figura 5.19. Cuando la función *DcAB()* termina los arreglos *Y[][]*, *Cb[][]* y *Cr[][]* contienen los coeficientes de la TWD cuantizados.

Algoritmo EBCOT

Tal como se explicó en la sección 4.5.3, para reconstruir un bloque de coeficientes se deben recuperar los planos de bits en el mismo orden en que fueron procesados en la compresión, para lo cual se necesita generar los mismos pares de datos CX, D que fueron generados en la compresión. En la figura 5.20 se muestra el diagrama de flujo de la función *d_Ebcot()*, el cual es muy similar al de la función *Ebcot()* del sistema de compresión debido a que los datos CX se generan de la misma forma en ambas funciones.

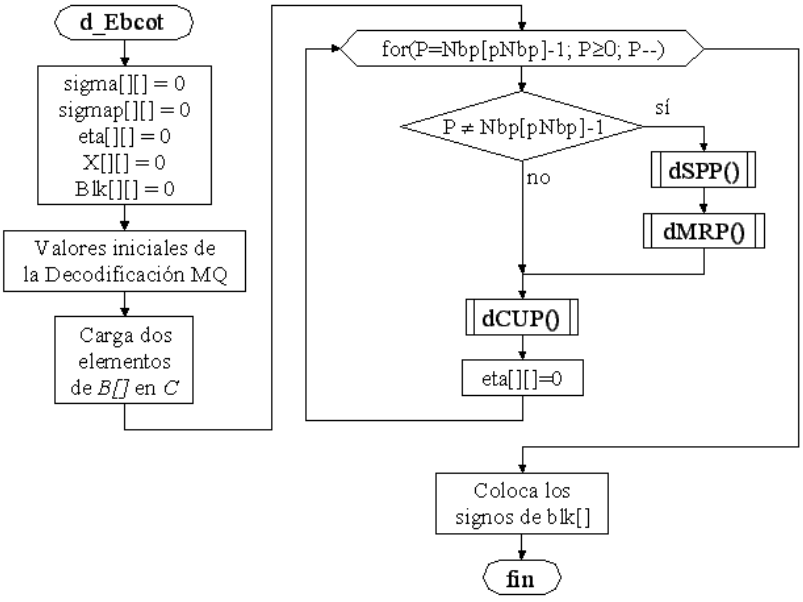


Figura 5.20 Diagrama de flujo de la función dEbcot()

El proceso comienza poniendo en 0 las banderas σ, σ' y η y los elementos de la matriz de signos χ y del arreglo *Blk[][]*, luego se asignan los valores iniciales a las variables de la

Decodificación MQ y se cargan los dos primeros elementos del arreglo *B[]* en los bits c del registro C de la figura 4.21 como se muestra en el siguiente segmento de código:

```
BP++;   C=B[BP];        C=C<<8;
BP++;   C=C+B[BP];      C=C<<16;
```

En el ciclo *for* de la figura 5.20 se reconstruye cada plano de bits llamando a las funciones *dSPP()*, *dMRP()* y *dCUP()* excepto por el primer plano de bits con el cual sólo se llama a la función *dCUP()*. El número de planos de bits que deben reconstruirse se encuentra almacenado en el arreglo *Nbp[]* (ver figura 5.19). Cuando el ciclo *for* termina las magnitudes de los coeficientes se encuentran en el arreglo *Blk[][]* y los signos en el arreglo *X[][]*, para colocar los signos negativos en los elementos del arreglo *Blk[][]* que así lo requieran se ejecuta el siguiente segmento de código:

```
for(j=0;j<=Nb-1;j++)
        for(i=0;i<=Nb-1;i++)
                if(X[j+Xc][i+Xc]==1)        Block[j][i]=Block[j][i]*-1;
```

En la figura 5.21 se muestra el diagrama de flujo de la función *dMRP()*, en la cual los datos CX se obtienen de la misma forma que en la función *MRP()*, es decir, a través de la tabla 4.6. El dato D que corresponde al dato CX se obtiene en la función *mq_decoder()*, la cual a través de Decodificación MQ determina si D es 0 ó 1. En la figura 5.21 se muestra que la forma de poner en 1 un bit en el plano de bits v^P es sumando al elemento *Blk[m][n]* el valor 2^P debido a que este valor sólo afecta a los bits del elemento *Blk[m][n]* que se encuentran en la posición P. Las funciones *dSPP()* y *dCUP()* se desarrollan de forma similar.

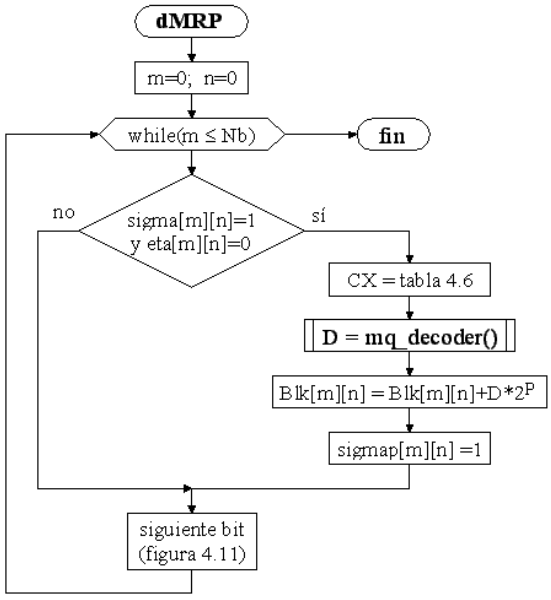


Figura 5.21 Diagrama de flujo de la función dMRP

Decodificación MQ

Tal como se explicó en la sección 4.5.3, en la Decodificación MQ primero se averigua si D es MPS o LPS y después se traduce en 0 ó 1 a través de la tabla 4.8. Como se puede

observar en la figura 4.21, D es MPS cuando C apunta a un lugar dentro del intervalo MPS y es LPS cuando C apunta a un lugar dentro del intervalo LPS.

En la figura 5.22 se muestra el diagrama de flujo de la función *mq_decoder()*, en el cual se puede observar que al principio a la variable *A* se le asigna el valor del intervalo MPS (*A-Qe[Ii]*), que la rama de la izquierda se toma cuando D es MPS y que la variable A sufre los mismos procesos que la variable que representa al intervalo A en la compresión, incluyendo el intercambio condicional y la renormalización, razón por la cual los diagramas de flujo de las funciones *mq_coder()* y *mq_decoder()* son similares.

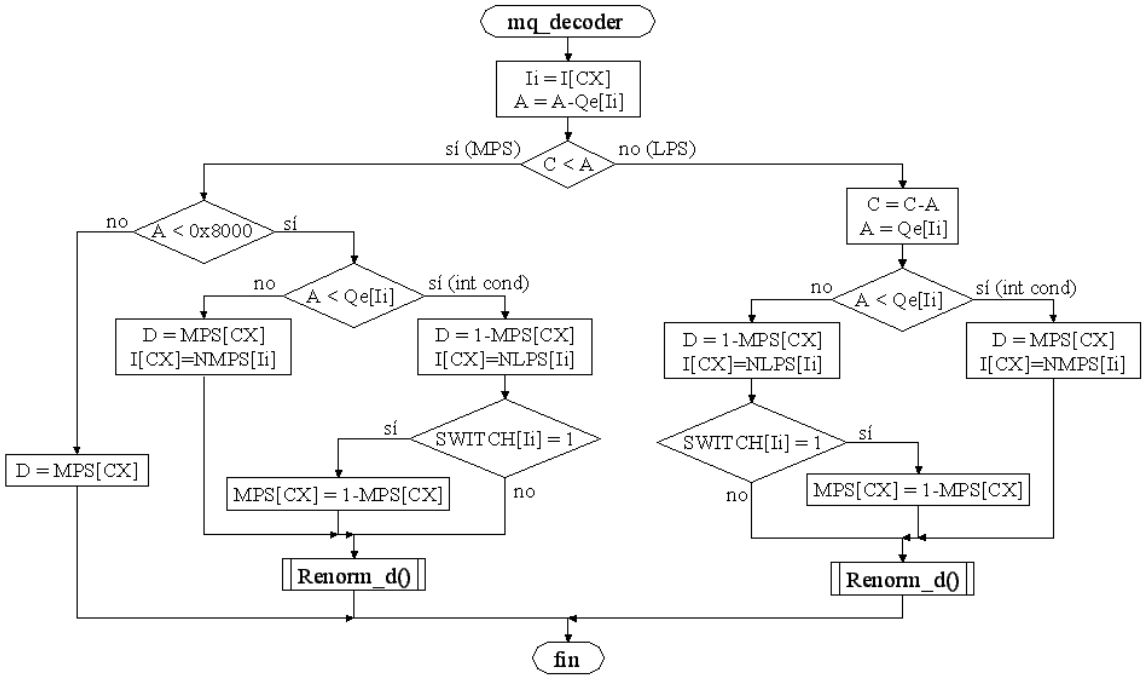


Figura 5.22 Diagrama de flujo de la función mq_decoder()

En la función *mq_decoder()* la tabla 4.8 está representada con los arreglos *MPS[]* e *I[]*. En la figura 5.22 se puede observar que cuando D es MPS a la variable *D* se le asigna un 0 ó 1 a través de la instrucción *D=MPS(CX)* y cuando es LPS se le asigna un 0 ó 1 mediante la instrucción *D=1-MPS(CX)*.

La renormalización de las variables *A* y *C* se realiza en la función *Renorm_d()* como se muestra en el siguiente segmento de código:

```
void Renorm_d(void)
{       while(A<0x80000000)
        {       if(CT==0)
                {       Byte_in();
                        CT=8; }

                A=A<<1;         C=C<<1;
                CT--; } }
```

Cuando el contador *CT* llega a cero los bits b del regirstro C (ver figura 4.21) están vacíos y se llama a la función *Byte_in()*, para transferir un nuevo dato del arreglo *B[]* a la variable *C*.

A continuación se muestra el código de la función *Byte_in()*, en la cual simplemente se transfiere un dato del arreglo *B[]* a los bits b de la variable *C* y en caso de ser encontrado un byte 0xFF, la variable *BP* se incrementa para que en la siguiente llamada a la función *Byte_in()* el siguiente dato no se transfiera a la variable *C*, ya que es un cero.

```
void Byte_in(void)
{       BP++;
        if(B[BP]!=0xFF)
        {       Bprov=B[BP];
                C=C+(Bprov<<8); }
        else
        {       BP++;
                C=C+0xFF00; } }
```

5.3.2 Decuantización

En la función *d_Quant()* de la figura 5.18 se aplica la Decuantización a los coeficientes recuperados en función *DcAB()*. En la figura 5.21 se muestra el diagrama de flujo de la etapa de Decuantización de los coeficientes de la componente Y, el cual es muy similar al diagrama de la Cuantización, sólo que esta vez, en lugar de ser dividido, cada elemento es multiplicado por el tamaño de paso utilizado en la Cuantización. La Decuantización de las componentes Cb y Cr se realiza de la misma forma.

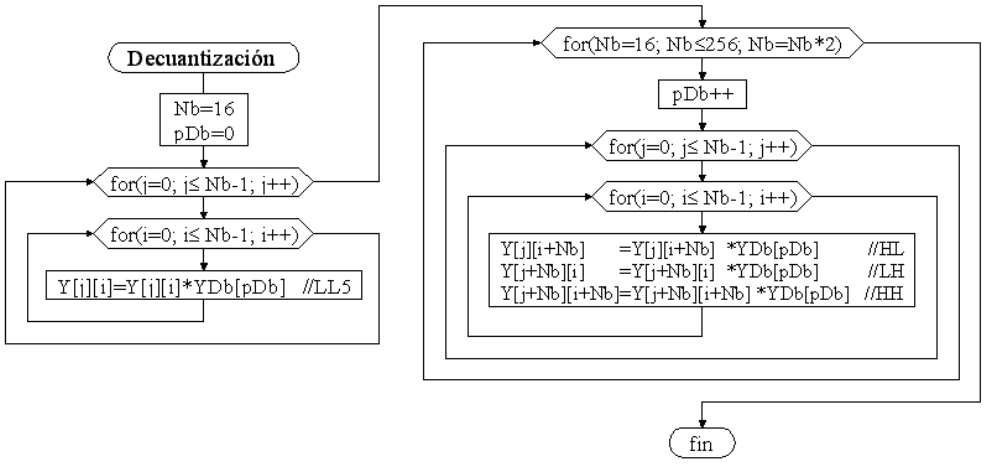


Figura 5.23 Decuantización de los coeficientes de la TWD

5.3.3 Transformada Wavelet Discreta Inversa

En la función *IDWT()* de la figura 5.18 se aplicó la ITWD de la forma en que se explicó en la sección 4.3.3 a los coeficientes almacenados en los arreglos *Y[][]*, *Cb[][]* y *Cr[][]*, con lo cual se recupera una imagen en el modelo de color YCbCr, en la figura 5.24 se muestra el diagrama de flujo de dicha función. La variable *Np* es el número de elementos de la línea o columna a la cual se aplica la TWDI, para aplicar la TWDI se comienza por el último nivel de descomposición, por lo tanto, el valor inicial de *Np* es 32 y su valor se duplica cada vez que se pasa de un nivel de descomposición a otro.

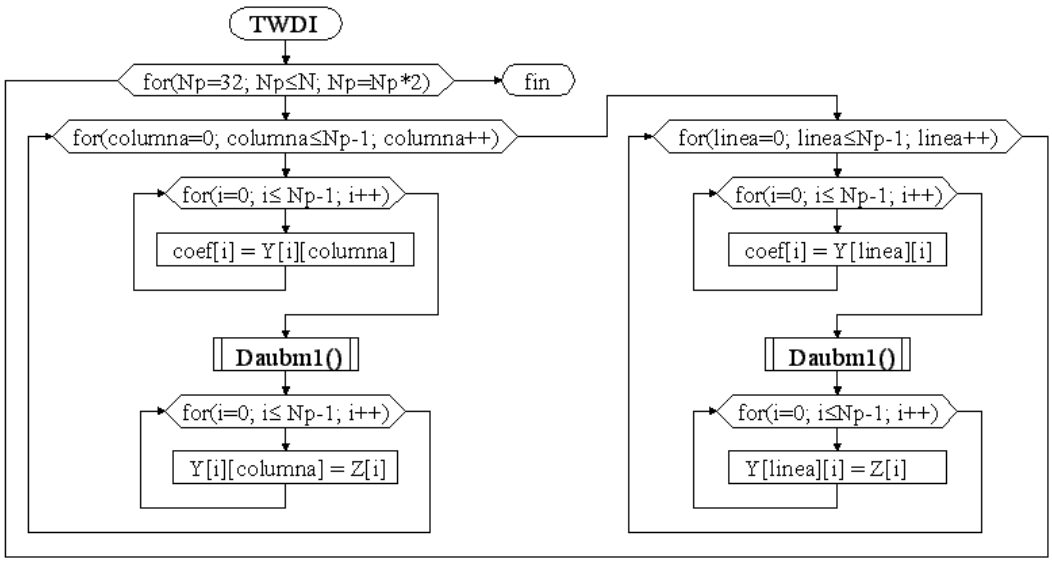


Figura 5.24 TWDI para recuperar la componente Y

La IDWT se aplica primero a las columnas, para lo cual se transfieren una por una a un arreglo intermedio llamado *coef[]* para aplicarle la TWDI en la función *Daubm1()*, los datos que resultan se guardan en el arreglo *Z[]* y éste se transfiere al arreglo *Y[][]* en el lugar que ocupaba la columna a la que se aplicó la TWDI. De forma similar se aplica la TWDI a las líneas.

En la figura 5.25 se muestra el diagrama de flujo de la función *Daubm1()*, en ella se realiza la combinación lineal entre los coeficientes que se encuentran en el arreglo *coef[]* y las funciones base φ_k y ψ_k para recuperar los elementos de la componente Y. Al igual que en la aplicación de la TWD, en la TWDI se ignoran las multiplicaciones por cero, por lo tanto, sólo se utilizan los primeros ocho elementos del vector *u(n)* y los primeros diez elementos del vector *v(n)*, los cuales fueron almacenados en el arreglo *u[]* y *v[]* del archivo Daub9_7.h que se muestra en el anexo F.

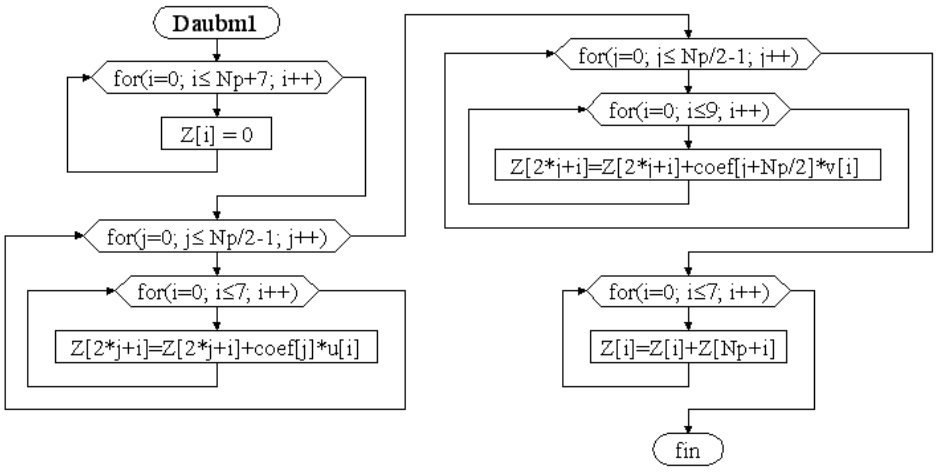


Figura 5.25 TWDI al arreglo intermedio coef[]

Debido a que las funciones base φ_k y ψ_k sólo son diferentes de cero en un número pequeño de elementos, los cuales se desplazan de dos en dos lugares a la derecha (ver ecuaciones (3.41) y (3.42)), cada una de ellas coopera con el resultado de la ecuación (3.44) en lugares específicos del vector *z(n)*, por ejemplo, en la tabla 5.4 se puede observar que la función base φ_0 sólo aporta información al arreglo *Z[]* del elemento 0 al elemento 7, que la función base φ_1 sólo aporta información al arreglo *Z[]* del elemento 2 al elemento 9 y así sucesivamente, lo cual se consigue en el diagrama de la figura 5.25 con el sumando *2*j* que aparece en el índice del arreglo *Z[]*.

$\alpha_0\varphi_0$	$\alpha_1\varphi_1$	$\alpha_j\varphi_j$
$z[0]=z[0]+coef[0]*u[0]$	$z[2]=z[2]+coef[1]*u[0]$	$z[2*j+0]=z[2*j+0]+coef[j]*u[0]$
$z[1]=z[1]+coef[0]*u[1]$	$z[3]=z[3]+coef[1]*u[1]$	$z[2*j+1]=z[2*j+1]+coef[j]*u[1]$
$\vdots$	$\vdots$	$\vdots$
$z[7]=z[7]+coef[0]*u[7]$	$z[9]=z[9]+coef[1]*u[7]$	$z[2*j+7]=z[2*j+7]+coef[j]*u[7]$

Tabla 5.4 recuperación del vector z(n) mediante combinación lineal

5.3.4 Posprocesamiento

En esta etapa se revierten los efectos que tuvo el preprocesamiento en la imagen original, es decir, se transforma la imagen del modelo de color YCbCr al modelo de color RGB a través de la ecuación (4.4) y se suma 128 a cada elemento de las componentes RGB. En el siguiente segmento de código se muestra la forma en que se obtuvo la componente R. La obtención de las componentes G y B y la suma de 128 se realizan de forma similar.

```
for(py=0;py<=N-1;py++)
        for(px=0;px<=N-1;px++)
                R[px][py]=Y[px][py]+1.402*Cr[px][py];
```

Una vez que son ejecutadas todas las funciones de la figura 5.18, se despliega la imagen recuperada en una ventana del CCS y se analizan los resultados obtenidos, los cuales se muestran en el siguiente capítulo.

5.4 Resumen

En este capítulo se han mencionado las características principales del módulo de desarrollo DSKC6416 y las razones por las cuales el DSP TMS320C6416 es apropiado para aplicaciones en las cuales se requiere realizar cálculos numéricos intensivos, como en el procesamiento de imágenes, y se ha descrito mediante diagramas de flujo, esquemas y segmentos de código la forma en que fueron programados en el DSP TMS320C6416 un compresor y un descompresor de imágenes basados en el estándar JPEG2000 desde el momento en que se tenía una imagen digital en formato TIFF en la memoria de la PC hasta que se guardó la imagen reconstruida en el modelo de color RGB en la memoria del DSKC6416.

Capítulo 6
Resultados

En los capítulos anteriores hemos visto que las diferentes etapas que componen un sistema de compresión basado en el estándar JPEG2000 contienen parámetros que pueden influir en la tasa de compresión de las imágenes, como son el número de *niveles de descomposición* en la Transformada Wavelet Discreta (TWD) y las dimensiones de los *bloques de coeficientes* en la Codificación Aritmética Binaria (CAB), sin embargo, los *tamaños de paso* utilizados en la Cuantización son los parámetros que tienen el mayor impacto en la tasa de compresión y en la distorsión de la imagen recuperada, por lo tanto, para lograr el equilibrio entre la calidad deseada de la imagen recuperada y la tasa de compresión, los parámetros que deben modificarse son los tamaños de paso.

En este capítulo se muestran los porcentajes de compresión que se lograron al comprimir con el sistema de compresión-descompresión implementado en este trabajo tres imágenes de 512x512 píxeles con diferentes tamaños de paso, se muestran las imágenes que se recuperaron al descomprimirlas, se evalúa su calidad visual y se comparan los tiempos de ejecución en el DSP TMS320C600 de los sistemas de compresión y descompresión con el tiempo que utiliza Matlab para ejecutar sistemas similares.

6.1 Tasas de compresión y calidad de las imágenes recuperadas

Con el propósito de encontrar los tamaños de paso que generaran la mayor tasa de compresión para las calidades visuales de imagen recuperada *muy buena* y *aceptable*, se realizó la prueba cuyos pasos se muestran en el diagrama de flujo de la figura 6.1 en repetidas ocasiones hasta que se obtuvieron los resultados deseados.

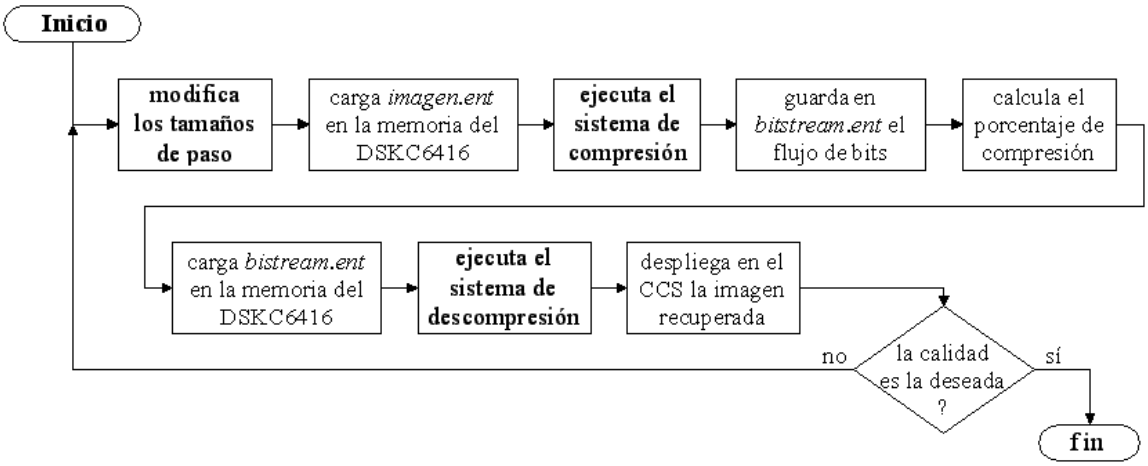


Figura 6.1 Prueba realizada para obtener los porcentajes de compresión

La razón por la cual las pruebas se realizan de esta forma es que diferentes imágenes requieren de diferentes tamaños de paso debido a que las imágenes en las que predominan las regiones sin bordes y los cambios graduales contienen menos información importante para su calidad visual en las sub-bandas HL, LH y HH que las imágenes en las que predominan las regiones con bordes, por lo tanto, para dichas sub-bandas se pueden usar tamaños de paso mayores en las imágenes en las que predominan los cambios graduales.

En el capítulo anterior vimos que el flujo de bits que resulta del proceso de compresión se almacena en el arreglo *B[]*, por lo tanto, para guardar el flujo de bits en el archivo *bitstream.ent* como se menciona en la figura 6.1 el arreglo *B[]* se almacena en dicho archivo. El porcentaje de compresión se calcula dividiendo el número de bytes que componen el flujo de bits contenido en el arreglo *B[]* entre el número de bytes que se requieren para almacenar la imagen original en la memoria del DSKC6416 de acuerdo con la ecuación (6.1).

$$\%\text{Comp} = \frac{\text{long}(\textit{flujo de bits})}{3(512)^2}100\% \qquad \textbf{(6.1)}$$

Tal como se explicó en la sección 5.2.3, en el arreglo *YDb[]* fueron almacenados los tamaños de paso de las sub-bandas de la componente Y de la forma en que se indica en la tabla 6.1. Los tamaños de paso de las componentes Cb y Cr se almacenaron de la misma forma en los arreglos *CbDb[]* y *CrDb[]*, respectivamente.

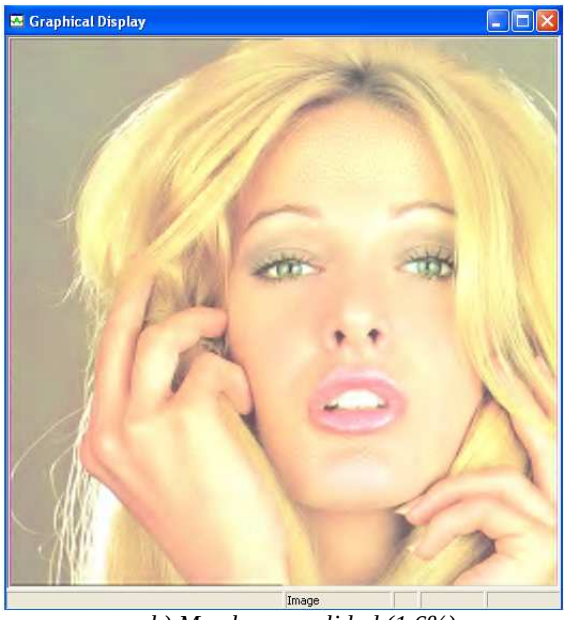
Tamaño de paso	Sub-banda
YDb[0]	LL5
YDb[1]	HL5, LH5, HH5
YDb[2]	HL4, LH4, HH4
YDb[3]	HL3, LH3, HH3
YDb[4]	HL2, LH2, HH2
YDb[5]	HL1, LH1, HH1

Tabla 6.1 Tamaños de paso de las diferentes sub-bandas

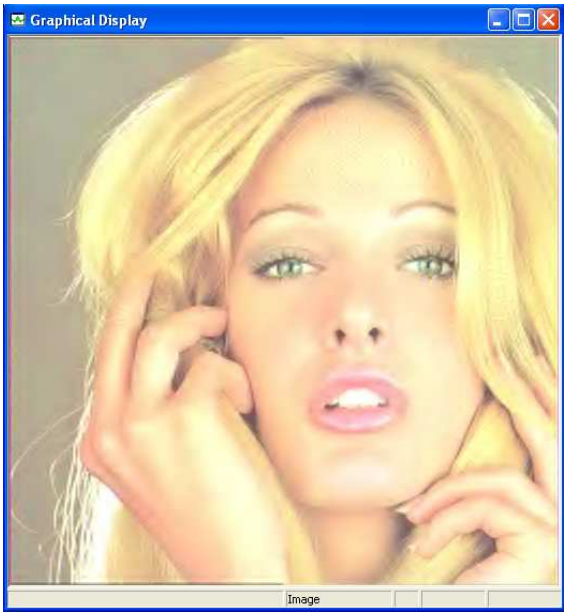
Luego de descomprimir una imagen sin utilizar los datos de las sub-bandas HL1, LH1 y HH1 de la componente de luminancia fue posible observar que la información que aportaban estas sub-bandas no afectaba visiblemente la calidad de la imagen recuperada, lo mismo sucedió con las sub-bandas HL1, LH1, HH1, HL2, LH2 y HH2 de las crominancias, por lo tanto, de ahora en adelante sólo se utilizaran los primeros cinco elementos del arreglo *YDb[]* y los primeros cuatro elementos de los arreglos *CbDb[]* y *CrDb[]*.

En las figuras 6.2b, 6.2c y 6.2d se muestran las imágenes que se recuperaron al descomprimir los flujos de bits que se obtuvieron al comprimir la imagen de la figura 6.2a, la cual tiene el nombre de *Tiffany*, con los tamaños de paso que se muestran en la tabla 6.2. La imagen de la figura 6.2b se considera de muy buena calidad debido a que es casi igual a la original excepto por una ligera pérdida de definición en el cabello, el cual es la parte de la imagen en la que se encuentran los cambios más rápidos. La imagen de la figura 6.2c se considera de calidad aceptable debido a que los dedos de la mano izquierda, la nariz y el borde que separa la palma de la mano derecha del cabello han perdido definición. La imagen de la figura 6.2d se considera de mala calidad debido a que la imagen completa ha perdido definición y ha aparecido una serie de patrones extraños en la frente.

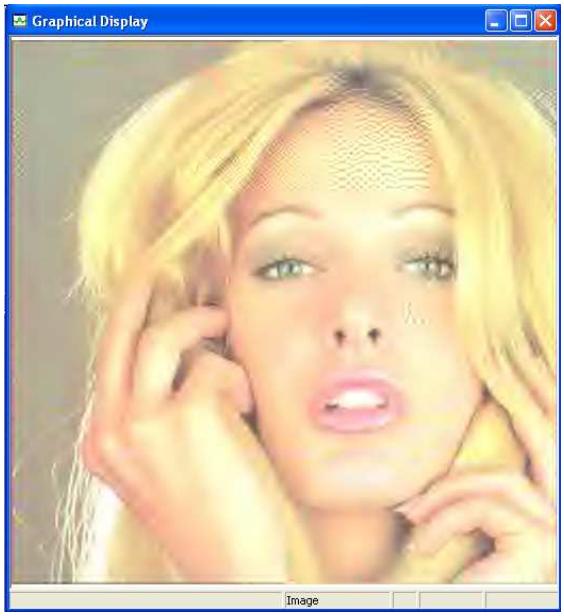
a) Imagen original



b) Muy buena calidad (1.6%)



c) Calidad aceptable (1.06%)



d) Mala calidad (0.53%)

Figura 6.2 Descompresión de Tiffany con diferentes calidades visuales

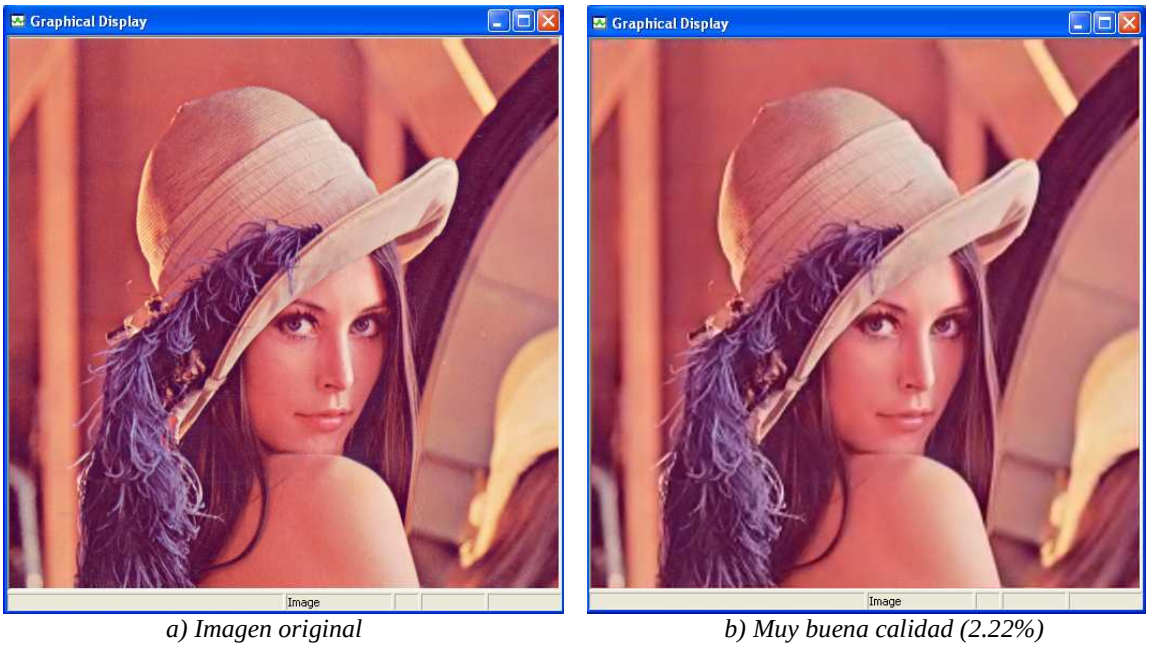
En la tabla 6.2 se muestran los tamaños de paso utilizados en la compresión de la imagen de la figura 6.2a y las tasas de compresión logradas, en ella se puede observar que conforme se incrementan los tamaños de paso, la tasa de compresión aumenta y la calidad disminuye, que se asignaron tamaños de paso mayores a las crominancias que a la luminancia debido a que nuestro sistema visual es menos sensible a los cambios de color que a los cambios de intensidad (sección 2.2.3) y que los tamaños de paso de las sub-bandas de los primeros niveles de descomposición son mayores que los de los últimos niveles debido a que en la mayoría de las imágenes los cambios que predominan son los

cambios graduales (sección 3.2.3) y a que las sub-bandas que contienen la información de estos cambios son las de los últimos niveles de descomposición (sección 4.3.3).

YDb[0] YDb[1] YDb[2] YDb[3] YDb[4] CbDb[0] CbDb[1] CbDb[2] CbDb[3] CrDb[0] CrDb[1] CrDb[2] CrDb[3]	Calidad	Tasa de Compresión
2 4 8 16 16 4 8 16 32 4 8 16 32	Muy buena	1.6%
2 4 12 24 24 16 32 48 64 16 32 48 64	Aceptable	1.06%
16 24 32 48 64 32 64 96 128 32 64 96 128	Mala	0.53%

Tabla 6.2 Tamaños de paso y tasas de compresión de Tiffany

En la figura 6.3a se muestra la imagen llamada *Lenna* y en las figuras 6.3b, 6.3c y 6.3d se muestran las imágenes que se recuperaron al descomprimir los datos que se generaron al comprimir la imagen *Lenna* con los tamaños de paso que se muestran en la tabla 6.3. La imagen de la figura 6.3b se considera de muy buena calidad debido a que es casi igual a la original, excepto por una ligera pérdida de definición en los patrones de rayas del sombrero y en el cabello. La imagen de la figura 6.3c se considera de calidad aceptable debido a que la pérdida de definición se ha extendido al resto de la imagen dándole un aspecto borroso y la imagen de la figura 6.3d es considerada de mala calidad debido a que,



a) Imagen original *b) Muy buena calidad (2.22%)*

Figura 6.3 Descompresión de Lenna con diferentes calidades visuales

c) Calidad aceptable (1.27%)

d) Mala Calidad (0.56%)

Figura 6.3 Descompresión de Lenna con diferentes calidades visuales (Continuación)

además de una mayor pérdida de definición, sobretodo en los ojos y la boca, se ha presentado una serie de patrones extraños en los bordes del hombro y el sombrero.

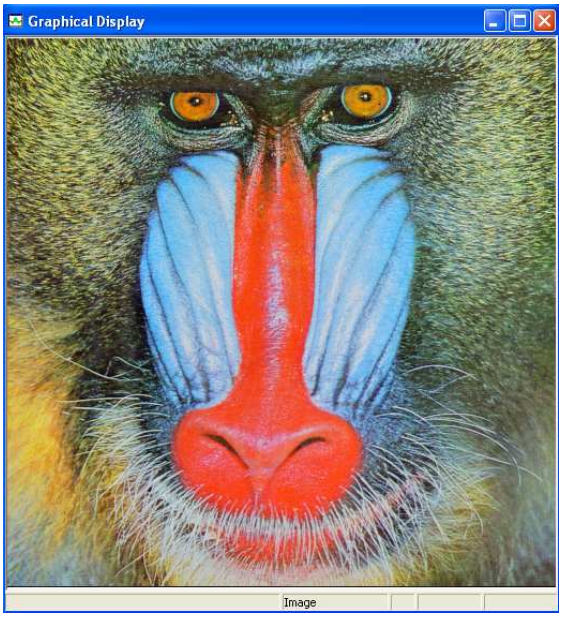
En la tabla 6.3 se muestran los tamaños de paso utilizados en la compresión de la imagen *Lenna* y las tasas de compresión logradas. En la primera columna de la tabla 6.3 se puede observar que los tamaños de paso utilizados son menores que los usados en la compresión de la imagen *Tiffany* y en la tercera columna se puede observar que se logró una menor compresión, esto se debe a que existe una menor cantidad de información en las sub-bandas HL, LH y HH de la imagen *Tiffany* que en las de la imagen *Lenna* debido a que la imagen *Tiffany* posee regiones sin bordes más grandes y sus cambios son más graduales que los de la imagen *Lenna*.

YDb[0] YDb[1] YDb[2] YDb[3] YDb[4] CrDb[0] CrDb[1] CrDb[2] CrDb[3] CrDb[0] CrDb[1] CrDb[2] CrDb[3]	Calidad	Tasa de compresión
2 4 8 8 8 16 32 64 128 16 32 64 128	Muy buena	2.22%
4 8 12 16 24 24 48 96 192 24 48 96 192	Aceptable	1.27%
8 16 32 64 96 48 96 192 250 48 96 192 250	Mala	0.56%

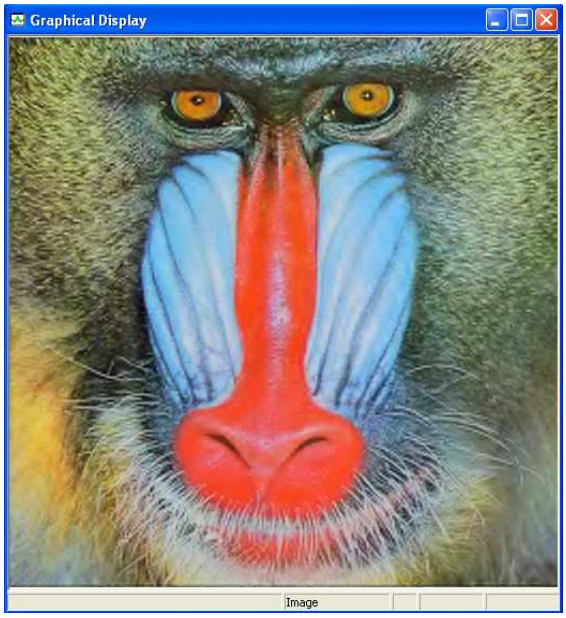
Tabla 6.3 Tamaños de paso y tasas de compresión de Lenna

En la figura 6.4a se muestra la imagen llamada *Baboon* y en las figuras 6.4b, 6.4c y 6.4d se muestran las imágenes que se recuperaron al descomprimir los flujos de bits que se

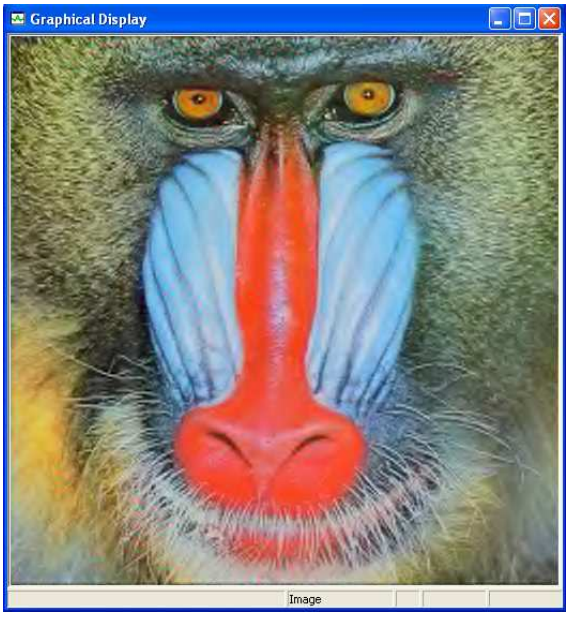
generaron al comprimir la imagen de la figura 6.4a con los tamaños de paso que se muestran en la tabla 6.4. La imagen de la figura 6.4b se considera de muy buena calidad debido a que es casi igual a la original, excepto por una ligera pérdida de definición en el pelo de los costados de la cara, la imagen de la figura 6.4c se considera de calidad aceptable debido a que la falta de definición se ha extendido a la parte central de la cara y el color de los ojos se ha vuelto más pálido y la imagen de la figura 6.4d se considera de mala calidad debido a que ha aparecido una serie de patrones extraños en casi toda la imagen.



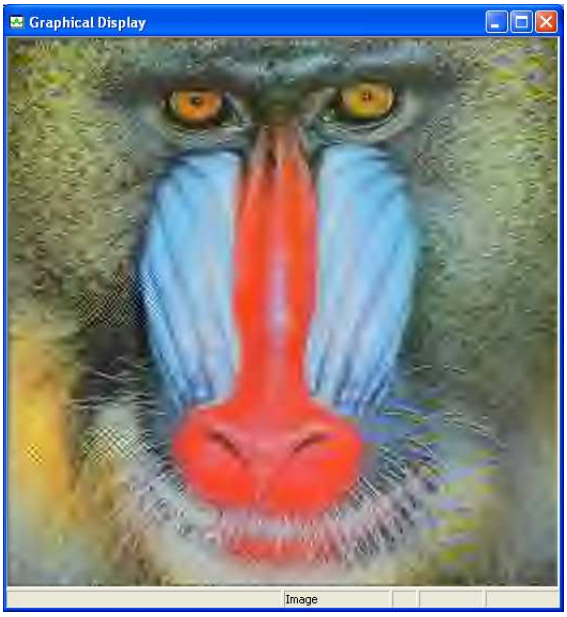
a) Imagen original



b) Muy buena calidad (2.89%)



c) Calidad aceptable (2.1%)



d) Mala calidad (0.57%)

Figura 6.4 Descompresión de Baboon con diferentes calidades visuales

En la tabla 6.4 se muestran los tamaños de paso usados en las imágenes de la figura 6.4 y las tasas de compresión logradas. En la tercera columna se puede observar que no se lograron tasas de compresión tan buenas como en las imágenes *Tiffany* y *Lenna*, esto se debe a que las regiones sin bordes que contiene la imagen *Baboon* son muy pequeñas y contiene muy pocas regiones en las que los cambios sean graduales, lo cual provoca que las sub-badas HL, LH y HH contengan más información y que la Cuantización no pueda llevar a cero a tantos coeficientes como en las imágenes *Tiffany* y *Lenna* sin que haya pérdidas notables en la calidad de la imagen recuperada.

YDb[0] YDb[1] YDb[2] YDb[3] YDb[4] CrDb[0] CrDb[1] CrDb[2] CrDb[3] CrDb[0] CrDb[1] CrDb[2] CrDb[3]	Calidad	Tasa de compresión
2 4 8 20 20 16 32 48 48 16 32 48 48	Muy buena	2.89%
4 6 12 24 32 24 48 96 96 24 48 96 96	Aceptable	2.1%
12 24 48 96 128 64 128 250 250 64 128 250 250	Mala	0.57%

Tabla 6.4 Tamaños de paso y tasas de compresión de Baboon

6.2 Promedio de los Errores Relativos Porcentuales

Hasta ahora se ha evaluado la calidad visual de las imágenes recuperadas de forma subjetiva. Una forma objetiva de evaluar que tan parecidas son las imágenes recuperadas a la original es calculando el *Error Relativo Porcentual* (ERP) de cada elemento de las componentes RGB y obteniendo el promedio. El ERP de un elemento de la componente R se calcula con ecuación (6.2)

$$ERP = \frac{\left|R_{ori}[j][i] - R_{rec}[j][i]\right|}{R_{ori}[j][i]} 100\% \qquad \textbf{(6.2)}$$

donde $R_{ori}[j][i]$ es el elemento en la línea j y la columna i de la componente R original y $R_{rec}[j][i]$ es el elemento en la línea j y la columna i de la componente R recuperada.

El Promedio de los ERPs de todos los elementos de las tres componentes RGB (PERP) se calcula a través de la ecuación (6.3).

$$PERP = \frac{100\%}{3(512)^2}\left[\sum_{j=1}^{512}\sum_{i=1}^{512}\left(\frac{\left|R_{ori}[j][i] - R_{rec}[j][i]\right|}{R_{ori}[j][i]} + \frac{\left|G_{ori}[j][i] - G_{rec}[j][i]\right|}{G_{ori}[j][i]} + \frac{\left|B_{ori}[j][i] - B_{rec}[j][i]\right|}{B_{ori}[j][i]}\right)\right] \qquad \textbf{(6.3)}$$

En la tabla 6.5 se muestran los PERP y las tasas de compresión logradas en la imagen *Tiffany* con respecto a la calidad visual de las imágenes recuperadas. En la tabla 6.5 se puede observar que conforme la tasa de compresión se incrementa el PERP también aumenta, lo cual se esperaba de acuerdo con lo observado en la figura 6.2.

Calidad visual	Tasa de Compresión	PERP
Muy buena	1.6%	2.68%
Aceptable	1.06%	3.02%
Mala	0.53%	3.72%

Tabla 6.5 PERP de la imagen Tiffany

En las tablas 6.6 y 6.7 se muestran los PERP de las imágenes recuperadas de *Lenna* y *Baboon*. En ambas imágenes se puede observar un aumento considerable del PERP con respecto a las imágenes recuperadas de *Tiffany* para las mismas calidades visuales, esto se debe principalmente a que *Lenna* y *Baboon* contienen una mayor cantidad de información en las sub-bandas HL, LH y HH y la Cuantización, la cual afecta en mayor medida a estas sub-bandas, provoca que parte de esta información se pierda, sin embargo, dado que nuestro sistema visual es poco agudo para distinguir cambios rápidos dentro de una imagen, esta pérdida de información no afecta la calidad visual de la imagen recuperada de la forma en que lo haría la pérdida de información en la sub-banda LL.

Calidad visual	Tasa de Compresión	PERP
Muy buena	2.22%	6.18%
Aceptable	1.27%	7.13%
Mala	0.56%	9.91%

Tabla 6.6 PERP de la imagen Lenna

Calidad visual	Tasa de Compresión	PERP
Muy buena	2.89%	15.71%
Aceptable	2.1%	17.01%
Mala	0.57%	22.43%

Tabla 6.7 PERP de la imagen Baboon

En la figura 6.5 se muestra una gráfica en la que se resumen los resultados que se obtuvieron al evaluar los PERP de las imágenes recuperadas de *Tiffany*, *Lenna* y *Baboon* y que fueron presentados en las tablas 6.5, 6.6 y 6.7.

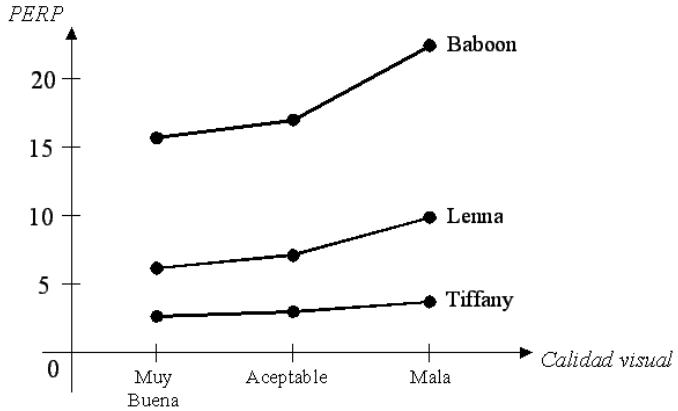


Figura 6.5 PERP de las imágenes Tiffany, Lenna y Baboon

En la gráfica de la figura 6.5 se puede observar que conforme la calidad visual de las imágenes disminuye, los PERP aumentan y que el PERP que se ve afectado en mayor medida con el aumento de la tasa de compresión es el de la imagen *Baboon* debido a que es la que contiene la mayor cantidad de cambios abruptos, como puede apreciarse en la figura 6.4.

6.3 Tiempos de ejecución en el DSP y recursos utilizados

Los DSPs de la familia C6000 están diseñados para aplicaciones que requieren de la realización de cálculos numéricos intensivos como el procesamiento de imágenes. El DSP TMS320C6416 trabaja con una frecuencia de reloj de 1GHz, lo cual significa que un ciclo de reloj se realiza en 1ns. En esta sección se evalúa el tiempo que le lleva al DSP TMS320C6416 la ejecución de los procesos de compresión y descompresión con el propósito de comprobar la capacidad de procesamiento de este DSP, se compara su desempeño con el de un programa que se implementó en Matlab que tiene la misma estructura que el del DSP y que realiza un número similar de operaciones y se muestra la cantidad de memoria del DSP que se utilizó en la implementación de los sistemas compresor y descompresor.

En la tabla 6.8 se muestra el número de ciclos de reloj y el tiempo que le lleva al DSP la ejecución de los procesos de compresión y descompresión de las imágenes *Tiffany*, *Lenna* y *Baboon* con muy buena calidad de la imagen recuperada y el tiempo que le lleva a Matlab comprimir y descomprimir las mismas imágenes con la misma calidad.

		DSP TMS320C6416		Matlab
		Ciclos de reloj	Tiempo	
Compresión	Tiffany	2,747,563,079	2.748s	43s
	Lenna	2,796,523,735	2.797s	56s
	Baboon	2,840,193,102	2.840s	66s
Descompresión	Tiffany	1,825,003,111	1.825s	48s
	Lenna	1,894,141,087	1.894s	60s
	Baboon	1,941,380,881	1.941s	71s

Tabla 6.8 tiempos de ejecución en el DSP y en Matlab

En la tabla 6.8 se puede observar que los tiempos de ejecución en el DSP son mucho menores que los tiempos de ejecución en Matlab, esto se debe a que el DSP opera con una frecuencia de reloj muy alta y a su capacidad de realizar operaciones en paralelo, la cual está basada principalmente en su capacidad de realizar varios accesos a memoria en un solo ciclo de reloj, en el Pipeline y en que su CPU está compuesto por ocho unidades funcionales diferentes.

Aunque los tiempos de compresión en el DSP son pequeños comparados con los de Matlab, para utilizar este sistema en una aplicación de tiempo real, los tiempos de compresión tienen que reducirse de tal forma que se puedan comprimir varias imágenes en un segundo. Una forma de reducir los tiempos de compresión es utilizando imágenes más pequeñas, por ejemplo, una imagen con resolución QCIF (Quarter Common Intermediate Format, 176x144 píxeles) tiene aproximadamente diez veces menos datos que las imágenes de 512x512 píxeles usadas en este trabajo, por lo tanto, comprimir una imagen en formato

QCIF tomaría diez veces menos tiempo. En la tabla 6.9 se muestran los tiempos de compresión aproximados para las imágenes *Tiffany*, *Lenna* y *Baboon* si tuvieran resolución QCIF, con los cuales se podrían comprimir casi cuatro imágenes por segundo.

Imagen	Resolución	Tiempo de compresión aproximado
Tiffany	176x144	0.2748s
Lenna	176x144	0.2797s
Baboon	176x144	0.2840s

Tabla 6.9 Tiempos aproximados de compresión para resolución QCIF

Por último, en la tabla 6.10 se muestra la cantidad de memoria dato y memoria programa que fue utilizada para la implementación de los sistemas compresor y descompresor.

Proceso	Memoria Dato			Memoria Programa
	Imagen RGB	Variables	Flujo de bits	
Compresión	768 kbytes	3073.42 kbytes	16 kbytes	21.9 kbytes
Descompresión	768 kbytes	3073.42 kbytes	16 kbytes	14.5 kbytes

Tabla 6.10 Memoria utilizada por los sistemas de compresión y descompresión

6.4 Resumen

En este capítulo se han presentado los resultados que se obtuvieron al comprimir tres imágenes diferentes con el sistema de compresión basado en el estándar JPEG2000 implementado en el DSP TMS320C6416, se han expuesto las razones por las cuáles los tamaños de paso usados en la Cuantización son los parámetros que se deben variar para lograr el equilibrio entre la tasa de compresión y la calidad visual deseada de la imagen recuperada, se han mostrado las imágenes que se recuperaron con el sistema descompresor con tamaños de paso y calidades visuales diferentes y con el propósito de mostrar el poder de procesamiento que posee el DSP TMS320C6416, se ha evaluado el tiempo que le toma a este DSP la ejecución de los sistemas de compresión y descompresión y se ha comparado con el tiempo que le toma a Matlab ejecutar sistemas similares que realizan casi exactamente el mismo número de operaciones.

Capítulo 7
Conclusiones

En este trabajo se ha logrado implementar un sistema de compresión-descompresión de imágenes a color en la modalidad de compresión con pérdidas, basado en el estándar de compresión JPEG2000, en el módulo de desarrollo DSKC6416, alcanzando tasas de compresión *menores al tres por ciento* en tres imágenes distintas con una muy buena calidad de la imagen recuperada.

En los sistemas de compresión con pérdidas, como el que fue implementado en este trabajo, los procesos que generan la mayor compresión son las transformadas y la Cuantización. Al implementar la Transformada Wavelet Discreta y aplicarla a diferentes imágenes fue posible observar que la mayor parte de la información de la imagen se concentraba en unos pocos coeficientes y que el resto de los coeficientes tenían valores muy pequeños y podían ser llevados a cero mediante Cuantización sin afectar notablemente la calidad de la imagen recuperada; sin embargo, los coeficientes que eran importantes para la calidad visual de la imagen estaban dispersos en tres arreglos distintos de 512x512 elementos. Con la aplicación del método de compresión sin pérdidas, Codificación Aritmética Binaria, fue posible formar un flujo de bits continuo y almacenarlo en un arreglo unidimensional, con lo cual se aprovechó de forma eficiente la memoria en que fueron almacenados los datos finales.

Durante el proceso de descompresión, se pudo observar que la degradación que sufren las imágenes al utilizar porcentajes de compresión muy altos con JPEG2000 corresponde a una menor resolución en las zonas de la imagen en que existen bordes, dándole un aspecto borroso, debido a que las pérdidas se concentran principalmente en la información de los cambios rápidos que existen dentro de la imagen. También fue posible observar que diferentes imágenes requieren de diferentes tamaños de paso para conseguir calidades visuales similares debido a que contienen diferentes cantidades de cambios abruptos y graduales.

Al evaluar el tiempo que le llevó al DSP TMS320C6416 la ejecución del sistema de compresión se pudo comprobar que su poder de procesamiento basado en una frecuencia de reloj muy alta y en la realización de operaciones en paralelo lo hace apropiado para aplicaciones de procesamiento de imágenes; sin embargo, los tiempos de ejecución obtenidos no son lo suficientemente pequeños para la implementación de sistemas en tiempo real, lo cual se puede solucionar sustituyendo el código en lenguaje C por código en lenguaje ensamblador y utilizando imágenes más pequeñas como las del formato QCIF.

El sistema que ha sido implementado en este trabajo puede utilizarse como núcleo de aplicaciones más complejas, por ejemplo, se puede modificar para que comprima imágenes de cualquier tamaño y en escala de grises, se puede adaptar una cámara de video digital al módulo DSKC6416 para que el sistema comprima imágenes en tiempo real y se pueden programar los procesos que requieren el mayor número de operaciones en lenguaje ensamblador, con lo cual se disminuiría significativamente el tiempo que le lleva al sistema la compresión y la descompresión.

Anexos

A. Wavelet padre y Wavelet madre de diferentes bases ortonormales.

Daubechies 4	Daubechies 6	Daubechies 8
$u'=\begin{bmatrix}0.4830\\0.8365\\0.2241\\-0.1294\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.8365\\0.4830\\0\\0\\\vdots\\0\\0.1294\\0.2241\end{bmatrix}$	$u'=\begin{bmatrix}0.3327\\0.8069\\0.4599\\-0.1350\\-0.0854\\0.0352\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.8069\\0.3327\\0\\0\\\vdots\\0\\-0.0352\\-0.0854\\0.1350\\0.4599\end{bmatrix}$	$u'=\begin{bmatrix}0.2304\\0.7148\\0.6309\\-0.0280\\-0.1870\\0.0308\\0.0329\\-0.0106\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.7148\\0.2304\\0\\0\\\vdots\\0\\0.0106\\0.0329\\-0.0308\\-0.1870\\0.0280\\0.6309\end{bmatrix}$
Daubechies 10	**Daubechies 12**	**Daubechies 14**
$u'=\begin{bmatrix}0.1601\\0.6038\\0.7243\\0.1384\\-0.2423\\-0.0322\\0.0776\\-0.0062\\-0.1258\\0.0033\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.6038\\0.1601\\0\\0\\\vdots\\0\\-0.0033\\-0.1258\\0.0062\\0.0776\\0.0322\\-0.2423\\-0.1384\\0.7243\end{bmatrix}$	$u'=\begin{bmatrix}0.1115\\0.4946\\0.7511\\0.3153\\-0.2263\\-0.1298\\0.0975\\0.0275\\-0.0316\\0.0006\\0.004\\-0.0011\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.4946\\0.1115\\0\\0\\\vdots\\0\\0.0011\\0.004\\-0.0006\\-0.0316\\-0.0275\\0.0975\\0.1298\\-0.2263\\-0.315\\0.7511\end{bmatrix}$	$u'=\begin{bmatrix}0.0779\\0.3965\\0.7291\\0.4698\\-0.1439\\-0.2240\\0.0713\\0.0806\\-0.0380\\-0.0166\\0.0126\\0.0004\\-0.0018\\0.0004\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.3965\\0.0779\\0\\0\\\vdots\\0\\-0.0004\\-0.0018\\-0.0004\\0.0126\\0.0166\\-0.0380\\-0.0806\\0.0713\\0.2240\\-0.1439\\-0.4698\\0.7291\end{bmatrix}$

Symmlet 4	Symmlet 5	Symmlet 6
$u'=\begin{bmatrix}-0.1071\\-0.0419\\0.7037\\1.1367\\0.4212\\-0.1403\\-0.0178\\0.0456\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}0.0419\\-0.1071\\0\\0\\\vdots\\0\\-0.0456\\-0.0178\\0.1403\\0.4212\\-1.1367\\0.7037\end{bmatrix}$	$u'=\begin{bmatrix}0.0387\\0.0417\\-0.0553\\0.2820\\1.0231\\0.8966\\0.0235\\-0.2480\\-0.0298\\0.0276\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.0417\\0.0387\\0\\0\\\vdots\\0\\-0.0276\\-0.0298\\0.2480\\0.0235\\-0.8966\\1.0231\\-0.2820\\-0.0553\end{bmatrix}$	$u'=\begin{bmatrix}0.0218\\0.0049\\-0.1669\\-0.0683\\0.6945\\1.1139\\0.4779\\-0.1027\\-0.0298\\0.0633\\0.0025\\-0.0110\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.0049\\0.0218\\0\\0\\\vdots\\0\\0.0110\\0.0025\\-0.0633\\-0.029\\0.1027\\0.4779\\-1.1139\\0.6945\\0.0683\\-0.1669\end{bmatrix}$
Coifman 1	Coifman 2	Coifman 3
$u'=\begin{bmatrix}0.0386\\-0.1270\\-0.0772\\0.6075\\0.7457\\0.2266\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}0.1270\\0.0386\\0\\0\\\vdots\\0\\-0.2266\\0.7457\\-0.6075\\-0.0772\end{bmatrix}$	$u'=\begin{bmatrix}0.0164\\-0.0415\\-0.0674\\0.3861\\0.8127\\0.4170\\-0.0765\\-0.0594\\0.0237\\0.0056\\-0.0018\\-0.0007\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}0.0415\\0.0164\\0\\0\\\vdots\\0\\0.0007\\-0.0018\\-0.0056\\0.0237\\0.0594\\-0.0765\\-0.4170\\0.8127\\-0.3861\\-0.0674\end{bmatrix}$	$u'=\begin{bmatrix}-0.0038\\0.0078\\0.0235\\-0.0658\\-0.0611\\0.4052\\0.7938\\0.4285\\-0.0718\\-0.0823\\0.0345\\0.0159\\-0.0090\\-0.0026\\0.0011\\0.0005\\-0.0001\\-0.00003\\0\\0\\\vdots\\0\end{bmatrix}$, $v'=\begin{bmatrix}-0.0078\\-0.0038\\0\\0\\\vdots\\0\\0.00003\\-0.0001\\-0.0005\\0.0011\\0.0026\\-0.0090\\-0.0159\\0.0345\\0.0823\\-0.0718\\-0.4285\\0.7938\\-0.4052\\-0.0611\\0.0658\\0.0235\end{bmatrix}$

B. Tabla de estimación de probabilidades de la Codificación MQ.

Índice	Qe	NMPS	NLPS	SWITCH
0	0x5601	1	1	1
1	0x3401	2	6	0
2	0x1801	3	9	0
3	0x0AC1	4	12	0
4	0x0521	5	29	0
5	0x0221	38	33	0
6	0x5601	7	6	1
7	0x5401	8	14	0
8	0x4801	9	14	0
9	0x3801	10	14	0
10	0x3001	11	17	0
11	0x2401	12	18	0
12	0x1C01	13	20	0
13	0x1601	29	21	0
14	0x5601	15	14	1
15	0x5401	16	14	0
16	0x5101	17	15	0
17	0x4801	18	16	0
18	0x3801	19	17	0
19	0x3401	20	18	0
20	0x3001	21	19	0
21	0x2801	22	19	0
22	0x2401	23	20	0
23	0x2201	24	21	0
24	0x1C01	25	22	0
25	0x1801	26	23	0
26	0x1601	27	24	0
27	0x1401	28	25	0
28	0x1201	29	26	0
29	0x1101	30	27	0
30	0x0AC1	31	28	0
31	0x09C1	32	29	0
32	0x08A1	33	30	0
33	0x0521	34	31	0
34	0x0441	35	32	0
35	0x02A1	36	33	0
36	0x0221	37	34	0
37	0x0141	38	35	0
38	0x0111	39	36	0
39	0x0085	40	37	0
40	0x0049	41	38	0
41	0x0025	42	39	0
42	0x0015	43	40	0
43	0x0009	44	41	0
44	0x0005	45	42	0
45	0x0001	45	43	0
46	0x5601	46	46	0

Tabla B.1 Estimación de la probabilidad Qe en la codificación MQ

C. Banderas en el Estándar JPEG2000.

	Nombre	Código
Banderas para delimitar		
Start Of Codestream	SOC	4F
Start Of Tile-part	SOT	90
Start Of Data	SOD	93
End Of Codestream	EOC	D9
End Of Packet header	EPH	92
Banderas de segmento		
Image and tile Size	SIZ	51
Coding Style Default	COD	52
Coding Style Component	COC	53
Region of Interes	RGN	5E
Quantization Default	QCD	5C
Quantization Component	QCC	5D
Progression Order Default	POD	5F
Tile-part Lengths, Main header	TLM	55
Packet Length, Main header	PLM	57
Packet Length, Tile-part header	PLT	58
Packet Packet header, Main header	PPM	60
Packet Packet header, Tile-part header	PPT	61
Star Of Packet	SOP	91
Comment and Extension	CME	64

Tabla C.1 Nombre y código de las banderas de JPEG2000

Nombre	Obligatoria/Opcional
SOC	Requerida como primer bandera
SIZ	Requerida como segunda bandera
COD	Requerida
COC	Opcional
QCD	Requerida
QCC	Opcional
RGN	Opcional
POD	Requerida
PPM	Opcional
TLM	Opcional
PLM	Opcional
CME	Opcional

Tabla C.2 Banderas utilizadas en el encabezado principal

Nombre	Obligatoria/Opcional
SOT	Requerida como primer bandera
COD	Opcional
COC	Opcional
QCD	Opcional
QCC	Opcional
RGN	Opcional
POD	Opcional
PPT	Opcional
PLT	Opcional
CME	Opcional
SOD	Requerida

Tabla C.3 Banderas utilizadas en los encabezados de mosaico

D. Código fuente del sistema de compresión.

```
#include <stdio.h>
#include <math.h>
#include "Daub9_7.h"
#include "tablaB_1.h"
#pragma DATA_SECTION(R,  "Rsect")
#pragma DATA_SECTION(G,  "Gsect")
#pragma DATA_SECTION(Bl, "Bsect")
#define N 512
#define sc 1
#define Xc 1

char R[N][N], G[N][N], Bl[N][N];
float Y[N][N], Cb[N][N], Cr[N][N];
unsigned char B[25000];
short BP;
//mq_coder
char MPS[19],I[19],SC,CT;
unsigned short A;
char bitplane[32][32];
unsigned int C,T;

void preprocess(void)
{	short px,py;

	for(py=0;py<=N-1;py++)
		for(px=0;px<=N-1;px++)
		{	R[px][py]=R[px][py]-128;
			G[px][py]=G[px][py]-128;
			Bl[px][py]=Bl[px][py]-128;}

	for(py=0;py<=N-1;py++)
		for(px=0;px<=N-1;px++)
		{	Y[px][py]=0.299*R[px][py]+0.587*G[px][py]+0.114*Bl[px][py];
			Cb[px][py]=-0.168736*R[px][py]-0.331264*G[px][py]+0.5*Bl[px][py];
			Cr[px][py]=0.5*R[px][py]-0.418688*G[px][py]-0.081312*Bl[px][py]; } }
//////////////////////**********************//////////////////
void Daub_aZ(short Np,float Z[N+7],float coef[N])
{	short i,j;

	for(i=0;i<=Np-1;i++) coef[i]=0;

	for(j=0;j<=Np/2-1;j++)
		for(i=0;i<=8;i++)	coef[j]=coef[j]+ut[i]*Z[2*j+i];

	for(j=Np/2;j<=Np-1;j++)
		for(i=0;i<=8;i++)	coef[j]=coef[j]+vt[i]*Z[2*(j-Np/2)+i]; }

void DWT(void)
{	short i,Np,linea,column;
	float Z[N+7],coef[N];

	for(Np=N;Np>=32;Np=Np/2)
	{	for(linea=0;linea<=Np-1;linea++)
		{	for(i=0;i<=Np-1;i++) Z[i]=Y[linea][i];
			for(i=0;i<=6;i++)		Z[Np+i]=Z[i];
			Daub_aZ(Np,Z,coef);
			for(i=0;i<=Np-1;i++) Y[linea][i]=coef[i]; }

		for(column=0;column<=Np-1;column++)
		{	for(i=0;i<=Np-1;i++) Z[i]=Y[i][column];
			for(i=0;i<=6;i++)		Z[Np+i]=Z[i];
			Daub_aZ(Np,Z,coef);
			for(i=0;i<=Np-1;i++) Y[i][column]=coef[i]; } }

	for(Np=N;Np>=32;Np=Np/2)
	{	for(linea=0;linea<=Np-1;linea++)
		{	for(i=0;i<=Np-1;i++) Z[i]=Cb[linea][i];
```

```
                        for(i=0;i<=6;i++)               Z[Np+i]=Z[i];
                        Daub_aZ(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cb[linea][i]=coef[i]; }

                for(column=0;column<=Np-1;column++)
                {       for(i=0;i<=Np-1;i++) Z[i]=Cb[i][column];
                        for(i=0;i<=6;i++)               Z[Np+i]=Z[i];
                        Daub_aZ(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cb[i][column]=coef[i]; } }

        for(Np=N;Np>=32;Np=Np/2)
        {       for(linea=0;linea<=Np-1;linea++)
                {       for(i=0;i<=Np-1;i++) Z[i]=Cr[linea][i];
                        for(i=0;i<=6;i++)               Z[Np+i]=Z[i];
                        Daub_aZ(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cr[linea][i]=coef[i]; }

                for(column=0;column<=Np-1;column++)
                {       for(i=0;i<=Np-1;i++) Z[i]=Cr[i][column];
                        for(i=0;i<=6;i++)               Z[Np+i]=Z[i];
                        Daub_aZ(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cr[i][column]=coef[i]; } } }
//////////////////**********////////////////////
void Quant(char YDb[6],char CbDb[6],char CrDb[6])
{       short Nb=16,i,j;
        char pDb=0;

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Y[j][i]=(int)(Y[j][i]/YDb[pDb]);

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Y[j][i+Nb]=(int)(Y[j][i+Nb]/YDb[pDb]);
                                Y[j+Nb][i]=(int)(Y[j+Nb][i]/YDb[pDb]);
                                Y[j+Nb][i+Nb]=(int)(Y[j+Nb][i+Nb]/YDb[pDb]); }
        Nb=16;  pDb=0;
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cb[j][i]=(int)(Cb[j][i]/CbDb[pDb]);

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Cb[j][i+Nb]=(int)(Cb[j][i+Nb]/CbDb[pDb]);
                                Cb[j+Nb][i]=(int)(Cb[j+Nb][i]/CbDb[pDb]);
                                Cb[j+Nb][i+Nb]=(int)(Cb[j+Nb][i+Nb]/CbDb[pDb]); }

        Nb=16;  pDb=0;
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cr[j][i]=(int)(Cr[j][i]/CrDb[pDb]);

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Cr[j][i+Nb]=(int)(Cr[j][i+Nb]/CrDb[pDb]);
                                Cr[j+Nb][i]=(int)(Cr[j+Nb][i]/CrDb[pDb]);
                                Cr[j+Nb][i+Nb]=(int)(Cr[j+Nb][i+Nb]/CrDb[pDb]); } }
//////////////////**********////////////
void Nbitps(int Block[32][32],short pNbp,short Nbp[780],short Nb)
{       int max=0,a=1;
        char i,j;

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++)
                        if(abs(Block[j][i])>max) max=abs(Block[j][i]);

        Nbp[pNbp]=0;
        while(max>a-1)
        {       Nbp[pNbp]++;        a=a*2; }
        BP++;   B[BP]=Nbp[pNbp]; }
```

```
void Byte_out(void)
{       T=C>>19;
        if(T>0xFF)
        {       B[BP]++;
                if(B[BP]==0xFF)
                {       BP++;   B[BP]=0; }
                while(SC>0)
                {       BP++;   B[BP]=0;  SC--; }
                BP++;   B[BP]=T&0x000000FF; }

        else if(T==0xFF)    SC++;

        else
        {       while(SC>0)
                {       BP++;   B[BP]=0xFF;
                        BP++;   B[BP]=0;
                        SC--; }
                BP++;   B[BP]=T; }

        C=C&0x7FFFF; }

void Renorm_e(void)
{       while(A<0x8000)
        {       A=A<<1;  C=C<<1;
                CT--;

                if(CT==0)
                {       Byte_out();
                        CT=8; } } }

void mq_coder(char CX, char D)
{       char Ii;

        Ii=I[CX];
        A=A-Qe[Ii];
        if(MPS[CX]==D)
        {       if(A<0x8000)
                {       if(A<Qe[Ii])
                        {       C=C+A;
                                A=Qe[Ii]; }
                        I[CX]=NMPS[Ii];
                        Renorm_e(); } }
        else
        {       if(A>=Qe[Ii])
                {       C=C+A;
                        A=Qe[Ii]; }

                if(SWITCH[Ii]==1)   MPS[CX]=1-MPS[CX];

                I[CX]=NLPS[Ii];
                Renorm_e(); } }

void ZC(char m,char n,char sigma[34][34],char ZCtable)
{       char D,CX,SH,SV,SD;

        SH=sigma[m+sc][n-1+sc]+sigma[m+sc][n+1+sc];
        SV=sigma[m-1+sc][n+sc]+sigma[m+1+sc][n+sc];
        SD=sigma[m-1+sc][n-1+sc]+sigma[m-1+sc][n+1+sc]+sigma[m+1+sc][n-1+sc]+sigma[m+1+sc][n+1+sc];

        if(ZCtable==1)//HL
        {       if(SV==2) CX=8;
                else if(SV==1)
                {       if(SH>=1) CX=7;
                        else
                        {       if(SD>=1) CX=6;
                                else      CX=5; } }
                else
                {       if(SH==2) CX=4;
                        else if(SH==1)      CX=3;
                        else
```

```
                        {       if(SD>=2) CX=2;
                                else if(SD==1)      CX=1;
                                else CX=0; } } }

        else if(ZCtable==2)//LH
        {       if(SH==2) CX=8;
                else if(SH==1)
                {       if(SV>=1) CX=7;
                        else
                        {       if(SD>=1) CX=6;
                                else      CX=5; } }
                else
                {       if(SV==2) CX=4;
                        else if(SV==1)      CX=3;
                        else
                        {       if(SD>=2) CX=2;
                                else if(SD==1)      CX=1;
                                else      CX=0; } } }

        else//HH
        {       if(SD>=3) CX=8;
                else if(SD==2)
                {       if(SH+SV>=1)        CX=7;
                        else        CX=6; }
                else if(SD==1)
                {       if(SH+SV>=2)        CX=5;
                        else if(SH+SV==1)   CX=4;
                        else        CX=3; }
                else
                {       if(SH+SV>=2)        CX=2;
                        else if(SH+SV==1)   CX=1;
                        else        CX=0; } }

        D=bitplane[m][n];
        mq_coder(CX,D); }

void SCoding(char m,char n,char sigma[34][34],char X[34][34])
{       char auxH=-1,auxV=-1,H=1,V=1,Xgor=0,CX,D;

        if(sigma[m+sc][n-1+sc]*(1-2*X[m+Xc][n-1+Xc])+sigma[m+sc][n+1+sc]*(1-2*X[m+Xc][n+1+Xc])>-1)
        auxH=sigma[m+sc][n-1+sc]*(1-2*X[m+Xc][n-1+Xc])+sigma[m+sc][n+1+sc]*(1-2*X[m+Xc][n+1+Xc]);

        if(sigma[m-1+sc][n+sc]*(1-2*X[m-1+Xc][n+Xc])+sigma[m+1+sc][n+sc]*(1-2*X[m+1+Xc][n+Xc])>-1)
        auxV=sigma[m-1+sc][n+sc]*(1-2*X[m-1+Xc][n+Xc])+sigma[m+1+sc][n+sc]*(1-2*X[m+1+Xc][n+Xc]);

        if(auxH<H)          H=auxH;
        if(auxV<V)          V=auxV;

        if((H==1 && V==1) || (H==-1 && V==-1))CX=13;
        else if((H==1 && V==0) || (H==-1 && V==0))      CX=12;
        else if((H==1 && V==-1) || (H==-1 && V==1))     CX=11;
        else if((H==0 && V==1) || (H==0 && V==-1))      CX=10;
        else if(H==0 && V==0)       CX=9;

        if(H==-1 || (H==0 && V==-1))  Xgor=1;
        if(Xgor==X[m+Xc][n+Xc])     D=0;
        else        D=1;
        mq_coder(CX,D); }

void SPP(short Nb,char sigma[34][34],char eta[32][32],char X[34][34],char ZCtable)
{       char m=0,n=0,prefnei,j,i;

        while(m<Nb)
        {       prefnei=0;
                for(j=m-1;j<=m+1;j+=2)
                        for(i=n-1;i<=n+1;i++)
                                prefnei=prefnei+sigma[j+sc][i+sc];
                for(i=n-1;i<=n+1;i+=2)
                        prefnei=prefnei+sigma[m+sc][i+sc];
//prefei>0 el bit está en un vecindario preferente
```

```
                if(sigma[m+sc][n+sc]==0 && prefnei>0)
                {       ZC(m,n,sigma,ZCtable);
                        eta[m][n]=1;
                        if(bitplane[m][n]==1)
                        {       SCoding(m,n,sigma,X);
                                sigma[m+sc][n+sc]=1; } }
                //siguiente bit
                m++;
                if(m%4==0)
                {       m=m-4;  n++;
                        if(n==Nb)
                        {       n=0;        m=m+4; } } } }

void MRP(short Nb,char sigma[34][34],char sigmap[32][32],char eta[32][32])
{       char j,i,D,CX;
        short m=0,n=0;

        while(m<Nb)
        {       if(sigma[m+sc][n+sc]==1 && eta[m][n]==0)
                {       //MRC
                        if(sigmap[m][n]==1)  CX=16;
                        else
                        {       CX=14;
                                for(j=m-1;j<=m+1;j+=2)
                                        for(i=n-1;i<=n+1;i++)
                                                if(sigma[j+sc][i+sc]==1)        CX=15;
                                for(i=n-1;i<=n+1;i+=2)
                                        if(sigma[m+sc][i+sc]==1)        CX=15; }
                        D=bitplane[m][n];
                        mq_coder(CX,D);

                        sigmap[m][n]=1; }
                //siguiente bit
                m++;
                if(m%4==0)
                {       m=m-4;  n++;
                        if(n==Nb)
                        {       n=0;        m=m+4; } } } }

void CUP(short Nb,char sigma[34][34],char eta[32][32],char X[34][34],char ZCtable)
{       char m=0,n=0,j,i,neighbor,D,CX,sum,DP;

        while(m<Nb)
        {       if(sigma[m+sc][n+sc]==0 && eta[m][n]==0)
                {       if(m%4==0)
                        {       neighbor=0;
                                for(j=m-1;j<=m+4;j++)
                                        for(i=n-1;i<=n+1;i++)
                                                neighbor=neighbor+sigma[j+sc][i+sc];
                                if(neighbor==0)
                                {       //RLC
                                        CX=17;
                                        sum=0;
                                        for(i=m;i<=m+3;i++)
                                                sum=sum+bitplane[i][n];
                                        if(sum==0)
                                        {       D=0; m=m+3; }
                                        else
                                        {       D=1;        mq_coder(CX,D);
                                                CX=18;
                                                if(bitplane[m][n]==1)
                                                {       D=0;        DP=0; }
                                                else if(bitplane[m+1][n]==1)
                                                {       D=0;        DP=1;       m++; }
                                                else if(bitplane[m+2][n]==1)
                                                {       D=1;        DP=0;       m+=2; }
                                                else
                                                {       D=1;        DP=1;       m+=3; }
                                                mq_coder(CX,D);
                                                D=DP; }
```

```
                                        mq_coder(CX,D); }

                                else ZC(m,n,sigma,ZCtable); }
                        else ZC(m,n,sigma,ZCtable);

                        if(bitplane[m][n]==1)
                        {       SCoding(m,n,sigma,X);
                                sigma[m+sc][n+sc]=1; } }
                //siguiente bit
                m++;
                if(m%4==0)
                {       m=m-4;  n++;
                        if(n==Nb)
                        {       n=0;    m=m+4; } } } }
void flush(void)
{       T=C+A-1;
        T=T&0xFFFF0000;
        if(T<C)  T=T+0x8000;
        C=T<<CT;
        Byte_out();
        C=T<<8;
        Byte_out();

        BP++;   B[BP]=0xFF;
        BP++;   B[BP]=127; }

void Ebcot(int v[32][32],short pNbpE,short Nbp[780],short Nb,char ZCtable)
{       char sigma[34][34];
        char sigmap[32][32];
        char eta[32][32];
        char X[34][34];
        short j,i;
        short P;
        int mascara;

        for(j=0;j<=Nb+1;j++)
                for(i=0;i<=Nb+1;i++)
                {       sigma[j][i]=0;          X[j][i]=0;  }

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++)
                {       sigmap[j][i]=0;         eta[j][i]=0;}

        if(Nbp[pNbpE]>0)
        {       for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                                if(v[j][i]<0)
                                {       X[j+Xc][i+Xc]=1;    v[j][i]=-1*v[j][i]; }
                //inicializa mq_coder
                for(i=0;i<=18;i++)
                {       MPS[i]=0; I[i]=0; }
                I[0]=4;   I[17]=3; I[18]=46;
                A=0x8000; C=0; T=0;
                CT=11;  SC=0;

                for(P=Nbp[pNbpE]-1;P>=0;P--)
                {       mascara=1;
                        if(P!=0)  for(i=1;i<=P;i++)     mascara=mascara*2;
                        for(j=0;j<=Nb-1;j++)
                                for(i=0;i<=Nb-1;i++)
                                {       if((v[j][i]&mascara)>0)          bitplane[j][i]=1;
                                        else bitplane[j][i]=0; }

                        if(P!=Nbp[pNbpE]-1)
                        {       SPP(Nb,sigma,eta,X,ZCtable);    MRP(Nb,sigma,sigmap,eta); }
                        CUP(Nb,sigma,eta,X,ZCtable);

                        for(j=0;j<=Nb-1;j++)
                                for(i=0;i<=Nb-1;i++)
                                        eta[j][i]=0; }
```

```
                flush(); } }

void CAB(char YDb[6],char CbDb[6],char CrDb[6])
{       char pDb,bn=1,enj,eni,ZCtable;
        short Nb=16,i,j,k,bj,bi;
        short Nbp[780];
        short pNbp=-1,pNbpE=-1;
        int Block[32][32];

        BP=-1;
        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x4F;     //SOC
        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x5C;     //QCD
        for(pDb=0;pDb<=5;pDb++)
        {       BP++;   B[BP]=YDb[pDb]; }
        for(pDb=0;pDb<=5;pDb++)
        {       BP++;   B[BP]=CbDb[pDb]; }
        for(pDb=0;pDb<=5;pDb++)
        {       BP++;   B[BP]=CrDb[pDb]; }

        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x90;     //SOT
        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x93;     //SOD
        //////////////* Paquete 0 */////////////////
        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x91;     //SOP
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Y[j][i];
        pNbp++; Nbitps(Block,pNbp,Nbp,Nb);

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Cb[j][i];
        pNbp++; Nbitps(Block,pNbp,Nbp,Nb);

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Cr[j][i];
        pNbp++; Nbitps(Block,pNbp,Nbp,Nb);
        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x92;     //EPH
        /*************************************/
        ZCtable=2;
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Y[j][i];
        pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable);

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Cb[j][i];
        pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable);

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Block[j][i]=Cr[j][i];
        pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable);

        while(bn<=8)
        {       //Paquete n>0
                BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x91;     //SOP
                if(bn<=4)
                {       enj=0;  eni=1;
                        for(k=1;k<=3;k++)
                        {       for(bj=0;bj<=bn-1;bj++)
                                        for(bi=0;bi<=bn-1;bi++)
                                        {       for(j=0;j<=Nb-1;j++)
                                                        for(i=0;i<=Nb-1;i++)
                                        Block[j][i]=Y[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                        pNbp++; Nbitps(Block,pNbp,Nbp,Nb); }
                                eni=enj;  enj=1; } }
                else
                {       for(i=0;i<=3*bn*bn-1;i++)
                        {       pNbp++; Nbp[pNbp]=0;
                                BP++;   B[BP]=0; } }
                if(bn<=2)
                {       enj=0;  eni=1;
                        for(k=1;k<=3;k++)
                        {       for(bj=0;bj<=bn-1;bj++)
```

```
                                        for(bi=0;bi<=bn-1;bi++)
                                        {       for(j=0;j<=Nb-1;j++)
                                                        for(i=0;i<=Nb-1;i++)
                                        Block[j][i]=Cb[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                        pNbp++;  Nbitps(Block,pNbp,Nbp,Nb); }
                                eni=enj;  enj=1; }

                        enj=0;  eni=1;
                        for(k=1;k<=3;k++)
                        {       for(bj=0;bj<=bn-1;bj++)
                                        for(bi=0;bi<=bn-1;bi++)
                                        {       for(j=0;j<=Nb-1;j++)
                                                        for(i=0;i<=Nb-1;i++)
                                                Block[j][i]=Cr[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                        pNbp++;  Nbitps(Block,pNbp,Nbp,Nb); }
                                eni=enj;  enj=1; } }
                else
                {       for(i=0;i<=6*bn*bn-1;i++)
                        {       pNbp++;  Nbp[pNbp]=0;
                                BP++;    B[BP]=0; } }
                BP++;   B[BP]=0xFF;     BP++;   B[BP]=0x92;         //EPH

                enj=0;  eni=1;  ZCtable=1;
                for(k=1;k<=3;k++)
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Block[j][i]=Y[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable); }
                        eni=enj;  enj=1;  ZCtable++; }

                enj=0;  eni=1;  ZCtable=1;
                for(k=1;k<=3;k++)
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Block[j][i]=Cb[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable); }
                        eni=enj;  enj=1;  ZCtable++; }

                enj=0;  eni=1;  ZCtable=1;
                for(k=1;k<=3;k++)
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Block[j][i]=Cr[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)];
                                pNbpE++; Ebcot(Block,pNbpE,Nbp,Nb,ZCtable); }
                        eni=enj;  enj=1;  ZCtable++; }
                bn=bn*2;
                if(Nb!=32)
                {       Nb=32;  bn=1; } }

        BP++;   B[BP]=0xFF;     BP++;   B[BP]=0xD9; /*EOC*/ }

////////////////**********////////////////
void main(void)
{       char YDb[6] ={2,4,8,16,32,32};
        char CbDb[6]={2,4,8,16,32,32};
        char CrDb[6]={2,4,8,16,32,32};

        preprocess();
        DWT();
        Quant(YDb,CbDb,CrDb);
        CAB(YDb,CbDb,CrDb);

        while(1==1)
        {       } }
```

E. Código fuente del sistema de descompresión.

```
#include <stdio.h>
#include "Daub9_7.h"
#include "tablaB_1.h"
#define N 512
#define sc 1
#define Xc 1

unsigned char B[20000];
short BP;
char fin;
float Y[N][N],Cb[N][N],Cr[N][N];
char R[N][N],G[N][N],Bl[N][N];
//mq_coder
char MPS[19],I[19],SC,CT;
unsigned int A,C,T;
int Block[32][32];

//////////////////**********///////////////
void Byte_in(void)
{       short Bprov;
        if(fin==0)
        {       BP++;
                if(B[BP]!=0xFF)
                {       Bprov=B[BP];
                        C=C+(Bprov<<8); }
                else
                {       BP++;
                        if(B[BP]==0)            C=C+0xFF00;
                        else if(B[BP]==0x7F)fin=1;
                        else            puts("error Byte_in"); } } }

void Renorm_d(void)
{       while(A<0x80000000)
        {       if(CT==0)
                {       Byte_in();
                        CT=8; }

                A=A<<1;  C=C<<1;
                CT--; } }

char mq_decoder(char CX)
{       char D,Ii;
        unsigned int Qeprov;

        Ii=I[CX];
        Qeprov=Qe[Ii];          Qeprov=Qeprov<<16;
        A=A-Qeprov;

        if(C>=A)
        {       if(A>=(Qeprov))
                {       C=C-A;
                        A=Qeprov;
                        D=1-MPS[CX];
                        if(SWITCH[Ii]==1)   MPS[CX]=1-MPS[CX];
                        I[CX]=NLPS[Ii]; }
                else
                {       C=C-A;
                        A=Qeprov;
                        D=MPS[CX];
                        I[CX]=NMPS[Ii]; }
                Renorm_d(); }
        else
        {       if(A>=0x80000000)   D=MPS[CX];
                else
                {       if(A>=(Qeprov))
                        {       D=MPS[CX];
                                I[CX]=NMPS[Ii]; }
```

```
                        else
                        {       D=1-MPS[CX];
                                if(SWITCH[Ii]==1)   MPS[CX]=1-MPS[CX];
                                I[CX]=NLPS[Ii]; }
                        Renorm_d(); } }

        return D; }

void ZC(char m,char n,char sigma[34][34],char ZCtable, int masc)
{       char D,CX,SH,SV,SD;

        SH=sigma[m+sc][n-1+sc]+sigma[m+sc][n+1+sc];
        SV=sigma[m-1+sc][n+sc]+sigma[m+1+sc][n+sc];
        SD=sigma[m-1+sc][n-1+sc]+sigma[m-1+sc][n+1+sc]+sigma[m+1+sc][n-1+sc]+sigma[m+1+sc][n+1+sc];

        if(ZCtable==1)//HL
        {       if(SV==2) CX=8;
                else if(SV==1)
                {       if(SH>=1) CX=7;
                        else
                        {       if(SD>=1) CX=6;
                                else      CX=5; } }
                else
                {       if(SH==2) CX=4;
                        else if(SH==1)      CX=3;
                        else
                        {       if(SD>=2) CX=2;
                                else if(SD==1)      CX=1;
                                else CX=0; } } }

        else if(ZCtable==2)//LH
        {       if(SH==2) CX=8;
                else if(SH==1)
                {       if(SV>=1) CX=7;
                        else
                        {       if(SD>=1) CX=6;
                                else      CX=5; } }
                else
                {       if(SV==2) CX=4;
                        else if(SV==1)      CX=3;
                        else
                        {       if(SD>=2) CX=2;
                                else if(SD==1)      CX=1;
                                else      CX=0; } } }

        else//HH
        {       if(SD>=3) CX=8;
                else if(SD==2)
                {       if(SH+SV>=1)      CX=7;
                        else      CX=6; }
                else if(SD==1)
                {       if(SH+SV>=2)      CX=5;
                        else if(SH+SV==1)   CX=4;
                        else      CX=3; }
                else
                {       if(SH+SV>=2)      CX=2;
                        else if(SH+SV==1)   CX=1;
                        else      CX=0; } }

        D=mq_decoder(CX);
        if(D==1)   Block[m][n]=Block[m][n]+masc; }

void SCoding(char m,char n,char sigma[34][34],char X[34][34], int masc)
{       char auxH=-1,auxV=-1,H=1,V=1,Xgor=0,CX,D;

        if(sigma[m+sc][n-1+sc]*(1-2*X[m+Xc][n-1+Xc])+sigma[m+sc][n+1+sc]*(1-2*X[m+Xc][n+1+Xc])>-1)
                auxH=sigma[m+sc][n-1+sc]*(1-2*X[m+Xc][n-1+Xc])+sigma[m+sc][n+1+sc]*(1-2*X[m+Xc][n+1+Xc]);

        if(sigma[m-1+sc][n+sc]*(1-2*X[m-1+Xc][n+Xc])+sigma[m+1+sc][n+sc]*(1-2*X[m+1+Xc][n+Xc])>-1)
                auxV=sigma[m-1+sc][n+sc]*(1-2*X[m-1+Xc][n+Xc])+sigma[m+1+sc][n+sc]*(1-2*X[m+1+Xc][n+Xc]);
```

```
		if(auxH<H)		H=auxH;
		if(auxV<V)		V=auxV;

		if((H==1 && V==1) || (H==-1 && V==-1))CX=13;
		else if((H==1 && V==0) || (H==-1 && V==0))		CX=12;
		else if((H==1 && V==-1) || (H==-1 && V==1))		CX=11;
		else if((H==0 && V==1) || (H==0 && V==-1))		CX=10;
		else if(H==0 && V==0)		CX=9;

		if(H==-1 || (H==0 && V==-1))	Xgor=1;

		D=mq_decoder(CX);
		if(Xgor==D)		X[m+Xc][n+Xc]=0;
		else		X[m+Xc][n+Xc]=1; }

void dSPP(char Nb, char sigma[34][34], char eta[32][32],char X[34][34], char ZCtable, int masc)
{		char m=0,n=0,prefnei,j,i;

		while(m<Nb)
		{		prefnei=0;
				for(j=m-1;j<=m+1;j+=2)
						for(i=n-1;i<=n+1;i++)
								prefnei=prefnei+sigma[j+sc][i+sc];
				for(i=n-1;i<=n+1;i+=2)
						prefnei=prefnei+sigma[m+sc][i+sc];
//prefei>0 el bit está en un vecindario preferente
				if(sigma[m+sc][n+sc]==0 && prefnei>0)
				{		ZC(m,n,sigma,ZCtable,masc);
						eta[m][n]=1;
						if((Block[m][n]&masc)>0)
						{		SCoding(m,n,sigma,X,masc);
						sigma[m+sc][n+sc]=1; }}
				//siguiente bit
				m++;
				if(m%4==0)
				{		m=m-4;	n++;
						if(n==Nb)
						{		n=0;		m=m+4; } } } }

void dMRP(char Nb,char sigma[34][34], char sigmap[32][32], char eta[32][32], int masc)
{		char j,i,D,CX;
		short m=0,n=0;

		while(m<Nb)
		{		if(sigma[m+sc][n+sc]==1 && eta[m][n]==0)
				{		//MRC
						if(sigmap[m][n]==1)	CX=16;
						else
						{		CX=14;
						for(j=m-1;j<=m+1;j+=2)
								for(i=n-1;i<=n+1;i++)
										if(sigma[j+sc][i+sc]==1)		CX=15;
						for(i=n-1;i<=n+1;i+=2)
								if(sigma[m+sc][i+sc]==1)		CX=15; }

						D=mq_decoder(CX);
						if(D==1)	Block[m][n]=Block[m][n]+masc;

						sigmap[m][n]=1; }
				//siguiente bit
				m++;
				if(m%4==0)
				{		m=m-4;	n++;
						if(n==Nb)
						{		n=0;		m=m+4; } } } }

void dCUP(char Nb, char sigma[34][34], char eta[32][32],char X[34][34], char ZCtable, int masc)
{		char m=0,n=0,j,i,neighbor,D,Dp,CX;

		while(m<Nb)
```

```
        {       if(sigma[m+sc][n+sc]==0 && eta[m][n]==0)
                {       if(m%4==0)
                        {       neighbor=0;
                                for(j=m-1;j<=m+4;j++)
                                        for(i=n-1;i<=n+1;i++)
                                                neighbor=neighbor+sigma[j+sc][i+sc];
                                if(neighbor==0)
                                {       //RLC
                                        CX=17;
                                        D=mq_decoder(CX);
                                        if(D==0)  m=m+3;//los cuatro bits son cero
                                        else
                                        {       CX=18;
                                                Dp=mq_decoder(CX);
                                                D=mq_decoder(CX);
                                                if(Dp==0 && D==0)
                                                Block[m][n]=Block[m][n]+masc;
                                                else if(Dp==0 && D==1)
                                                {       m++;    Block[m][n]=Block[m][n]+masc; }
                                                else if(Dp==1 && D==0)
                                                {       m+=2;   Block[m][n]=Block[m][n]+masc; }
                                                else
                                                {       m+=3;   Block[m][n]=Block[m][n]+masc; } } }

                                        else ZC(m,n,sigma,ZCtable,masc); }
                                else ZC(m,n,sigma,ZCtable,masc);

                                if((Block[m][n]&masc)>0)
                                {       SCoding(m,n,sigma,X,masc);
                                        sigma[m+sc][n+sc]=1; } }
                        //siguiente bit
                        m++;
                        if(m%4==0)
                        {       m=m-4;  n++;
                                if(n==Nb)
                                {       n=0;    m=m+4; } } } }
void d_Ebcot(short pNbp,short Nbp[780],char Nb,char ZCtable)
{       char sigma[34][34]=0;
        char sigmap[32][32]=0;
        char eta[32][32]=0;
        char X[34][34]=0;
        short j,i,P;
        int masc;

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++)
                        Block[j][i]=0;

        fin=0;
        if(Nbp[pNbp]>0)
        {       //inicializa mq_coder
                for(i=0;i<=18;i++)
                {       MPS[i]=0; I[i]=0; }
                I[0]=4;     I[17]=3; I[18]=46;
                A=0x80000000; C=0; T=0;
                CT=0;       SC=0;

                BP++;       C=B[BP]; C=C<<8;
                BP++;       C=C+B[BP];          C=C<<16;
                if(B[BP]==255)      BP++;

                for(P=Nbp[pNbp]-1;P>=0;P--)
                {       masc=1;
                        if(P!=0)    for(i=1;i<=P;i++)       masc=masc*2;

                        if(P!=Nbp[pNbp]-1)
                        {       dSPP(Nb,sigma,eta,X,ZCtable,masc);
                        dMRP(Nb,sigma,sigmap,eta,masc); }
                        dCUP(Nb,sigma,eta,X,ZCtable,masc);
```

```
                    for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                                eta[j][i]=0; }

        if(fin==0) BP+=2;
        else fin=0;

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++)
                        if(X[j+Xc][i+Xc]==1)        Block[j][i]=Block[j][i]*-1; } }

void DcAB(short YDb[6],short CbDb[6],short CrDb[6])
{       char pDb,Nb,ZCtable=2,bn=1,bi,bj,eni,enj,k;
        short Nbp[780],j,i,pNbp=-1;

        Nb=16;  BP=-1;
        BP++;
        if(B[BP]==0xFF)     BP++;   if(B[BP]==0x4F)     BP++;  //SOC
        if(B[BP]==0xFF)     BP++;
        if(B[BP]==0X5C)                 //QCD
        {       for(pDb=0;pDb<=5;pDb++)
                {       BP++;   YDb[pDb]=B[BP]; }
                for(pDb=0;pDb<=5;pDb++)
                {       BP++;   CbDb[pDb]=B[BP]; }
                for(pDb=0;pDb<=5;pDb++)
                {       BP++;   CrDb[pDb]=B[BP]; } }

        BP++;
        if(B[BP]==0xFF)     BP++;   if(B[BP]==0X90)     BP++;  //SOT
        if(B[BP]==0xFF)     BP++;   if(B[BP]!=0X93)     puts("error SOD");  //SOD
        BP++;
        if(B[BP]==0xFF)     BP++;
        if(B[BP]==0X91)     //SOP
                for(i=0;i<=2;i++)
                {       BP++;   Nbp[i]=B[BP]; }
        BP++;
        if(B[BP]==0xFF)     BP++;   if(B[BP]!=0X92)     puts("error EPH"); //EPH

        pNbp++;             d_Ebcot(pNbp,Nbp,Nb,ZCtable);
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Y[j][i]=Block[j][i];
        pNbp++;             d_Ebcot(pNbp,Nbp,Nb,ZCtable);
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cb[j][i]=Block[j][i];
        pNbp++;             d_Ebcot(pNbp,Nbp,Nb,ZCtable);
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cr[j][i]=Block[j][i];

        while(bn<=8)
        {       BP++;
                if(B[BP]==0xFF)     BP++;
                if(B[BP]==0X91)     //SOP
                        for(pNbp=0;pNbp<=9*bn*bn-1;pNbp++)
                        {       BP++;   Nbp[pNbp]=B[BP]; }
                BP++;
                if(B[BP]==0xFF)     BP++;   if(B[BP]!=0X92)     puts("error EPH"); //EPH
        ////////////////////////
                pNbp=-1;
                enj=0;  eni=1;  ZCtable=1;
                for(k=1;k<=3;k++)
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       pNbp++;             d_Ebcot(pNbp,Nbp,Nb,ZCtable);
                                        for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Y[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)]=Block[j][i]; }
                        eni=enj;    enj=1;  ZCtable++; }

                enj=0;  eni=1;  ZCtable=1;
                for(k=1;k<=3;k++)
```

```
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       pNbp++;         d_Ebcot(pNbp,Nbp,Nb,ZCtable);
                                        for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Cb[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)]=Block[j][i]; }
                        eni=enj;    enj=1;      ZCtable++; }

                enj=0;      eni=1;      ZCtable=1;
                for(k=1;k<=3;k++)
                {       for(bj=0;bj<=bn-1;bj++)
                                for(bi=0;bi<=bn-1;bi++)
                                {       pNbp++;         d_Ebcot(pNbp,Nbp,Nb,ZCtable);
                                        for(j=0;j<=Nb-1;j++)
                                                for(i=0;i<=Nb-1;i++)
                                        Cr[j+Nb*(bn*enj+bj)][i+Nb*(bn*eni+bi)]=Block[j][i]; }
                        eni=enj;    enj=1;      ZCtable++; }

                bn=bn*2;
                if(Nb==16)
                {       Nb=32;      bn=1; } }

        if(B[BP]==0xFF)     BP++;       if(B[BP]!=0XD9)     puts("error EOC"); //EOC }
//////////////**********////////
void d_Quant(short YDb[6],short CbDb[6],short CrDb[6])
{       char pDb=0;
        short i,j,Nb=16;

        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Y[j][i]=Y[j][i]*YDb[pDb];

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Y[j][i+Nb]=Y[j][i+Nb]*YDb[pDb];
                                Y[j+Nb][i]=Y[j+Nb][i]*YDb[pDb];
                                Y[j+Nb][i+Nb]=Y[j+Nb][i+Nb]*YDb[pDb]; }

        Nb=16;      pDb=0;
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cb[j][i]=Cb[j][i]*CbDb[pDb];

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Cb[j][i+Nb]=Cb[j][i+Nb]*CbDb[pDb];
                                Cb[j+Nb][i]=Cb[j+Nb][i]*CbDb[pDb];
                                Cb[j+Nb][i+Nb]=Cb[j+Nb][i+Nb]*CbDb[pDb]; }

        Nb=16;      pDb=0;
        for(j=0;j<=Nb-1;j++)
                for(i=0;i<=Nb-1;i++) Cr[j][i]=Cr[j][i]*CrDb[pDb];

        for(Nb=16,pDb=1;Nb<=256;Nb=Nb*2,pDb++)
                for(j=0;j<=Nb-1;j++)
                        for(i=0;i<=Nb-1;i++)
                        {       Cr[j][i+Nb]=Cr[j][i+Nb]*CrDb[pDb];
                                Cr[j+Nb][i]=Cr[j+Nb][i]*CrDb[pDb];
                                Cr[j+Nb][i+Nb]=Cr[j+Nb][i+Nb]*CrDb[pDb]; } }
//////////////**********////////
void Daubm1(short Np,float Z[N+8],float coef[N])
{       short i,j;

        for(i=0;i<=Np+7;i++)Z[i]=0;
        for(j=0;j<=Np/2-1;j++)
                for(i=0;i<=7;i++)       Z[2*j+i]=Z[2*j+i]+coef[j]*u[i];

        for(j=0;j<=Np/2-1;j++)
                for(i=0;i<=9;i++)       Z[2*j+i]=Z[2*j+i]+coef[j+Np/2]*v[i];
        for(i=0;i<=7;i++)       Z[i]=Z[i]+Z[Np+i]; }
```

```
void IDWT(void)
{       short i,Np,linea,column;
        float Z[N+8],coef[N];

        for(Np=32;Np<=N;Np=Np*2)
        {       for(column=0;column<=Np-1;column++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Y[i][column];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Y[i][column]=Z[i]; }

                for(linea=0;linea<=Np-1;linea++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Y[linea][i];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Y[linea][i]=Z[i]; } }

        for(Np=32;Np<=N;Np=Np*2)
        {       for(column=0;column<=Np-1;column++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Cb[i][column];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cb[i][column]=Z[i]; }

                for(linea=0;linea<=Np-1;linea++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Cb[linea][i];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cb[linea][i]=Z[i]; } }

        for(Np=32;Np<=N;Np=Np*2)
        {       for(column=0;column<=Np-1;column++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Cr[i][column];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cr[i][column]=Z[i]; }

                for(linea=0;linea<=Np-1;linea++)
                {       for(i=0;i<=Np-1;i++) coef[i]=Cr[linea][i];
                        Daubm1(Np,Z,coef);
                        for(i=0;i<=Np-1;i++) Cr[linea][i]=Z[i]; } } }
//////////////**********////////
void Posprocess(void)
{       short px,py;

        for(py=0;py<=N-1;py++)
                for(px=0;px<=N-1;px++)
                {       if(Y[px][py]+1.402*Cr[px][py]<-128)                     R[px][py]=-128;
                        else if(Y[px][py]+1.402*Cr[px][py]>127)  R[px][py]=127;
                        else        R[px][py]=Y[px][py]+1.402*Cr[px][py];

                        if(Y[px][py]-0.344136*Cb[px][py]-0.714136*Cr[px][py]<-128)  G[px][py]=-128;
                        else if(Y[px][py]-0.344136*Cb[px][py]-0.714136*Cr[px][py]>127)          G[px][py]=127;
                        else        G[px][py]=Y[px][py]-0.344136*Cb[px][py]-0.714136*Cr[px][py];

                                if(Y[px][py]+1.722*Cb[px][py]<-128)                     Bl[px][py]=-128;
                                else if(Y[px][py]+1.722*Cb[px][py]>127)  Bl[px][py]=127;
                                else        Bl[px][py]=Y[px][py]+1.722*Cb[px][py]; }

        for(py=0;py<=N-1;py++)
                for(px=0;px<=N-1;px++)
                {       R[px][py]=R[px][py]+128;
                        G[px][py]=G[px][py]+128;
                        Bl[px][py]=Bl[px][py]+128; } }
//////////////**********////////
void main(void)
{       short YDb[6],CbDb[6],CrDb[6];

        DcAB(YDb,CbDb,CrDb);
        d_Quant(YDb,CbDb,CrDb);
        IDWT();
        Posprocess();

        while(1==1)
        {        } }
```

F. Archivos *Daub9_7.h* y *tablaB_1.h*.

```
/************** Daub9_7.h ***************/

#define a1 0.0378284554956993
#define a2 -0.0238494650131592
#define a3 -0.110624404085811
#define a4 0.377402855512633
#define a5 0.852698678836979
#define a6 0.377402855512633
#define a7 -0.110624404085811
#define a8 -0.0238494650131592
#define a9 0.0378284554956993

#define b1 0.0645388826835489
#define b2 -0.0406894176455255
#define b3 -0.418092272881996
#define b4 0.788485616984644
#define b5 -0.418092272881996
#define b6 -0.0406894176455255
#define b7 0.0645388826835489

float u[8]={0,-b1,b2,-b3,b4,-b5,b6,-b7};
float v[10]={0,a1,-a2,a3,-a4,a5,-a6,a7,-a8,a9};
float ut[9]={a1,a2,a3,a4,a5,a6,a7,a8,a9};
float vt[9]={0,0,b1,b2,b3,b4,b5,b6,b7};
```

```
/************** tablaB_1.h ***************/

unsigned short Qe[47] = {0x5601,0x3401,0x1801,0x0AC1,0x0521,0x0221,0x5601,0x5401,0x4801,0x3801,
        0x3001,0x2401,0x1C01,0x1601,0x5601,0x5401,0x5101,0x4801,0x3801,0x3401,
        0x3001,0x2801,0x2401,0x2201,0x1C01,0x1801,0x1601,0x1401,0x1201,0x1101,
        0x0AC1,0x09C1,0x08A1,0x0521,0x0441,0x02A1,0x0221,0x0141,0x0111,0x0085,
        0x0049,0x0025,0x0015,0x0009,0x0005,0x0001,0x5601};
char NMPS[47] = {1,2,3,4,5,38,7,8,9,10,
          11,12,13,29,15,16,17,18,19,20,
          21,22,23,24,25,26,27,28,29,30,
          31,32,33,34,35,36,37,38,39,40,
          41,42,43,44,45,45,46};
char NLPS[47] = {1,6,9,12,29,33,6,14,14,14,
          17,18,20,21,14,14,15,16,17,18,
          19,19,20,21,22,23,24,25,26,27,
          28,29,30,31,32,33,34,35,36,37,
          38,39,40,41,42,43,46};
char SWITCH[47] = {1,0,0,0,0,0,1,0,0,0,
            0,0,0,0,1,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0};
```

BIBLIOGRAFÍA

[1] Efford, N. *Digital Image Processing: A Practical Introduction Using Java.* Pearson Education. USA. 2000.

[2] González, R. *Digital Image Processing.* Prentice Hall. New Jersey, USA. 2002.

[3] Miano, J. *Compressed Image File Formats JPEG, PNG, GIF, XBM, BMP*. Addison Wesley. New York, USA. 1999.

[4] Baxes, G. A. *Digital Image Processing Principles and Applications.* John Wiley & Sons. USA. 1994

[5] Jain, A. K. *Fundamentals of Digital Image Processing*. Prentice Hall. USA. 1989.

[6] Perry, S. *Adaptive Image Processing.* CRC Press. Florida, USA. 2002.

[7] Acharya T. *Image Processing Principles and Applications*. John Wiley & Sons. Hoboken, NJ. 2005.

[8] Pajares, G. *Imágenes Digitales Procesamiento Práctico con Java*. Alfaomega, RA-MA. 2003.

[9] Richardson, I. *Video Codec Design*. John Wiley & Sons. United Kingdom. 2002

[10] Tarrés, F. *Sistemas Audiovisuales I. Televisión Analógica y Digital*. Edicions UPC. Barcelona, España. 2000.

[11] Ghanbari, M. *Standard Codecs: Image Compression to Advanced Video Coding*. IEE Press. London, United Kingdom. 2003.

[12] Sayood, K. *Introduction to Data Compresión.* Elsevier. USA. 2006

[13] Shi, Y. Q. *Image and Video Compression for Multimedia Engineering.* CRC Press. USA. 2000

[14] Salomon, D. *Data Compression the Complete Reference.* Springer. USA. 2007

[15] Rioul, O. *Wavelet and Signal Processing.* IEEE Signal Processing Magazine, págs. 14-38, octubre 1991.

[16] Frazier. M. W. *An Introduction to Wavelets Through Linear Algebra.* Springer. New York, USA. 1999.

[17] Acharya T. *JPEG2000 Standard for Image Compression.* John Wiley & Sons. New Jersey, USA. 2005.

[18] Vetterly, M. *Wavelets and Subband Coding.* Prentice Hall. New Jersey, USA. 1995.

[19] Walker, J. *A Primer on Wavelets and Their Scientific Applications.* Chapman & Hall/CRC. Florida, USA. 2008.

[20] Watkinson, J. *The MPEG handbook*. Focal Press. London, United Kingdom. 2001.

[21] Hernández, J. *Codificador-Decodificador JPEG en una Arquitectura DSP*. Tesis de Licenciatura, UNAM. 2006.

[22] Vitela, M. *Compresión MJPEG de video digital en un DSP*. Tesis de Maestría, UNAM. 2004.

[23] Chassaing, R. *Digital Signal Processing and Applications with the C6713 and C6416 DSK.* John Wiley & Sons. New Jersey, USA. 2005.

[24] Kehtarnavaz, N. *Real-Time Digital Signal Processing.* Elsevier. USA. 2005

[25] *TMS320C6416T DSK Technical Reference.* Spectrum Digital. USA. 2004