UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

COMPRESIÓN DE IMÁGENES UTILIZANDO LA TRANSFORMADA COSENO 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 :

JESÚS HERNÁNDEZ RIWES CRUZ

DIRECTOR DE TESIS: M.I. LARRY HIPÓLITO ESCOBAR SALGUERO

CIUDAD UNIVERSITARIA **MÉXICO D. F. OCTUBRE 2004**

Universidad Nacional Autónoma de México
UNAM
Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

ESTA TESIS NO SALE
DE LA BIBLIOTECA

The Knowledge of Time: Memory (that house of the Dead) – is the knowledge of inevitable personal destruction & at the same time immortality of a kind in human memory; for no word uttered, nor stone turned; no act, can ever be undone.

CHRIS CUTLER.

Autorizo a la Dirección General de Bibliotecas de la UNAM a difundir en formato electrónico e impreso el contenido de mi trabajo recepcional.
NOMBRE: Jesús Hernández Cruz
FECHA: 4 de octubre de 2004
FIRMA:

Resumen

El mundo contemporáneo se encuentra influenciado por los avances de la electrónica, las comunicaciones, el procesamiento digital de señales (PDS), etc. Gran parte de esta tecnología que ha caracterizado a estos últimos años han sido pequeños dispositivos de gran velocidad a bajo costo. Algunos de estos dispositivos permiten la manipulación de una gran variedad de señales como las imágenes y el sonido; ya sea permitiendo su transferencia, capturando su información o modificándola. Estos dispositivos los encontramos tanto en herramientas para la persona promedio hasta en usos muy especializados.

Asimismo, la digitalización ha hecho posible que el manejo de señales se torne más sencilla y en algunos casos sea más efectiva; el filtrado se vuelve más preciso, los procesos de encriptación son más diversos que en el caso de las señales analógicas, etc.

Uno de los dispositivos que apareció para aprovechar la digitalización de fenómenos fue el *Procesador Digital de Señales* (DSP). El objetivo de dicho microprocesador fue de entregar los resultados de sus cálculos de manera rápida y mantener un costo bajo mediante el diseño de su arquitectura.

Este trabajo se enfoca a la exploración y manejo de dicha tecnología en la compresión de imágenes mediante la implementación de algoritmos utilizados frecuentemente en el procesamiento digital de señales, tal como la *Transformada Discreta Coseno* (TC). Esta transformada aunada a otros procesos constituye el estándar de imágenes JPEG.

El objetivo de este trabajo es la creación de un bloque básico que se pueda utilizar en diversas aplicaciones que utilicen imágenes fijas y que al mismo tiempo sea la base de aplicaciones de imágenes en movimiento. Este bloque básico se producirá mediante la implementación en un DSP de la Transformada Coseno inmersa en un proceso basado en el estándar JPEG.

Mediante el uso del DSP TMS320C6711 de Texas Instruments (TI), se pretende obtener porcentajes de compresión de hasta un 10% con respecto al tamaño de las imágenes originales y al mismo tiempo mantener una muy buena calidad en las imágenes recuperadas. Asimismo se proyecta mejorar los tiempos de ejecución que se requerirían para calcular la TC haciendo uso de otros procesadores.

Índice General

RESUMEN I

ÍNDICE GENERAL III

CAPÍTULO 1 1
INTRODUCCIÓN 1

CAPÍTULO 2 5
ANÁLISIS DE IMÁGENES Y TÉCNICAS DE COMPRESIÓN 5
2.1 Imágenes 5
2.1.1 Conceptos básicos 6
2.1.2 Percepción visual 7
2.1.3 Modelos de color 8
2.1.4 Gráficos vectoriales y mapa de bits 9
2.1.5 Imágenes de tono continuo 10
2.2 Compresión de datos 10
2.2.1 Compresión con pérdidas vs compresión sin pérdidas 11
2.2.2 Técnicas de compresión 11
2.2.3 Formatos de imágenes comprimidas 12
2.3 Resumen 13

CAPÍTULO 3 15
LA TRANSFORMADA DISCRETA COSENO Y EL ESTÁNDAR JPEG 15
3.1 Formatos comunes de imágenes comprimidas 15
3.2 Transformaciones 16
3.2.1 La transformada Karhunen-Loeve 16
3.3 La Transformada Coseno 17
3.3.1 Definición 17
3.3.2 Efecto de la Transformada Coseno 18
3.3.3 Cuantización 23
3.3.4 Definición de la FDCT e IDCT unidimensional 24
3.3.5 Derivación de la DCT bidimensional a partir de la DCT unidimensional 25
3.3.6 Algoritmos rápidos para la DCT 26
3.4 El estándar JPEG 27
3.4.1 Codificación basada en la DCT 28
3.4.2 Modalidades de JPEG con pérdidas 29
3.4.3 Componentes de los modelos de color en JPEG 31
3.4.4 Frecuencia de muestreo 32
3.4.5 Codificación de datos intercalados y no intercalados 32
3.4.6 Corrimiento de nivel 33
3.4.7 Codificación de los coeficientes DC y AC 34
3.4.8 Marcadores o banderas en JPEG 41
3.5 Resumen 41

CAPÍTULO 4 43
IMPLEMENTACIÓN DE COMPRESIÓN DE IMÁGENES EN UN DSP 43
4.1 Elección de un procesador para la ejecución de los algoritmos 43
4.2 Descripción de la tarjeta DSK del TMS320C6711 44
4.2.1 Paquetes de ejecución y *pipeline* 46
4.2.2 Lenguaje ensamblador y *C* 47
4.2.3 Interfaz entre PC y DSP 48
4.3 Descripción del algoritmo de compresión implementado en el DSP 48
4.3.1 Almacenamiento de la imagen en el DSP 48
4.3.2 Lectura de datos de la imagen original 49

4.3.3 Aplicación de la Transformada Coseno a las unidades de datos 51
4.3.4 Cuantización de los coeficientes DCT 56
4.3.5 Codificación de las diferencias entre coeficientes DC 57
4.3.6 Codificación de los coeficientes AC 63
4.4 Módulos programados en el DSP para la compresión de imágenes *71*
4.5 Descripción del algoritmo de recuperación de la imagen implementado en el DSP *73*
4.5.1 Lectura de la información de la imagen comprimida 74
4.5.2 Decodificación de los coeficientes DC 79
4.5.3 Decodificación de los coeficientes AC 90
4.5.4 Decuantización de los coeficientes DCT 96
4.5.5 Aplicación de la Transformada Coseno Inversa a las unidades de datos 96
4.5.6 Escritura de las unidades de datos recuperadas en un arreglo de salida 98
4.6 Módulos programados en el DSP para la descompresión de imágenes *99*
4.7 Implementación de la DCT optimizada *100*
4.8 Resumen *102*

CAPÍTULO 5 105
RESULTADOS 105
5.1 Compresión y calidad de las imágenes recuperadas *105*
5.2 Desempeño del DSP en la implementación de la Transformada Coseno *115*
5.3 Resumen *118*

CAPÍTULO 6 119
CONCLUSIONES 119

ANEXOS 121
Anexo A 121
Tablas para la cuantización y para la codificación Huffman 121
Anexo B 130
Mapa de memoria del TMS320C6711 DSK 130
Anexo C 131
Código fuente de los programas para el DSP 131

GLOSARIO 137

BIBLIOGRAFÍA 141

Capítulo 1
Introducción

En la actualidad el gran avance de la tecnología ha permitido la transmisión de información a niveles difícilmente imaginados hace cien años, logrando el envío de altos volúmenes de información, incluyendo el intercambio de sonidos y de imágenes estáticas o en movimiento. Sin embargo, para que los equipos alcancen su máximo desempeño se requieren algoritmos que operen de manera eficiente.

Un procedimiento indispensable para equipos de comunicación de punta es la compresión de datos; ésta optimiza los recursos de almacenamiento y minimiza el tiempo de transmisión a cambio de tiempo de procesamiento en los equipos terminales, es decir, en la fuente y en el destino de la transmisión.

Dentro del ámbito de la compresión de imágenes existe un estándar llamado JPEG, cuyas siglas significan "Joint Photographic Experts Group"; nombre del grupo que lo desarrolló [1]. El estándar se ha hecho tan popular que se ha ganado un lugar en casi todas las aplicaciones multimedia, éste se divide principalmente en dos métodos, uno *sin pérdidas de información* y el otro *con pérdidas*.

El porcentaje de compresión con pérdidas es considerablemente más alto que el proceso sin pérdidas, y tiene la capacidad de mantener una calidad de imagen cuyas diferencias con la imagen original no son perceptibles por el ojo humano. La Transformada Coseno es la base del procedimiento con pérdidas y su alcance no se limita a JPEG, es la transformación más utilizada después de la transformada de Fourier en el procesamiento digital de señales [8].

De igual manera, los procesadores digitales de señales (DSP) han sido diseñados para tecnologías portátiles que requieran de una gran capacidad de procesamiento de datos a altas velocidades, lo cual se debe a su arquitectura que le permite realizar operaciones matemáticas intensivas en paralelo. Estas cualidades hacen de los DSP's una herramienta idónea para trabajar con algoritmos tales como JPEG.

La Transformada Coseno requiere una gran cantidad de sumas y multiplicaciones, asimismo las otras partes del estándar hacen uso de estas operaciones además de corrimientos y diversos procesos. En particular, el diseño de la familia de DSP's TMS320C6000 de la marca Texas Instruments (TI) se orientó hacia el procesamiento digital de imágenes [12].

Es evidente que ya existen ciertos programas que han implementado la Transformada Coseno, sin embargo la gran variedad de optimizaciones que han sido programadas no son parte del dominio público.

El *propósito de este trabajo* consiste en comprimir imágenes utilizando la Transformada Coseno, tomando como base el estándar JPEG y aprovechando la

capacidad de los DSP's de punto flotante; en este caso en particular el DSP TMS320C6711 de TI.

A continuación en la figura 1.1 se presenta un diagrama de un codificador basado en el estándar de compresión:

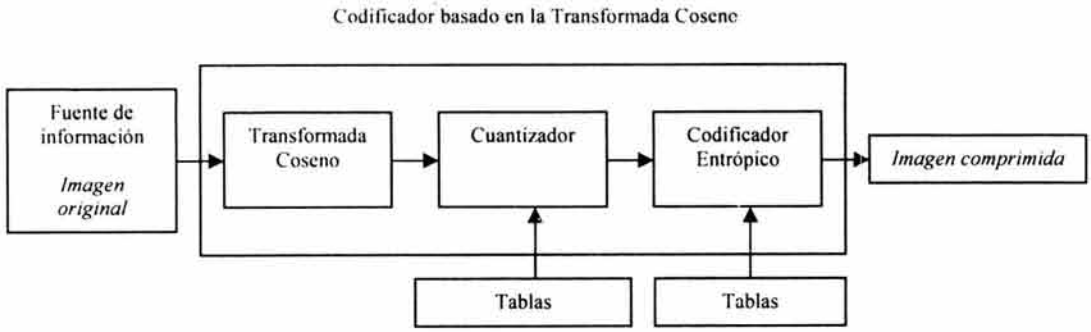


Fig.1.1 Codificador basado en la Transformada Coseno

De los bloques de la figura 1.1 podemos observar:

- Una fuente de información, que consiste en una imagen digitalizada; ya sea en tono de grises o en color.

Las tres etapas de compresión siguientes son propiamente un codificador JPEG base:

- La Transformada Coseno, siendo un procedimiento similar a la Transformada de Fourier, nos entrega coeficientes de señales senoidales que en conjunto representan nuestra imagen en el dominio de la frecuencia. La energía de la imagen se concentra en un cúmulo pequeño de los coeficientes que entrega la Transformada Coseno.
- Posteriormente se reduce la precisión de los coeficientes, es decir, se redondean a una cantidad que no sacrifique información discernible por el ojo humano.
- El último paso dentro del codificador antes de entregar la información comprimida consiste en asignar un código, donde los datos más probables en la imagen se representen con menos bits que aquellos menos probables. Obteniendo así máxima compresión de nuestra información.

En estos dos últimos pasos se hace uso de tablas de códigos y de datos que de igual manera debe tener el decodificador para reconstruir la imagen original. A la salida del codificador podemos observar los datos comprimidos. El proceso de decodificación es exactamente el inverso del mostrado en la figura 1.1.

Para implementar los procesos descritos en la figura 1.1, en la figura 1.2 se presenta el diagrama del equipo a utilizar.

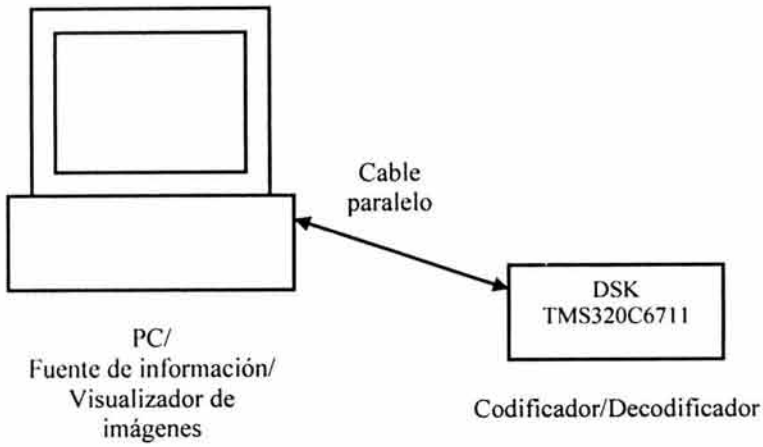


Fig. 1.2 Diagrama del equipo a utilizar en el proyecto

En la figura 1.2 observamos que la PC provee al DSP los datos de una imagen digitalizada a través del puerto paralelo LPT. El DSP realizará tanto la labor de codificación, así como la de decodificación y enviará de regreso por el mismo medio la imagen recuperada después de la compresión.

Dentro de la PC se harán las comparaciones pertinentes entre la imagen original y la imagen recuperada, para determinar porcentajes de compresión máximos en los cuales la distorsión no sea apreciable. A su vez en el DSP se pueden determinar los tiempos de ejecución del algoritmo. Con estos dos parámetros, porcentaje de compresión y tiempo de ejecución, obtendremos una referencia base para seguir buscando la forma optimizada del algoritmo que computacionalmente se acople mejor al DSP, permitiéndonos obtener resultados más eficientes.

El proyecto se enfoca al primer paso de la compresión a través de la Transformada Coseno, implementando las otras etapas de la compresión ilustradas en la figura 1.1 siguiendo algunas recomendaciones del estándar JPEG.

Se pretende probar el desempeño de arquitecturas que realicen las operaciones en paralelo, en donde la mayor parte de las operaciones multiplicación y adición se ejecuten en el mismo número de ciclos de reloj.

Esta aplicación puede ser un bloque fundamental para otras aplicaciones mayores, como por ejemplo, la transmisión de imágenes comprimidas (fijas o en movimiento) vía red.

Asimismo, la implementación de la Transformada Coseno y el uso de arquitecturas en paralelo no se limitan únicamente a imágenes, sino al Procesamiento Digital de Señales en general.

Debido a que el modo de operación del JPEG elegido es el modo de operación base, las imágenes a manejar tendrán una *resolución de 8 bits* para cada espacio de color, o para la componente de luminiscencia y para cada componente de crominancia. Los

niveles de compresión que se piensan obtener en imágenes de color se enumeran a continuación:

- 3-6% del tamaño original: desde calidad moderada hasta buena.
- 6-10% del tamaño original: desde calidad buena hasta muy buena.
- 10-20% del tamaño original: calidad excelente.
- 20-25% del tamaño original: indistinguible del original.

Es importante tener en cuenta que éstos son sólo resultados aproximados, debido a que la calidad y compresión de la imagen pueden variar significativamente debido a las características de la imagen y el contenido de la escena.

Inicialmente se espera que el proyecto terminado siente las bases para el aprovechamiento de la tecnología de los DSP's, acortando la distancia entre los manuales y los ingenieros, así como aportar algoritmos al conjunto de programas en lenguaje *C*.

Este trabajo está distribuido de la siguiente forma:

El *segundo capítulo* aborda conceptos de imágenes necesarios para entender y realizar un nivel básico de procesamiento digital de imágenes, y así pasar a una breve descripción de las técnicas de compresión de datos más utilizados en la actualidad. Con los conceptos de imágenes y las técnicas de compresión se hace una justificación de los métodos elegidos.

En el *tercer capítulo* se investiga más a fondo los pormenores de estos métodos; el estándar JPEG y particularmente la Transformada Coseno.

El *cuarto capítulo* resulta ser la piedra angular del trabajo, ya que se propone una arquitectura para procesar los algoritmos elegidos, se describe la arquitectura así como la interfaz utilizada para entrada y salida de los datos. Posteriomente se describe el porqué y el cómo de los diseños realizados para la compresión de imágenes. Asimismo se proponen comparaciones entre programas ya existentes no optimizados y aquellos diseñados en este proyecto, específicos para la arquitectura y compresión seleccionadas.

En los *últimos dos capítulos*, Resultados y Conclusiones, se hacen comparaciones, evaluaciones y análisis de los resultados obtenidos, llegando a las conclusiones del trabajo. Al final se anexan documentos relevantes del DSK TMS320C6711 así como de los códigos fuente de los programas realizados.

Capítulo 2
Análisis de imágenes y técnicas de compresión

En este capítulo se abordan conceptos básicos de imágenes y de compresión de imágenes que son necesarios para el entendimiento de los procesos a realizar en este trabajo. Se da una breve descripción de algunos métodos de compresión utilizados en las imágenes.

2.1 Imágenes

La mayoría de veces la gente da por sentado la definición de imagen, sin embargo, rara vez se tiene la noción del alcance de este concepto.

Una de tantas definiciones nos dice que es una "Representación de alguna cosa en pintura, escultura, dibujo, fotografía, etc." [2]. Esta definición queda abierta no sólo a las imágenes ópticas sino a otro tipo de imágenes; tomemos como ejemplo a una persona ciega. Para la persona en cuestión, una escultura no es una representación visual, sino táctil de la forma de un objeto, animal o individuo. Con base en el ejemplo anterior se puede inferir que todos los sentidos del ser humano, ya sea de manera individual o en conjunto, pueden generar la percepción de una imagen.

Es decir, que para obtener una imagen no es necesario percibir directamente el fenómeno a representar; mencionemos el caso de las imágenes infrarrojas. La luz infrarroja está fuera del espectro visible, nuestro ojo no registra su presencia, no obstante, con dispositivos especiales podemos crear una representación del espectro infrarrojo en algo que podamos interpretar. Es una práctica común representar los niveles de intensidad infrarroja mediante una tabla de colores, en donde cada color representa un nivel de intensidad en particular.

Sin embargo, nuestro campo de interés son las imágenes ópticas, es decir, aquellas que pueden percibirse directamente por el ojo humano, en particular las que son bidimensionales. Podemos dividir este grupo en dos, *las imágenes continuas* y *las discretas* y utilizar la siguiente definición de éstas:

"*Una imagen continua es aquélla donde la variación de tonos de gris o color se presenta sin discontinuidades, sin líneas, o fronteras, aparte de las que pudiera tener la escena misma. Una imagen discreta, por su parte, es la que está compuesta por elementos definidos y diferenciados como puntos o cuadrados.*" [3]

Lo que implica que dentro de una imagen discreta siempre habrá pérdida de información, independientemente de que nos dé la ilusión de que es continua. Dentro de las imágenes discretas podemos encontrar a su vez a las imágenes digitales, cuyos elementos en lugar de presentar tonalidades o colores guardan un número. En esta

modalidad, las imágenes pueden ser procesadas por las computadoras como un conjunto de datos numéricos.

A los puntos o cuadrados que componen la imagen se les ha llamado "campo instantáneo de vista" (CIV) y a sus equivalentes numéricos "píxel", aunque a veces se les maneja de manera indistinta [3]. Una representación gráfica se muestra en la figura 2.1.

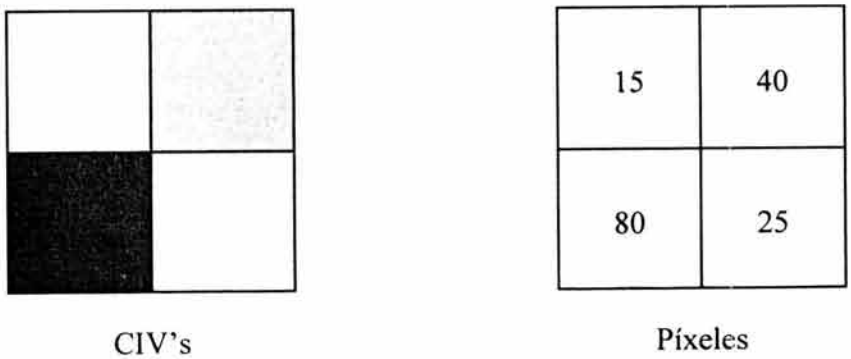


Figura 2.1 Ejemplos de CIV's y de píxeles

2.1.1 Conceptos básicos

Para poder representar una imagen es necesario un número finito de muestras suficientes para que el ojo humano perciba la continuidad. Cada muestra corresponde a un CIV. Siendo el CIV la partícula más pequeña que podemos representar, el tono que se le asigna es el promedio de los tonos que existen en el espacio equivalente en la escena original.

La precisión en la tonalidad dependerá de qué tan grande sea el conjunto numérico que utilicemos para representar la gama de tonos. En una computadora los números se manejan en forma binaria, y mientras más grande sea nuestra precisión se necesitarán más bits para representar un tono, por lo que es necesario mantener una relación coherente entre la precisión y el espacio disponible para guardar y manipular los datos.

En el caso de que no exista un tono exacto para representar el promedio de tonos contenido en la muestra original, se elegirá al tono más cercano que se pueda generar. Si nuestra gama de tonos no es lo suficientemente grande, nuestro ojo percibirá una diferencia notable entre la imagen original y la digital.

El nivel de detalle que nos proporciona el CIV debe ser lo suficientemente grande para que solamente abarque cambios pequeños en la tonalidad, de no ser así, la imagen reproducida puede ser totalmente distinta de la original; como ejemplo tomemos un cuadro compuesto por líneas alternadas de blanco y negro, ya sea en forma horizontal o vertical, si nuestro CIV abarca un par de líneas, el tono resultante del promedio del tono blanco y el tono negro en cada CIV nos generará un cuadro completamente gris en lugar del original.

El CIV define la *resolución espacial*; es decir, el grado de detalle con el que cuenta la silueta o figura que presenta una imagen. Es importante mencionar que no existe un criterio definido para dictaminar si la resolución de las imágenes es buena o mala, debido a que ciertas imágenes no necesitan un alto grado de detalle para representarse

correctamente y así afirmar que cuentan con buena resolución; la calidad de la resolución dependerá en gran medida del uso que pretendamos darle a una imagen [3].

2.1.2 Percepción visual

La información sensorial que nos ofrece una imagen puede ser examinada gracias a conceptos íntimamente ligados a la percepción visual. Con base en ellos podemos realizar análisis cualitativos y cuantitativos en una imagen. Sin embargo, estos análisis distarían mucho de ser útiles si no contamos con información acerca de las limitaciones del sistema visual humano. Cuando se nos presenta una imagen en dos dimensiones, aparte de poder discernir formas, nos encontramos que cada color que la compone cuenta con su propio tono, saturación y brillantez [4].

- Los *tonos* en sí, tienen que ver con la longitud de onda predominante de la luz que percibimos, es decir, qué tan "rojizo", "verduzco", etc. es un color.
- Su *saturación* es una cualidad de pureza, si realmente predomina una longitud de onda el color será intenso, sin embargo si existe una interferencia considerable entre ellas el color aparecerá deslavado.
- La *brillantez* depende de la intensidad de luz que percibamos del color; el color más brillante tenderá hacia el blanco, en contraste a uno poco brillante que hará parecer nuestro color al negro. Es importante mencionar que la brillantez no sólo depende de la cantidad de luz en sí, sino también del contraste con lo que lo rodea.

Tomando en cuenta los conceptos anteriores podemos empezar a notar las limitantes de presentar imágenes en cualquier dispositivo de salida con que contemos. Si nuestro dispositivo es por ejemplo, un monitor, la brillantez de las imágenes variará del original dependiendo de la iluminación del cuarto en el que nos encontremos.

Un concepto muy ligado tanto al CIV como a la resolución espacial, es la *frecuencia espacial*. En lugar de describir el grado de detalle propiamente, la frecuencia espacial se refiere a los cambios que sufre una imagen tanto vertical como horizontalmente. Mientras el CIV sea más pequeño, nuestra imagen digital podrá contener elementos con una frecuencia espacial mayor.

Existen curvas características para representar los límites en que el ojo puede percibir los cambios de intensidad (en brillo o en tono) dependiendo de la frecuencia espacial. Cálculos aproximados indican que la sensibilidad del ojo a los cambios de intensidad horizontales es mayor para objetos de aproximadamente 0.2cm si se observan a una distancia de un metro [1]. La respuesta del ojo a variaciones verticales es similar a la respuesta a las variaciones horizontales, sin embargo, la sensibilidad a cambios diagonales es significativamente menor.

El ojo humano puede distinguir aproximadamente 1,000 niveles de gris bajo condiciones ideales [1], con lo cual se podría sugerir un límite superior de 10 bits por píxel en una imagen digital. Sin embargo, en los sistemas reales 8 bits por píxel suele ser suficiente, ya sea por variaciones que afectan directamente nuestra percepción (distancia del dispositivo de salida, iluminación, etc.) o bien por las limitaciones de los dispositivos de salida para presentar los cambios entre los tonos.

2.1.3 Modelos de color

En la sección 2.1 se mencionó que el equivalente numérico del CIV es el píxel. Los sistemas que representan colores mediante números se denominan "*modelos de color*" [5], y el diseño de cada uno de ellos se enfoca en aprovechar las características del dispositivo visualizador o de salida en que está basado.

Entre los modelos más conocidos se encuentra el **RGB** (Red-Green-Blue), debido a que la mayoría de los monitores a color cuentan con estos tres tipos de fósforo y a que con la combinación de estos tres (rojo, verde y azul) generan todos los colores que el dispositivo presenta. A los colores "primarios" con los cuales se generan los demás se les ha denominado *componentes*. Si las tres componentes brillan a su mayor intensidad, la luz resultante será blanca, por el contrario, si no emiten luz el color que presentarán será negro. Cada componente en el modelo RGB aporta una parte del tono, saturación y brillantez de la imagen resultante.

A los sistemas de coordenadas que pueden representar modelos de color se le llama "*espacio de color*". En la figura 2.2 podemos observar la representación del espacio de color RGB.

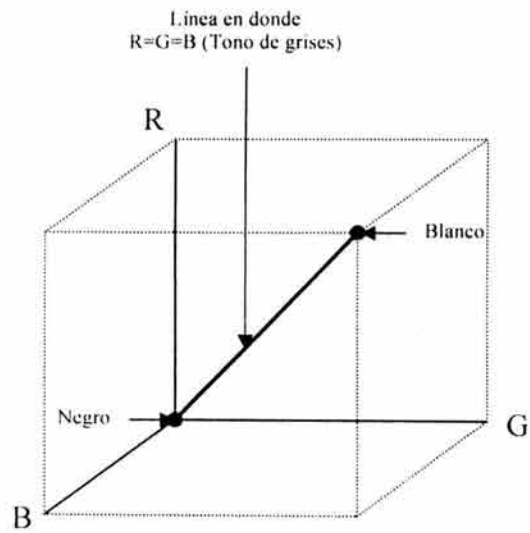


Figura 2.2 Representación del espacio de color RGB

El espacio está delimitado por un cubo que contiene los valores posibles a obtener en un dispositivo. Si trazamos una diagonal en donde las tres componentes tengan la misma magnitud, encontraremos que la diagonal sólo contiene intensidades diferentes del color gris. Dicha línea incluye el origen del espacio de color (el color negro), y el punto en donde las componentes alcanzan su valor máximo posible (el color blanco).

Para fines matemáticos, los valores para el espacio de color se normalizan a números reales entre el intervalo del 0.0 al 1.0, mientras que en programación se utilizan números enteros no signados.

Otro modelo de color relevante es el espacio de *tonos de grises*, el cual se representa por una sola componente cuyo intervalo del negro al blanco depende de la precisión con la que se tomen las muestras del tono. Al tono de grises se le suele llamar *blanco y negro*, a pesar de que el blanco y negro es un caso particular del tono de grises; aquel en que las muestras del tono se pueden representar con 1 bit, es decir, literalmente una imagen que sólo cuente con blanco y negro en ella.

Un acercamiento diferente a la representación de colores seguida en el modelo RGB lo maneja el modelo de color **YCbCr**. Al igual que el RGB, el modelo YCbCr consta de tres componentes. Sin embargo, una componente aporta la brillantez de la imagen y las otras dos aportan el tono y la saturación. A la componente que nos provee la brillantez de la imagen se le llama *componente de luminiscencia*, mientras que a las otras dos se les denomina *componentes de crominancia*. Aclaremos un poco más el concepto de crominancia:

"Se define al término *crominancia* como la diferencia entre un color y una referencia de color blanco al mismo nivel de luminiscencia" [1].

Para obtener los equivalentes en coordenadas entre los modelos RGB y YCbCr existen transformaciones lineales dadas por las siguientes ecuaciones:

$$Y = 0.3R + 0.6G + 0.1B \tag{2.1}$$

para la luminiscencia, donde Y es la luminiscencia, R la coordenada en rojo, G la coordenada en verde y B la coordenada en azul;

$$V = R - Y \tag{2.2}$$

$$U = B - Y \tag{2.3}$$

donde V y U son las componentes de otro espacio de color llamado YUV. U y V son las mismas componentes de crominancia que Cb y Cr, sólo que a Cb y a Cr se les aplica un escalamiento y un corrimiento en su origen, lo cual nos entrega:

$$Cb = (U/2) + 0.5 \tag{2.4}$$

$$Cr = (V/1.6) + 0.5 \tag{2.5}$$

El espacio YUV rebasa las coordenadas del espacio RGB; por lo que con el corrimiento y escalamiento se asegura que los valores de Cb y Cr siempre se encuentren dentro de los límites de 0 a 1 del espacio RGB.

2.1.4 Gráficos vectoriales y mapa de bits

Generalmente, se cuenta con dos modos de representar las imágenes: el *mapa de bits* y los *gráficos vectoriales*.

El *mapa de bits* es un arreglo de píxeles en dos dimensiones, en donde cada uno de los píxeles cuenta con un valor que representa directamente su color, o bien cuenta con una referencia a una tabla que a su vez nos refiere a cada color.

Por otro lado, los *gráficos vectoriales* constan de instrucciones de dibujo para representar las imágenes. Por ejemplo: comandos para dibujar arcos, líneas, etc., con referencia en un punto determinado. En otras palabras, generemos una imagen a partir de funciones y píxeles de referencia, sin necesidad de contar con el valor de cada píxel por separado.

En la figura 2.3 se muestra una imagen vectorial constituida por dos arcos. Cada arco se genera mediante un punto, un radio y un ángulo.

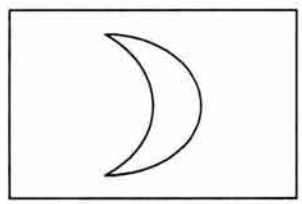


Figura 2.3 Ejemplo de una imagen vectorial

Mientras que en los gráficos vectoriales resulta sencillo modificar elementos individuales con procesos tales como escalamientos o rotaciones, en cualquier proceso aplicado a un mapa de bits, aún en los más sencillos como una ampliación produce dificultades.

En contraste, los mapas de bits son ideales para representar fotografías y cualquier imagen mucho más complicada. Para representar una fotografía en un gráfico vectorial se necesitaría de miles de comandos.

2.1.5 Imágenes de tono continuo

Una de las definiciones de una imagen de tono continuo nos dice que es aquella que necesita más de un bit para su representación [1]. Una acepción más general del concepto comprende a las imágenes que dan la apariencia de continuidad al observador.

2.2 Compresión de datos

Como se ha mencionado previamente, el mapa de bits resulta ser un formato ideal para archivar imágenes con escenas complejas. No obstante, el espacio que ocupan en memoria resulta ser grande y mientras se requiera más detalle en la imagen, requerirá mucho más espacio. Afortunadamente podemos optimizar nuestros recursos de memoria utilizando la compresión de datos.

2.2.1 Compresión con pérdidas vs compresión sin pérdidas

Dentro de la compresión de datos podemos encontrar dos vertientes generales; una vertiente con miras a recuperar toda la información suministrada originalmente y la otra sólo un aproximado de la información.

A la compresión en la que podemos recuperar toda la información del conjunto de datos original se le llama *compresión sin pérdidas*; cuya tasa de compresión en la mayoría de los casos no es mayor al 10% del total de la imagen. Esta modalidad de compresión está orientada a almacenar información con la cual se trabajará posteriormente; evitando procesar datos degradados.

En contraste tenemos los casos de imágenes que queremos archivar permanentemente o bien transmitirlas por algún medio. En el caso de contar con datos definitivos, podemos desechar la parte de los datos que no son relevantes en la aplicación para la cual fueron diseñados; por ejemplo, si tenemos una fotografía que solo apreciaremos mediante el ojo y eliminamos las frecuencias espaciales que no podamos distinguir mediante nuestra percepción visual, habremos comprimido nuestra información sin comprometer el uso para el cual fue diseñada.

Cuando transmitimos datos es deseable que el tiempo requerido para su envío y recepción sea mínimo, logrando maximizar la cantidad de información propagada hacia el destino y haciendo menos probable que un incidente interrumpa el proceso.

A la compresión en la que se recupera un aproximado del total de la información se le denomina *compresión con pérdidas*. Los compresores de datos de punta pueden comprimir la información desde 1/10 hasta 1/50 de su total [6].

2.2.2 Técnicas de compresión

La compresión aprovecha los patrones dentro del conjunto de datos para poder representar ese conjunto con otro de menor dimensión; es decir, los datos completamente aleatorios no se pueden compactar [5].

A continuación mencionamos algunas de las técnicas de compresión de datos junto con una breve descripción de cada una de ellas.

Codificación de datos iguales consecutivos

Si existe una serie de valores consecutivos de igual magnitud, éstos se podrán representar mediante un par de números: uno de los números indica las veces que el valor se repite y el otro número representa el valor. De esta manera al siguiente arreglo

7 7 7 7 7 7 7 7 7 7

también lo podemos representar como

10 7

Compresión mediante sustitución

Es un método de compresión que consiste en crear un "diccionario" de secuencias existentes dentro de los datos. Al momento de compactar los datos, las secuencias se reemplazan por los códigos que contiene el diccionario. Mientras más largas sean las secuencias con código en el diccionario y se repitan más, la compresión será mayor.

Entre los algoritmos más famosos se encuentran los esquemas LZ77, LZ78 y una variación de este último, el LZW [5]. La diferencia entre ellos radica en cómo se formula y transmite el diccionario dentro de los datos comprimidos; LZ77 formula su diccionario ayudándose de una ventana que recorre la información, y el LZ78 lo genera dinámicamente a partir de todos los datos.

Compresión Huffman

"*La codificación Huffman es una técnica de compresión de datos estadística, que reduce la longitud promedio del código utilizado para representar los símbolos de un alfabeto.*" [7] A los símbolos más frecuentes le asigna un código más corto y un código más largo a los menos frecuentes. Esta codificación es óptima en el caso de que la probabilidad de todos los símbolos dentro de un mensaje sean potencias de ½; en caso contrario, la longitud del código asignado no será entera y por lo tanto se perderá eficiencia debido a que no se pueden asignar códigos de fracción de bit.

Compresión aritmética

La codificación aritmética también es una técnica estadística, pero a diferencia de la codificación Huffman no necesita que las probabilidades de los símbolos en un mensaje sean potencias de ½. Se omitirá una descripción más detallada por dos razones, la más importante es que este esquema de compresión así como los LZ77 y LZ78 no pertenece al dominio público; la otra razón consiste en que el algoritmo toma más tiempo para procesar y es posible que no sea adecuado para implementaciones en que la rapidez sea indispensable, a pesar de que la compresión resultante mejora de un 5 a un 10% [6].

Compresión mediante Transformada Coseno

La Transformada Coseno aplicada a un grupo de píxeles nos regresa un grupo de coeficientes de señales senoidales a diferentes frecuencias con las cuales podemos representar a la imagen. Posteriormente se aprovecha el hecho de que las señales senoidales con mayor frecuencia espacial aportan muy poca información acerca de nuestros datos y se descartan. Es importante mencionar que esto es cierto sólo en el caso de que la imagen sea de tono continuo. Se dedicará gran parte del capítulo 3 a describir de una manera más completa las propiedades y el alcance de la Transformada Coseno.

2.2.3 Formatos de imágenes comprimidas

Para que diversos sistemas puedan leer una misma imagen, es necesario delimitar el principio y el fin de la información para acceder a ella. Asimismo, se pueden especificar otros parámetros como precisión de las muestras, número de muestras horizontales y verticales, etc., que dependerán del sistema y aplicación para el cual esté orientado el uso de los datos. Debido a lo anterior se han diseñado diversos *formatos de imágenes*, los cuales son estándares para ordenar la información de una imagen.

En nuestro caso particular necesitamos que nuestro formato también incluya parámetros para realizar el proceso inverso a la compresión. Cuando hablamos de compresión, es importante hacer notar que no nos referimos solamente a un proceso individual de los mencionados en el apartado anterior, sino a un conjunto de ellos. Los *formatos de imágenes comprimidas* guardan en sí una colección de algoritmos de compresión.

No existe un estándar de imágenes para elegir el proceso de compresión, sin embargo, podemos elegir entre la colección de formatos existentes que cuentan con algoritmos predeterminados. Una buena compresión dependerá en buena medida del formato de imagen seleccionado. Mencionemos como ejemplo el caso de una imagen en blanco y negro; si elegimos un formato que maneje tono de grises, no se perderá información al momento de representarla en el espacio de color. Probablemente se obtendrá una compresión de los datos, sin embargo la compresión no será comparable con la de un formato que se especialice en imágenes blanco y negro.

2.3 Resumen

En este capítulo se ha explicado que nuestro campo de interés lo componen las imágenes digitales. Una imagen digital se compone de un arreglo en dos dimensiones de muestras de una escena; a las muestras se les conoce como CIV y a su equivalente numérico píxel. Se puede describir el estado de un CIV mediante su tono, saturación y brillantez. Los dispositivos que despliegan imágenes aprovechan las limitaciones de la percepción visual para presentar una imagen que aparente ser igual al original. La información de una imagen puede manejarse utilizando diversos modelos de color. Los mapas de bits son ideales para representar imágenes fotográficas o relativamente complejas; pero su desventaja radica en la gran cantidad de memoria que se necesita para su almacenamiento. Como solución al almacenamiento de datos se propone la compresión; en particular un estándar que agrupe diversos algoritmos de compresión. La compresión no sólo redituará en una optimización de los recursos de almacenamiento, sino también en la disminución de los tiempos requeridos para la transmisión de datos.

Capítulo 3
La Transformada Discreta Coseno y el estándar JPEG

En este capítulo se explica la elección del estándar de compresión de imágenes JPEG y se analizan las etapas fundamentales que lo componen, asimismo se describe el efecto que produce en las imágenes cada proceso que forma parte del estándar. Se comienza por analizar el efecto de la Transformada Coseno en las señales, para proseguir con un proceso de eliminación de información. Finalmente se aborda el tema de la codificación de los datos para obtener así la información comprimida de la imagen.

3.1 Formatos comunes de imágenes comprimidas

Existe una gran cantidad de formatos para compresión de imágenes, pero entre los más utilizados se encuentran los formatos: **GIF** (Graphics Interchange Format), **PNG** (Portable Network Graphics) y **JFIF** (JPEG File Interchange Format). La estructura de JFIF está basada en el estándar JPEG.

Antes de la aparición del formato JFIF, el formato que predominaba en la mayoría de aplicaciones de transmisión y almacenamiento de imágenes era GIF. El porcentaje de compresión obtenido para cualquier imagen es aproximadamente el mismo, ya sea en blanco y negro, tono de grises o color. Sus principales características son [5]:

- Representación de hasta 256 colores utilizando de 1 a 8 bits por píxel
- Posibilidad de varias imágenes dentro de un mismo archivo

Al aparecer el estándar JPEG se dejó de utilizar GIF en las imágenes fotográficas debido a que JPEG ofrece una mejor compresión para estos casos y al mismo tiempo una mayor riqueza de colores.

Hoy en día GIF aparece cada vez en menos aplicaciones debido a asuntos legales; el algoritmo de compresión LZW dejó de formar parte del dominio público y por lo tanto cualquier persona que desarrolle una aplicación utilizando esa tecnología deberá pagar derechos. Existe la posibilidad de realizar una compresión en GIF sin utilizar LZW, sin embargo la ventaja de utilizar el formato en cuestión era la tasa de compresión.

PNG ha sido en gran medida la respuesta generada a partir del vacío que dejó GIF. La compresión del formato es sin pérdidas, lo que implica que no es un estándar óptimo para el almacenamiento final de imágenes. No obstante, PNG es un formato muy versátil; cuenta con elementos para revisar la integridad de la información, puede representar más tonos que JPEG e incluye el canal alfa que le permite generar capas de tonos

transparentes. PNG es ideal para imágenes que requieran de una edición posterior, su razón de compresión todavía dista de acercarse a la alcanzada por JPEG [5].

El estándar JPEG fue diseñado para comprimir imágenes de tono continuo, ya sea en color o en tono de grises. Lo anterior no implica que no se pueda aplicar a cualquier imagen, sin embargo la compresión será baja para imágenes que no sean de tono continuo. JPEG no es un formato de imágenes comprimidas, es un estándar que agrupa diversas técnicas de compresión.

JPEG se divide en dos modalidades base: una modalidad de *compresión con pérdidas* y otra de *compresión sin pérdidas*. El nivel de compresión alcanzado por la modalidad sin pérdidas es tan bajo que en algunos casos resulta provechoso utilizar cualquier otro formato de imágenes comprimidas que no esté basado en el estándar JPEG [7].

En los últimos años, la modalidad con pérdidas ha permitido disminuir de manera significativa el ancho de banda utilizado por todas las aplicaciones que manejen imágenes, al mismo tiempo de optimizar los recursos de almacenamiento. La base de dicha modalidad resulta ser la Transformada Coseno.

3.2 Transformaciones

Las transformaciones son operaciones cuyo fin es facilitar los procedimientos necesarios para resolver los problemas matemáticos. Por ejemplo, si se aplica la transformada de Laplace a una ecuación diferencial, el trabajo a realizar para resolver la ecuación se reducirá a manipulaciones algebraicas simples [8].

Algunas de las transformaciones más conocidas dentro del ámbito del *procesamiento digital de señales* (PDS) son la *transformada de Fourier* y la *transformada Karhunen-Loeve* (KLT). La transformada de Fourier descompone las señales y las representa mediante componentes dentro del dominio de la frecuencia, mientras que la KLT decorrelaciona completamente una función aleatoria y nos entrega coeficientes dentro del dominio de la transformada [8].

3.2.1 La transformada Karhunen-Loeve

Para describir esta transformación de una manera intuitiva, supongamos que deseamos transmitir una señal senoidal en un medio digital. Un método podría consistir en tomar muestras de la señal y enviarlas una por una. El receptor necesitaría procesar todos los datos para recuperar la señal original durante todo el tiempo en que la señal existiese. Eso sin tomar en cuenta que la señal se puede reconstruir mejor y de una manera más fácil mientras más muestras se envíen. Ahora bien, para representar adecuadamente una señal senoidal sólo se necesitan cinco elementos: su magnitud, fase, frecuencia, sincronía y el hecho de que es una señal senoidal. Así, en lugar de enviar muestras, bastaría con mandar estos cinco elementos para que el receptor pudiese generar una señal idéntica; en otras palabras, el nivel de correlación entre los valores de las muestras es alto, y por lo tanto la información que transmite cada muestra es poca. En

contraste, el total de la información que proveen todas las muestras de la señal senoidal, puede ser contenida en cinco datos con correlación nula. La KLT toma un número N de muestras de una señal y nos entrega datos sin correlación alguna con los que podemos recuperarla [8].

La decorrelación es una cualidad deseable en el PDS; gracias a ella cada coeficiente que compone a una señal puede ser manejado de manera independiente sin que disminuya la eficiencia en los procesos de compresión [1].

La transformada KLT es ideal para obtener un conjunto de coeficientes sin correlación, a partir de un número finito de muestras de una señal. Sin embargo, rara vez se encuentra una solución analítica para aplicarla. Es por ello que resulta ser una transformada impráctica para implementar en cualquier aplicación. Para resolver este problema, se han utilizado aproximaciones a la transformada. Una aproximación cuyo desempeño sobresale entre las demás es la *Transformada Discreta Coseno* (DCT), también llamada *Transformada Coseno*.

3.3 La Transformada Coseno

3.3.1 Definición

En realidad, no existe sólo una Transformada Coseno. La Transformada Coseno original fue clasificada como DCT-II por Wang [8]. A continuación se presenta su núcleo (*kernel*):

$DCT - II$:

$$\left[C_N^{II}\right]_{mn} = \left(\frac{2}{N}\right)^{1/2}\left[k_m \cos\left(\frac{m\left(n+\frac{1}{2}\right)\pi}{N}\right)\right] \qquad m,n = 0,1,\ldots,N-1; \qquad (3.1)$$

donde

$$k_m = 1 \qquad \text{para } m \neq 0$$

$$k_m = \frac{1}{\sqrt{2}} \qquad \text{para } m = 0$$

- $\left[C_N^{II}\right]$ es la matriz de la DCT-II de tamaño ($N \times N$)
- $\left[C_N^{II}\right]_{mn}$ es el elemento mn-ésimo de dicha matriz
- N es el número de muestras de la señal a transformar

La transformación se puede aplicar a arreglos multidimensionales. Al transformar un vector columna de entrada $x = [x(0), x(1), \ldots, x(N-1)]^T$, obtenemos el vector $X = [X(0), X(1), \ldots, X(N-1)]^T$ mediante la operación

$$X = \left[C_N^{II}\right] x \tag{3.2}$$

Para transformar una matriz V en una matriz T mediante la DCT, el procedimiento tomará esta forma

$$T = \left[C_N^{II}\right] V \left[C_N^{II}\right]^T \tag{3.3}$$

Los datos entregados por la DCT son números reales, lo cual disminuye la complejidad de su cálculo. Esta versión de la DCT mantiene un comportamiento asintótico a la KLT independientemente del número de muestras, siempre y cuando cada muestra tenga un nivel de correlación alto con respecto a las muestras adyacentes.

3.3.2 Efecto de la Transformada Coseno

La DCT es una transformación que descompone una señal en un grupo de señales senoidales ortogonales llamadas *funciones base*. La transformada toma un conjunto de muestras y nos entrega un conjunto de coeficientes; el número de elementos en ambos conjuntos es el mismo. Al escalarse la amplitud de las funciones base con los coeficientes resultantes de la transformación y calcularse la sumatoria de ellas se recupera la señal. Las funciones base se obtienen a partir de la matriz DCT. Cada renglón de la matriz DCT corresponde a una función base y cada valor representa la amplitud de la función base correspondiente en cada punto de muestreo de la señal a transformar.

DCT en 1-D

Consideremos el caso de la DCT para una dimensión. En la figura 3.1 se observa un ejemplo de la forma de ocho funciones base para $N = 8$. El eje de las ordenadas contiene la amplitud de las señales base sin escalar, y el de las abscisas el número correspondiente a cada muestra. Se puede observar que la frecuencia de la primera función base es igual a cero, y por lo tanto es una constante; las funciones consecutivas van aumentando en frecuencia. El coeficiente que corresponde a la función base constante se le llama *coeficiente DC*, a todos los demás se les conoce por *coeficientes AC*.

Debido a que las funciones base son ortogonales, ninguna de ellas podrá ser obtenida por cualquier combinación de las mismas. Sin embargo, cualquier otra función (en este caso compuesta por ocho muestras) podrá ser representada mediante una combinación lineal de las funciones base.

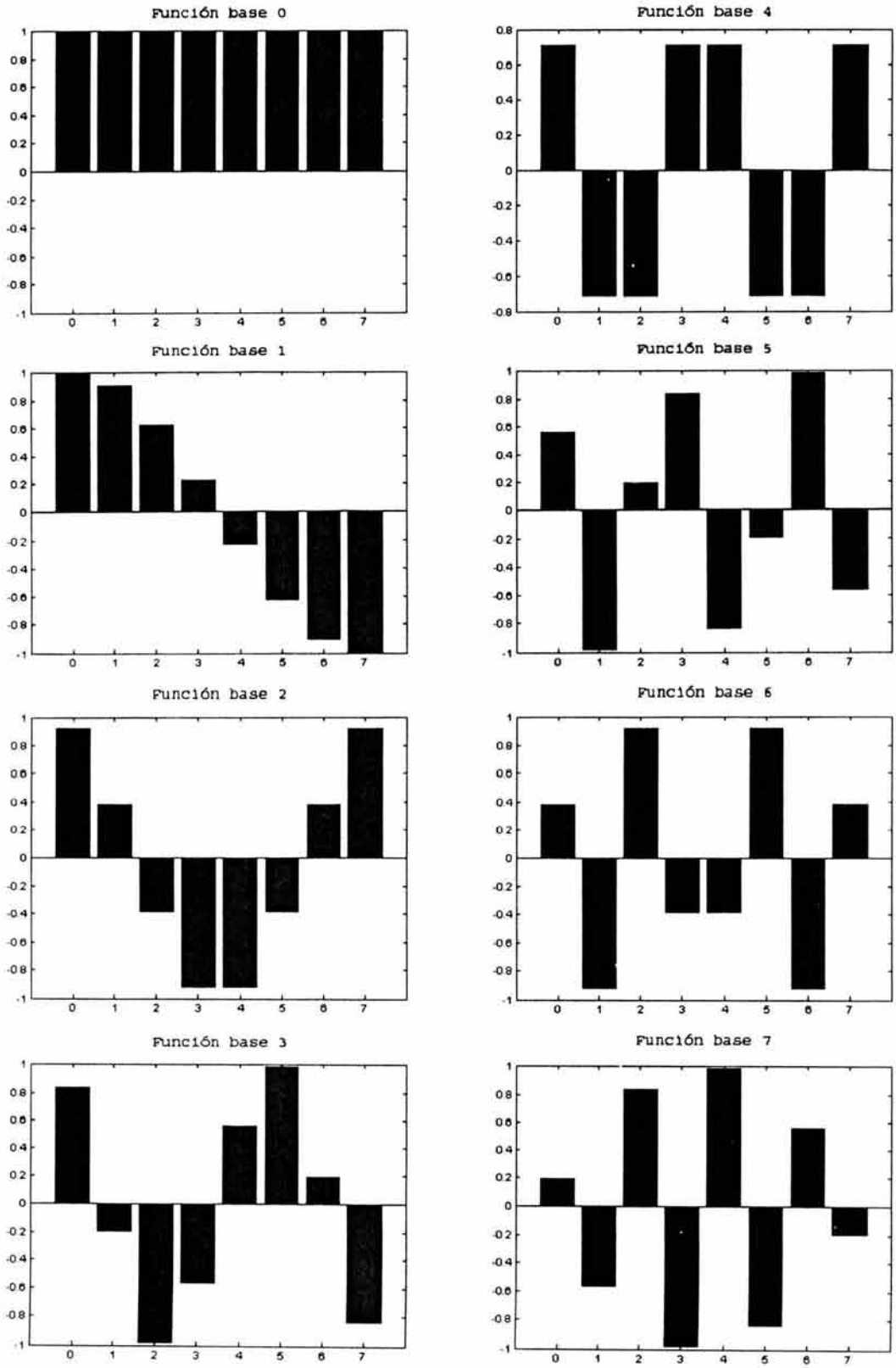


Fig. 3.1 Ocho formas de onda de funciones base de la DCT

Al hecho de descomponer un conjunto de muestras en un conjunto de funciones base escaladas se le llama *Transformada Discreta Coseno Directa*, o por sus siglas en inglés FDCT. A la reconstrucción del conjunto de muestras mediante las funciones base escaladas se le denomina *Transformada Discreta Coseno Inversa*, y sus siglas en inglés son IDCT.

Es importante remarcar que la reconstrucción de una señal sólo es posible idealmente. A pesar de que la FDCT nos entrega números reales, se necesita de una precisión numérica infinita para almacenarlos [5]. Debido a que no existe ningún sistema realizable que pueda lograrlo, la información que se le provee a la IDCT es incompleta; como resultado sólo obtenemos una reconstrucción aproximada de la señal original.

Se ha mencionado anteriormente que el estándar JPEG fue diseñado para comprimir imágenes de tono continuo, la razón de este hecho proviene de la DCT.

La tabla 3.1 contiene un conjunto de ocho muestras con un nivel de correlación alto, es decir que la curva que describen las muestras no contiene cambios drásticos.

n	0	1	2	3	4	5	6	7
Entrada	170	175	180	178	168	127	80	40
Coeficientes DCT	395.272	118.943	−72.773	16.010	−2.121	−1.148	−2.001	3.421

Tabla 3.1 Conjunto de ocho muestras con un nivel de correlación alto y sus respectivos coeficientes DCT

De igual manera se incluyen en la tabla 3.1 los coeficientes DCT correspondientes a cada muestra. Los coeficientes de dicha tabla se pueden obtener mediante la operación matricial descrita en la ecuación 3.2; donde x es el conjunto de muestras y X los coeficientes obtenidos.

La figura 3.2 muestra la colaboración de cada función base escalada en la reconstrucción de la señal x; los asteriscos representan las muestras originales y la línea la aproximación generada.

Para $n = 0$ encontramos que la IDCT aporta una función constante a la reconstrucción que, como se recordará, corresponde a la primera función base modificada por el coeficiente de DC. Las siguientes gráficas para diferentes n's corresponden a los coeficientes de AC, aquí notamos cómo con cada nueva función base la reconstrucción se va asemejando cada vez más al original.

Con este ejemplo es evidente que la información que aportan las funciones base con más alta frecuencia es poca; a partir de $n = 3$ ó $n = 4$ se puede afirmar que la semejanza entre la señal original y la recuperada es muy grande.

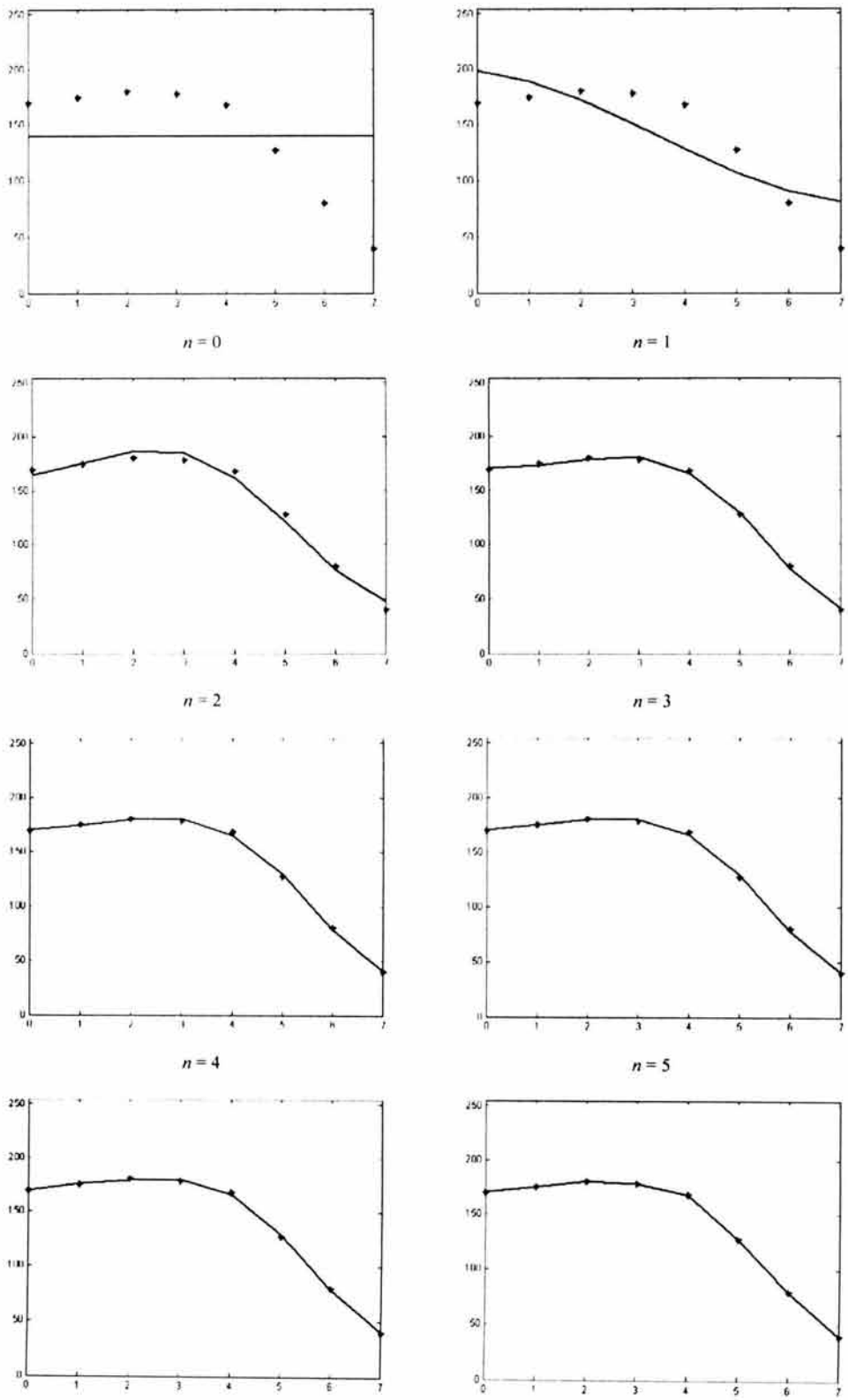


Figura 3.2 Reconstrucción de una señal con alto nivel de correlación entre sus muestras

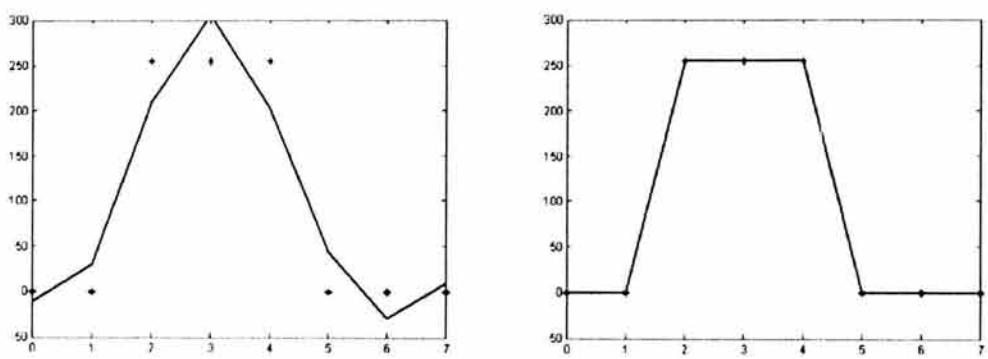
Para contrastar el ejemplo anterior, tomemos como fuente de información los datos con bajo nivel de correlación de la tabla 3.2.

n	0	1	2	3	4	5	6	7
Entrada	0	0	255	255	255	0	0	0
Coeficientes DCT	270.468	70.835	−284.381	−125.050	90.156	24.874	20.210	106.012

Tabla 3.2 Conjunto de ocho muestras con un nivel de correlación bajo y sus respectivos coeficientes DCT

La figura 3.3 incluye la señal de aproximación a la señal original con siete y ocho funciones base. En ella se puede apreciar que a pesar de que se hayan procesado siete funciones base de la IDCT, la función recuperada dista mucho de la original.

En el caso de los datos de la tabla 3.1 podemos descartar muchos coeficientes y aún así obtener una reconstrucción fiel de la señal original, si intentamos desechar cualquier coeficiente de la tabla 3.2 se producirá una distorsión notable. La diferencia entre los casos de la tabla 3.1 y la 3.2 radicó en el nivel de correlación que los datos de una señal mantienen entre sí.



$n = 6$ $n = 7$

Figura 3.3 Reconstrucción de una señal con bajo nivel de correlación entre sus muestras

DCT en 2-D

En una imagen, las funciones base corresponden a las frecuencias espaciales; podemos descartar aquellas que aportan poca información o ninguna que sea capaz de percibir el ojo. Al representar una imagen con menos datos que los originales habremos producido compresión de la información. Las imágenes de tono constante contienen datos con un nivel de correlación alto, por ello es posible lograr una gran razón de compresión utilizando la DCT.

Ahora bien, como las imágenes no son arreglos en una sola dimensión sino en dos, la DCT en una dimensión se puede extender para transformar arreglos de dos dimensiones. La figura 3.4 muestra un conjunto de 64 funciones base en dos dimensiones que se generan a partir de la multiplicación de dos conjuntos de una dimensión, ambos conjuntos con 8 funciones base; un conjunto unidimensional representando las frecuencias espaciales horizontales y el otro las frecuencias espaciales verticales. El

conjunto orientado horizontalmente representa las frecuencias horizontales, y el otro las frecuencias verticales. Por convención, el coeficiente DC de las frecuencias base horizontales se encuentra en el extremo izquierdo del arreglo y el coeficiente de las frecuencias base verticales en la parte superior. En consecuencia, el único término de DC es el que se encuentra en la esquina superior izquierda.

Con las 64 funciones base de este ejemplo podemos generar cualquier conjunto de 64 muestras de una imagen. Este conjunto de funciones base es el que utiliza el estándar JPEG, es decir, divide todos los píxeles en una imagen en grupos de 8×8 píxeles. Su elección no sólo radica en que resulta extremadamente complejo procesar un gran número de muestras al mismo tiempo (en cuanto a manejo de memoria), sino que al manejar un bloque pequeño nos permite desarrollar un proceso adaptable a las muestras, debido a que la variación de las frecuencias espaciales en un bloque de 8×8 píxeles es mucho menor que en una imagen completa. Además, como se ha mencionado, si la variación de las frecuencias espaciales en un conjunto de datos es pequeña, la compresión que se puede generar para ese conjunto de datos será mayor.

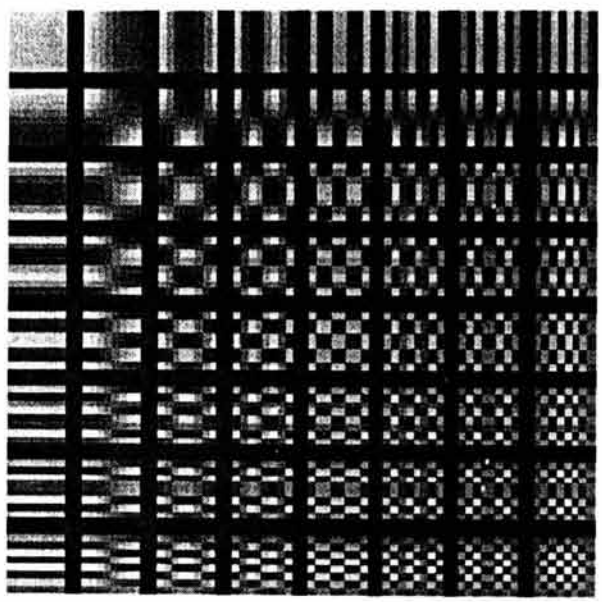


Figura 3.4 Funciones base de la DCT en 2-D

3.3.3 Cuantización

Al desechar coeficientes de la DCT se genera la compresión de imágenes. Al proceso que utiliza el estándar JPEG para descartar coeficientes se le llama *cuantización*; la cuantización consiste en una división del coeficiente por un número y el posterior redondeo del resultado a un número entero[1], es decir:

$$Sq_{vu} = redondeo\left(\frac{S_{vu}}{Q_{vu}}\right) \tag{3.4}$$

donde S_{vu} es un coeficiente DCT, Q_{vu} es el elemento cuantizador y Sq_{vu} es el coeficiente cuantizado.

En la cuantización se producen dos efectos de compresión:

- La precisión requerida para representar los coeficientes en un formato de número entero es menor a la precisión requerida para representar un número real; por lo tanto, la memoria o espacio utilizado para almacenar la información disminuye.
- En la división, cualquier coeficiente que aporte poca información a la señal tenderá a cero y finalmente se descartará al ser redondeado.

El conjunto modificado de coeficientes DCT se recupera mediante el proceso inverso, es decir, mediante la multiplicación de los datos por el valor con el que se les cuantizó.

Se pueden elegir diferentes valores de cuantización para cada muestra. El criterio para asignar valores de cuantización se basa en la visibilidad de las funciones base en las imágenes [1]. Es posible modificar el grado de perceptibilidad de una función base manipulando su amplitud; para cada coeficiente DCT existe un umbral en donde la magnitud del coeficiente apenas permite notar la aportación de la función base a la imagen, si durante la cuantización las magnitudes de los divisores Q_{vu} no rebasan dichos umbrales, al momento de recuperar los coeficientes DCT no se generará una diferencia visual entre la imagen original y la reconstruida, a pesar de que exista una diferencia entre los valores previos y los valores posteriores a la cuantización. Los umbrales de la luminiscencia son muy diferentes a los de crominancia, por lo que usualmente se utilizan dos tipos de tablas, una para la componente de luminiscencia y otra para las componentes de crominancia. Los estudios mediante los cuales se obtiene la visibilidad de las funciones base en las imágenes incluyen no sólo las características de la imagen, sino del dispositivo de salida y condiciones de iluminación que van más allá del alcance de este proyecto. Debido a lo anterior, se utilizarán tablas de cuantización prefabricadas que se incluyen en los anexos [1].

El estándar JPEG está diseñado de tal manera que para un valor de cuantización igual a uno, la precisión del coeficiente DCT sea de un máximo de 11 bits para muestras o píxeles de 8 bits de precisión.

3.3.4 Definición de la FDCT e IDCT unidimensional

La definición matricial vista en el apartado 3.3.1 puede aplicarse a n dimensiones, sin embargo, el uso de la transformada en tres dimensiones es limitado; sólo las DCT en una y dos dimensiones forman parte de las herramientas elementales en el PDS. Debido a que el estándar JPEG sólo utiliza la DCT para conjuntos de 8×8 muestras, se manejará esta modalidad de la transformada en lo que resta de este trabajo.

La DCT en dos dimensiones se puede realizar directamente aplicando un algoritmo bidimensional, o bien aplicando el algoritmo unidimensional primero horizontalmente y luego verticalmente. Esto es posible gracias a que la DCT cuenta con la propiedad de ser separable [13].

Definimos la Transformada Discreta Coseno Directa para un conjunto de ocho muestras en una dimensión como [1]:

FDCT:

$$S(u) = \frac{C(u)}{2} \sum_{x=0}^{7} s(x) \cos[(2x+1)u\pi/16] \qquad (3.5)$$

$$u: 0 \rightarrow 7$$

y la Transformada Discreta Coseno Inversa para ocho coeficientes en una dimensión como:

IDCT:

$$s(x) = \sum_{u=0}^{7} \frac{C(u)}{2} S(u) \cos[(2x+1)u\pi/16] \qquad (3.6)$$

$$x: 0 \rightarrow 7$$

donde

$C(u) = 1/\sqrt{2}$ cuando $u = 0$
$C(u) = 1$ cuando $u > 0$
$s(x)$ es el valor de una muestra en 1-D
$S(u)$ es un coeficiente DCT en 1-D

3.3.5 Derivación de la DCT bidimensional a partir de la DCT unidimensional

La FDCT e IDCT para 64 muestras en dos dimensiones se pueden generar a partir de los productos de dos DCT unidimensionales [1]:

FDCT:

$$S(v,u) = \frac{C(v)}{2} \frac{C(u)}{2} \sum_{y=0}^{7} \sum_{x=0}^{7} s(y,x) \cos[(2x+1)u\pi/16] \cos[(2y+1)v\pi/16] \qquad (3.7)$$

$$u, v: 0 \rightarrow 7$$

IDCT:

$$s(y,x) = \sum_{v=0}^{7} \frac{C(v)}{2} \sum_{u=0}^{7} \frac{C(u)}{2} S(v,u) \cos[(2x+1)u\pi/16] \cos[(2y+1)v\pi/16] \qquad (3.8)$$

$$x, y: 0 \rightarrow 7$$

donde

$C(u) = 1/\sqrt{2}$ cuando $u = 0$
$C(u) = 1$ cuando $u > 0$
$C(v) = 1/\sqrt{2}$ cuando $v = 0$
$C(v) = 1$ cuando $v > 0$
$s(y,x)$ es el valor de una muestra en 2-D
$S(v,u)$ es un coeficiente DCT en 2-D
x,y son los índices horizontal y vertical respectivamente de las muestras
u,v son los índices horizontal y vertical respectivamente de los coeficientes DCT

3.3.6 Algoritmos rápidos para la DCT

La forma más directa de aplicar la Transformada Coseno consiste en aplicar el algoritmo básico para la FDCT en una dimensión, no obstante, no es el método más eficiente. El número de operaciones requeridas para procesar un bloque de 8×8 muestras asciende a 1,024 multiplicaciones y 896 sumas; esto sin tomar en cuenta que posteriormente se deberá cuantizar el resultado.

A pesar de que en la actualidad los procesadores pueden realizar gran número de operaciones en tiempos cortos, se han desarrollado algoritmos que reducen drásticamente la intensidad de los cálculos. Dichos algoritmos aprovechan las propiedades de la DCT en una o en dos dimensiones. Las soluciones planteadas son diversas en forma; se utilizan algoritmos de la *transformada rápida de Fourier* (FFT), factorización de la matriz DCT, algoritmos de *decimación en el tiempo* (DIT) y *decimación en frecuencia* (DIF), se obtiene la DCT mediante otras transformaciones u otros métodos [8]. Para ejemplificar podemos nombrar que Feig desarrolló un modelo que entrega la DCT de un bloque de 8×8 muestras con sólo 54 multiplicaciones, 462 sumas y 6 corrimientos [1].

El diseño de algoritmos rápidos busca encontrar factores comunes dentro de la transformación, fusionar la cuantización con las operaciones propias de la DCT; pero principalmente generar un algoritmo recursivo. Un algoritmo recursivo es ideal para cualquier tecnología, sin embargo, es deseable aprovechar al máximo las particularidades de cualquier sistema en el que se vaya a implementar.

3.4 El estándar JPEG

En la introducción de este capítulo se mencionó la clasificación más general de las modalidades de JPEG: compresión con pérdidas y compresión sin pérdidas. Se ha elegido en este trabajo estudiar la modalidad con pérdidas (basada en la DCT) debido a que ningún estándar ha podido competir hasta ahora con este nivel de compresión en imágenes de tono constante [5].

La figura 3.5 muestra el proceso general de codificación que maneja JPEG, y en la figura 3.6 se puede apreciar el proceso de decodificación.

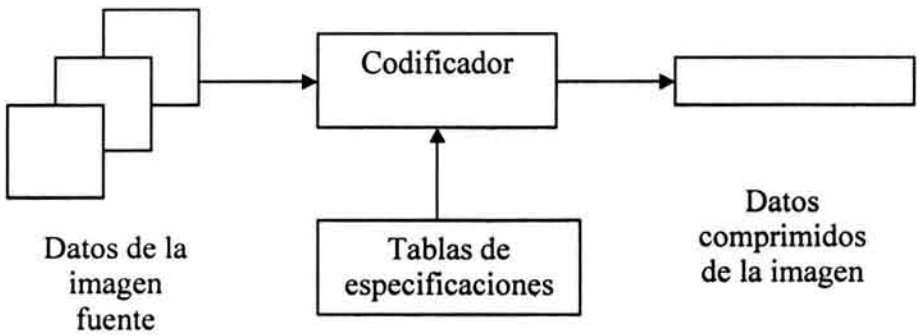


Fig. 3.5 Codificador JPEG

En la figura 3.5, se suministran los datos digitales de la imagen fuente y las tablas de especificaciones al codificador. A partir de un proceso de eliminación de información no indispensable y de un proceso codificación estadístico, el codificador entrega los datos comprimidos de la imagen.

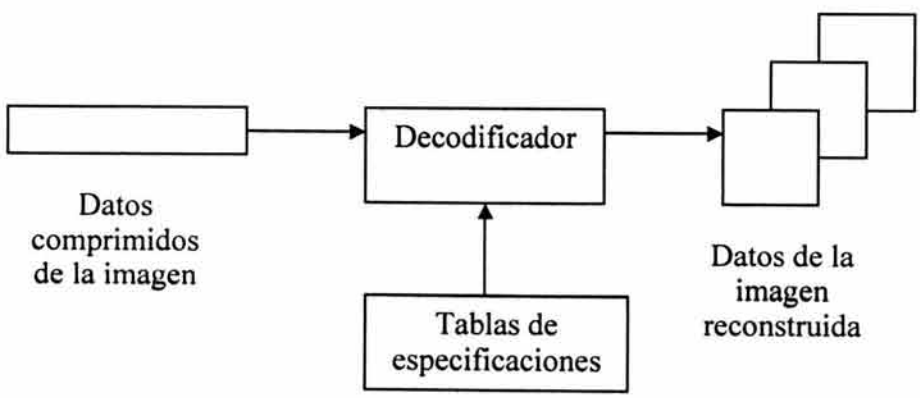


Fig. 3.6 Decodificador JPEG

Al decodificador se le suministra los datos comprimidos de la imagen y las tablas de especificaciones. Mediante un proceso inverso al de codificación entrega una imagen reconstruida.

3.4.1 Codificación basada en la DCT

La figura 3.7 muestra los procedimientos principales para todos los procesos de compresión de JPEG basados en la DCT e ilustra el caso particular de una imagen cuyo espacio de color está compuesto por una sola componente.

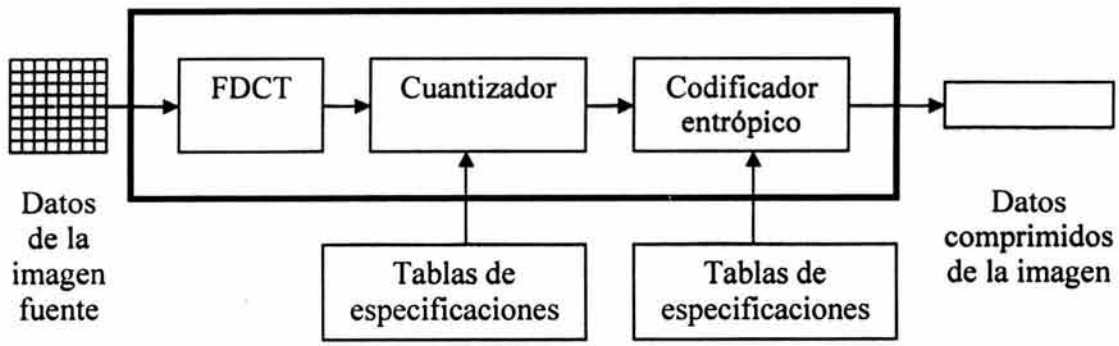


Fig. 3.7 Diagrama simplificado de un codificador basado en la DCT

En el proceso de compresión se agrupan las muestras de la componente de la imagen suministrada en bloques de 8×8 píxeles. Dichos bloques se denominan *unidad de datos*. El orden en que se procesan los bloques es de izquierda a derecha y de arriba abajo, siendo el bloque de 8×8 de la esquina superior izquierda el primero en ser procesado.

a) A cada bloque se aplica la FDCT para obtener un conjunto de 64 coeficientes DCT.

b) Cada uno de los coeficientes de la FDCT se cuantiza utilizando el valor que le corresponde en una *tabla de cuantización.*[•]

c) Una vez cuantizados los coeficientes, éstos se preparan para codificarse entrópicamente.[†] La información se traduce a símbolos que podrán representarse posteriormente mediante *códigos de longitud variable*. Los coeficientes de DC y de AC se manejan por separado. Más adelante se verá con más detalle este procedimiento.

[•] La forma de obtener dicha tabla no está especificada dentro de JPEG, y como se ha mencionado en un apartado anterior, su construcción implica estudios de las características particulares de la imagen, del dispositivo en que se vaya a desplegar, de las condiciones de iluminación del lugar en donde se encuentre el dispositivo, etc.

[†] La codificación entrópica es un proceso estadístico que obtuvo su nombre por su símil con el concepto termodinámico; mientras un sistema tienda a un estado estable, su cantidad de entropía aumentará. Se puede describir cualitativamente a la entropía como la medida del nivel de "sorpresa" o qué tan inesperada será la información, si es que dicha información es diferente a la que se espera de un sistema. Debido a esta propiedad de los sistemas, se asignan códigos pequeños a la información esperada y códigos largos a la información inesperada; reduciendo de esta manera la longitud de la información más no su contenido [1].

d) Ya preparada la información, se puede optar por aplicar codificación aritmética o codificación Huffman. Para realizar la codificación se utilizan tablas condicionales o tablas Huffman dependiendo del caso.

La figura 3.8 muestra los procedimientos básicos para cualquier decodificador basado en la DCT. Esencialmente, cada paso es el inverso del correspondiente en la codificación. La información de entrada se decodifica entrópicamente con el uso de tablas entregando los coeficientes cuantizados. Las tablas pueden estar incluidas en los datos comprimidos o bien la aplicación puede tener conocimiento previo de ellas. Los coeficientes se decuantizan y se les aplica la IDCT. Como resultado obtenemos bloque por bloque la información reconstruida de la imagen.

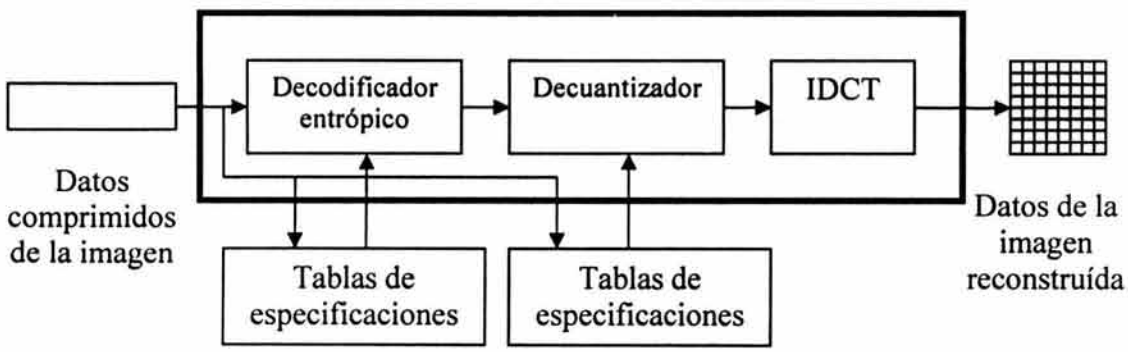


Fig. 3.8 Diagrama simplificado de un decodificador basado en la IDCT

3.4.2 Modalidades de JPEG con pérdidas

Dentro de la compresión con pérdidas podemos encontrar diversas modalidades de operación en JPEG:

- Secuencial
- Progresiva
- Jerárquica
- Básica

Modalidad secuencial

La modalidad secuencial realiza solamente una lectura de los datos contenidos en una imagen; el orden de procesamiento de los bloques de 8×8 píxeles es de izquierda a derecha y de arriba abajo. Esta manera de procesar la imagen permite que cada bloque de 8×8 píxeles pueda ser cuantizado y codificado inmediatamente después de haber sido transformado, minimizando los recursos de memoria utilizados.

Modalidad Progresiva

En la modalidad progresiva los datos se procesan en el mismo orden que en la modalidad secuencial, sin embargo, en este caso ocurren varias lecturas de la información de una imagen. Ya cuantizados, los coeficientes DCT se codifican parcialmente en cada lectura. Existen dos procedimientos para codificar parcialmente los coeficientes, al primero se le llama *selección espectral* debido a que los coeficientes de un bloque se dividen dependiendo de su frecuencia espacial y se procesan parte por parte; iniciando los coeficientes con frecuencia espacial baja, siguiéndoles los coeficientes con frecuencias más altas. Al segundo procedimiento se le conoce como *aproximación sucesiva*; en la primera lectura de datos se codifican los bits más significativos de todos los coeficientes, y en las lecturas posteriores se codifican los bits menos significativos. Ambos procedimientos pueden utilizarse juntos o por separado.

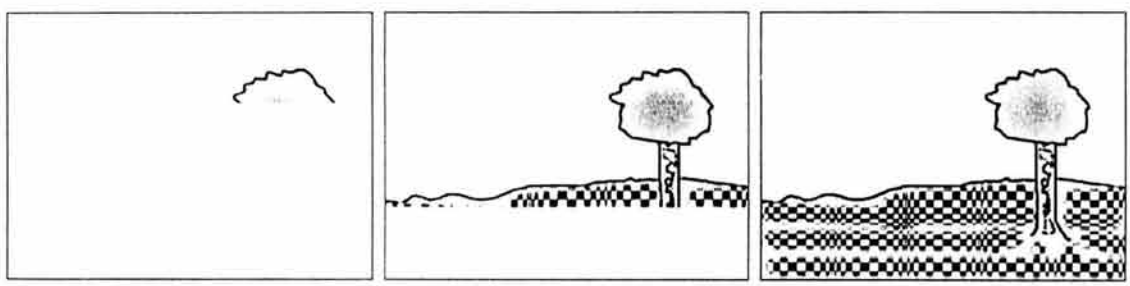


Fig. 3.9 JPEG secuencial

En las figuras 3.9 y 3.10 se observa cómo se despliegan las imágenes al momento de la descompresión para los casos secuencial y progresivo respectivamente. En el caso de la modalidad secuencial observamos que segmentos de la imagen con toda la definición aparecen consecutivamente hasta formar la imagen completa. En contraste, la modalidad progresiva presenta en su primera etapa a la imagen completa en baja definición y las siguientes etapas complementan su nitidez.

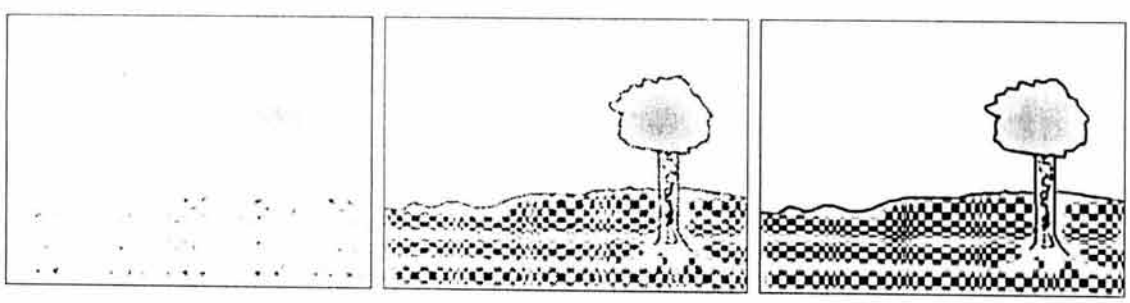


Fig. 3.10 JPEG progresivo

Modalidad jerárquica

La modalidad jerárquica descompone la imagen en un conjunto de subimágenes. Dichas subimágenes contienen un número menor de muestras que la imagen original, es decir, cuentan con menos píxeles. La información contenida en cada subimagen puede codificarse en varias lecturas, utilizando los métodos de la modalidad progresiva. A partir de una subimagen se genera otra que tenga mayor número de muestras, y así sucesivamente hasta llegar al número de muestras original. El efecto visual al momento de descomprimir una imagen en modo jerárquico es similar al modo progresivo.

Modalidad básica

Todas las modalidades mencionadas hasta ahora tienen la posibilidad de manejar 8 ó 12 bits para representar una componente de un espacio de color cualquiera. Sin embargo, JPEG estipula una modalidad básica que debe de poder manejar cualquier aplicación que implemente el estándar. La modalidad básica es un caso particular de la modalidad secuencial; utiliza sólo 8 bits por componente de espacio de color, de 1 a 4 componentes por espacio de color. Los datos de las componentes pueden presentarse intercalados o separados y sólo se podrá aplicar la codificación Huffman. La modalidad básica está limitada en cuanto las tablas de cuantización y de codificación Huffman que puede utilizar.

En la práctica, sólo las modalidades secuencial y progresiva del estándar JPEG se utilizan [5]. Pocos programas manejan 12 bits por componente de color en una imagen codificada con JPEG, la mayoría sólo puede codificar y decodificar imágenes de 8 bits por componente de color. Asimismo, el uso de la codificación aritmética dentro de JPEG es limitado; el algoritmo está patentado y la diferencia en porcentajes de compresión con la codificación Huffman no es suficiente como para considerar el pago de las cuotas correspondientes [7].

Este proyecto se centra en el papel de la Transformada Coseno en la compresión de imágenes, debido a ello se ha elegido implementar la modalidad básica de JPEG, aunado al hecho de que esta modalidad se encuentra presente en cualquier aplicación que maneje una imagen codificada siguiendo dicho estándar.

3.4.3 Componentes de los modelos de color en JPEG

Los datos de la imagen fuente pueden contener una o más componentes para construir un modelo de color. Las componentes están formadas por arreglos rectangulares de muestras de la imagen. Todas las muestras de todas las componentes en una imagen deben contar con la misma precisión numérica.

El estándar JPEG no especifica qué modelo de color se debe utilizar, sin embargo, para imágenes en color los algoritmos de compresión tienen un mejor desempeño al utilizar un sistema de coordenadas basado en luminiscencia/crominancia [1].

La componente que cuenta con más cambios en su frecuencia espacial es la de luminiscencia, las componentes de crominancia casi no presentan cambios de tono; consecuentemente, la compresión para las componentes de crominancia es grande. En contraste, en un espacio como el RGB los cambios de tono se reflejan en todas las componentes, por lo tanto, no se podrá compactar la información tanto como en el caso de un modelo de color basado en luminiscencia y crominancia. Debido a lo anterior, el modelo YCbCr es el más utilizado en las aplicaciones JPEG para imágenes en color.

3.4.4 Frecuencia de muestreo

Una imagen a color se representa con varias componentes; en el caso del modelo YCbCr se representa con tres. JPEG puede codificar una imagen cuyas componentes tengan diferente frecuencia de muestreo e inclusive aprovechar esta característica para aumentar la compresión.

La frecuencia de muestreo depende del número de píxeles con que cuente una componente, es decir, una imagen de 512×512 píxeles contará con una mayor frecuencia de muestreo que una componente con 480×480 píxeles de resolución. En el apartado anterior se mencionó que en una imagen de tono continuo las componentes Cb y Cr contienen cambios poco drásticos en sus frecuencias espaciales. Como consecuencia, se puede utilizar un número menor de píxeles para representar una componente de crominancia sin que la imagen sufra una distorsión significativa.

La reducción de muestras se puede aplicar horizontal, verticalmente o en ambos sentidos. Por ejemplo, una componente de crominancia puede corresponder a dos componentes de luminiscencia horizontalmente o verticalmente, y si el muestreo es la mitad en ambas dimensiones, corresponderá a cuatro píxeles. Esta técnica va mas allá del alcance de este proyecto, por lo que se optará por utilizar componentes que utilicen el mismo número de píxeles de resolución.

3.4.5 Codificación de datos intercalados y no intercalados

En la figura 3.7 se mostró el caso particular en que se lee por separado cada componente de un modelo de color en una imagen. Para que la información en una imagen de múltiples componentes pueda ser codificada en una sola lectura, los datos de cada componente del modelo de color deben estar intercalados.

Tomemos en cuenta la figura 3.11, en ella la imagen cuenta con tres componentes A, B, y C. Si se procesan primero todas las unidades de datos de la componente A, después las de B y al último las de C, su codificación será *no-intercalada*. Para el caso particular en que todas las componentes tengan la misma frecuencia de muestreo, se codificará una unidad de datos de la componente A, una unidad de B y una de C, para volver a procesar la siguiente unidad de A y así sucesivamente obteniendo una *codificación intercalada*.

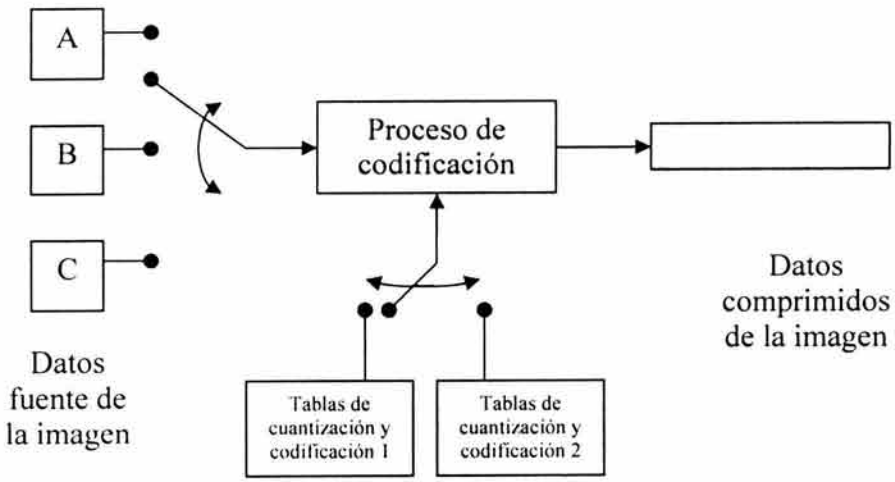


Fig. 3.11 Control de componentes intercaladas y cambio de tablas

Las tablas de cuantización y de codificación pueden ser diferentes para las componentes. Es posible intercalar componentes con distintas frecuencias de muestreo; no obstante, se decidió simplificar el problema de la codificación manejando componentes con la misma resolución, por ello no se describirá el procedimiento para intercalar estos datos ni las condiciones particulares que deben cumplir.

Si los datos en una imagen comprimida no están intercalados, la *unidad mínima codificada* (MCU) la conformará una unidad de datos. Por el contrario, si los datos están intercalados, la MCU (en nuestro caso en particular) la conformarán una unidad de datos de A, una de B y una de C.

No se pueden procesar MCU's incompletas debido a que sólo se utiliza la DCT de 8×8; por lo que si la resolución de la imagen no es un múltiplo de ocho algunas MCU's quedarán en esta situación. JPEG soluciona esta situación rellenando las MCU's; es decir, en todas las columnas faltantes se utilizarán los mismos datos de la última columna, asimismo, si existen filas sin datos se rellenarán con la información de la última fila.

3.4.6 Corrimiento de nivel

El estándar JPEG estipula que debe realizarse un corrimiento de nivel de los datos que componen una imagen antes de aplicar la DCT. Dicho corrimiento depende del nivel de precisión numérica con que cuenten las muestras. Para el caso de 8 bits por componente, las componentes se representan con formato entero no signado en el dominio de 0 a 255; el corrimiento será de 128, es decir, a cada muestra se le restará la constante 128 obteniendo una representación signada entera. Después de aplicar la IDCT se deberá aplicar el corrimiento inverso con el fin de recuperar el formato entero no

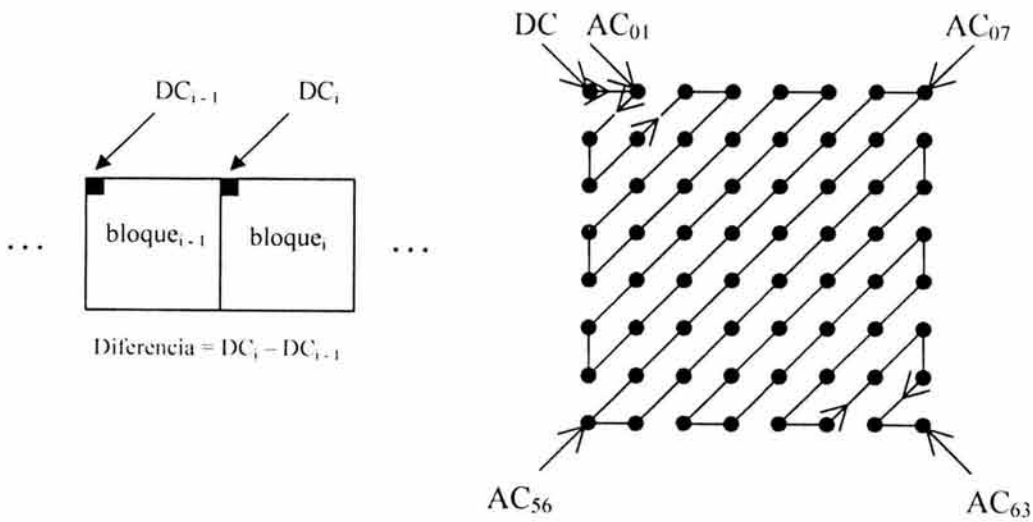
signado. El algoritmo DCT que se implementó en el proyecto incluye el corrimiento dentro de sus operaciones.

3.4.7 Codificación de los coeficientes DC y AC

Una vez transformada y cuantizada la información, ésta se preparará para codificarse entrópicamente. Como se ha mencionado, los coeficientes DC se codifican separados de los coeficientes AC. La razón para ello consiste en que un coeficiente DC representa el valor promedio de los 64 píxeles en una unidad de datos, y su nivel de correlación con el coeficiente DC del siguiente bloque de 8×8 será alta, mientras que la mayoría los coeficientes AC son cero.

A los coeficientes DC se les aplica la *modulación por codificación diferencial de pulsos* (DPCM). En otras palabras, en lugar de codificar cada coeficiente DC por separado, se codifican las diferencias entre dos coeficientes DC. En contraste con la información sin modular, la información en diferencias se aproxima al valor cero, aunado al hecho de que el intervalo en que se representa la información es menor. Si las ocurrencias de valores o eventos se encuentran en un intervalo pequeño se beneficiará la compresión [1].

Los 63 coeficientes restantes de cada unidad de datos se ordenan en un arreglo unidimensional mediante una secuencia zig-zag. Al agrupar los coeficientes en este arreglo, los coeficientes de más baja frecuencia tienden a presentarse al inicio del arreglo y los coeficientes de alta frecuencia al final. Usualmente los coeficientes de baja frecuencia son diferentes de cero y los de alta frecuencia son cero.



Codificación diferencial de DC Ordenamiento zig-zag

Fig. 3.12 Preparación de los coeficientes cuantizados para la codificación entrópica

La figura 3.12 ilustra la codificación DPCM y el ordenamiento zig-zag; la diferencia entre dos coeficientes DC se obtiene sustrayendo al coeficiente i-ésimo el coeficiente que

le precede, asimismo muestra la relación entre los índices zig-zag y las posiciones de los coeficientes DCT.

Al comenzar una lectura de los datos de una imagen, el valor que precede al primer coeficiente DC se inicializa con valor cero, por lo que la primera diferencia será el valor absoluto del primer coeficiente DC

Inmediatamente después de usar codificación DPCM en los coeficientes DC, estos se ordenan en un modelo estadístico Huffman que les permitirá ser codificados. Dicho modelo estadístico divide las diferencias DPCM en grupos. Las magnitudes de un grupo a otro aumentan en forma cuasilogarítmica. La tabla 3.3 contiene los diferentes grupos o categorías para el caso en que la precisión numérica de nuestra imagen es de 8 bits.

SSSS	**Diferencia DPCM**
0	0
1	-1,1
2	-3,-2,2,3
3	-7, ,-4,4, 7
4	-15, ,-8,8, 15
5	-31, -16,16, ,31
6	-63, ,-32,32, ,63
7	-127, ,-64,64, ,127
8	-255, ,-128,128, ,255
9	-511, ,-256,256, ,511
10	-1023, ,-512,512, ,1023
11	-2047, ,-1024,1024, ,2047

Tabla 3.3 Categorías *SSSS* para las diferencias entre coeficientes DC

Cada grupo de diferencias representa un símbolo al cual se le asignará un código Huffman. La categoría describe el valor exacto de la diferencia sólo en el caso en que la diferencia es cero, por ello necesitamos de *SSSS* bits adicionales para especificar adecuadamente la magnitud y el signo de la diferencia. Si la diferencia es positiva se debe concatenar a la categoría codificada un número binario con *SSSS* bits de longitud con el valor de la diferencia; si la diferencia es negativa, se usarán los *SSSS* bits en complemento a uno. En la tabla 3.4 se muestran algunos ejemplos de bits adicionales.

SSSS	**Diferencia DPCM**	**Bits adicionales (binario)**
0	0	-
1	-1,1	0,1
2	-3,-2,2,3	00,01,10,11
3	-7, ,-4,4, ,7	000, ,011,100, ,111
4	-15, ,-8,8, ,15	0000, ,0111,1000, ,1111

Tabla 3.4 Ejemplos de bits adicionales para especificar signo y magnitud

Dentro de la tabla 3.4 se observa que el bit más significativo en las diferencias positivas siempre es uno, mientras que en las diferencias negativas es cero.

Los coeficientes AC requieren de preparación extra. Dado que muchos de los coeficientes AC son iguales a cero, se les aplica un proceso de codificación de datos iguales consecutivos, es decir, se agrupan los coeficientes consecutivos con el valor cero.

Para los coeficientes con magnitud diferente de cero se genera un modelo estadístico similar al aplicado a las diferencias de la codificación DPCM; se agrupan los valores en conjuntos dependiendo del número de bits adicionales que se necesitarán para recuperar su signo y magnitud exacta. La tabla 3.5 contiene las categorías en que se agrupan los coeficientes AC diferentes de cero para imágenes con precisión de 8 bits por componente de espacio de color.

SSSS	**Coeficientes AC**
1	-1,1
2	-3,-2,2,3
3	-7, ,-4,4, ,7
4	-15, ,-8,8, ,15
5	-31, ,-16,16, ,31
6	-63, ,-32,32, ,63
7	-127, ,-64,64, ,127
8	-255, ,-128,128, ,255
9	-511, ,-256,256, ,511
10	-1023, ,-512,512, 1023

Tabla 3.5 Categorías *SSSS* para los coeficientes AC diferentes de cero

La información queda compuesta por dos símbolos en la siguiente forma:

símbolo - 1	*símbolo - 2*
(ceros consecutivos, número de bits adicionales)	(valor)

El primer símbolo provee dos datos, y es éste símbolo al que se le aplica la codificación Huffman. El primer dato nos entrega el número *RRRR* de ceros consecutivos que ocurren en la secuencia zig-zag hasta un coeficiente diferente de cero, el segundo dato contiene el número de bits *SSSS* adicionales que nos permitirán recuperar la magnitud y signo de dicho coeficiente.

Al *símbolo - 1* se le representa mediante 1 byte, los cuatro bits menos significativos corresponden a *SSSS*, y los cuatro más significativos *RRRR*. El segundo símbolo constituye la magnitud y signo del coeficiente AC diferente de cero y se representa mediante los bits adicionales.

El primer dato dentro del *símbolo - 1* puede representar de 0 a 15 ceros consecutivos antes de un coeficiente a codificar, sin embargo, el *símbolo – 1* puede representar hasta 16 ceros consecutivos si *RRRR* es igual a 15 y *SSSS* es 0; a este caso particular se le denomina ZRL (*zero run length*).

Otro caso importante de la información contenida en el *símbolo - 1* es la combinación de *RRRR* y *SSSS* igual a 0; dicho suceso nos indica que la información en el unidad de datos ha terminado y todos los coeficientes que restan en la secuencia zig-zag son iguales a cero; a este caso se le llama fin de bloque, o por sus siglas en inglés, EOB (*end of block*).

En la tabla 3.6 se muestra el arreglo 2-D del modelo estadístico Huffman para los coeficientes AC; el modelo contiene las combinaciones de ceros consecutivos y bits adicionales. Aparte de EOB y ZRL, las combinaciones cuando *SSSS* es igual a cero no son permitidas (N/A). Cada valor de la tabla se obtiene mediante la operación $(16 \times RRRR) + SSSS$.

	SSSS											
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
RRRR	**0**	EOB	01	02	03	04	05	06	07	08	09	OA
	1	N/A	11	12	13	14	15	16	17	18	19	1A
	2	N/A	21	22	23	24	25	26	27	28	29	2A
	3	N/A	31	32	33	34	35	36	37	38	39	3A
	4	N/A	41	42	43	44	45	46	47	48	49	4A
	5	N/A	51	52	53	54	55	56	57	58	59	5A
	6	N/A	61	62	63	64	65	66	67	68	69	6A
	7	N/A	71	72	73	74	75	76	77	78	79	7A
	8	N/A	81	82	83	84	85	86	87	88	89	8A
	9	N/A	91	92	93	94	95	96	97	98	99	9A
	10	N/A	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA
	11	N/A	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA
	12	N/A	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA
	13	N/A	D1	D2	D3	D4	D5	D6	D7	D8	D9	D
	14	N/A	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA
	15	ZRL	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA

Tabla 3.6 Arreglo 2-D del modelo estadístico Huffman para los coeficientes AC

A continuación se ejemplifican la preparación de los coeficientes AC y DC. Se tienen dos unidades de datos en las tablas 3.7 y 3.8.

11	4	0	0	0	0	0	0
-7	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	9	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	0	0	0	0	-1	1

Tabla 3.7 Ejemplo de unidad de datos con coeficientes DCT cuantizados

11	-2	0	0	0	0	0	0
3	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	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabla 3.8 Ejemplo de unidad de datos con coeficientes DCT cuantizados

Consideremos que la tabla 3.7 es la primera MCU y le sigue la tabla 3.8. Primero codificamos los coeficientes DC; como se mencionó previamente, el valor anterior del coeficiente DC se inicializa como cero. Se obtiene la diferencia, se clasifica en una categoría y se generan los bits adicionales que especificarán completamente a la cantidad. La tabla 3.9 contiene la información de los pasos descritos.

Coeficientes DC				
Coeficiente DC anterior	Coeficiente DC actual	Diferencia	# de bits adicionales	Bits adicionales
0	11	11	4	1011

Tabla 3.9 Obtención de diferencia DC y asignación de bits adicionales para la tabla 3.7

Posteriormente se procesan los coeficientes AC de la tabla 3.7. El siguiente coeficiente distinto de cero (al seguir el orden zig-zag) es el número cuatro, se necesitaron 0 ceros consecutivos para llegar a él y pertenece a la categoría que se puede representar con 3 bits adicionales. Se continúa con el siguiente coeficiente AC de la misma MCU hasta el fin de la unidad. En la tabla 3.10 se observa el resultado de la preparación de los coeficientes AC de la primera unidad de datos correspondientes a los coeficientes de la tabla 3.7.

Coeficientes AC				
Ceros consecutivos	Valor del coeficiente	# de bits adicionales	*RRRRSSSS* (hexadecimal)	Bits adicionales (binario)
0	4	3	03	100
0	-7	3	03	000
8	9	4	84	1001
15	0	0	F0 (ZRL)	-
15	0	0	F0 (ZRL)	-
15	0	0	F0 (ZRL)	-
2	-1	1	21	0
0	1	1	01	1

Tabla 3.10 Obtención de *RRRR*, *SSSS* y asignación de bits adicionales para la tabla 3.7

Los resultados de aplicar el modelo estadístico a la segunda unidad de datos (tabla 3.8) aparecen en las tablas 3.11 y 3.12.

Coeficientes DC				
Coeficiente DC anterior	Coeficiente DC actual	Diferencia	# de bits adicionales	Bits adicionales
11	11	0	0	-

Tabla 3.11 Obtención de diferencia DC y asignación de bits adicionales para la tabla 3.8

Coeficientes AC				
Ceros consecutivos	Valor del coeficiente	# de bits adicionales	*RRRRSSSS* (hexadecimal)	Bits adicionales (binario)
0	-2	2	02	01
0	3	2	02	11
Fin de bloque	-	0	00 (EOB)	-

Tabla 3.12 Obtención de *RRRR*, *SSSS* y asignación de bits adicionales para la tabla 3.8

Marcadores de reinicio

JPEG prevé y soluciona la corrupción de los datos dentro de un archivo añadiendo dentro del formato marcadores de reinicio **RST**. Dichos marcadores se colocan en intervalos arbitrarios y deben de seguir un orden del marcador 0 al marcador 7 (RST_0-RST_7); después de llegar al marcador 7 se empezará de nuevo con el marcador cero y así sucesivamente. Si uno de los marcadores no está presente en la secuencia, la información de la imagen está corrupta. Si se desea recuperar el resto de la información, se debe buscar el siguiente marcador y, dependiendo del diseño del codificador, ignorar la información dañada, ya sea asignándole un color (negro, por ejemplo) o cualquier otro artificio que permita continuar desplegando la información de los bloques intactos en su lugar correspondiente. Si la falla se extiende más allá de 8 marcadores toda la información será irrecuperable. Inmediatamente después de cada marcador, el valor del coeficiente DC anterior se reiniciará de nuevo a cero.

Una vez obtenidos los símbolos para los coeficientes DC y AC, se pueden utilizar tablas de Huffman prefabricadas o se pueden generar tablas Huffman específicas para su codificación entrópica.

Evidentemente, el uso de tablas generadas con base en las estadísticas de la imagen resultará en una mejor compresión; sin embargo, se requiere de dos lecturas a toda la información. En la primera lectura se obtendrán los datos estadísticos y en la segunda se codificará.

El alcance de este proyecto no abarca la generación y optimización de la codificación Huffman, por lo que sólo se incluirán unas tablas predeterminadas que aparecen como ejemplo en los documentos del estándar JPEG [1]. Al igual que las tablas de cuantización, se manejan dos tablas Huffman, una para la componente de luminiscencia y la otra para las componentes de crominancia. Las tablas a utilizar en este proyecto se encuentran en el Anexo A.

Consideremos el caso de la siguiente unidad de datos de coeficientes DCT cuantizados de la tabla 3.13:

11	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-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	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Tabla 3.13 Coeficientes DCT cuantizados

Suponiendo que el valor anterior del coeficiente DC fuese 15, la diferencia entre el coeficiente de DC anterior y en actual sería de -4 (*coeficiente DC actual – coeficiente DC anterior*). Tendremos los siguientes símbolos intermedios para representar la información:

Para la diferencia DC = -4, *SSSS* = 3.

Siguiendo el orden zig-zag, los coeficientes AC diferentes de cero proveen las siguientes combinaciones de *RRRR* y *SSSS*:

Coeficiente AC	*RRRRSSSS*
-2	12
-1	01
-1	01
-1	01
-1	21
EOB	00

Se asignará el código Huffman correspondiente utilizando las tablas para las diferencias DC en componentes de luminiscencia suministradas como ejemplo en la especificación JPEG y se concatenarán los bits adicionales:

Diferencia	*SSSS*	Código Huffman	Bits adicionales
-4	3	100	011

Los códigos para la luminiscencia en coeficientes AC y los bits adicionales en este ejemplo son:

Coeficientes AC	*RRRRSSSS*	Código Huffman	Bits adicionales
EOB	00	1010	-
-1	01	00	0
-2	12	11011	01
-1	21	11100	0

Concatenamos primero, los bits adicionales para la diferencia DC al código para dicha diferencia:

100011

posteriormente se concatena el código para la combinación *RRRR* y *SSSS* obtenido a partir del primer coeficiente AC diferente de cero y los ceros que le preceden, así como los bits adicionales:

1000111101101

y así sucesivamente concatenamos los restantes códigos AC más sus bits adicionales hasta concatenar finalmente el código de EOB:

10001111011010000000001110001010

Por lo tanto, con los 32 bits anteriores representamos el conjunto de 64 coeficientes del ejemplo.

3.4.8 Marcadores o banderas en JPEG

Todo formato de imágenes necesita de marcadores o banderas para determinar el inicio y el final de la información. Los marcadores necesarios para compactar una imagen mediante el estándar JPEG en su modalidad base son [5]:

- La información debe comenzar con un marcador de *inicio de imagen* **SOI** (start of image) y debe terminar con un marcador de *fin de imagen* **EOI** (end of image).
- Dentro de la información debe haber un marcador **SOF** (start of frame) que indique el *inicio de una serie de lecturas de la información*, y debe estar antes de que aparezca un marcador de *inicio de lectura a la información* **SOS** (start of scan).
- Todas las tablas Huffman y de cuantización que se utilicen en una lectura deben definirse previamente a la aparición del marcador SOS de dicha lectura. El encabezado de una tabla o tablas Huffman consta de un marcador de *definición de tabla Huffman* **DHT** (define Huffman table); al encabezado de las tablas de cuantización se le denomina marcador para *definición de tabla de cuantización* **DQT** (define quantization table)

3.5 Resumen

Al inicio del capítulo se eligió implementar la compresión de imágenes siguiendo ciertas recomendaciones encontradas dentro del estándar de compresión de imágenes JPEG. El fundamento de JPEG es la Transformada Coseno DCT, la cual es una transformación que nos permite decorrelacionar un conjunto de datos. La transformada KLT realiza la decorrelación de datos de una señal, sin embargo, la complejidad que implica implementarla la margina de algunas aplicaciones. La DCT descompone un

conjunto de muestras en otro conjunto de coeficientes que representan la amplitud de funciones base; la sumatoria de dichas funciones base escaladas por los coeficientes reconstruyen nuestro conjunto original de muestras. Los coeficientes que entrega la DCT son números reales, lo cual implica el uso de precisión numérica infinita para representar los resultados. Debido a que no existe un sistema que pueda almacenar números que no sean racionales de precisión numérica infinita, es un proceso que acarrea pérdida de información. Mientras una imagen sea de tono continuo, podremos descartar ciertos coeficientes DCT sin percibir la diferencia entre la imagen original y la reconstruida. Los coeficientes se descartan mediante la cuantización. El estándar JPEG cuenta con tres pasos básicos para compactar una imagen: la Transformada Coseno, la cuantización y la codificación entrópica. Se eligió la modalidad base para desarrollarse en este proyecto; dicha modalidad maneja 8 bits por cada componente de color. La codificación entrópica codifica a los coeficientes DC y AC por separado. Para la codificación de todos los coeficientes se emplea una combinación de *códigos de longitud variable* y de *enteros de longitud variable* que complementan la información del código.

Capítulo 4
Implementación de compresión de imágenes en un DSP

En este capítulo se describirá la arquitectura elegida para procesar las imágenes, el porqué de su elección y su comunicación con la fuente y destino de la información. Posteriormente se describen todos los procesos efectuados tanto en la compresión como en la descompresión. Dentro de los procesos se incluyen dos implementaciones de la Transformada Coseno: la definición unidimensional y una optimización de dicha transformada.

4.1 Elección de un procesador para la ejecución de los algoritmos

En las figuras 3.7 y 3.8 observamos los procedimientos que son necesarios para la compresión de imágenes, entre éstos están: la Transformada Coseno de una señal, la cual requiere de un gran número de operaciones aritméticas y lógicas (1,024 multiplicaciones y 896 sumas para un bloque de 8×8 muestras utilizando la definición 1-D de la DCT), y las operaciones de los procesos de cuantización y codificación, por lo que es necesario elegir una arquitectura que pueda realizar toda esta carga de trabajo y que entregue los resultados en un tiempo razonable. Dentro del gran conjunto de procesadores que existen hoy en día resaltan por su capacidad de manejar imágenes dos grupos: los *procesadores digitales de señales* (**DSP**) y los *procesadores de conjunto reducido de instrucciones* (**RISC**: reduced instruction set computer).

Los procesadores RISC se encuentran usualmente en estaciones de trabajo dedicadas al procesamiento de video, sin embargo, su uso se extiende a otras aplicaciones debido a su bajo costo. Dichos procesadores manejan un lenguaje de programación cuyo conjunto de instrucciones es relativamente pequeño y consiste en instrucciones simples. El que las instrucciones del procesador no sean complicadas le permite al procesador ejecutarlas más rápido [9].

El diseño de los DSP está enfocado a dos objetivos: al procesamiento de señales en tiempo real y a que el usuario lo pueda adquirir a bajo costo. Debido a lo anterior es que los DSP's se pueden encontrar en todo tipo de aplicaciones portátiles que impliquen conversión de señales analógicas a digitales como teléfonos celulares, fax, módems, control, etc. Los DSP's orientan sus instrucciones y hardware a la operación *multiplicación y acumulación* (MAC), la cual optimiza el cálculo de la FFT y en nuestro caso el de la DCT.

Tomemos como ejemplo de nuevo la ecuación 3.5 para la Transformada Coseno en 1-D:

$$S(u) = \frac{C(u)}{2} \sum_{x=0}^{7} s(x) \cos\left[(2x+1)u\pi/16\right]$$

Notamos en la ecuación que cada valor de la función base correspondiente dada por $\cos\left[(2x+1)u\pi/16\right]$ debe *multiplicarse* por el valor *s(x)* correspondiente y *acumularse* a un total. Como mencionamos anteriormente, el número de las multiplicaciones y acumulaciones necesarias para obtener el resultado es elevado.

Debido a que las instrucciones y el hardware de los DSP's están dedicados a determinadas aplicaciones; entonces, en dichas aplicaciones los DSP's mejoran el desempeño y eficiencia de procesadores que tienen un reloj interno más rápido.

Tanto los procesadores RISC como los DSP's utilizan intensivamente la técnica de procesamiento en paralelo *pipeline* para optimizar su desempeño; dicho de otra manera, el procesador reduce el tiempo en que se llevan a cabo los algoritmos optimizando el orden y los tiempos en que se ejecutan las instrucciones. En nuestro caso particular no es necesario que los algoritmos se realicen en tiempo real, sin embargo, implementar nuestra aplicación en una arquitectura DSP o RISC nos da la flexibilidad de incluirla en procesos más complejos.

Pensando en un futuro, la elección de un DSP nos permitirá extender más fácilmente el proyecto a la adquisición de imágenes. Sin embargo, la elección no es simplemente económica; es necesario dar uso a dichos equipos y dejar sentados elementos de investigación y aplicación para generaciones posteriores.

Ahora bien, entre las arquitecturas a elegir se encuentran las de punto fijo y de punto flotante. Las arquitecturas de punto fijo son más eficientes debido a que su uso de energía es menor que los de punto flotante, no obstante, la utilización de una arquitectura de punto flotante facilita enormemente la implementación de algoritmos como los de la Transformada Coseno y la cuantización.

Otro punto esencial es la cantidad de memoria utilizada para el procesamiento de imágenes. Si utilizamos un registro de 1 byte para almacenar la información de un píxel de cada componente de color, en una imagen de 256×256 píxeles serán necesarios 65536 bytes (64Kbytes); entonces para tres componentes se requieren 3×65536 bytes. Además, es preciso considerar memoria adicional para los resultados de compresión.

Debido a estas necesidades de memoria y pontecialidad de cálculo, la arquitectura que se seleccionó por su alto desempeño fue la del DSP **TMS320C6711**.

4.2 Descripción de la tarjeta DSK del TMS320C6711

El procesador TMS320C6711 (**C6711**) es un DSP's que forma parte de la familia TMS320C6000 (**C6x**) de Texas Instruments (**TI**). La familia C6x está diseñada para realizar cálculos numéricos intensivos y se basa en una arquitectura del tipo **VLIW** (very long instruction word); es decir, cada instrucción cuenta dentro de sí con varias

operaciones simples que no son interdependientes. El procesador divide el trabajo dirigiendo cada operación a diferentes *unidades de ejecución.*

El DSP C6711 es una versión económica del DSP de punto flotante C6701. Dentro de la arquitectura del C6711 encontramos ocho unidades de ejecución: seis unidades lógicas aritméticas (ALUs) y dos unidades multiplicadoras. Dichas unidades operan en paralelo y pueden realizar hasta seis instrucciones de punto flotante durante un ciclo de reloj [10].

La extensión de la memoria interna del C6711 es de 72kBytes y se encuentra separada en memoria de "alto" nivel y de "bajo" nivel. En el nivel alto se dedican 4kBytes a la memoria programa y 4kBytes para la memoria dato. Los 8kBytes tienen acceso directo al núcleo del procesador. Los 64kBytes restantes guardan ambos tipos de memoria, programa y dato, para separarse posteriormente y pasar al nivel alto. Los bancos de memoria independientes en los C6x permiten dos accesos a memoria dentro de un ciclo de instrucción mediante buses independientes; dicho de otra manera se pueden realizar dos instrucciones de lectura o dos instrucciones de escritura en paralelo (e inclusive una de lectura y una de escritura). Asimismo, los DSP's C6x pueden realizar lecturas, escrituras y realizar varias ejecuciones dentro de un mismo ciclo, gracias a que cuenta con buses separados para datos, programa y acceso directo a la memoria (**DMA**).

El procesador a utilizar se encuentra en una tarjeta llamada *DSP Starter Kit* (**DSK**), la cual contiene 16 megabytes de memoria externa SDRAM, 128kB de memoria ROM flash, un codec para conversión A/D y D/A y los periféricos necesarios para comunicación. Un diagrama del DSK se observa en la figura 4.1.

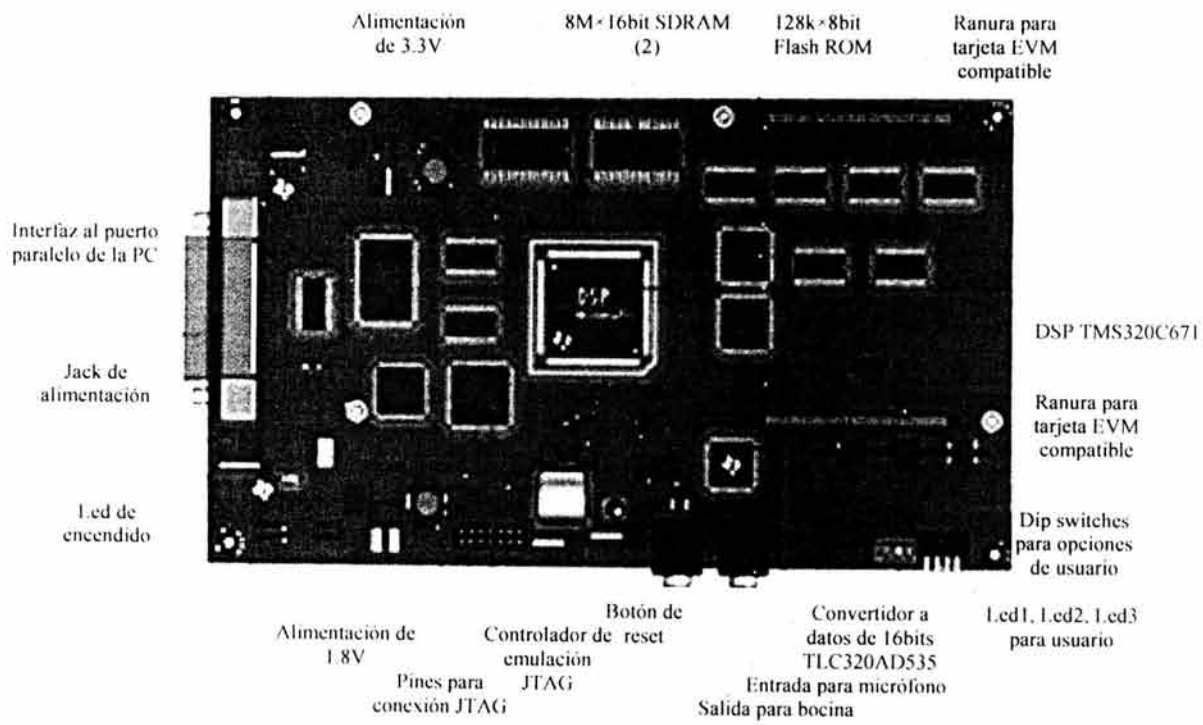


Figura 4.1 Diagrama de la tarjeta DSK del TMS320C6711*

* Imagen tomada de la página *http://focus.ti.com/graphics/tool/C6711brd1.gif* de Texas Instruments.

Dentro de los periféricos se incluyen dos puertos seriales multicanal con capacidad de almacenar temporalmente muestras en la memoria (**McBSPs**), una interfaz de 16 bits para la comunicación entre la tarjeta y la PC (*Host Port Interface*, **HPI**) y una interfaz de 32 bits para memoria externa (**EMIF**).

La herramienta que provee Texas Instruments para su programación es el *Code Composer Studio* (CCS), que es un ambiente integrado de desarrollo con compilador en lenguaje *C* y ensamblador. El DSK cuenta con un reloj de 150MHz; con él, idealmente se pueden realizar dos multiplicaciones y acumulaciones por ciclo, es decir, un total de 300 millones de operaciones MAC por segundo [11].

4.2.1 Paquetes de ejecución y *pipeline*

El C6711 se basa en la arquitectura **VELOCITI** de TI; en ella siempre se le proporcionan ocho instrucciones al núcleo del procesador en cada *búsqueda y recuperación* (**fetch**) de instrucciones. A este conjunto de instrucciones se le denomina *fetch packet* (**FP**). Dichas instrucciones pueden ser o no ser ejecutadas en paralelo; el conjunto de aquellas que se ejecuten en paralelo constituyen un paquete de ejecución *execute packet* (**EP**). Un EP puede contener hasta ocho instrucciones y cada instrucción dentro de un EP debe utilizar una unidad de ejecución diferente [12].

Para que un programa sea eficiente se necesita que no existan tiempos entre una y otra ejecución del procesador; es decir, que al momento en que el resultado de una unidad de ejecución se haga disponible para manipulación, el proceso siguiente comience a utilizarlo inmediatamente. A este proceso de hacer eficiente el manejo de datos se le llama *pipeline*.

Para poder utilizar el pipeline en el DSP es necesario tomar en cuenta que no todas las instrucciones se realizan en un solo ciclo. Mientras una instrucción se encuentre en *estado latente* dentro de una unidad de ejecución no se podrá empezar a ejecutar otra instrucción en dicha unidad.

Es importante resaltar que los ciclos en que una instrucción se mantiene en estado latente no son los mismos que los que son necesarios para completar la instrucción. A los ciclos posteriores al inicial, que utiliza una instrucción para terminar su ejecución, se les denomina *delay slots*. Los delay slots son iguales o mayores a los ciclos en que una instrucción se encuentra latente en una unidad de ejecución.

El pipeline puede aprovecharlo el programador asignando las unidades de ejecución para las distintas operaciones o bien utilizando las herramientas de optimización incluidas dentro del CCS. El pipeline disminuye el tiempo en que las operaciones se ejecutan y consecuentemente nuestro algoritmo se hace más eficiente. La ventaja del pipeline se refleja en gran medida en las operaciones MAC.

4.2.2 Lenguaje ensamblador y *C*

El DSP C6711 puede ser programado de diferentes maneras, aunque su lenguaje nativo es el *lenguaje ensamblador*. Con el lenguaje ensamblador se manejan directamente los registros a utilizar para el procesamiento de datos y, asimismo, se puede especificar en qué unidades de ejecución se realicen las operaciones; en resumen, se tiene más control del DSP. Sin embargo, a veces resulta demasiado complicado o extenso crear un proceso en lenguaje ensamblador en comparación a introducir una instrucción en lenguaje *C*.

El CCS permite la programación en *lenguaje C* y en *C++*, dado que el compilador de *C* traduce el código fuente a lenguaje ensamblador. La desventaja radica en que algunas funciones en *C* toman una gran cantidad de ciclos en ejecutarse. Existe un tercer lenguaje con el que se puede programar el C6711, al que se le denomina *lenguaje ensamblador lineal*, y que se puede describir como una mezcla entre en el lenguaje ensamblador y el lenguaje *C*. Al igual que en el lenguaje *C*, el CCS utiliza el código en lenguaje ensamblador lineal para generar un programa en lenguaje ensamblador.

Al programar en lenguaje *C* se tiene la ventaja de que la función principal puede llamar y utilizar funciones que se encuentren escritas en lenguaje ensamblador y en ensamblador lineal, aparte de las funciones que estén escritas en *C*. Esto permite que los procesos que abarcan operaciones matemáticas intensivas se puedan optimizar fácilmente. El CCS también admite que se encuentren líneas en lenguaje ensamblador (delimitadas) dentro de un programa en *C*, sin embargo no es recomendable debido a que en la optimización del código fuente mediante las herramientas del CCS se pueden generar errores.

Una vez que se cuenta con un archivo en lenguaje ensamblador, el CCS genera un *archivo objeto*, lo une con las librerías que se necesiten incluir para que el programa funcione y finalmente entrega un archivo de código ejecutable (en este caso ejecutable en el DSP). Adicionalmente, antes de generar el archivo objeto, CCS puede optimizar el código en ensamblador. En la tabla 4.1 se observa una descripción general de los cuatro niveles de optimización disponibles.

Opción	Descripción
-o0	Optimiza el uso de los registros
-o1	Utiliza las optimizaciones de –o0 y optimiza localmente
-o2	Utiliza las optimizaciones de –o1 y optimiza globalmente
-o3	Utiliza las optimizaciones de –o2 y optimiza el archivo

Tabla 4.1 Nivel de optimizaciones del CCS

Independientemente de que se elija o no alguna de las opciones anteriores, el CCS siempre realizará algunas labores de optimización al código fuente adicionales a las mencionadas [13].

Para este proyecto se utilizará el lenguaje *C*, tomando en cuenta la posibilidad de optimizar algunos procesos posteriormente con lenguaje ensamblador o lenguaje ensamblador lineal.

4.2.3 Interfaz entre PC y DSP

La comunicación entre el C6711 es posible gracias al puerto huésped HPI en el DSK, el puerto paralelo en la PC y un protocolo del CCS.

Debido a que el interés de este proyecto no reside en la eficacia de los algoritmos en tiempo real, no es necesaria una comunicación entre el DSK y la PC en tiempo real, por lo tanto, no se eligió desarrollar alguna aplicación de comunicación entre el DSP y la PC. Nuestra aplicación está enfocada a ser un bloque fundamental, ya sea que necesite o no de una PC para el proceso de adquisición o carga de las imágenes.

La preparación de los datos para su procesamiento en el DSP y para su despliegue en la PC se realizará mediante el uso de *Matlab*. Dichos procedimientos los podemos enumerar de la manera siguiente:

- Se lee la información contenida en un archivo de mapa de bits
- Se hace la conversión entre el espacio RGB al YC_bC_r para aquellas imágenes en donde sea necesario
- Con Matlab se maneja la información de la imagen como un arreglo bidimensional y mediante programación se genera un archivo con las características adecuadas para que el ambiente CCS pueda acceder y trabajar con él.
- Una vez procesada la imagen, se realizan los procesos inversos a los tres procedimientos mencionados anteriormente.

4.3 Descripción del algoritmo de compresión implementado en el DSP

4.3.1 Almacenamiento de la imagen en el DSP

En el primer bloque de la figura 3.7 se observa que a la entrada del proceso de codificación se tiene una imagen dividida en píxeles; lo cual equivaldría al momento en que se cargan a la memoria los datos de la imagen.

Tomando en cuenta que el ambiente CCS puede leer información que se encuentre en archivos compatibles con lenguaje C, se incluye la información de la imagen como un archivo con extensión *.h* dentro del proyecto en el CCS. Dicho archivo contiene un arreglo formado por todos los píxeles de la imagen. Debido a que la información original proviene de un archivo bitmap, cuyos valores en el espacio de color varían de 0 a 255, se definió el arreglo como tipo char; el cuál en el C6711 consta de 8 bits:

char original[n] = {dato 0, dato 1, , dato n-1};

Ahora bien, previamente a la carga de la información, la imagen original (a color en espacio RGB) se transformó mediante Matlab a una imagen de tono de grises con la finalidad de que sólo fuese necesario una componente del espacio de color (ver figura 2.2) para representar una imagen. Al mismo tiempo, esta componente única resulta ser también la componente de luminiscencia en el espacio de color YCbCr.

El ambiente CCS permite asignar el lugar en el mapa de memoria donde se almacenarán datos mediante la especificación de la localidad en un archivo de comando *cmd*, y posteriormente mediante el comando escrito en el archivo *main* (archivo principal en *C*):

#pragma DATA_SECTION (variable o arreglo, nombre o referencia del lugar en el mapa de memoria)

asegura el lugar para almacenar los datos e impide que ocupen *memoria programa* y la saturen, con la consecuencia de que el programa no se compile.

Mientras el programa no sea muy largo y no se utilicen muchas variables el compilador puede utilizar un modelo de memoria donde las variables y las funciones se asignen localmente; en caso contrario, es necesario utilizar un modelo de llamadas a variables y funciones lejanas; es decir, que no se almacenen en memoria caché. La opción para elegir entre estos modelos de compilación se puede encontrar en las opciones de compilación del ambiente CCS.

Se eligió ejecutar el algoritmo para compresión en imágenes de 256×256 píxeles, evitando así el rellenado de bits (ver apartado 3.4.5). Mientras el número de píxeles horizontales y verticales sea múltiplo de 8, no se necesitará agregar un algoritmo adicional para rellenado de bits, tan sólo modificar el número de veces que se realiza el algoritmo de compresión.

4.3.2 Lectura de datos de la imagen original

Antes de que la información de la imagen pase al siguiente bloque del algoritmo de compresión de la figura 3.7, que consiste en la Transformada Coseno, el estándar JPEG recomienda que se introduzca un *offset* a la información sustrayéndole el valor 128 a cada uno de los píxeles de la imagen. Dicha sustracción se realizó conforme los bloques de unidades de datos se iban procesando.

La selección y extracción de cada bloque de 8×8 píxeles para su procesamiento se logró mediante un conjunto de ciclos anidados conteniendo una instrucción de direccionamiento en su núcleo. Dentro de la instrucción de direccionamiento se incluyó la sustracción del valor 128 para cada píxel. Este bloque de código se muestra a continuación:

```
/* ciclos anidados para lectura de las unidades de datos */

    for(l=0;l<32;l++)
/* Un incremento en l equivale a un desplazamiento de 8 pixeles hacia abajo es decir, nos remite
a una unidad de datos abajo */
    {
        for(k=0;k<32;k++)
/* Un incremento en k equivale a un desplazamiento de 8 pixeles hacia la derecha; es decir, nos
remite a una unidad de datos a la derecha */
        {
                for(j=0, m=0;j<8;j++)
/* Un incremento en j nos remite a la segunda línea de la unidad de datos en turno */
                {
                        for(i=0;i<8;i++)
/* Un incremento en i equivale a un desplazamiento de un pixel hacia la derecha en la unidad de
datos en turno */
                        {
                        bloque[m]=((short) original[(l*2048)+(8*k)+(256*j)+i])-128;
/* La instrucción de índices de MCU's incluye la sustracción del valor 128 */
                        m++;
/* Se incrementa el índice de la unidad de datos temporal bloque[m] */
                        }
                }
    /* Espacio para implementar la compresión */
        }
    }
```

donde *bloque* es un arreglo unidimensional de 64 elementos y almacena la unidad de datos a procesar. Cada elemento de *bloque* consta de 8 bits y almacena un número entero.

La rutina anterior suministra las unidades de datos en un orden de derecha a izquierda y de arriba hacia abajo para un total de 1024 unidades de datos, donde cada unidad de datos es un segmento de 8×8 píxeles para un total de 65536 píxeles. En la figura 4.2 se puede ver una representación del orden de acceso a cada unidad de datos de la imagen.

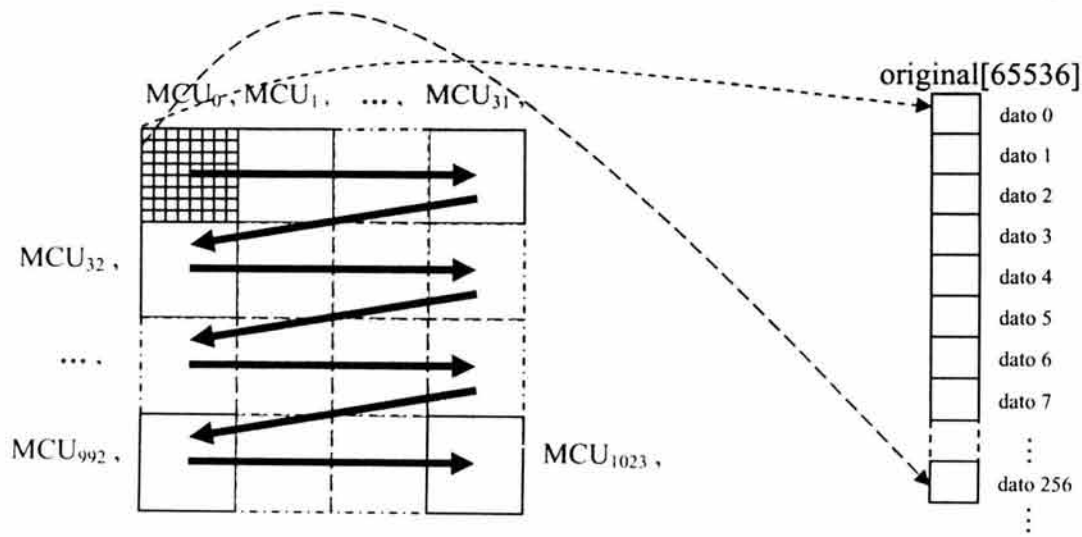


Figura 4.2 Orden de acceso a cada unidad de datos de la imagen

La lectura de datos en memoria consiste en leer 8 bits consecutivos de un mismo renglón, y posteriormente saltar al siguiente renglón para repetir el procedimiento con otros 8 bits y así hasta juntar 64 bits. Como ejemplo se puede ver en la tabla 4.2 la lectura de la primera unidad de datos en una imagen de 256×256 píxeles mostrada en la figura 4.2:

dato 0	*dato 1*	*dato 2*	*dato 3*	*dato 4*	*dato 5*	*dato 6*	*dato 7*
dato 256	*dato 257*	*dato 258*	*dato 259*	*dato 260*	*dato 261*	*dato 262*	*dato 263*
dato 512	*dato 513*	*dato 514*	*dato 515*	*dato 516*	*dato 517*	*dato 518*	*dato 519*
dato 768	*dato 769*	*dato 770*	*dato 771*	*dato 772*	*dato 773*	*dato 774*	*dato 775*
dato 1024	*dato 1025*	*Dato 1026*	*dato 1027*	*dato 1028*	*dato 1029*	*dato 1030*	*Dato 1031*
dato 1280	*dato 1281*	*Dato 1282*	*dato 1283*	*dato 1284*	*dato 1285*	*dato 1286*	*Dato 1287*
dato 1536	*dato 1537*	*Dato 1538*	*dato 1539*	*dato 1540*	*dato 1541*	*dato 1542*	*Dato 1543*
dato 1792	*dato 1793*	*Dato 1794*	*dato 1795*	*dato 1796*	*dato 1797*	*dato 1798*	*Dato 1799*

Tabla 4.2 Elementos de la primera unidad de datos en una imagen de 256×256 píxeles

Dentro de la tabla 4.2 encontramos que un salto de renglón equivale a un incremento de 256 en el índice de la unidad de datos, así como un salto a la siguiente unidad de datos implica un incremento en 8 al mismo índice.

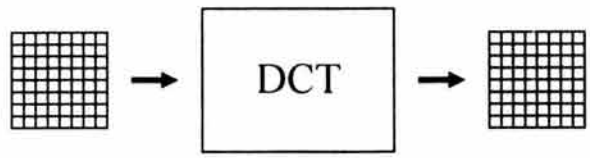
En el código fuente del algoritmo se puede ver que los incrementos de 256 píxeles en la posición de lectura están dados por *j*, los de un píxel por *i*. Para procesar la siguiente MCU de la derecha se incrementa el valor de *k* en 8, y para procesar la siguiente línea de MCU's se incrementa la posición de lectura por 2048 píxeles al incrementarse el valor de *l*.

Dado que esta rutina funciona a manera de índice para los bloques a procesar, dentro de la misma se puede ejecutar la rutina de compresión, es decir, de esta manera reduce el espacio de memoria utilizado.

4.3.3 Aplicación de la Transformada Coseno a las unidades de datos

Se eligió realizar dos Transformadas Coseno unidimensionales a cada unidad de datos debido a la equivalencia de este proceso a una Transformada Coseno en 2-D. La operación unidimensional se aplicó primero a renglones y después a columnas, aunque para efectos prácticos es indistinto el orden en que se aplique la DCT [13].

Es importante remarcar que a pesar de recibir un arreglo de números enteros, la Transformada Coseno nos entrega un arreglo de números reales, por lo que es necesario almacenar los coeficientes resultantes en otro arreglo diferente al *bloque[64]*. Se puede ver una representación del proceso en la figura 4.3:



bloque
Arreglo de
números enteros

bloque_t
Arreglo de
números reales

Figura 4.3 Representación del proceso de la DCT

En este caso se utilizó un arreglo unidimensional de números de punto flotante *bloque_t*, que consta también de 64 elementos, para recibir los coeficientes resultantes.

En las ecuación 3.5 se puede observar que los valores de las funciones base a multiplicar por las muestras son siempre los mismos para cualquier conjunto de muestras. Se ahorró tiempo de procesamiento almacenando en un arreglo bidimensional *basisf* de 8×8 números reales los valores de las funciones base multiplicadas por los coeficientes $\frac{C(u)}{2}$ correspondientes. Lo anterior se puede expresar mediante la forma expresada en la ecuación 4.1:

$$S(u) = \sum_{x=0}^{7} s(x) \underbrace{\left[\frac{C(u)}{2} \cos\left[(2x+1)u\pi/16\right] \right]}_{\substack{elemento \\ basisf}} \qquad (4.1)$$

En la ecuación 4.1 se encuentran, dentro de una llave con nombre *basisf*, los elementos previamente calculados.

Para obtener la DCT en una dimensión, se puede suponer que se cuenta con 8 conjuntos de muestras, cada conjunto a su vez con 8 muestras. El arreglo *bloque* es unidimensional, pero puede considerarse que del elemento 0 al 7 es un conjunto de 8 muestras, del elemento 8 al 15 es otro conjunto y así sucesivamente.

Asimismo, para una DCT unidimensional, se puede considerar que se tienen 8 funciones base, y cada función base cuenta con 8 valores distintos como se mostró en la figura 3.1.

Para obtener el primer coeficiente DCT unidimensional de un conjunto de 8 muestras, debe multiplicarse cada muestra por el valor que le corresponda de la primera función base y hacerse la sumatoria de los resultados.

El segundo coeficiente se obtiene multiplicando, de nuevo cada muestra por el valor que le corresponda de la segunda función base y haciéndose la sumatoria de los resultados.

Para esclarecer el procedimiento, supongamos que queremos obtener la DCT del primer renglón de la unidad de datos.

Dada la ecuación 4.1, sabemos que los valores de la función base que se multiplicarán por las muestras *s(x)* para obtener un coeficiente *S(u)* están dados por $\cos[(2x+1)u\pi/16]$.

Cuando u toma el valor de cero, se calculará entonces *S(0)*, es decir, el primer coeficiente DCT. Si sustituimos $u = 0$ en los argumentos de los cosenos, dichos cosenos tomarán el valor 1 independientemente del valor de la muestra x, quedándonos la siguiente modificación de la ecuación 4.1:

$$S(u) = \sum_{x=0}^{7} s(x) \cdot \frac{C(u)}{2} \qquad (4.2)$$

Asimismo, cuando $u = 0$, $C(u) = 1/\sqrt{2}$.

Tenemos entonces, que la primera función base es una función constante de valor igual a uno y, por lo tanto, cada muestra sólo se multiplica por el mismo valor $\frac{C(u)}{2} \approx 0.353553$. Los resultados de cada multiplicación se suman como se muestra en la figura 4.4:

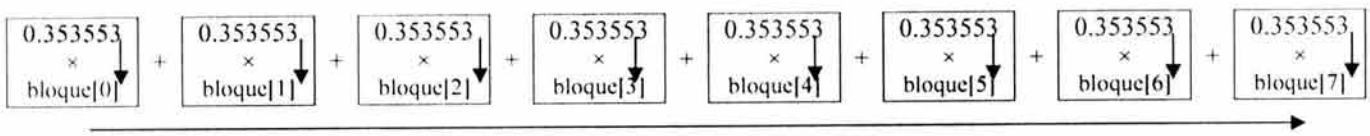


Figura 4.4 *Multiplicación* de las muestras del primer renglón de la MCU por los valores de la primera función base de la DCT en 1-D y *acumulación* de los resultados

El resultado de la sumatoria en la figura 4.4 nos entregará el valor del primer coeficiente DCT para el conjunto de ocho muestras compuestos por los elementos de *bloque* con índices del 0 al 7.

Para obtener el segundo coeficiente se repite el procedimiento pero ahora entre las muestras y los valores de la segunda función base. En este caso $C(u) = 1$ y los cosenos no trazan una función base constante como en el caso anterior. Esto se muestra en la figura 4.5:

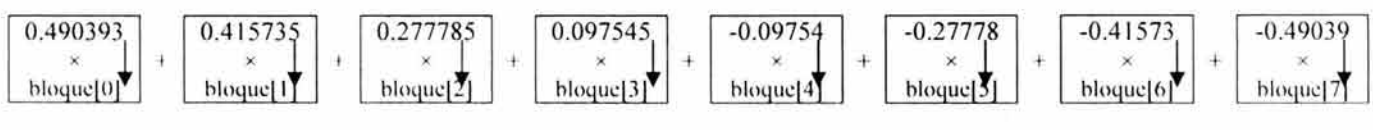


Figura 4.5 *Multiplicación* de las muestras del primer renglón de la MCU por los valores de la segunda función base de la DCT en 1-D y *acumulación* de los resultados

Los coeficientes restantes se calculan de la misma manera con cada función base.

Para obtener los coeficientes de los demás renglones de muestras de la unidad de datos debe de seguirse el procedimiento con cada renglón.

Una vez calculada la DCT en una dimensión, en este caso los renglones, debe de ejecutarse el procedimiento para las columnas con el fin de obtener la DCT bidimensional.

En lugar de que se agrupen muestras horizontalmente, se harán verticalmente. Ejemplos para obtener el primer y segundo coeficiente de un grupo de muestras verticales se muestran en las figuras 4.6 y 4.7.

Para calcular el primer coeficiente del primer conjunto de muestras verticales:

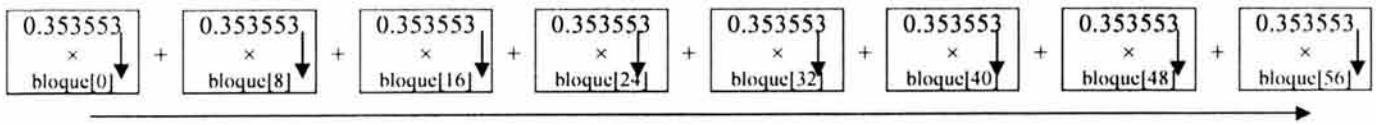


Figura 4.6 *Multiplicación* de las muestras de la primera columna de la MCU por los valores de la primera función base de la DCT en 1-D y *acumulación* de los resultados

Y para calcular el segundo coeficiente del primer conjunto de muestras verticales:

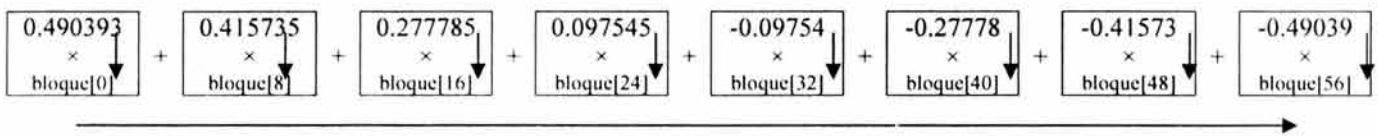


Figura 4.7 *Multiplicación* de las muestras de la primera columna por los valores de la segunda función base de la DCT en 1-D y *acumulación* de los resultados

El diseño anterior permite realizar la DCT en una dimensión sobre los elementos de *bloque* y almacenarlos directamente en el arreglo *bloque_t*. Sin embargo, la DCT en la siguiente dimensión se debe aplicar a los elementos de *bloque_t* y los resultados de la transformación almacenarse de nuevo en *bloque_t*. En este escenario no pueden modificarse los elementos de *bloque_t* hasta el momento en que ya no sean utilizados para la transformación, por lo cual es necesario el uso de un arreglo temporal que almacene los resultados.

Se agregó al código un vector de 8 números reales de nombre *vectort*, cuya función consiste en ir almacenando los coeficientes resultantes de la segunda DCT unidimensional. Al momento en que se termine el proceso de la DCT sobre una columna, *vectort* contará con los coeficientes resultantes. Los elementos de la columna transformada ya no serán necesarios para las demás iteraciones, y por ende los valores de *vectort* podrán sobrescribirse en la columna de *bloque_t* a la que se le haya aplicado la segunda transformada.

La interfaz entre los elementos del arreglo bidimensional *basisf* y los arreglos unidimensionales *bloque* y *bloque_t* se logró mediante manipulaciones aritméticas a los índices dentro de tres ciclos. Como ejemplo se incluye a continuación parte del código fuente correspondiente a la primera DCT aplicada a la unidad de datos:

```
/* DCT en los renglones de bloque */
    for(k=0;k<8;k++)
    {
            for(i=0;i<8;i++)
            {
                    bloque_t[i+(k*8)]=0; /* Inicialización de los elementos de
                                            bloque_t a cero */
                    for(j=0;j<8;j++)
                            bloque_t[i+(k*8)] += bloque[j+(k*8)]*basisf[i][j];

            }
    }
```

- El ciclo central utiliza un contador *j*, que sirve al mismo tiempo como índice para los elementos de cada renglón para la unidad de datos guardada en *bloque* y para las funciones base en *basisf*. La sumatoria de los resultados de la multiplicación entre las muestras de un renglón y los valores de la función base en turno se almacena en un elemento de *bloque_t*.
- Sólo hasta que se ejecutan 8 multiplicaciones y 8 sumas, termina el ciclo central y el segundo ciclo aumenta el índice *i* para calcular el siguiente coeficiente del renglón.
- Asimismo, sólo hasta el momento en que se hayan calculado los 8 coeficientes de un renglón, el ciclo exterior aumenta *k* para calcular los elementos del siguiente renglón.

Hemos visto a profundidad que la operación *multiplicación y acumulación* se utiliza ampliamente en el procedimiento de la Transformada Coseno, lo que justifica el uso de un DSP. Adicionalmente, la programación en módulos utilizada en el DSP nos permite utilizar cualquier otro algoritmo para calcular la DCT; habilitándonos así buscar el proceso más eficiente para nuestra aplicación. No sólo se pueden intercambiar optimizaciones de la DCT programadas en C, sino también se pueden utilizar algoritmos que se desarrollen en lenguaje ensamblador; dándonos así mayor control sobre el Hardware y reduciendo ciclos de ejecución.

Como se mencionó en el apartado 4.2.2, el ambiente CCS puede mejorar el desempeño de nuestro algoritmo si se modifica para que las instrucciones se ejecuten en *pipeline*. En el nivel más alto de optimización del CCS puede considerarse que el código fue programado en lenguaje ensamblador.

4.3.4 Cuantización de los coeficientes DCT

El control de la compresión en el estándar JPEG se encuentra en la cuantización de los coeficientes; mientras más coeficientes resulten ser cero después de dicho proceso, el código asignado a la información ocupará menos espacio.

Como se mencionó en el apartado 3.3.3, cada coeficiente DCT se puede cuantizar con un valor diferente, dependiendo de la aportación de la función base a la imagen. Dentro de las páginas del estándar JPEG se incluye una tabla como ejemplo para realizar la cuantización a la componente de luminiscencia y otra tabla para las componentes de crominancia. La generación de dichas tablas requiere de estudios externos al tema tratado, por lo que estos ejemplos de tablas fueron las que se utilizaron en el proyecto.

El Grupo Independiente de JPEG (*Independent JPEG Group*) propone una manera de controlar el grado de cuantización de cada coeficiente, y ésta consiste en ajustar una tabla fija mediante una escala [5]. Para modificar la cuantización, se eligió dividir los valores de las tablas entre un número entero, ej. : 2, 4, etc., o bien, si se desea cuantizar aún más los coeficientes, se divide la tabla por el inverso de un entero.

La siguiente ecuación se obtiene de la ecuación 3.4 y define el proceso de control de la cuantización:

$$Sq_{vudiv} = redondeo\left(\frac{S_{vu}}{\left(\frac{Q_{vu}}{div}\right)}\right) \qquad (4.3)$$

donde:

- S_{vu} es un coeficiente DCT
- Q_{vu} es el elemento cuantizador
- Sq_{vudiv} es el coeficiente cuantizado modificado por *div*
- *div* es el número que modifica el nivel de cuantización de la unidad de datos

Es necesario tener cuidado de que *div* no rebase ningún valor de la tabla de cuantización, ya que la razón $\left(\frac{Q_{vu}}{div}\right)$ resultaría en un número positivo menor a 1, dicho caso ocasionaría el aumento de valor en algunos coeficientes DCT y disminución en otros. Sobrepasar este límite implicaría obtener resultados inesperados.

Adicionalmente, en la práctica no es recomendable utilizar números mayores a 5, debido a que la imagen no sufre mayores cambios arriba de este número, ya sea para el aumento o disminución del nivel de cuantización.

Una vez obtenida la unidad de datos cuantizada, se procede a la codificación del coeficiente DC.

La figura 4.8 representa el proceso de cuantización en el DSP:

bloque_t
Arreglo de
números reales

bloque
Arreglo de
números enteros

Figura 4.8 Representación del proceso de cuantización

Se puede observar en la figura 4.8 que se toman los coeficientes reales almacenados en *bloque_t* y se les aplica el proceso de cuantización, los coeficientes resultantes al estar redondeados pueden almacenarse de nueva cuenta en *bloque*.

La parte del código fuente de la función que realiza la cuantización (ec. 4.3) se incluye a continuación:

```
for(i=0;i<64;i++)
        bloque[i]= roundf(bloque_t[i]/(tql[i]/div));
```

El código consiste en un ciclo de 64 repeticiones, una para cada elemento de *bloque_t*. El arreglo *tql* contiene el valor de cuantización para cada elemento de la unidad de datos, y el número divisor tiene el mismo nombre que en la ecuación 4.1: *div*. Como se mencionó, los coeficientes cuantizados se almacenan en *bloque*.

4.3.5 Codificación de las diferencias entre coeficientes DC

Como se planteó en el apartado 3.4.7, son las diferencias entre coeficientes DC y no los coeficientes DC en sí las que son codificadas.

La codificación DC consiste en representar las diferencias mediante dos símbolos: un código Huffman que nos indique el número mínimo de bits adicionales con el cual se puede representar dicha diferencia, es decir *SSSS*, y la diferencia en sí, expresada en *SSSS* bits.

Ahora bien, tanto el código Huffman como los bits sin codificar son de longitud variable y su eficiencia consiste en que pueden utilizar menos memoria de la que ocupan los registros convencionales: 8, 16, 32 bits. Las tablas de códigos utilizados se encuentran en los documentos del estándar JPEG [1]. El estándar en sí, no define como obtener las tablas de códigos, sin embargo incluye algunas como ejemplo para poder desarrollar las aplicaciones. En dichas tablas, el código más extenso para un coeficiente DC consta de 11 bits, mientras que el código más extenso para un coeficiente AC consta de 16 bits.

Para que la codificación sea funcional es necesario que los bits de los códigos y de las magnitudes no codificadas sean concatenados y que varios coeficientes codificados puedan ocupar un mismo registro. Debido a lo anterior, se eligió diseñar un algoritmo de concatenado de bits que utiliza un registro de 16 bits en donde se acumulan temporalmente los códigos y magnitudes concatenadas. El bit más significativo del código en turno se concatena al bit menos significativo del código anterior y así sucesivamente hasta completar 16 bits.

Al momento de que el registro temporal tenga 16 bits de información codificada, se duplica el registro temporal en una localidad de memoria predestinada para recibir esos datos (un arreglo de localidades de 16 bits) como se ve en la siguiente figura:

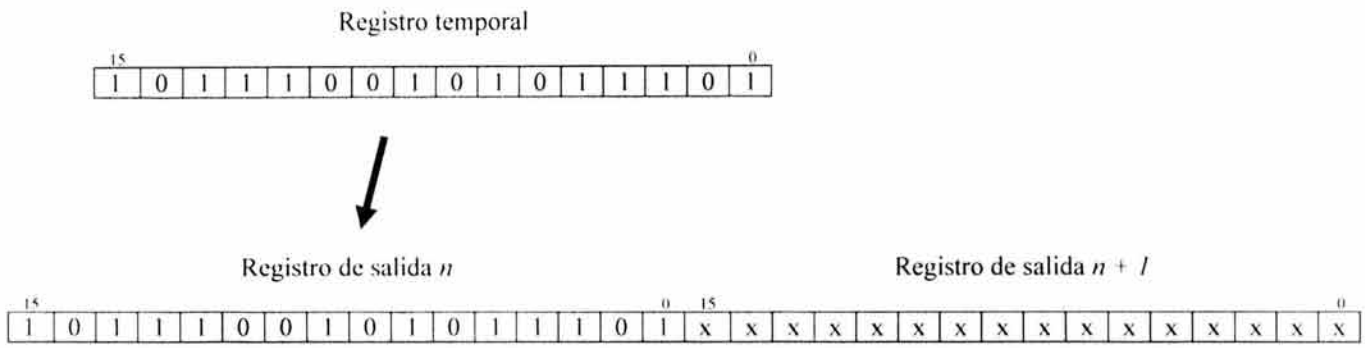


Figura 4.9 Escritura en el arreglo de salida de la información del registro temporal

La figura 4.9 muestra cómo el registro temporal, al contar con 16 bits de información, duplica los datos en un registro de salida *n*. El registro *n* es elemento de un arreglo de registros de 16 bits. El registro *n + 1* contiene información aleatoria y espera a que el arreglo temporal se llene de nuevo para recibir su información.

Algoritmo de concatenado

El algoritmo de concatenado se muestra en el diagrama de flujo de la figura 4.10.

Cuando el número de bits concatenados es mayor que 16, los 16 bits que ocupan el registro temporal se duplican al arreglo de salida. Los bits restantes del último código o magnitud concatenada se acumulan en la localidad de memoria temporal para concatenarse con los datos siguientes.

Almacenar la información en un arreglo de memoria de localidades de 16 bits nos asegura que la información resultante de la codificación no estará dispersa y se encuentre en un bloque definido.

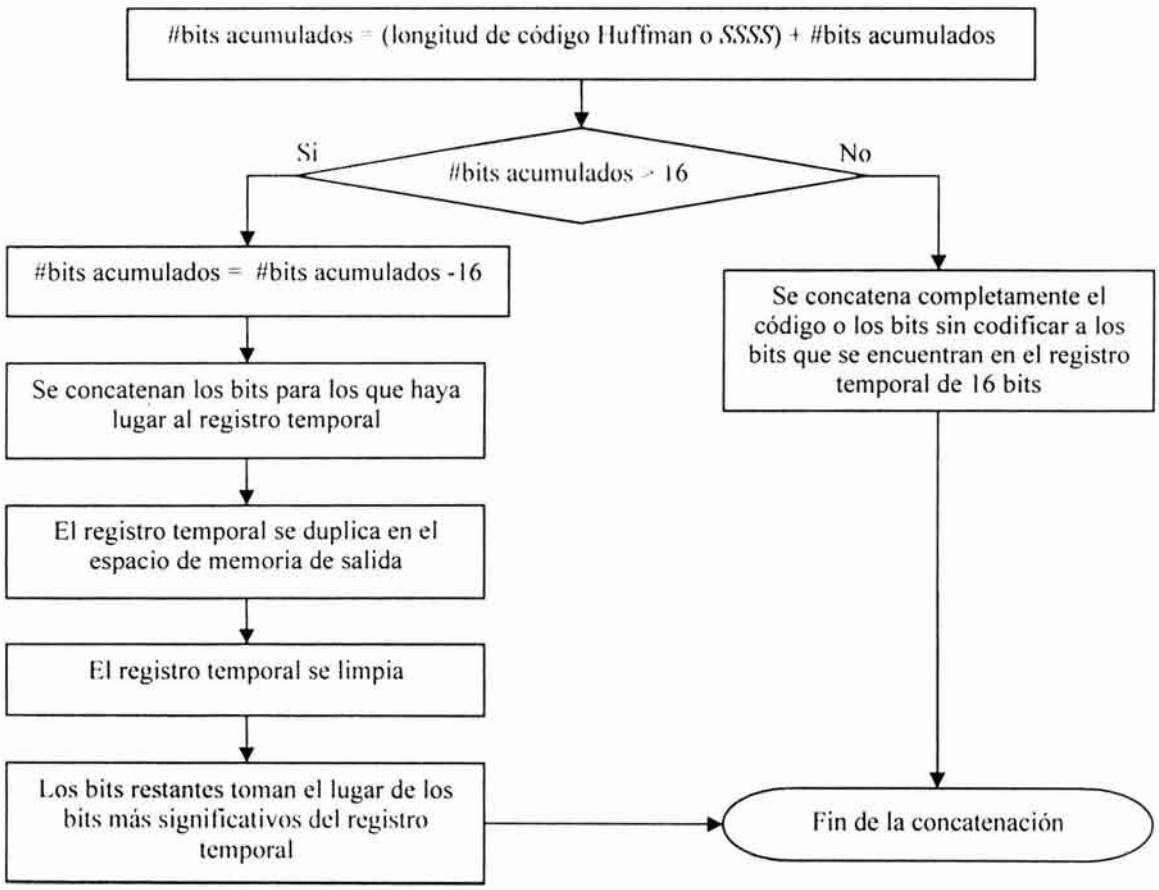


Figura 4.10 Diagrama de Flujo del algoritmo de concatenación de la información de salida

Este algoritmo de concatenado sirve tanto para los códigos variables así como para los bits sin codificar que se agreguen al arreglo de salida que guarda la imagen comprimida. Una vez analizado este bloque esencial de la codificación, podemos hacer referencia a él como un simple bloque de ejecución en los siguientes diagramas.

Algoritmo general para la codificación DC

En la figura 4.11 se muestra el diagrama de flujo general del algoritmo de codificación DC.

Como se mencionó, el primer paso de la codificación de los coeficientes DC consiste en calcular la diferencia entre el valor anterior y el actual de los coeficientes DC. Inmediatamente después del cálculo de la diferencia, el valor actual pasa a duplicarse en la localidad de memoria del valor anterior, preparándose así para la iteración de la siguiente unidad de datos.

A estas operaciones les sigue una subrutina diseñada para determinar *SSSS* y las diferencias expresadas en *SSSS* bits.

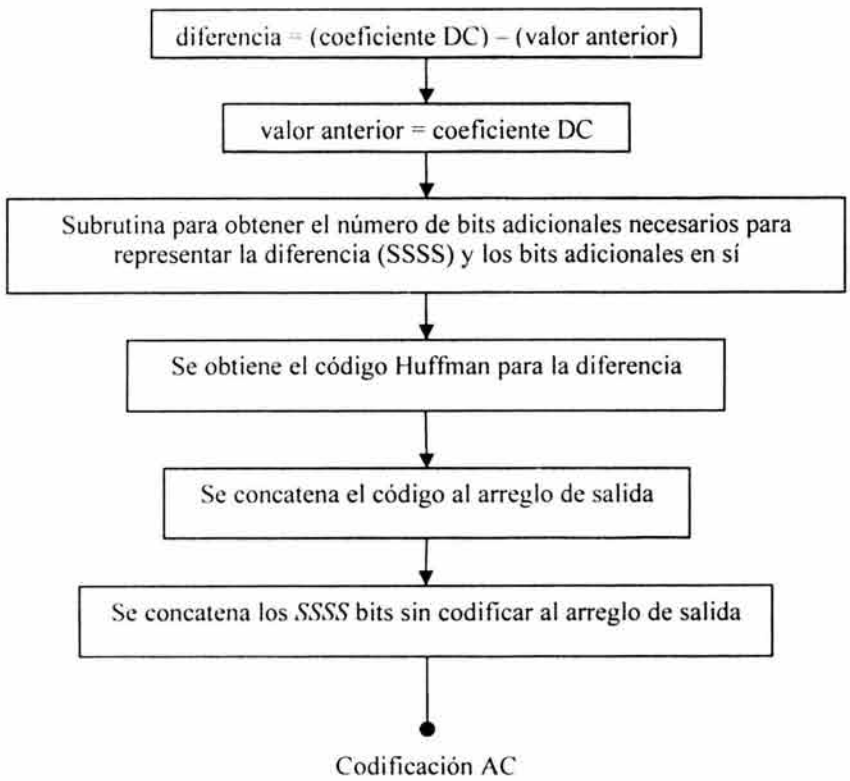


Fig. 4.11 Diagrama de flujo general de la codificación de coeficientes DC

Posteriormente se busca el código Huffman correspondiente a *SSSS* para que una vez encontrado se concatene el código al arreglo que almacena la información de la imagen comprimida.

Asimismo se concatena la diferencia expresada en *SSSS* bits al arreglo de salida para después pasar a la codificación de los coeficientes AC.

Clasificación de las diferencias DC en las categorías SSSS

Dentro de la tabla 3.3 encontramos las categorías *SSSS* en que se clasifican las magnitudes posibles de las diferencias DC dependiendo de los bits necesarios para representarlas; el proceso de clasificación en las categorías *SSSS* forma parte del tercer bloque de la figura 4.11 y se muestra a detalle en el diagrama de flujo representado en la figura 4.12.

El algoritmo contiene dos casos generales; un caso en que la diferencia es cero y el otro en que no lo es. Siendo la diferencia igual a cero, la clasificación es inmediata, además de que es el único caso que no se requiera concatenar al código bits sin codificar que representen dicha magnitud. Si la diferencia es diferente de cero, se determina primero si la cantidad es negativa o positiva; si es positiva, dicha cantidad no se modificará y pasará tal cual al siguiente bloque del proceso, sin embargo, si es negativa

una bandera guardará el hecho de que la cantidad a codificar es negativa y se substituirá la diferencia por su inverso aditivo.

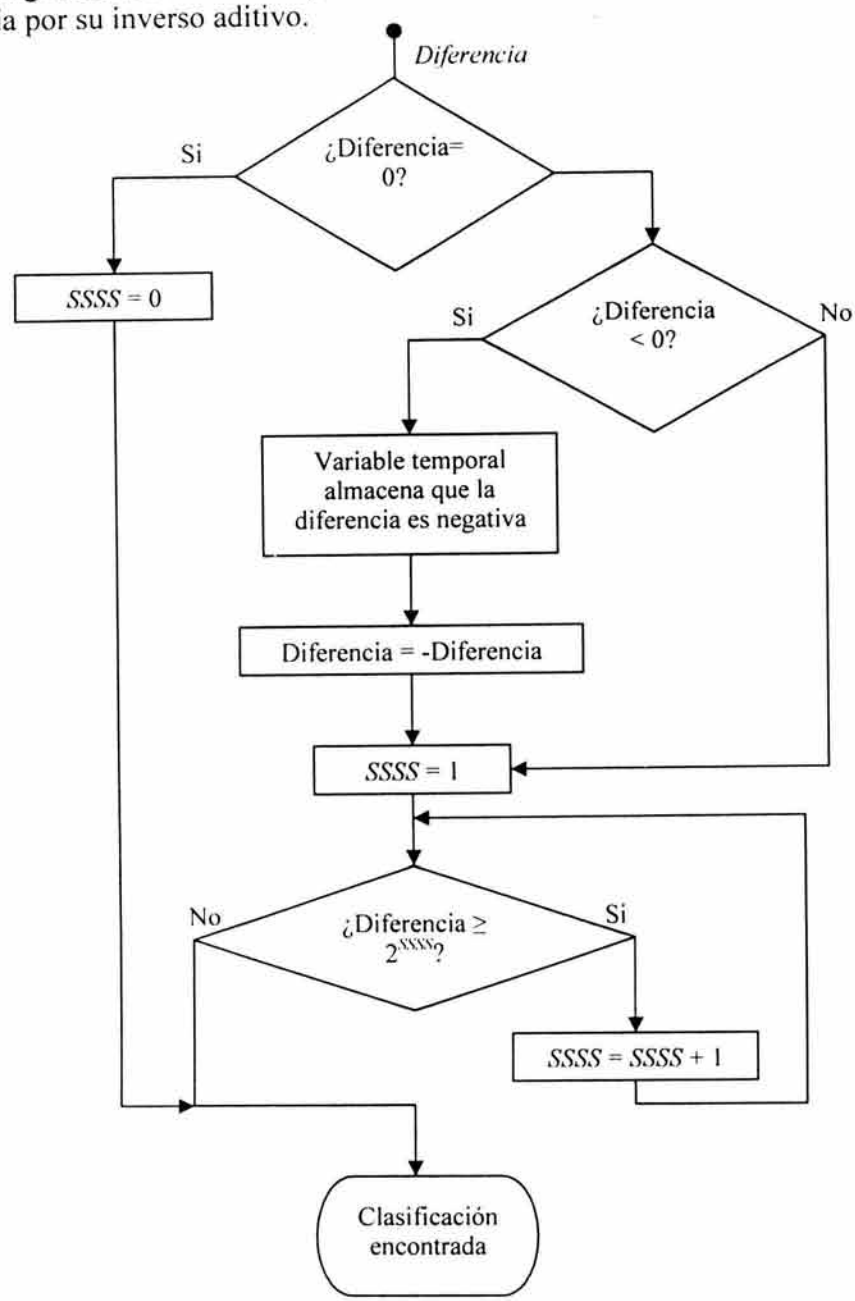


Figura 4.12 Diagrama de flujo del algoritmo para encontrar *SSSS*

El hecho de convertir las diferencias negativas a su inverso aditivo nos permite clasificar todos los valores con un simple algoritmo en que se compara la diferencia con potencias de 2. Mientras que la diferencia sea mayor o igual a 2^{SSSS}, *SSSS* tendrá incrementos iguales a 1; en el momento en que dicha relación no se cumpla se habrá encontrado la clasificación para la cantidad a codificar, es decir, el valor de *SSSS* para la diferencia.

Preparación de los bits sin codificar para su concatenado

Una vez clasificada la diferencia, ésta se preparará para concatenarse al código variable. Debido a que los bits se concatenarán mediante la operación booleana "*O*" u en inglés "*OR*", si la diferencia originalmente era positiva, la cantidad no requerirá de mayor preparación. Sin embargo, si era negativa, es necesario aplicarle la operación de complemento a uno pero sólo en los bits *SSSS*. Por ejemplo, si tenemos un registro de 16 bits parcialmente almacenando bits codificados de la forma siguiente:

XXXXXXXXXXXXXX00

Donde *X* representa un 0 o un 1, y los últimos bits aún no almacenan parte de la imagen comprimida. Si quisiéramos concatenar, digamos el número 2 (10_b), sólo sería necesaria la operación

XXXXXXXXXXXXXX00 OR *0000000000000010*

para obtener

XXXXXXXXXXXXXX10

En contraste, si la diferencia fuese negativa, al aplicarle el complemento a uno al número 2 obtendríamos en primer lugar

1111111111111101

cantidad que, si intentásemos concatenar con el mismo procedimiento, nos llevaría a modificar los bits codificados X que no fueran ceros. Existen múltiples métodos para lograr que el complemento a uno se aplique tan sólo en los primeros *SSSS* bits del registro.

Durante el diseño del programa se abordó el problema de diferentes maneras, y en está sección del programa de compresión de imágenes, se eligió la forma tal vez más directa y fácil; los corrimientos.

Si recorremos en primera instancia este registro *n* bits a la izquierda, en donde *n* = (Número de bits en el registro – *SSSS*) y posteriormente *n* bits a la derecha, el resultado de la diferencia después de aplicarle este proceso nos dará el registro deseado. En este caso en particular *n* = (16-2) = 14, dando como resultado de los corrimientos la siguiente cantidad:

0000000000000001

En este momento, es posible aplicar la operación "*OR*" con el registro parcialmente codificado:

XXXXXXXXXXXXXX00 OR *0000000000000001*

obteniendo

XXXXXXXXXXXXXX01

sin modificar los bits *X*.

No obstante, como se muestra en la figura 4.11, antes de concatenar los bits sin codificar es necesario concatenar los bits codificados.

Almacenamiento de los códigos Huffman para las diferencias DC

Los códigos para las diferencias entre coeficientes de luminiscencia DC se pueden encontrar en la tabla A.3 dentro del Anexo A.

Dichos códigos se almacenaron en un arreglo de estructuras con dos elementos: uno que almacenará la longitud del código y otro para el código en sí.

A continuación se muestra la definición y declaración respectivamente del arreglo de estructuras que almacenan los códigos DC.

```
struct chufdc {
        unsigned cdl   :4;
        unsigned cod   :9;
                        };

struct chufdc codDC[12];
```

El arreglo de estructuras cuenta con doce elementos debido a que las categorías de *SSSS* van de 0 a 11. Los índices del arreglo corresponden a la categoría *SSSS*.

Los elementos o miembros de la estructura se declaran como una cantidad de bits no signados; el código más largo para la codificación de la componente de luminiscencia DC cuenta con 9 bits, por ello sólo serán necesarios 4 bits para almacenar la longitud del código ($1001_b = 9_d$) y 9 bits para el código en sí, por lo que declarar miembros con longitudes menores a los registros convencionales nos permite optimizar el manejo de memoria ya que se desperdicia menos espacio de memoria dato en el almacenamiento.

4.3.6 Codificación de los coeficientes AC

A la codificación del coeficiente DC de una unidad de datos, le sigue la codificación de los 63 coeficientes AC restantes.

En el apartado 3.4.7 se mencionó que previamente a la codificación de los coeficientes AC es necesario reordenarlos en zig-zag. Para ello se diseñó una función que duplica los coeficientes AC en un arreglo temporal y trabaja en este nuevo arreglo temporal, al terminar el proceso sobrescribe los coeficientes AC en el nuevo orden al arreglo original.

El arreglo temporal cuenta con sólo 63 elementos a diferencia del arreglo original, ya que no contempla al coeficiente DC.

La función para reordenamiento zig-zag consta de dos ciclos generales; uno para los primeros 35 coeficientes reordenados y otro para los restantes.

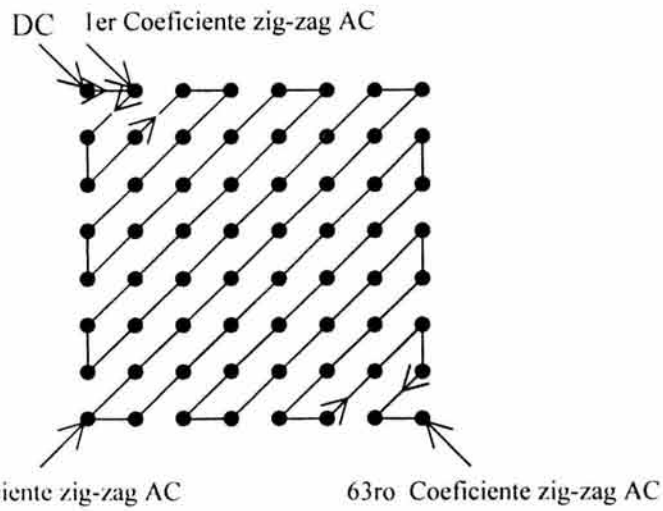


Figura 4.13 Referencia de coeficientes en el ordenamiento zig-zag

Ordenamiento de los primeros 35 coeficientes zig-zag

En la figura 4.13 se muestra que los índices de los coeficientes AC de la unidad de datos en sí es diferente a los índices de los coeficientes en el orden zig-zag; notamos que el 56to coeficiente original AC corresponde al 35to coeficiente zig-zag. Asimismo podemos observar que el ordenamiento de los coeficientes AC mantiene una progresión durante los primeros 35 elementos, para posteriormente ir disminuyendo hasta el último elemento.

Como se puede ver en la figura 4.13, en el ordenamiento de los primeros 35 elementos se realiza uno de cuatro tipos de incrementos o decrementos en el índice de los *coeficientes originales* para encontrar el siguiente *coeficiente zig-zag*:

- Incremento de un elemento
- Incremento de siete elementos
- Incremento de ocho elementos
- Decremento de siete elementos

Tomando en cuenta dichas relaciones tenemos el diagrama de flujo de la figura 4.14 para ordenar los primeros 35 coeficientes AC.

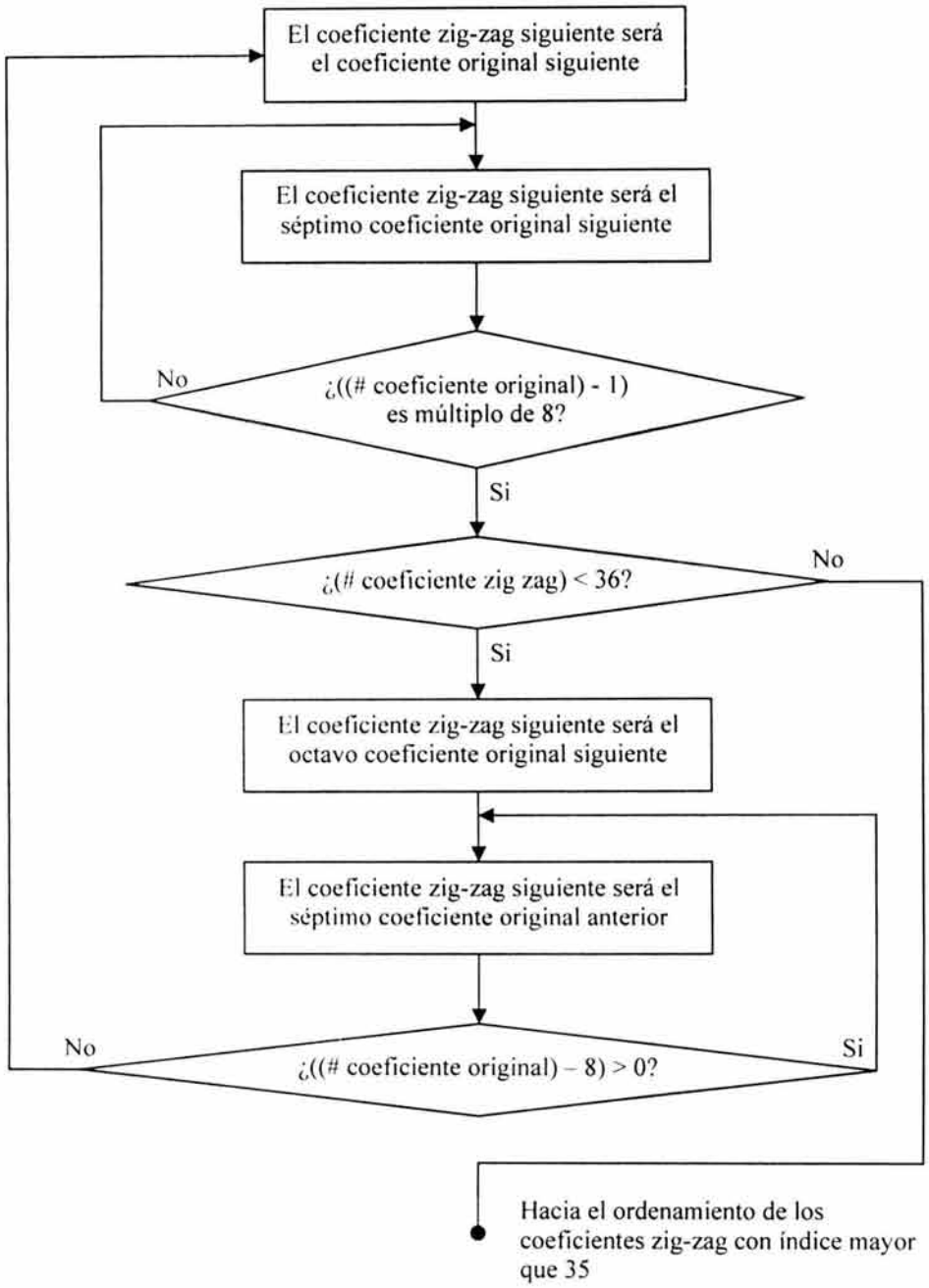


Figura 4.14 Diagrama de flujo para el ordenamiento zig-zag de la primera parte de los coeficientes AC de una unidad de datos

El proceso inicia con una asignación y se debe tomar en cuenta que el primer coeficiente zig-zag siguiente es el primero del arreglo zig-zag, mientras que el coeficiente original siguiente es el segundo de la unidad de datos; es decir, el primer coeficiente AC.

El funcionamiento correcto de este algoritmo depende de que los índices de la unidad de datos se incrementen previamente a la asignación de datos y que los índices del arreglo zig-zag se incrementen posteriormente a ella. Tomemos como ejemplo unas cuantas iteraciones:

1er coef. zig-zag = 2do coef. de la unidad de datos

Se incrementó previamente el índice de la unidad de datos y se incrementa posteriormente el del arreglo zig-zag, por lo que obtenemos la siguiente asignación:

2do coef. zig-zag = 9no coef. de la unidad de datos

Y así sucesivamente siguiendo los bloques de ejecución del diagrama de flujo; tenemos que el índice de la unidad de datos es 9, (9-1) es un múltiplo de 8, y también cumple la condición de ser menor que 36, por lo que la siguiente asignación resulta ser:

3er coef. zig-zag = 17mo coef. de la unidad de datos

El índice de los coeficientes zig-zag se continúa incrementando posteriormente y las siguientes asignaciones del diagrama de flujo hasta el momento en que se deje de cumplir la condición ((indice original – 8) > 0):

4to coef. zig-zag = 10mo coef. de la unidad de datos

5to coef. zig-zag = 3er coef. de la unidad de datos

Después de esta última iteración, la condición arriba mencionada deja de cumplirse y se ejecuta un salto al primer bloque de ejecución para que el algoritmo se repita hasta que se cumpla la condición en que el índice del coeficiente zig-zag es igual a 36.

Ordenamiento de los coeficientes zig-zag restantes

Una vez ordenados los primeros 35 coeficientes de la unidad de datos concluye el primero de los dos ciclos generales para reordenar la unidad de datos; el segundo ciclo prosigue con los coeficientes restantes.

Como se puede observar en la figura 4.13, los incrementos y decrementos que suceden en este ordenamiento son los mismos que en los 35 primeros elementos, sin embargo se aplican en orden diferente mientras que los bloques condicionales son similares. Debido a estas cualidades es posible diseñar el algoritmo para el segundo ciclo simplemente modificando un poco el algoritmo del primer ciclo.

Tomando en cuenta lo anterior tenemos el diagrama de flujo que se encuentra en la figura 4.15:

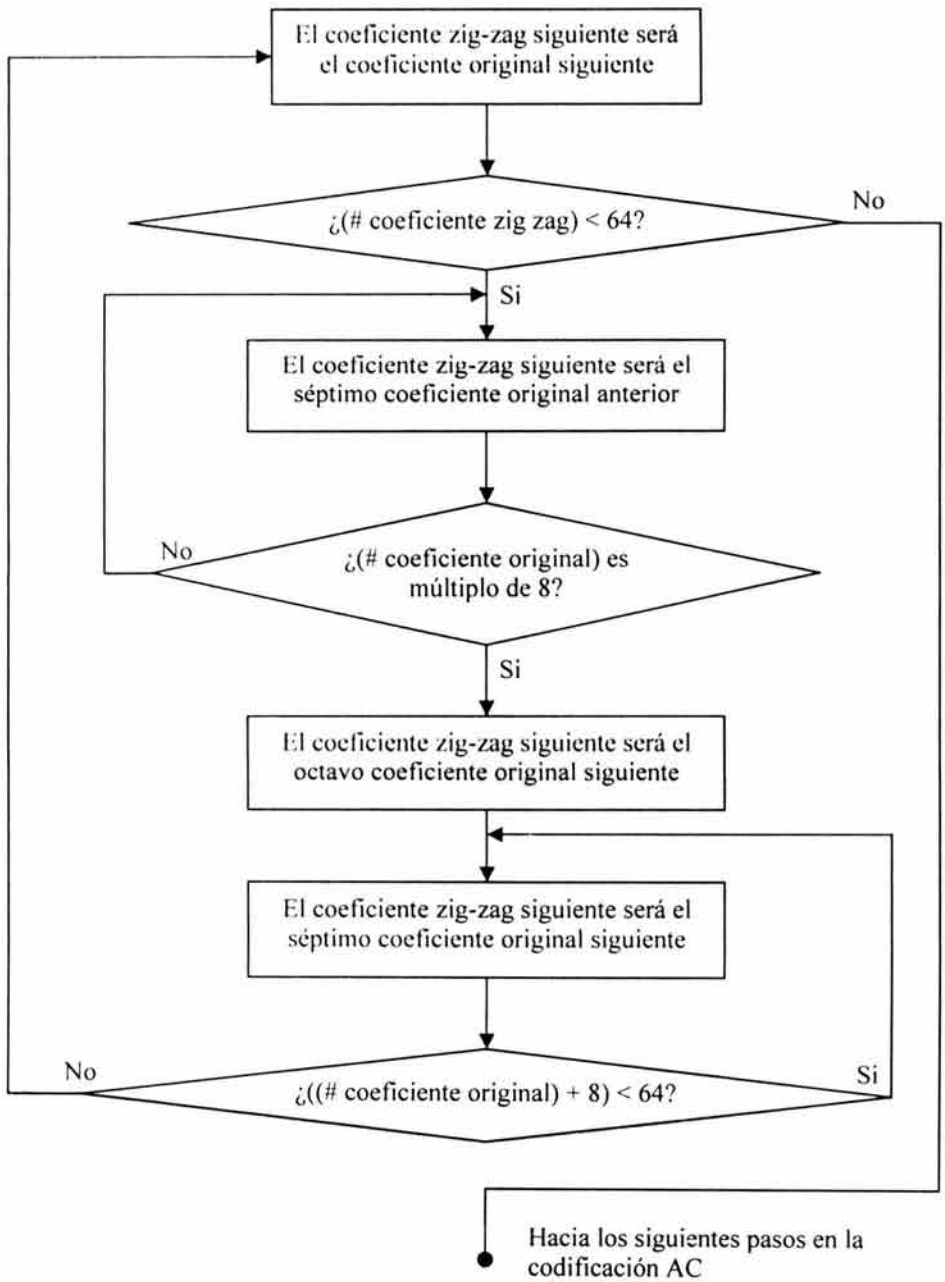


Figura 4.15 Diagrama de flujo para el ordenamiento zig-zag de la segunda parte de los coeficientes AC de una unidad de datos

El diagrama del segundo ciclo de reordenamiento se muestra en la figura 4.15. Éste funciona al igual que el primero, incrementado los índices originales previamente a la asignación de datos e incrementando posteriormente a ella los índices zig-zag.

Al contarse con los coeficientes DCT de una unidad de datos arreglados en un orden zig-zag, puede representarse dicha unidad de datos como una serie de pares de símbolos, *RRRR* y *SSSS*, como se mencionó en el apartado 3.4.7.

Como se recordará, *RRRR* representa el número de ceros consecutivos que ocurren en la secuencia zig-zag hasta el acontecimiento de un coeficiente diferente de cero, mientras que *SSSS* contiene el número de bits adicionales que nos permitirán recuperar la magnitud y signo de dicho coeficiente.

El algoritmo diseñado para obtener dichos símbolos debe analizar en la unidad de datos un coeficiente a la vez siguiendo el orden zig-zag, e inspeccionar si el coeficiente en turno es o no diferente de cero. Este algoritmo quedó inmerso dentro de la función de codificación general de los coeficientes AC, el cual asigna los códigos dependiendo de los símbolos y los concatena al arreglo de salida.

Almacenamiento de los códigos Huffman para los coeficientes AC

Los códigos Huffman AC se almacenaron en un arreglo bidimensional de estructuras cuyos índices están determinados por *RRRR* y *SSSS*. Cada estructura contiene, al igual que las estructuras destinadas para los coeficientes DC, dos miembros; uno que almacena la longitud del código y otro que almacena el código en sí. Dado lo anterior, la definición de las estructuras quedó de la siguiente manera:

```
struct chufac {
        unsigned cdl  :5;
        unsigned cod  :16;
                      };
```

En el caso de los códigos AC, la longitud del código más extenso alcanza los 16 bits (10000_b), por lo que los bits destinados al miembro que almacena la longitud del código son cinco.

La declaración, a su vez, quedó de la siguiente forma:

```
struct chufac codAC[16][10];
```

En donde el primer valor entre corchetes corresponde a los valores de *RRRR* posibles, y el segundo a los valores *SSSS* posibles.

Una inspección a la tabla 3.6 permite detectar que la declaración del arreglo de estructuras de códigos AC no es suficiente para contener todos los casos de combinaciones de *RRRR* y *SSSS*; la razón fue que en la primera columna de dicha tabla sólo existen dos combinaciones válidas para *RRRR* y *SSSS*, es decir, cuando *SSSS* = 0, *RRRR* sólo puede adoptar dos valores válidos: 0 y 16.

En lugar de desperdiciar memoria declarando casi toda una columna que no se utilizará, se definieron y declararon por separado estructuras para los casos *EOB* y *ZRL*.

```
struct chufeob {
        unsigned cod  :4;
                              };

struct chufzrl {
        unsigned cod  :11;
                              };
struct chufeob codEOB;

struct chufzrl codZRL;
```

Cada definición cuenta con un miembro con el espacio suficiente para guardar el código de cada caso.

Así, para referirnos a cualquier caso contamos con los símbolos provistos por las combinaciones de *RRRR* y *SSSS* y los símbolos *EOB* y *ZRL* por separado. El símbolo *EOB* sólo puede ser uno de dos valores; 0 ó 1 para determinar si se presentó el fin de la unidad de datos. En contraste, *ZRL* puede presentarse hasta tres veces seguidas antes de que se presente *EOB*, por lo que sus valores probables sólo son 1, 2 y 3.

Algoritmo general para la codificación AC

El diagrama de flujo presentado en la figura 4.16 ilustra el funcionamiento del algoritmo general de codificación AC.

Dicho algoritmo cuenta con las condiciones iniciales *RRRR* y *ZRL* igual a cero, la condición inicial de *SSSS* no importa debido a que su valor no se acumula. El proceso comienza con la lectura de un coeficiente AC respetando el orden zig-zag, dependiendo si el coeficiente es igual a cero el proceso se divide en dos ramas principales:

- Si el coeficiente es igual a cero, se contabiliza dentro de *RRRR*, y si *RRRR* es igual a 16 se tiene una cadena de 16 ceros consecutivos o *ZRL*. Al momento en que se contabiliza *ZRL*, a la cantidad *RRRR* se le asigna el valor cero. Es importante hacer notar que la ramificación que se toma cuando el coeficiente es igual a cero no incluye ninguna instrucción de escritura a la cadena de salida, solamente en el caso en que el último coeficiente sea igual a cero: es decir, si se encuentra el fin de la unidad de datos no importa que haya ceros o *ZRL* acumuladas, tan sólo se escribirá el código para el fin de bloque *EOB*.

- En el caso en que el coeficiente sea diferente de cero, se verifica si la bandera *ZRL* es diferente de cero. Si la bandera tiene valor de uno, dos o tres, se escribirá el código de *ZRL* y posteriormente se disminuirá en uno el valor de dicha bandera. En el momento en que esta bandera *ZRL* tome el valor cero, se habrán escrito en el arreglo de salida el código para las cadenas de 16 ceros consecutivos encontrados (si acaso hubo). A continuación se obtiene *SSSS* y

los bits sin codificar a partir del coeficiente diferente de cero. Con *RRRR* y *SSSS* se busca el código correspondiente y se concatena el código y los bits sin codificar al arreglo de salida. Posteriormente *RRRR* toma el valor cero para quedar listo en la siguiente iteración. Por último, si el coeficiente codificado fue el último coeficiente de la unidad de datos, la codificación de la unidad de datos se da por terminada sin agregar más código.

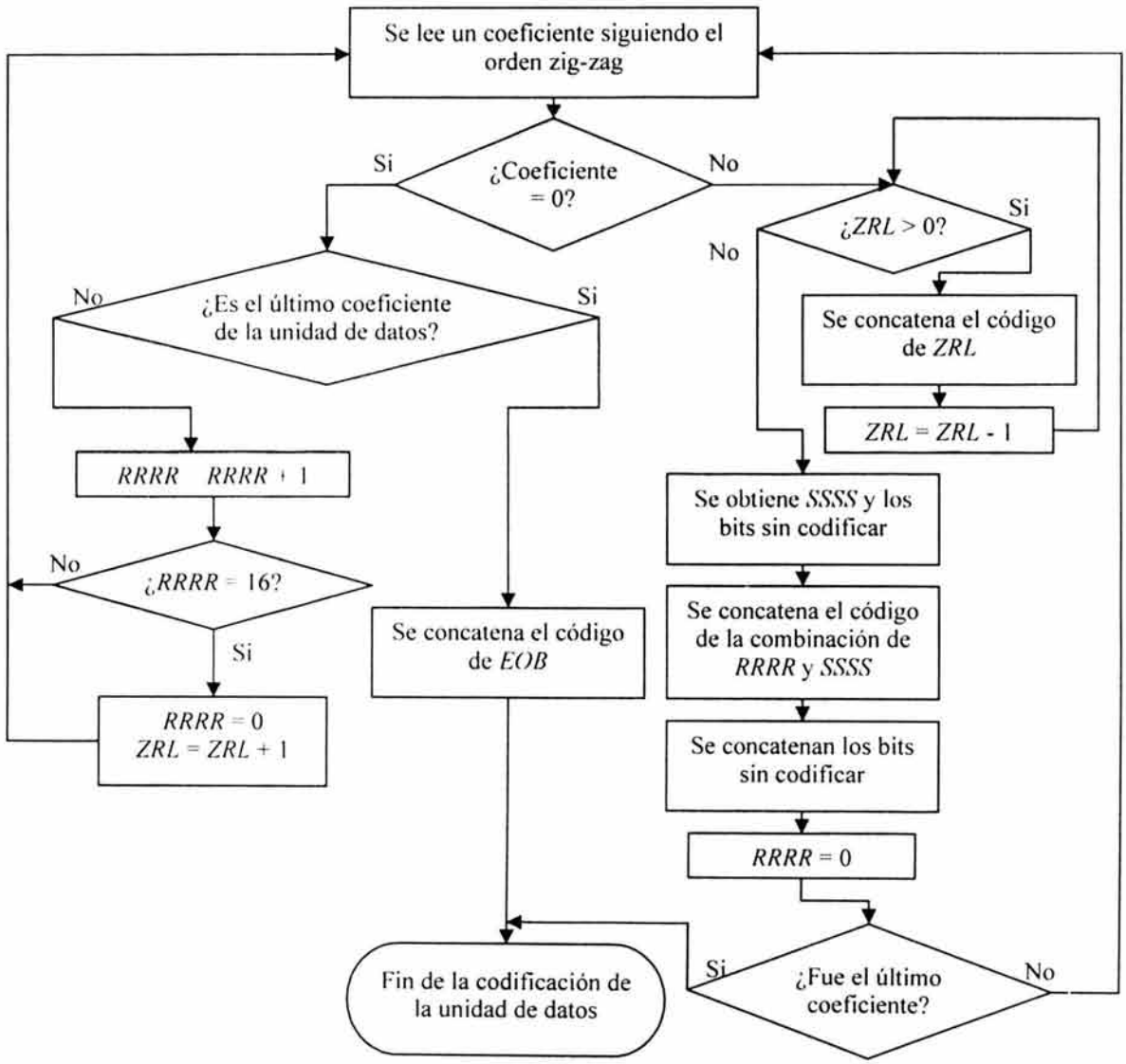


Figura 4.16 Diagrama de flujo general de la codificación AC

Los algoritmos para codificación de coeficientes DC y AC son completamente funcionales para todas las unidades de datos de la imagen de entrada a excepción del último bloque. Debido a la manera en que se compaginan los algoritmos de concatenado con los de codificación, el registro temporal siempre conserva bits codificados que

esperan ser concatenados en la siguiente iteración; así, al no haber siguiente iteración, dichos bits no se escriben al arreglo de salida.

Dado lo anterior, es necesario un algoritmo de detección de la última unidad de datos a codificar, para que después de ser procesada, los bits del registro temporal se concatenen al arreglo de salida.

En nuestro caso es sencillo debido a que las imágenes a procesar que utilizamos en este proyecto son de 256×256 píxeles; si los índices utilizados en las instrucciones *for* anidadas para el procesamiento de las unidades de datos (apartado 4.3.2) están en su valor tope ($l = 31$ y $k = 31$), el arreglo temporal se concatena después del procesamiento de la unidad de datos.

Una vez desarrollado este último proceso se pueden considerar descritos todos los algoritmos de codificación de la imagen y se puede proceder a detallar la integración de dichos algoritmos en el DSP.

4.4 Módulos programados en el DSP para la compresión de imágenes

Los módulos programados para la compresión de imágenes y las interrelaciones entre los mismos se pueden ilustrar mediante la figura 4.17:

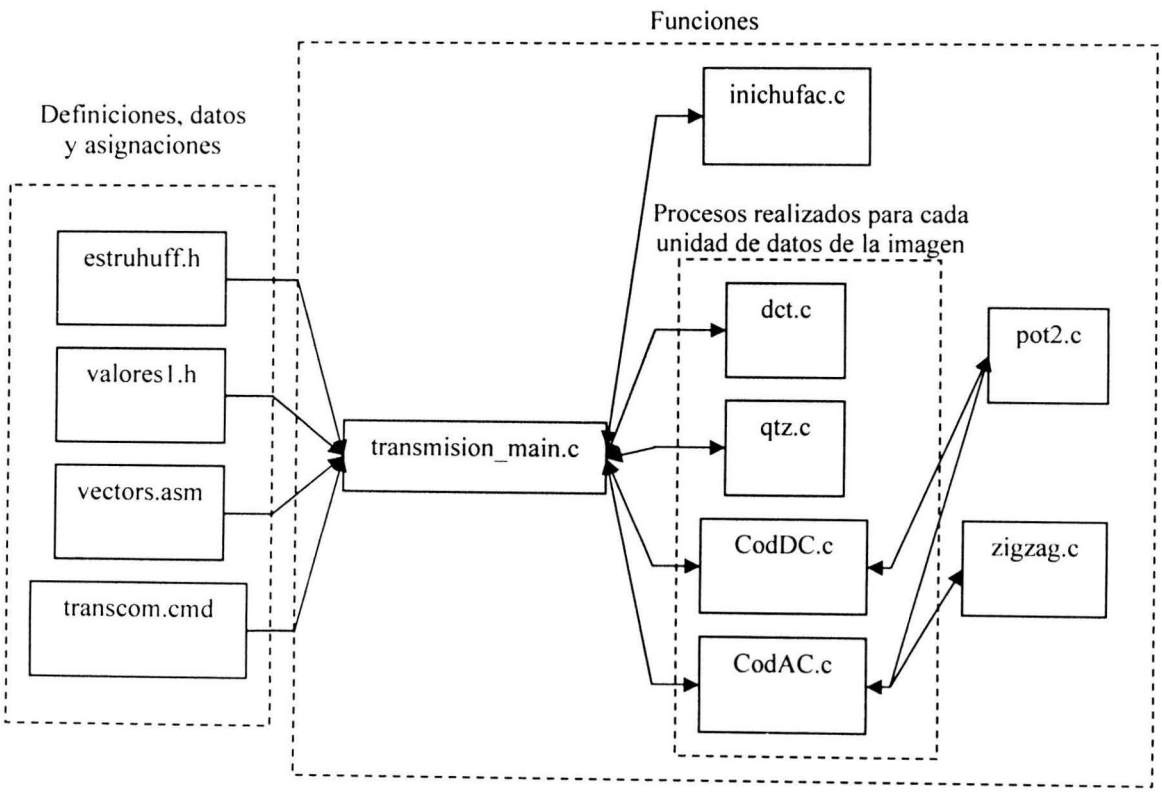


Figura 4.17 Relaciones entre los módulos programados para la compresión de una imagen

En primera instancia, se pueden observar en la figura 4.17 dos bloques: el bloque de la izquierda consta de definiciones, datos y asignaciones necesarias para la ejecución del programa, mientras que al bloque de la derecha lo conforman las funciones que realizan la compresión en sí.

Al bloque de definiciones, datos y asignaciones lo componen, a su vez, cuatro bloques que representan archivos:

- *estruhuff.h*
  Este archivo contiene las declaraciones de las estructuras que guardan los códigos Huffman, así como declaraciones de algunos valores constantes que se utilizan ya sea en el bloque principal de ejecución o en las funciones emergentes.

- *valores1.h*
  Aquí se incluye la información de la imagen original (tan solo una componente del espacio de color en este caso) en forma de un arreglo de números enteros de 8 bits.

- *vectors.asm*
  Cuando el código del programa no inicia en la primera localidad de la memoria programa del DSP, como ocurre en el caso de la programación en lenguaje *C*, es necesario señalar al DSP la localidad en donde debe empezar a ejecutarse el programa; en este archivo se define dicha localidad. Adicionalmente, en este bloque se pueden definir las interrupciones a utilizar; como en nuestro caso no estamos obteniendo la información en tiempo real, no utilizamos interrupciones y por lo tanto se deshabilitan las llamadas a interrupciones.

- *transcom.cmd*
  Se puede elegir con exactitud el lugar en la memoria donde se almacenen los datos mediante un archivo *cmd*; ya sea en la memoria *SDRAM*, en la memoria *Flash*, etc.

Estos cuatro bloques no ocupan tiempo de ejecución en el algoritmo de compresión, tan sólo ocupan tiempo en que el DSP realiza la lectura del programa a ejecutar.

Dentro del bloque de funciones encontramos el módulo principal *transmisión_main.c*. Dicho módulo contiene en un principio las definiciones de las demás funciones a las que llama mediante su ejecución, inicializaciones de las tablas de cuantización y de las estructuras de codificación para los coeficientes DC. Estas definiciones e inicializaciones se realizan antes de la ejecución del programa, sin embargo, debido a que las inicializaciones de los códigos AC eran demasiadas para el compilador no hubo otra solución que realizar las inicializaciones AC dentro del tiempo de ejecución del programa.

Por lo anterior, la primera función a la que llama el módulo principal es a *inichufac.c*, que se encarga de las inicializaciones de las estructuras de códigos que no pudo realizar el compilador.

Una vez que el programa dispone de la imagen original y demás datos necesarios para la compresión se procede a ejecutar los algoritmos descritos en el apartado 4.3:

1. *dct.c*: aplica la Transformada Coseno a una unidad de datos en nuestra imagen original.
2. *qtz.c*: cuantiza los coeficientes DCT entregados por la transformada coseno.
3. *CodDC.c*: codifica el coeficiente DC de la unidad de datos en turno.
4. *CodAC.c*: codifica los coeficientes AC de la unidad de datos en turno.

Estas cuatro funciones se repiten tantas veces como el número de unidades de datos, y el *pipeline* del DSP puede ser aprovechado especialmente en las dos primeras funciones debido a que constan principalmente de operaciones aritméticas inmersas en ciclos y no contienen saltos condicionales, los cuales rompen la continuidad del *pipeline*.

Las flechas en ambas direcciones de los bloques nos indican que después de haberse ejecutado un bloque, regresa la ejecución del programa al bloque de donde surgió la llamada a la función.

Por último, podemos ver que las funciones *CodDC.c* y *CodAC.c* llaman a su vez a dos funciones para poder realizar su fin; *pot2.c* y *zigzag.c*; dichas funciones pudieron haberse incluido dentro de las funciones que las llaman, sin embargo, este proyecto se orientó a la programación modular. Mientras dividamos las tareas complejas a un conjunto de tareas más simples, el código fuente resultante podrá volver a utilizarse o modificarse más fácilmente [14].

Una ventaja de la programación modular en el DSP, como se ha mencionado, es que el ambiente CCS permite utilizar código ensamblador en conjunción con código en lenguaje *C*.

Una vez ejecutadas las cuatro funciones para todas las unidades de datos, la etapa de compresión habrá terminado dando como resultado un arreglo de registros de 16 bits con los códigos de cada unidad de datos concatenados.

4.5 Descripción del algoritmo de recuperación de la imagen implementado en el DSP

La recuperación de una imagen comienza al contar con un arreglo de información comprimida de dicha imagen. En la figura 3.8 se observa que los procesos para obtener una imagen a partir de los datos codificados son los inversos a la compresión.

En primer lugar tenemos la decodificación de los coeficientes DCT cuantizados, tras lo cual se procede a la decuantización de dichos coeficientes para finalizar con la

Transformada Inversa Coseno; la cual nos entregará información aproximada a la información de la imagen original, dependiendo del nivel de cuantización que se haya utilizado. El procesamiento para la recuperación de la imagen se realiza en una unidad de datos siguiendo el orden visto en la figura 4.2.

4.5.1 Lectura de la información de la imagen comprimida

Como se recodará del apartado 3.4.7, el arreglo de información comprimida cuenta con dos tipos de elementos, códigos de longitud variable seguidos de magnitudes representadas en un número variable *SSSS* de bits.

Leyendo el código podemos determinar a *SSSS* en el caso de las diferencias DC y a *RRRR* y *SSSS* en el caso de los coeficientes AC.

La principal característica a tomar en cuenta en la información comprimida es que los códigos y las magnitudes están concatenados, por lo cual es necesario leer el arreglo bit a bit.

Agrupamiento de los códigos según su longitud en bits

Quizá la forma más directa de encontrar en el arreglo de información comprimida una cadena de bits que represente un código, consistiría en comparar cada código con los bits leídos hasta encontrar el código correcto (evidentemente sólo comparando códigos con la misma longitud a los bits leídos).

Sin embargo, *John Miano* propone una opción menos intensiva en iteraciones [5] a cambio de más espacio utilizado en la memoria. Dicha opción consiste primero en comparar dos valores; la cadena de bits leída y el código con mayor valor absoluto posible que conste del mismo número de bits que la cadena leída. Si esta condición se cumple, el código consta de ese número de bits.

El código exacto se obtiene a partir de la diferencia de la cadena de bits leída y el código con menor valor absoluto que está conformado con el mismo número de bits de la cadena de bits leída

Como ejemplo, supongamos que al inicio del arreglo de la imagen comprimida tengamos el siguiente registro de 16 bits:

15															0
1	1	0	1	1	1	1	0	0	0	1	0	1	1	0	0

La información al principio del arreglo corresponderá no sólo a la primera unidad de datos, sino a la primera diferencia entre coeficientes DC codificada de la componente de luminiscencia. Por lo tanto, con los códigos de las diferencias de luminiscencia DC encontrados la tabla A.3 del Anexo A podemos generar una tabla que contenga los

valores mayores y los menores que puede adoptar un código para cada número de bits. La tabla se muestra a continuación:

Número de bits del código	Código con el valor mayor	Código con el valor menor
1	Sin código	Sin código
2	00	00
3	110	010
4	1110	1110
5	11110	11110
6	111110	111110
7	1111110	1111110
8	11111110	11111110
9	111111110	111111110
10	1111111110	1111111110
11	11111111110	11111111110

Tabla 4.3 Valores máximos y mínimos de código para un determinado número de bits

Contando con la tabla 4.3 se inicia la lectura bit a bit del registro. Se puede observar que ningún código consta en este caso de un solo bit, por lo tanto se leerán los dos primeros:

11

Una de las condiciones que presenta el estándar JPEG es que ningún código Huffman puede constar solamente de unos [5]; es decir que el valor anterior $(11)_2$ es un código ilegal, por lo tanto se leerá otro bit.

110

El valor máximo para 3 bits es *110*, y los bits leídos cumplen con la condición de tener un valor igual o menor a ese valor máximo, por lo que la longitud correcta del código en bits ha sido encontrada.

Como se mencionó, el siguiente paso consiste en restarle a la cadena de bits leída el valor menor posible del código, que en nuestro caso corresponde a la siguiente operación

$110 - 010 = 100 = (4)_{10}$

Esta cantidad obtenida se la podemos sumar al valor *SSSS* del primer código que conste de 3 bits y obtener así, el valor de *SSSS* para nuestro código leído

$SSSS = 1 + 4 = 5$

Siendo $SSSS = 5$, sabemos que necesitamos leer 5 bits después de los bits del código para obtener la diferencia DC. Evidentemente, no se necesitan leer bit a bit estos 5 bits,

sin embargo, es necesario leer uno por uno de los bits que le siguen a esta magnitud sin codificar para encontrar el siguiente código.

Para que el algoritmo anterior funcione es necesario que el código Huffman asignado a las categorías *SSSS* y las categorías en sí mantengan un orden ascendente; por ejemplo, al código *010* correspondiente a *SSSS* = 1, le sigue el código *011* correspondiente a *SSSS* = 2 y así sucesivamente.

Si los códigos se asignaran en una manera desordenada sería difícil generar un algoritmo que necesitara de pocas iteraciones para realizarse.

En resumen, para ejecutar el algoritmo anterior necesitamos de cuatro datos:

1. La longitud del código
2. El valor máximo posible para esa longitud de código
3. El valor mínimo posible para esa longitud de código
4. El valor *SSSS* correspondiente al valor mínimo del código

Se eligió almacenar dicha información en estructuras con miembros de longitud en bits; esto con el objetivo de optimizar de nuevo el espacio utilizado de memoria.

Estructuras para la decodificación DC

La definición y declaración de las estructuras para la decodificación de los códigos DC fue de la siguiente manera:

```
struct dchufdc {
        unsigned lgt    :4;   /* Longitud del código */
        unsigned minc   :9;   /* Valor mínimo */
        unsigned maxc   :9;   /* Valor máximo */
        unsigned fv     :4;   /* SSSS del valor mínimo con la misma
                              longitud de código */
                        };

struct dchufdc decodDC[8];
```

En la definición se pueden observar los miembros correspondientes a los cuatro datos necesarios para realizar el algoritmo. Es necesaria una estructura por cada longitud posible en bits de los códigos, por lo que para el caso de la decodificación de las diferencias DC fue necesario declarar un arreglo de ocho estructuras de este tipo.

Tomando en cuenta lo anterior, encontramos que el número de estructuras de códigos necesarias para codificar las diferencias DC son once, tan solo una diferencia de tres estructuras en comparación con las necesarias para la decodificación. Adicionalmente las estructuras de codificación sólo cuentan con dos miembros a diferencia de las de decodificación, que cuentan con cuatro.

Estructuras para la decodificación AC

En el caso de la decodificación de diferencias DC, la ganancia aparente en velocidad comparando 8 en lugar de 11 estructuras resulta pequeña para el espacio de memoria utilizado por las estructuras. Sin embargo, para el caso de la decodificación de coeficientes AC la eficiencia de este algoritmo aumenta considerablemente.

Lo anterior se debe a que la búsqueda del código se realiza por *longitud de código* y que los códigos AC tienden a tener la misma longitud en bits. Esto permite que la declaración de estructuras necesaria para decodificación de coeficientes AC (omitiendo *EOB* y *ZRL*) conste tan solo de un arreglo de 128 estructuras, a diferencia de las 160 estructuras para la codificación. Se optó verificar la ocurrencia de *EOB* y *ZRL* en todos los casos independientemente del número de bits leídos, debido a que su inclusión dentro de los índices de las estructuras sólo traería consigo memoria desperdiciada y dificultaría el diseño del algoritmo.

Es necesario un arreglo de 16×8 estructuras para almacenar las estructuras de decodificación de *SSSS* y que al mismo tiempo genere un índice de longitud de códigos que nos permita recuperar *RRRR*. Sin embargo, debido al diseño del algoritmo de decodificación, sólo es necesario inicializar 50 estructuras completamente y 15 estructuras parcialmente para la decodificación AC. Como resultado de utilizar el algoritmo propuesto por *John Miano* el número de iteraciones necesarias para encontrar un código AC se reduce aproximadamente a un tercio de la búsqueda directa del código.

Con el fin de ver el proceso más detalladamente, se incluyen primeramente la definición de la estructura para la decodificación de ceros consecutivos y de los coeficientes AC:

```
struct dchufac {
        unsigned lgt    :5;     /* Longitud de código */
        unsigned minc   :16;    /* Valor mínimo */
        unsigned maxc   :16;    /* Valor máximo */
        unsigned fv     :4;     /* SSSS del valor mínimo con la misma
                                   longitud de código */
                        };
```

En el código fuente notamos que la estructura incluye los mismos miembros que la estructura para decodificación de las diferencias DC, tan sólo varían las longitudes en bits de dichos miembros. Esta variación, evidentemente es debido a que el código AC con mayor longitud cuenta con 16 bits en comparación a 11 bits del código de mayor longitud DC.

Con los miembros de las estructuras y el algoritmo descrito anteriormente podemos obtener solamente *SSSS*; para obtener *RRRR* se pueden aprovechar los índices inherentes al arreglo de estructuras para la decodificación AC:

- En la tabla A.5 del Anexo A que corresponde a la tabla de códigos AC para la luminiscencia encontramos que *RRRR* puede tomar uno de dieciséis valores (de 0 a 15), y que a su vez, a cada caso de *RRRR* le corresponde 11 casos de *SSSS* para magnitudes de coeficientes AC.
- La búsqueda del código se realiza con base en grupos de códigos de la misma longitud en bits, y el conjunto que cuenta con la colección más amplia de longitudes de bits contiene ocho longitudes de bits.
- Así, se puede aplicar el algoritmo de lectura a los primeros ocho casos de cuando *RRRRR* es igual a cero; de no encontrarse el código, se prosigue con los siguientes casos con *RRRR* igual a uno y así sucesivamente hasta encontrar el código.

Por lo tanto, si declaramos el arreglo de estructuras de decodificación de la siguiente manera:

struct dchufac decodAC[16][8];

El arreglo contará con un índice para los 16 casos de *RRRR* y el otro índice le permitirá almacenar hasta 8 longitudes de bits; por lo que al momento de encontrar el código correspondiente a la magnitud AC se encontrará simultáneamente el valor de *RRRR*. Posteriormente se calcula la diferencia entre el mayor valor posible de los códigos para ese valor de *RRRR* y finalmente, al igual que en la lectura de códigos DC, se suma la diferencia al valor *SSSS* del código con valor mínimo que cuente con una longitud en bits igual al código encontrado.

Ahora bien, se mencionó que no todas las estructuras de decodificación AC se inicializaron y que, adicionalmente, en algunos casos sólo se les asignó un valor a ciertos miembros. La razón de que algunas estructuras no se inicializaran radicó en el hecho de que sólo en el caso en que *RRRR* es igual a cero, se cuenta con 8 longitudes distintas en bits. En los valores de *RRRR* subsecuentes, los grupos de longitudes en bits van disminuyendo hasta llegar a casos en los que sólo se tiene una longitud en bits.

Por lo anterior, sólo se inicializaron completamente las estructuras suficientes para almacenar los grupos de longitudes en bits existentes para cada *RRRR*.

Como elemento para facilitar la implementación del algoritmo de decodificación, si no se utilizaban las ocho estructuras para un determinado valor de *RRRR* se inicializó parcialmente la estructura siguiente a la última estructura inicializada. Al único miembro que se le asignó un valor fue al de longitud en bits, y el valor asignado fue 17; una longitud en bits mayor a la de cualquier código válido, por lo tanto nos permitiría generar un algoritmo de decodificación AC que detectaría con facilidad los incrementos en *RRRR*.

Ejemplificando el párrafo anterior aparece a continuación la inicialización de algunas estructuras para diferentes casos de *RRRR*:

RRRR = 13

```
/* Este caso sólo cuenta con dos longitudes en bits: 11 y 16 */

decodAC[13][0].lgt = 11;          /* Se les asigna un valor a todos los */
decodAC[13][0].minc = 2040;       /* miembros de la estructura */
decodAC[13][0].maxc = 2040;
decodAC[13][0].fv = 1;

decodAC[13][1].lgt = 16;          /* Se les asigna un valor a todos los */
decodAC[13][1].minc = 65506;      /* miembros de la estructura */
decodAC[13][1].maxc = 65514;
decodAC[13][1].fv = 2;

/* Al no completarse 8 estructuras, se inicializa la siguiente estructura
parcialmente con una longitud de código inválida */

decodAC[13][2].lgt = 17;          /* Sólo se asigna el valor al miembro */
                                  /* que almacena la longitud */
```

RRRR = 14

```
/* Este caso solo cuenta con una longitud en bits: 16 */

decodAC[14][0].lgt = 16;          /* Se les asigna un valor a todos los */
decodAC[14][0].minc = 65515;      /* miembros de la estructura */
decodAC[14][0].maxc = 65524;
decodAC[14][0].fv = 1;

/* Al no completarse 8 estructuras, se inicializa la siguiente estructura
parcialmente con una longitud de código inválida *

decodAC[14][1].lgt = 17;          /* Sólo se asigna el valor al miembro */
                                  /* que almacena la longitud */
```

4.5.2 Decodificación de los coeficientes DC

El objetivo de la decodificación de los coeficientes DC consiste en recuperar el primer coeficiente de la unidad de datos en turno que se esté procesando y almacenarlo en el registro *bloque[0]*, es decir, el lugar para el coeficiente DC del arreglo que almacena temporalmente una unidad de datos.

La forma en que se implementó el algoritmo de decodificación DC puede dividirse en tres partes:

ESTA TESIS NO SALE
DE LA BIBLIOTECA

1. Lectura bit a bit del arreglo que contiene la información comprimida de la imagen.
2. Comparación de los bits leídos con los códigos Huffman.
3. Recuperación de las diferencias DC mediante la lectura de los bits sin codificar y su posterior suma al valor del coeficiente DC anterior.

Ahora bien, la forma que se implementó la lectura bit a bit fue mediante a corrimientos del registro en turno. Al iniciar la decodificación contamos con un arreglo de registros de 16 bits *cadcod[N]*, en donde *N* representa el número total de registros del arreglo que cuentan con información comprimida y que depende del nivel de compresión que se haya utilizado. A continuación se presenta el diagrama de flujo de la rutina de lectura de bits:

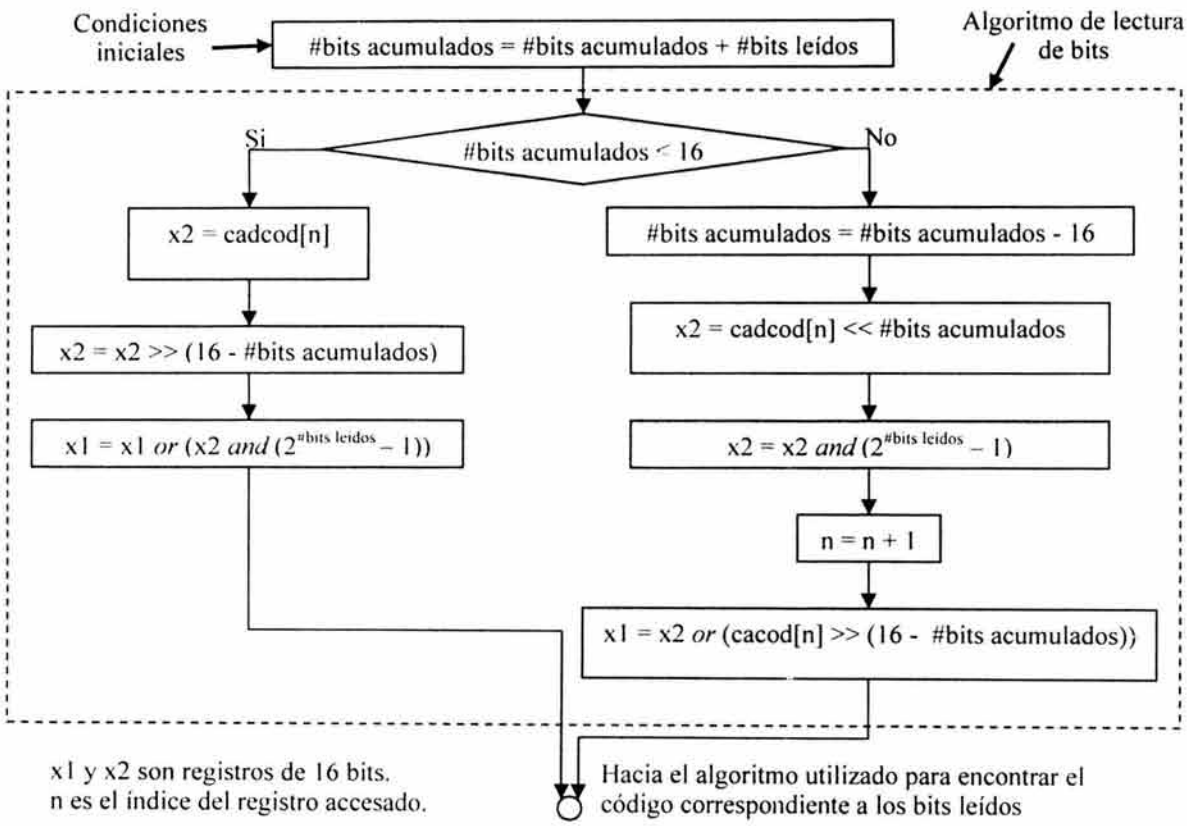


Figura 4.18 Diagrama de flujo del algoritmo de lectura de bits

En la figura 4.18 se observa que previamente al algoritmo de lectura, contenido en un cuadro de líneas punteadas, se encuentran las condiciones iniciales las cuales sólo son válidas para la primera iteración.

El *número de bits acumulados* representa el número de bits del registro *cadcod[n]* que ya han sido decodificados. A esta cantidad se le sumará el *número de bits leídos* que representan los bits a leer para decodificar un nuevo código. Mientras que la suma de estas dos cantidades no sobrepase 16, los bits a decodificar se encontrarán dentro de un

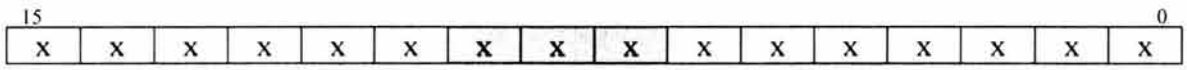
registro. Si la suma es mayor a 16, los bits a decodificar se encontrarán divididos en dos registros, el actual y el siguiente.

Lectura de bits en un solo registro

Para el caso en que los bits a leer se encuentren dentro de un solo registro será necesario ejecutar un corrimiento y una operación *and*. Con el fin de manipular la información del registro en turno, se duplica la información en un registro temporal *x2*.

Supongamos que hemos duplicado la información en un registro *x2* y que necesitamos leer tres bits:

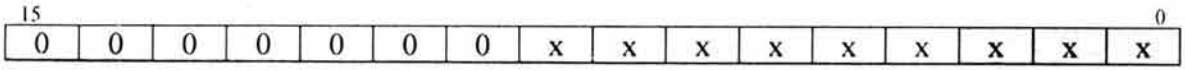
x2



Al registro anterior lo componen bits *x* que pueden ser un uno o un cero. Los bits que queremos recuperar se han sombreado. Los bits que ya han sido decodificados se encuentran a la izquierda de los bits sombreados, y el número de bits acumulados sería la suma de estos bits a la izquierda más los bits sombreados. Como resultado, se tienen 9 bits acumulados.

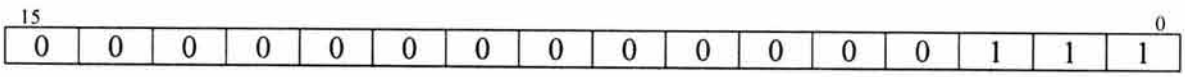
El siguiente bloque de ejecución del algoritmo de la figura 4.13 nos indica que debemos recorrer hacia la derecha (16 - #bits acumulados) bits el registro *x2*, esto es en nuestro ejemplo, 7 bits hacia la derecha (16-9).

x2



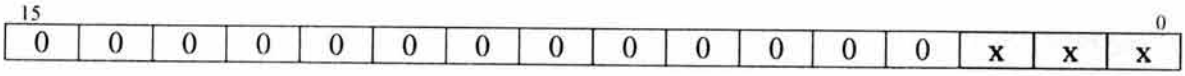
A este registro y a la cantidad ($2^{\#bits\ leidos} - 1$) les aplicamos el operador *and* para depurar los bits que deseemos leer.

$2^3 - 1 = 7 = (111)_2$, y en un registro de 16 bits se representa:



Y como resultado de la operación *and* tenemos:

x2

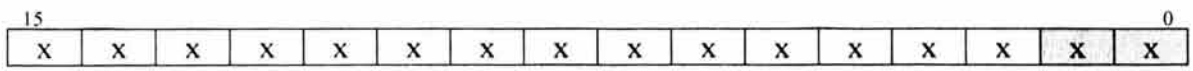


La última operación del último bloque de ejecución de esta rama del algoritmo consiste en duplicar la información final de *x2* en otro registro temporal *x1* el cual se comparará con los códigos Huffman. En caso de que no se encuentre el código será necesario seguir iterando el algoritmo de lectura de bits.

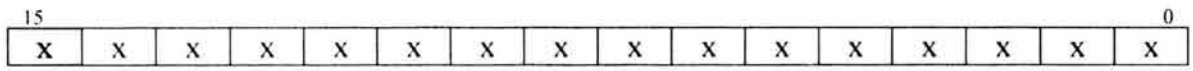
Lectura de bits compartidos entre dos registros

Si en la primera iteración o después de varias iteraciones se llega a la frontera de un registro de 16 bits, el algoritmo de lectura de bits toma otra rama condicional. Supongamos que necesitamos leer tres bits compartidos entre dos registros consecutivos *cadcod[n]* y *cadcod[n+1]*:

cadcod[n]



cadcod[n+1]



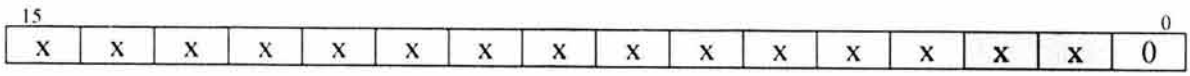
De nuevo para este ejemplo los bits que se desean leer se han sombreado. Encontramos que los bits previamente decodificados se encuentran a la izquierda de los bits sombreados del registro *cadcod[n]*, y que junto con los bits a leer constituyen un total de 17 bits acumulados.

El primer bloque de ejecución para el caso en que el número de bits acumulados es mayor a 16, consiste en sustraer la cantidad 16 al número de bits acumulados. Esta sustracción deja listo el contador de bits acumulados para la siguiente iteración de lectura y al mismo tiempo se aprovecha su resultado para la lectura actual. Así, tenemos que el número de bits acumulados para nuestro ejemplo pasa a ser

 # bits acumulados 17 -16 = 1

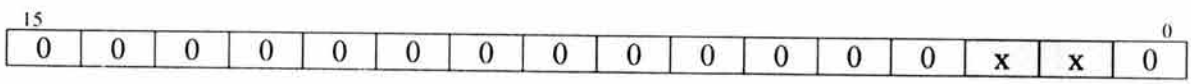
El registro *cadcod[n]* se recorre a la izquierda el número actual de bits acumulados (1 en este caso) y posteriormente se duplica su contenido en *x2*:

x2

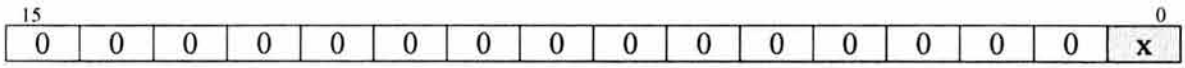


Inmediatamente se aplica la operación *and* entre el registro *x2* y $(2^{\#\text{bits leídos}} - 1)$. De nuevo en este ejemplo la operación resulta ser *x2 and* $(111)_2$, dando como resultado la eliminación de los bits a la izquierda de los bits sombreados.

x2

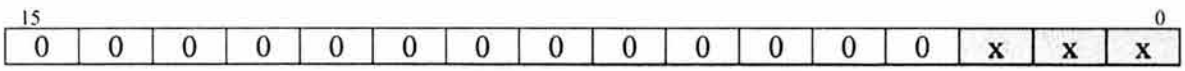


A continuación, se prepara la información del registro *cadcod[n+1]* recorriéndose los bits de este registro (16 - #bits acumulados) lugares a la derecha. Para nuestro ejemplo (16 – 1 = 15) bits a la derecha, dando como resultado lo siguiente:



A este resultado junto con el registro *x2* se les aplica el operador *or*, y el resultado se asigna al registro *x1*:

x1



De nuevo contamos con el registro *x1* para comparar con los códigos Huffman.

La finalidad del registro *x1* no aparece a la vista de manera inmediata; al parecer bien podría utilizarse *x2* para comparar los bits leídos con los códigos Huffman sin importar la rama del algoritmo de lectura que se siguiese. Sin embargo, es necesario almacenar los bits leídos hasta que se encuentre el código correspondiente: la razón reside en el caso que se necesite leer más bits después de haber leído los bits que se encuentren compartidos en dos registros. En cualquier otro caso la asignación de *x1* es innecesaria, pero incluir dicho registro nos evita tener que leer de nuevo el registro anterior o generar una rama en el algoritmo para este caso especial.

Comparación entre los bits leídos y los códigos Huffman DC

Una vez descrito el algoritmo de lectura de bits, se puede proceder a describir la forma en que se compararon de los bits leídos con los códigos Huffman. Se hace mención de la lectura de bits en forma de un bloque de ejecución.

La comparación de bits con códigos puede realizarse mediante un conjunto de bloques condicionales en cascada como los que se muestran en el diagrama de flujo en la figura 4.19.

Al inicio del diagrama de la figura 4.19 encontramos que a la condición inicial para el algoritmo de lectura de bits "*#bits acumulados = #bits acumulados + # bits leídos*", se le agregan otras asignaciones: a saber, se inicializan con el valor cero los valores *x1*, *x2* e *i*, adicionalmente el *número de bits leídos* toma el valor 2. La razón para esta última asignación consiste en que el código más pequeño para las diferencias DC consta de dos bits, y por lo tanto, el menor número de bits a decodificar es dos.

La variable *i* funcionará como índice para las estructuras de decodificación y se inicializa con el valor cero, que es el índice para la primera estructura.

El segundo bloque de ejecución de la figura 4.19 representa el algoritmo de lectura de bits representado en la figura 4.18.

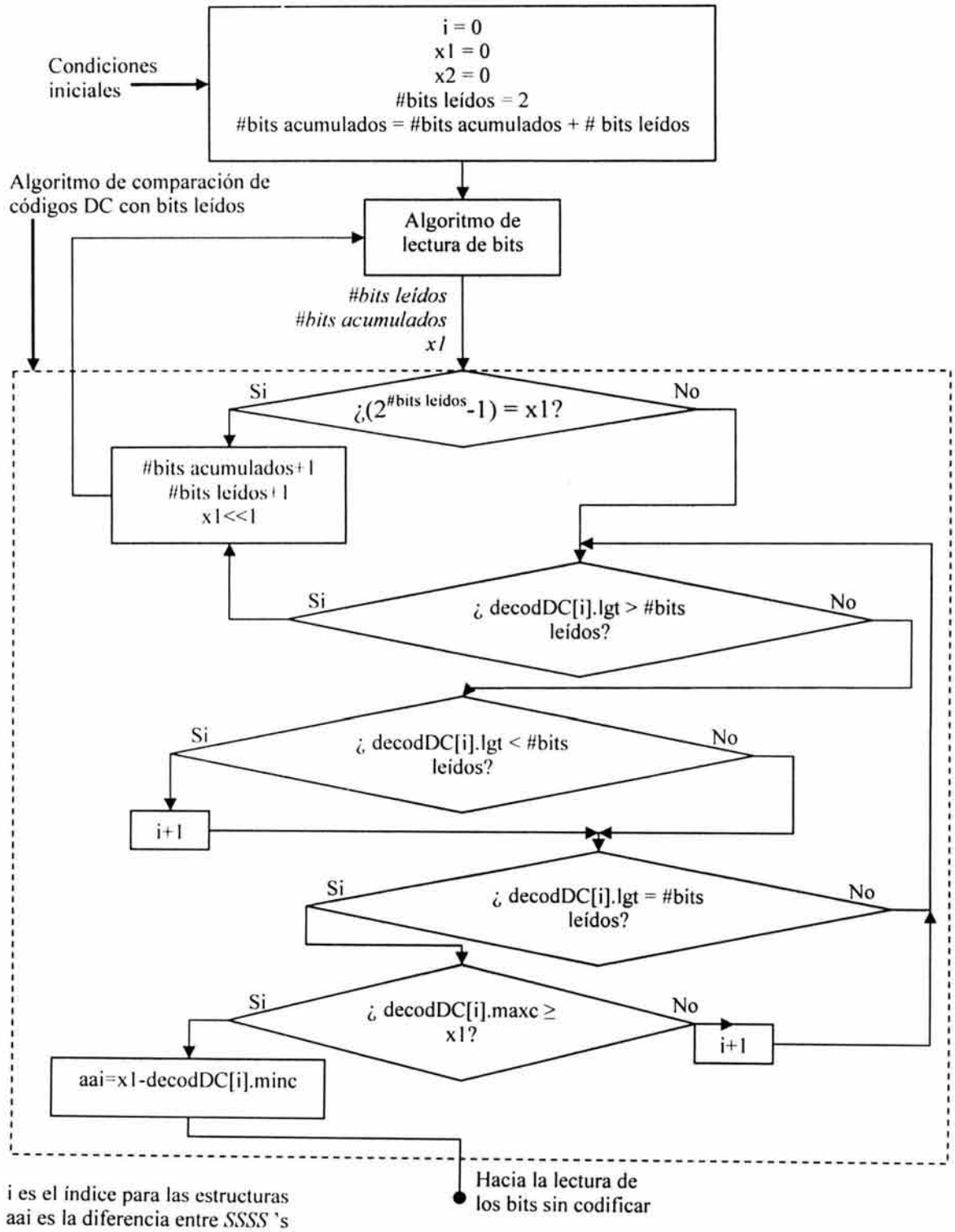


Figura 4.19 Diagrama de flujo del algoritmo para comparar los códigos DC con los bits leídos

La lectura de bits entrega los siguientes valores:

- El número de *bits acumulados*
- El número *bits leídos*
- El registro *x1* conteniendo los bits leídos

El siguiente bloque condicional verifica si los bits leídos encontrados en *x1* constan solamente de unos; como se mencionó anteriormente, no existe ningún código válido compuesto en su totalidad por unos. La prueba del bloque consiste simplemente en estos pasos:

- Se eleva el número dos a una potencia igual a los bits leídos
- Al resultado de dicha potencia se le resta 1
- Se compara la diferencia obtenida con los bits leídos

Como ejemplo supongamos que hemos leído 4 bits, por lo que los bits leídos se compararan con la cantidad

$2^{\#\text{bits leidos}} - 1 = 2^4 - 1 = 16 - 1 = 15 = (1111)_2$

Si los bits leídos son iguales a la cantidad resultante de la resta arriba mencionada, el código de los bits leídos será inválido por constar solamente de unos. En consecuencia, el algoritmo hace un salto hacia un bloque de ejecución donde se preparan las condiciones para que el algoritmo de lectura de bits lea otro bit: se aumenta en uno el número de bits a decodificar (*#bits leídos* y *#bits acumulados*) y se recorre a la izquierda el registro *x1*.

En caso de que los bits leídos no consten solamente de unos, se comparará su longitud en bits con los bits del código de la estructura de decodificación en turno, indicada por la variable *i*. Si el código de la estructura de decodificación cuenta con más bits que los bits leídos, se procederá a leer otro bit, de lo contrario el proceso continúa con otra verificación.

La siguiente prueba determina si el código a comparar consta de menos bits que los bits leídos. En caso de ser afirmativo, el índice de las estructuras aumenta en uno. En el caso opuesto, la ejecución del algoritmo pasa directamente a la última prueba de longitud en bits, en la que se determina si el número bits leídos corresponden al número de bits del código de la estructura de decodificación de ese instante (decodDC[i].lgt). Si la condición anterior no se cumple, el algoritmo salta al principio de las comparaciones de longitud en bits. Si la condición se cumple, la ejecución llega a su último bloque condicional.

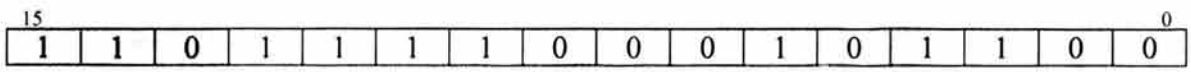
Como se mencionó en el apartado 4.5.1, se comparan los bits leídos *x1* con el valor máximo válido para un código de determinada cantidad de bits *decodDC[i].maxc*; en caso que el código sea mayor o igual a *x1*, se ejecutará un bloque para obtener la diferencia de *SSSS* 's, de lo contrario se hará un salto hacia el inicio de las pruebas de longitud en bits.

La diferencia entre *SSSS* 's se obtiene mediante la sustracción a *x1* del código con el valor mínimo posible para la estructura de decodificación en turno *decodDC[i].minc*. El resultado se almacena en una variable de nombre arbitrario *aai*.

La variable *aai* es necesaria para obtener la longitud en bits *SSSS*. Esta longitud dicta los bits sin codificar a leer para recuperar el valor de la diferencia DC.

Recuperación de la diferencia entre coeficientes DC

Tomemos como ejemplo de nuevo el registro de 16 bits utilizado para explicar la lectura bit a bit en el apartado 4.5.1:



Los bits leídos se encuentran sombreados y corresponden al valor almacenado en *x1*.

La estructura con índice $i = 2$ corresponde a los códigos de tres bits. El valor máximo que puede guardar un código de tres bits para la decodificación DC está dado por el miembro $decodDC[2].maxc = 6 = (110)_2$.

Debido a que los bits leídos son iguales o menores al miembro mencionado, se ha encontrado un código válido. Para encontrar el valor *SSSS* del código es necesario obtener *aai* mediante la siguiente operación:

$$x1 - decodDC[2].minc = aai$$

Y sustituyendo los valores correspondientes tenemos:

$$110 - 010 = 100 = (4)_{10}$$

Sumamos *aai* al valor *SSSS* del primer código que conste de 3 bits y obtener así, el valor de *SSSS* para nuestro código leído:

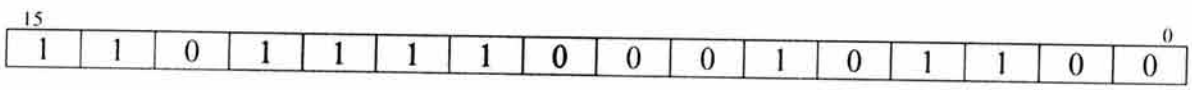
$$aai + decodDC[2].fv = SSSS$$

o bien,

$$SSSS = 1 + 4 = 5$$

Dado lo anterior, los bits que contienen la diferencia DC son los 5 bits que siguen a los bits recién decodificados:

cadcod[n]



En los procedimientos de lectura hemos visto cómo se pueden separar los bits sombreados mediante operaciones *and* y corrimientos, por lo cual se obviará la descripción de estos procedimientos.

Las diferencias a leer se almacenan en un registro temporal *x2*:

x2

15															0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0

La manera de determinar si la diferencia es positiva o negativa con respecto al coeficiente DC anterior consiste en determinar si el bits más significativo de los *SSSS* bits es uno o cero. Si la diferencia es positiva el bit más significativo será uno, en caso de ser una diferencia negativa el bit más significativo será cero.

La manera en que se implementó dicha prueba consistió simplemente en realizar en primera instancia un corrimiento del registro *x2* hacia la derecha de un valor igual a *(SSSS bits – 1 bit)*.

En nuestro ejemplo, el corrimiento consiste en (5 bits -1 bit) 4 bits hacia la derecha, dándonos como resultado un registro temporal como el siguiente:

15															0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Con este registro temporal es sencillo determinar si la diferencia es positiva o negativa; basta sólo con un bloque condicional que verifique si el registro es igual a uno o cero.

Si la diferencia fuese positiva, los *SSSS* bits (5 en nuestro caso) se suman directamente al valor del coeficiente DC anterior para encontrar el coeficiente DC actual a decodificar:

Coeficiente DC anterior + x2 = Coeficiente DC actual

El registro de muestra *cadcod[n]* de nuestro ejemplo es en realidad el primer registro para una de las imágenes procesadas en este proyecto, el coeficiente a decodificar es el primer coeficiente DC de la imagen y por ende, el coeficiente anterior tiene un valor igual a cero.

Para terminar el ejemplo, tendríamos entonces como coeficiente decodificado:

$Coeficiente\ actual = (11110)_2 + 0 = 30$

Sin embargo, si la diferencia es negativa, los *SSSS* bits se encuentran en complemento a uno, por lo que será necesario primero aplicarle de nuevo complemento a uno a *x2* para posteriormente sólo conservar los *SSSS* bits y restar el resultado de éstas dos operaciones al coeficiente anterior DC. En síntesis, las operaciones aplicadas a *x2* para recuperar el coeficiente DC se pueden resumir en la siguiente expresión:

$Coeficiente\ DC\ anterior - ((complemento\ a\ uno(x2))\ and\ (2^{SSSS} - 1)) = Coeficiente\ DC\ actual$

El algoritmo de la recuperación del coeficiente DC a partir del valor *aai* se presenta en la figura 4.20.

El diagrama de flujo de dicha figura inicia con una serie de valores obtenidos dentro de la fase de comparación de los bits leídos con los códigos; entre dichos valores encontramos a *aai*, el índice de las estructuras *i*, el índice *n* para arreglo de registros de la imagen comprimida y el número de bits de *cadcod[n]* que han sido decodificados (*bits acumulados*).

Un primer bloque de ejecución obtiene el valor de *SSSS* a partir del valor *aai* y el miembro de la estructura de decodificación en turno *decodDC[i].fv*; como mencionamos, el miembro de la estructura contiene el valor mínimo que puede tener un código para el número de bits que se han leído, y al sumarse a *aai* da como resultado *SSSS* para el código leído.

El algoritmo se divide entonces en dos casos generales, un bloque condicional pregunta si el valor *SSSS* es igual a cero.

Si se cumple la condición anterior, sabremos que la diferencia DC no necesitó de ningún bit adicional al código Huffman para codificarse, y el único caso en que esto se cumple es cuando la diferencia codificada fue igual a cero. Siendo así, sólo se asigna el valor del coeficiente DC anterior al coeficiente DC actual *bloque[0]* y el algoritmo finaliza.

Cuando los *SSSS* bits adicionales al código Huffman son diferentes a cero, es necesario leer dichos *SSSS* bits que se encuentran después de los bits que ya han sido decodificados. Por ello, se siguen los procedimientos que ya hemos mencionado en los otros dos pasos de la decodificación para la lectura y separación de los bits a trabajar, es decir, corrimientos y operaciones *and* y *or*.

Una vez que los *SSSS* bits que representan a la diferencia se han almacenado en el registro *x2*, el último bloque condicional verifica si la diferencia es negativa o positiva. Si es negativa, un bloque de ejecución le aplica al registro *x2* la operación complemento a uno y depura los bits de la diferencia mediante una operación *and*. Como resultado se obtiene el valor absoluto de la diferencia y se vuelve a almacenar en *x2*; así se puede substraer *x2* al valor del coeficiente DC anterior y almacenarlo en su lugar correspondiente en la MCU, es decir, *bloque[0]*.

Si la *x2* fue un valor positivo, simplemente se suma al valor del coeficiente anterior y se almacena en *bloque[0]*.

Al final del algoritmo, se asigna el valor del *bloque[0]* al espacio de memoria reservado para almacenar el valor del código DC anterior, preparando así los datos para la siguiente iteración de la decodificación de coeficientes DC.

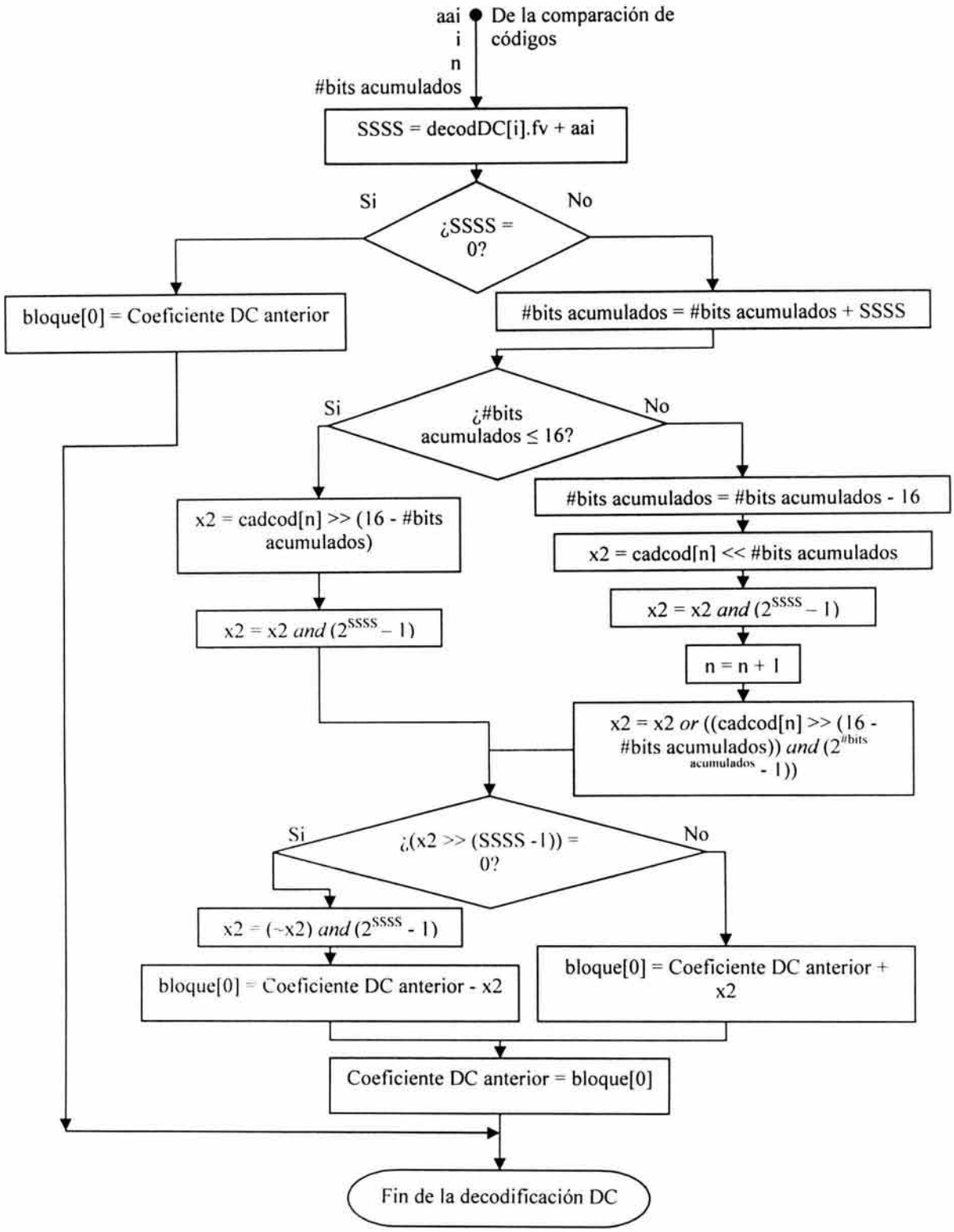


Figura 4.20 Diagrama de flujo de la recuperación de coeficientes DC

4.5.3 Decodificación de los coeficientes AC

Habiéndose recuperado el coeficiente DC y ocupado su lugar en *bloque[0]*, se procede a decodificar más bits con el fin de completar una unidad de datos.

El algoritmo para la decodificación DC es muy similar al de la decodificación DC; incluye la lectura de bits, su comparación con el código y la recuperación de un coeficiente diferente de cero a partir de los bits sin codificar encontrados después del código Huffman.

Comparación entre los bits leídos y los códigos Huffman AC

El algoritmo de lectura de bits es idéntico al utilizado en la decodificación DC y que se muestra en la figura 4.18; entrega, de la misma manera que en la decodificación DC, un registro *xl* conteniendo los bits leídos del registro de la imagen comprimida.

Sin embargo, durante la comparación de los bits leídos con los códigos AC aparecen un par de discrepancias: la comparación es necesaria hacerla a un arreglo bidimensional de estructuras que guardan los códigos y se incluye un bloque condicional para detectar los casos en que aparece *EOB* o *ZRL*.

Para una mejor apreciación del proceso, dentro de la figura 4.21 se muestra el diagrama de flujo del algoritmo de comparación de códigos con los bits leídos.

Encontramos al inicio del algoritmo que se inicializan variables adicionales a las que se encontraban en la decodificación DC; se aumenta un índice *j* para poder acceder correctamente al arreglo bidimensional de estructuras y se incluyen un par de banderas *eob* y *zrl* para los casos en que se detecten *EOB* y *ZRL* respectivamente.

Se realiza la lectura de bits y el registro *xl* queda listo para compararse con los códigos Huffman para los coeficientes AC.

De la misma manera que en la decodificación DC, el primer bloque condicional verifica que los bits leídos no consten solamente de unos y, por ende, conformen un código inválido. De ser inválido el código, el algoritmo lee otro bit.

Si los bits leídos no están compuestos solamente por unos, se compara el valor con los valores 10 y 2041. Estos valores en su representación binaria corresponden a los códigos de *EOB* y *ZRL* respectivamente. De encontrarse una correspondencia entre los bits leídos y estos códigos, no son necesarias más pruebas y el proceso tiene un salto hacia la última parte del algoritmo de decodificación de coeficientes AC.

Ahora bien, si los bits leídos no corresponden a *EOB* ni a *ZRL*, el algoritmo continúa con una serie de bloques condicionales en cascada que descartan y aumentan los índices *i* y *j* hasta que se compara una estructura de decodificación cuyos miembros tengan la misma longitud en bits que los bits leídos.

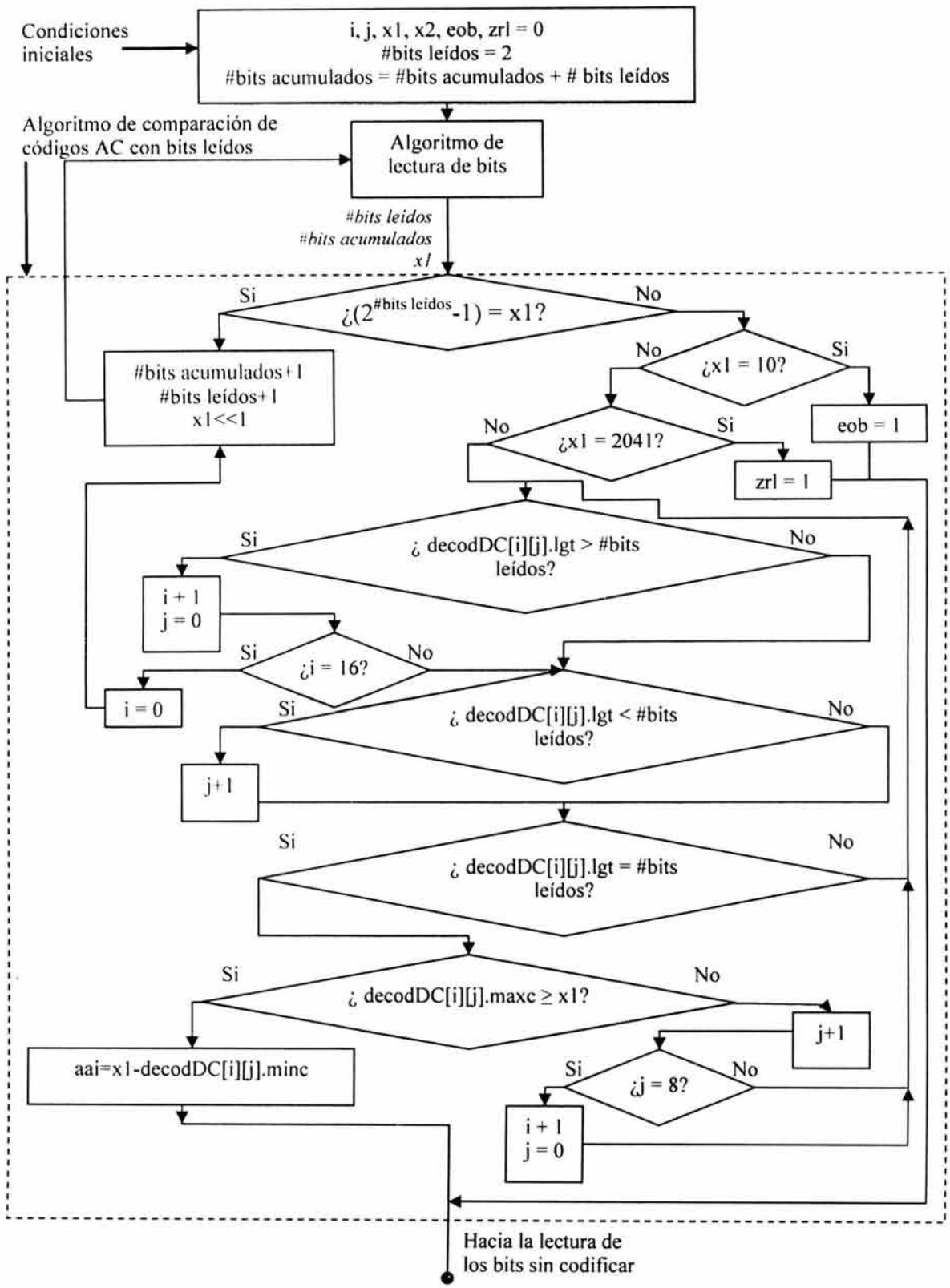


Figura 4.21 Diagrama de flujo del algoritmo de comparación de códigos AC con los bits leídos

Al cumplirse la condición anterior, se verifica si el valor mayor de un código contenido en la estructura en turno es mayor o igual a los bits leídos: de ser así, se calcula el valor *aai* con el fin de obtener *SSSS*, y el valor de *RRRR* estará dado directamente por el valor actual del índice *i*.

Si los bits leídos no pertenecen al grupo de códigos contenidos por la estructura de decodificación en turno, se aumentarán los subíndices.

Es prudente mencionar que el subíndice *i* sólo toma valores del 0 a 15 ya que representa a *RRRR*, y que *j* varía entre 0 y 7 dado el diseño de las estructuras de decodificación mencionado en el apartado 4.5.1.

Una vez contando con la diferencia *aai*, o bien si una de las banderas *EOB* o *ZRL* tomó el valor de uno, el algoritmo se torna hacia el proceso de ir almacenando los coeficientes AC dentro del arreglo *bloque*.

Algoritmo general para la decodificación AC

En la figura 4.22 se puede observar el diagrama de flujo general de la decodificación de coeficientes AC.

En primera instancia encontramos al algoritmo de lectura de bits, y a éste le sigue el algoritmo de comparación de los códigos AC con los bits leídos. Si el código encontrado corresponde al caso *EOB* a todos los elementos restantes del arreglo *bloque* se les asigna el valor cero. En el caso contrario un bloque condicional pregunta si el código encontrado correspondió al caso de *ZRL*; de cumplirse *ZRL*, se asigna a los 16 elementos siguientes de *bloque* el valor cero para posteriormente regresar al algoritmo de lectura.

Si el código no fue el de *ZRL*, se asigna a los *RRRR* elementos de *bloque* el valor cero. Posteriormente se obtiene el valor de un coeficiente diferente de cero a partir de bits adicionales a los bits del código leído y se almacena en el lugar en turno de *bloque*.

Al momento de asignarse un valor al último elemento de la unidad de datos, ya sea resultado de la recuperación de un coeficiente diferente de cero, o bien consecuencia del rellenado de ceros al encontrarse *EOB*, se ejecuta un proceso para reordenar a los elementos de *bloque* en su orden original. Lo anterior se debe a que al momento de recuperarse los coeficientes de la unidad de datos, estos coeficientes se encuentran en orden zig-zag.

Una vez reordenados los coeficientes AC, termina el proceso de decodificación para una unidad de datos.

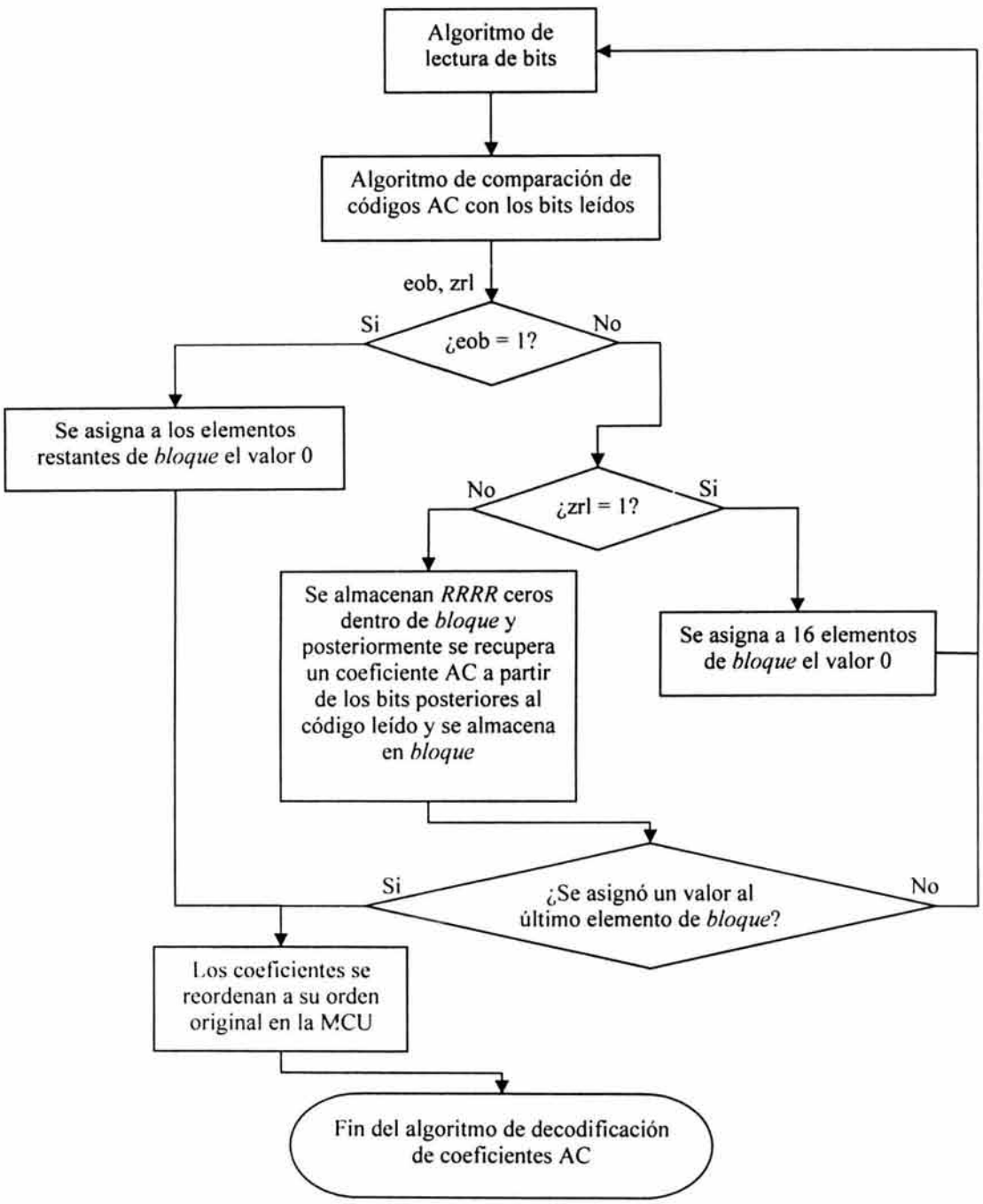


Figura 4.22 Diagrama de flujo del algoritmo general de la decodificación de coeficientes AC

Recuperación de los coeficientes AC

El algoritmo de recuperación de un coeficiente AC diferente de cero es casi idéntico al de recuperación de un coeficiente DC, asimismo el reordenamiento de coeficientes es muy similar al proceso de ordenamiento zig-zag. Sin embargo, es prudente aclarar ambos procesos.

En la figura 4.22 encontramos un bloque en donde se almacenan *RRRR* ceros en el arreglo *bloque*, y posteriormente se recupera un coeficiente distinto de cero para igualmente almacenarse. Estos procesos se ilustran más a detalle en la figura 4.23.

El diagrama de flujo de la figura 4.23 recupera coeficientes mientras no se haya presentado *EOB* ni *ZRL*. Aparecen dos variables nuevas: ***run*** y ***k***. La primera variable sirve como contador para asignar *RRRR* ceros al arreglo ***bloque***, y la segunda es una variable que sirve de índice para los elementos del mismo arreglo ***bloque***.

En el primer bloque de ejecución se asigna el valor de la variable *i* a la variable *run*; como se había mencionado, el índice *i* de la estructura que contiene el código es igual a la cantidad *RRRR* de ceros antes del coeficiente distinto de cero.

Se obtiene el valor de *SSSS* sumando los elementos *decodAC[i][j].fv* y *aai*, posteriormente mediante un bloque de ejecución y uno condicional se almacenan en *bloque* los *RRRR* ceros antes del bit diferente de cero. El algoritmo sigue con una serie de procesos idénticos a los utilizados en la decodificación DC para recuperar un coeficiente distinto de cero hasta el penúltimo bloque condicional.

A diferencia de la decodificación DC, los valores recuperados a partir de los bits posteriores al código constituyen los coeficientes en sí y no una diferencia. Al final del diagrama encontramos un bloque condicional; si se ha asignado un valor al último elemento de *bloque* ($k = 63$), el proceso se torna hacia el reordenamiento de la unidad de datos, de lo contrario se hace un salto hacia el algoritmo de lectura de bits para seguir decodificando coeficientes AC.

El algoritmo de reordenamiento de coeficientes en una unidad de datos sigue los mismos procesos que el algoritmo de ordenamiento zig-zag; sólo se intercambian los índices entre los arreglos *bloque* y *bloquez*. Adicionalmente en lugar de utilizar un arreglo temporal con nombre *bloquez* se usó uno llamado ***bloquedz*** para denotar el proceso inverso. El intercambio de índices se puede observar en los códigos fuente de las funciones *zigzag.c* y *dzigzag.c* incluídos en el Anexo C.

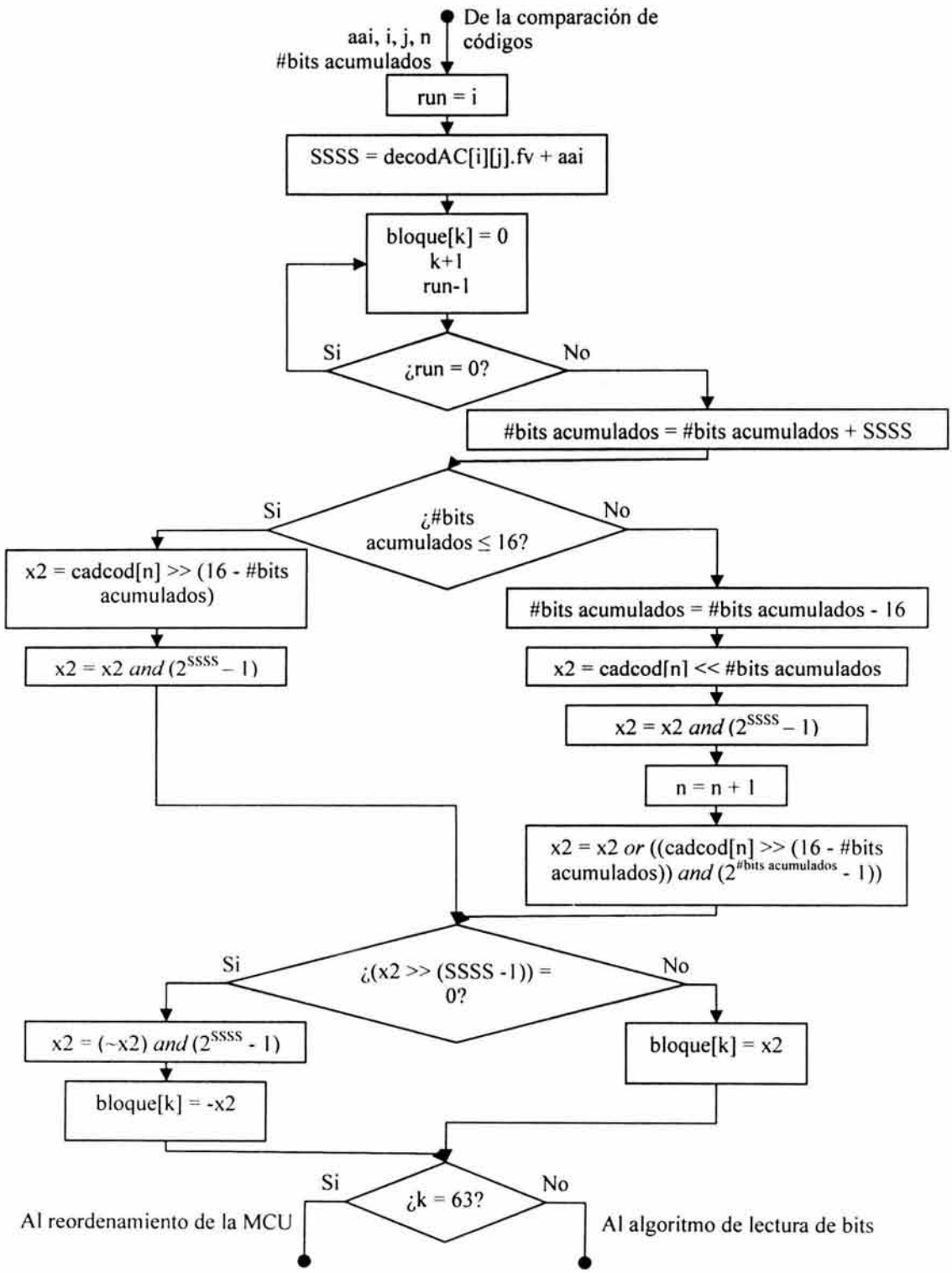


Figura 4.23 Diagrama de flujo del algoritmo de recuperación de coeficientes AC

4.5.4 Decuantización de los coeficientes DCT

Al momento de haberse decodificado una unidad de datos y haberse reordenado en su orden original, la unidad de datos sufre un proceso inverso al de la cuantización. La ecuación implementada se puede ver en la ecuación 4.4:

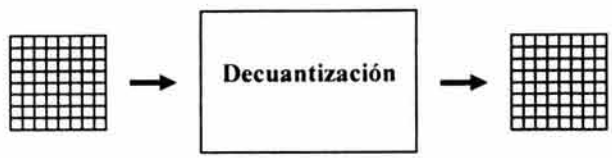
$$S_{vu} = Sq_{vudiv} \times \left(\frac{Q_{vu}}{div} \right) \qquad (4.4)$$

donde:

- S_{vu} es un coeficiente DCT
- Q_{vu} es el elemento cuantizador
- Sq_{vudiv} es el coeficiente cuantizado modificado por *div*
- *div* es el número que modifica el nivel de cuantización de la unidad de datos.

Los coeficientes decodificados son coeficientes cuantizados; el efecto que tiene la decuantización sobre la unidad de datos es recuperar un valor aproximado de aquellos coeficientes que no tomaron valor de cero al momento de la cuantización.

La representación de la decuantización se puede observar en la figura 4.24:



bloque
Arreglo de
números enteros

bloque_t
Arreglo de
números reales

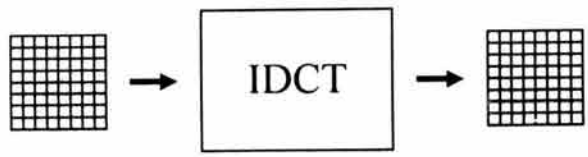
Figura 4.24 Representación del proceso de decuantización

A cada elemento de una unidad de datos almacenada en el arreglo de números enteros *bloque*, se le aplica la ecuación 4.2 y los resultados se almacenan en el arreglo de números reales *bloque_t*. La tabla de números divisores para cada elemento es la misma que se utilizó en el proceso de cuantización.

4.5.5 Aplicación de la Transformada Coseno Inversa a las unidades de datos

El último proceso en la recuperación de la imagen consiste en obtener los píxeles aproximados de la imagen original a partir de la transformación inversa de los coeficientes DCT.

La figura 4.25 muestra una representación del proceso de la Transformada Inversa Coseno para una unidad de datos:



bloque_t
Arreglo de
números reales

bloque
Arreglo de
números enteros

Figura 4.25 Representación del proceso de la IDCT

Los coeficientes DCT recuperados se encuentran en el arreglo de números reales *bloque_t*, y se les aplica la definición de IDCT en 1-D descrita en la ecuación 3.6 con el fin de obtener los valores aproximados de las muestras de la imagen original. Las muestras recuperadas se almacenan el arreglo de 64 números enteros *bloque*.

Si consideramos al arreglo de funciones *basisf* utilizado en la DCT como una matriz, y se desarrolla la ecuación 3.6, se verá que los valores de las funciones base para la IDCT serán iguales a la matriz *basisf* transpuesta. Dado lo anterior, no es necesario tener otro arreglo con los valores de las funciones base para la IDCT, bastará con cambiar los índices del arreglo utilizados en el código:

DCT
basisf[i][j] ⟶ IDCT
basisf[j][i]

Como resultado obtenemos la siguiente ecuación:

$$s(x) = \sum_{u=0}^{7} S(u) \underbrace{\left[\frac{C(u)}{2} \cos\left[(2x+1)u\pi/16\right] \right]}_{\substack{elemento \\ basisf \\ transpuesto}} \tag{4.5}$$

Otra diferencia entre los procesos de la IDCT y la DCT, es que en la primera será necesario utilizar el arreglo vectorial *vectort* tanto en las columnas como en los renglones. La razón para ello es que no podemos almacenar los resultados de la primera IDCT unidimensional en el arreglo de números enteros *bloque*, ya que perderíamos información de la imagen al eliminar los números decimales necesarios para la segunda IDCT unidimensional. Sólo hasta finalizar el proceso de la IDCT es que se deben redondear las muestras para poder almacenarlas en *bloque*.

Posterior al redondeo de las muestras se incluye una rutina diseñada para corregir los desbordamientos probables (*overflows*) en los valores posibles de la imagen provocados por el redondeo. Los valores de las muestras se habían normalizado al intervalo de valores comprendido entre -128 y 127. Por consiguiente, cualquier número que resulte mayor a 127 por el redondeo se sustituirá por 127, y cualquier número menor a -128 se sustituirá por -128.

4.5.6 Escritura de las unidades de datos recuperadas en un arreglo de salida

Todos los procesos necesarios para la decodificación de una unidad de datos se encuentran inscritos dentro de una serie de ciclos. Los ciclos son idénticos a los necesarios para la lectura de datos en la codificación como se puede ver a continuación:

```
for(l=0;l<32;l++)
{
    for(k=0;k<32;k++)
    {
/* Espacio para la decodificación */

/* Inicia escritura de datos al arreglo de salida */
        for(j=0, m=0;j<8;j++)
        {
                for(i=0;i<8;i++)
                {
/* De-normalización de las muestras y almacenado de las mismas en el arreglo de salida
"recuperada" */

        recuperada[(l*2048)+(8*k)+(256*j)+i]=((unsigned char)(bloque[m]+128));
                m++;
                }
        }
    }
}
```

Los aumentos en los índices utilizados en el código anterior representan los mismos cambios descritos en el código encontrado en el apartado 4.3.2 para la lectura de unidades de datos.

Podemos notar que a la decodificación de una unidad de datos le sigue la de-normalización de las muestras de la unidad y su colocación en el espacio destinado para ella en el arreglo de salida *recuperada*. Se puede observar que se de-normalizan las unidades de datos simplemente sumándole el valor 128 a cada muestra de la unidad.

Al término del último ciclo se contará con la información recuperada de una componente de color de la imagen dentro del arreglo *recuperada* y con ello terminará el proceso de decodificación.

Tanto en la codificación como en la decodificación, el proceso para cada componente de color es el mismo, sólo cambian las tablas de cuantización y de codificación, por lo cual para procesar una imagen a color tan sólo es necesario repetir todos los pasos para cada componente.

4.6 Módulos programados en el DSP para la descompresión de imágenes

Los módulos de programación y las interrelaciones entre los mismos se pueden ilustrar mediante la figura 4.26.

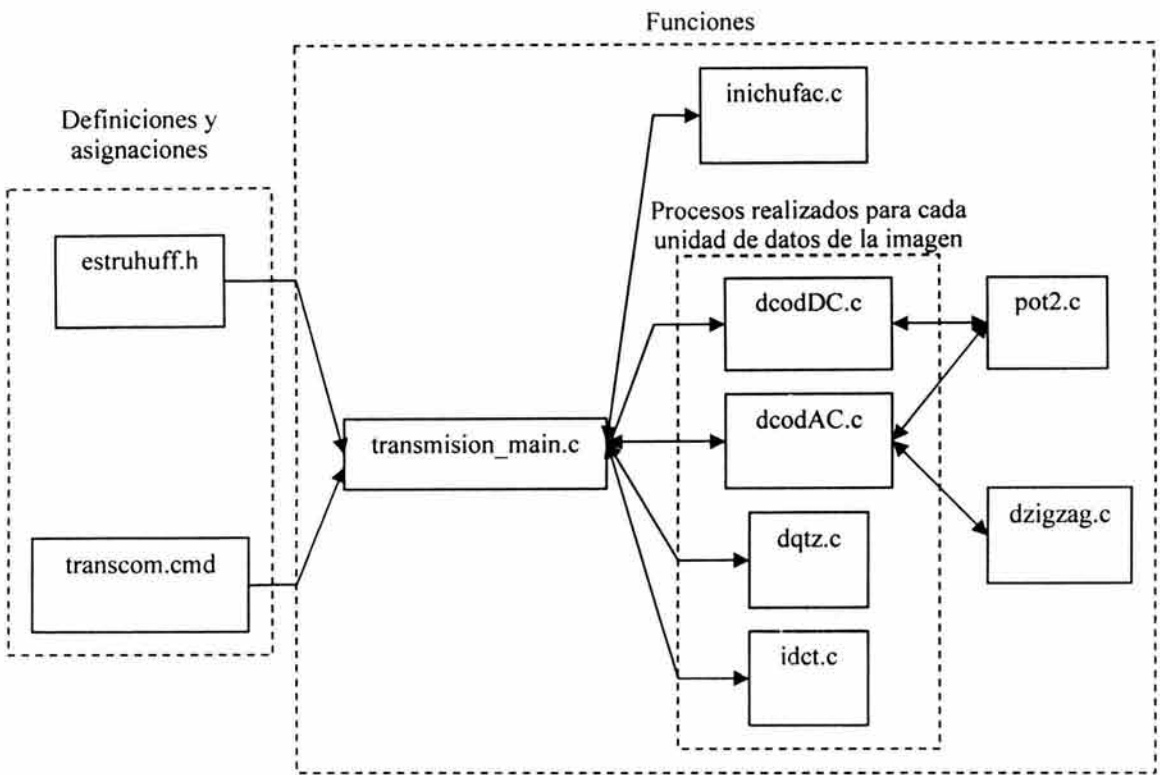


Figura 4.26 Relaciones entre los módulos programados en la decodificación de la imagen

Dentro de la decodificación se utilizan definiciones y asignaciones dadas por los siguientes archivos:

- *estruhuff.h*
  Contiene las definiciones para las estructuras de decodificación

- *transcom.cmd*
  Contiene las asignaciones de memoria del DSP para el almacenamiento de la imagen recuperada

Al terminar la codificación de la imagen, el programa principal *transmisión_main.c* se torna a la recuperación de la imagen llamando a distintas funciones que decodifican el código en unidades de datos y al final almacena cada unidad de datos en su destino final.

Previamente a la decodificación, al inicio de la ejecución, la función *inichufac.c* inicializa las estructuras de decodificación.

Para recuperar cada unidad de datos, la función principal llama ciertas funciones en el siguiente orden:

1. *dcodDC.c*: decodifica el coeficiente DC de una unidad de datos. A su vez llama a la función *pot2.c* para cumplir su cometido.
2. *dcodAC*.c: decodifica los coeficientes AC de una unidad de datos. Utiliza y llama a las funciones *pot2.c* y *dzigzag*. Deja lista la unidad de datos completa para los siguientes procesos.
3. *dqtz.c*: decuantiza los coeficientes DCT de la unidad de datos recién decodificada.
4. *idct.c*: aplica la IDCT a la unidad de datos dando como resultado muestras recuperadas de la imagen.

Estas 4 funciones se aplican para cada unidad de datos.

4.7 Implementación de la DCT optimizada

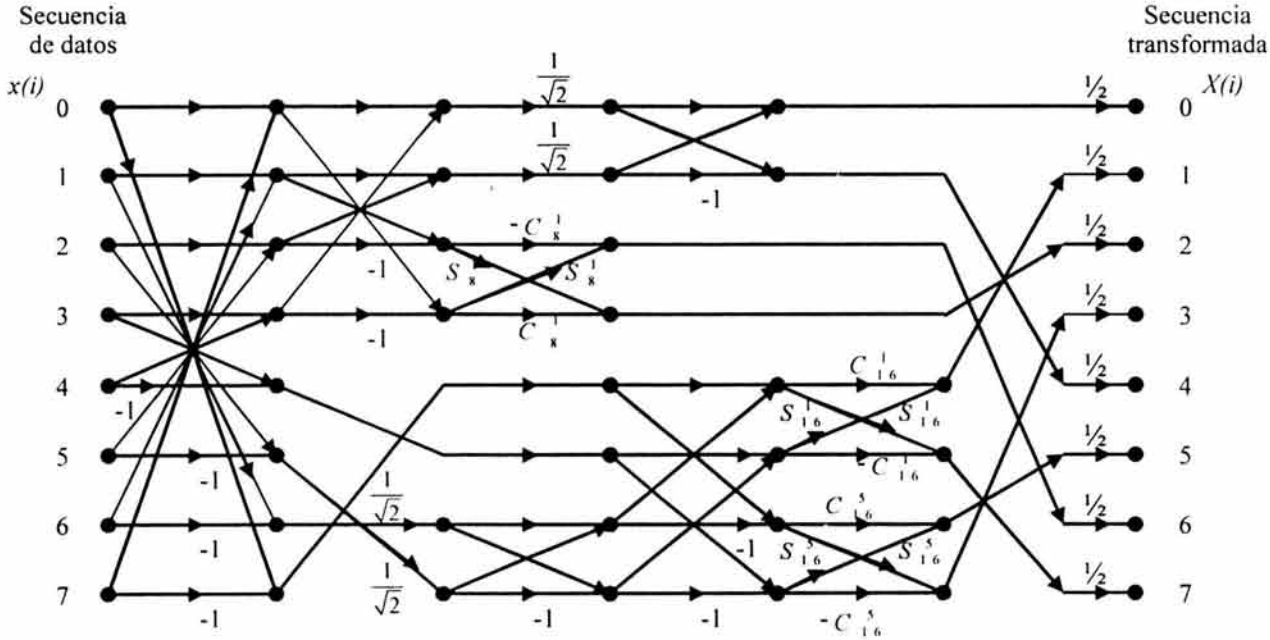
Como se mencionó en el apartado 3.3.6, debido a que la definición en 1-D de la transformada Coseno es muy intensiva en operaciones, se han diseñado formas optimizadas de la Transformada Coseno.

Este apartado contiene la descripción de algoritmos rápidos de la DCT e IDCT que suplantaron a las funciones *dct.c* e *idct.c* originalmente utilizadas. Gracias a la programación en módulos utilizada, dicha substitución se realizó con cambios mínimos al código principal (*main*) del programa.

Se eligió arbitrariamente implementar una optimización desarrollada por Wang, Suehiro y Hatori [5]. Ésta consiste en la factorización de los elementos de la matriz original de la Transformada Coseno (ec. 3.1) en pequeñas matrices con pocos elementos diferentes de cero. Dicha optimización, consta de una DCT e IDCT en 1-D para ocho elementos, por lo que se aplicó a cada renglón y posteriormente a cada columna para obtener la transformada de una unidad de datos.

Los diagramas de flujo para la DCT y la IDCT se muestran en las figuras 4.27 y 4.28 respectivamente.

En la figura 4.27, entran al proceso ocho muestras y salen ocho coeficientes DCT. Las flechas indican la dirección en que fluye la información, y en el caso de que las flechas estén modificadas por un factor, la información se multiplicará por ese factor antes de llegar a un nodo. Cuando dos cantidades se encuentran en un nodo, éstas se suman.



donde

$C_8^1 = \cos(\pi/8)$ $\quad C_{16}^1 = \cos(\pi/16)$ $\quad C_{16}^5 = \cos(5\pi/16)$

$S_8^1 = \text{sen}(\pi/8)$ $\quad S_{16}^1 = \text{sen}(\pi/16)$ $\quad S_{16}^5 = \text{sen}(5\pi/16)$

Figura 4.27 Diagrama de flujo para la optimización de Wang, Suehiro y Hatori de la DCT

El diagrama de la figura 4.27 cuenta con 26 sumas y 34 multiplicaciones. El algoritmo se implementó en *forma directa*, es decir, todas las sumas y multiplicaciones aparecen literalmente en el código fuente. Por lo anterior, fueron necesarias 416 sumas y 544 multiplicaciones para calcular la Transformada Coseno de una unidad de datos. Comparando el número de operaciones necesarias al utilizar la definición unidimensional de la DCT (apartado 3.3.6), notamos que esta optimización implementada disminuye aproximadamente a la mitad el número de sumas y de multiplicaciones en el cálculo de la transformada.

Apreciamos asimismo en el diagrama de la figura 4.27 que los valores de las funciones base *basisf*, utilizados en la implementación basada en la definición de la DCT, ya no son necesarios. Sin embargo, se precisan otras constantes para el cálculo de la DCT; es por ello que se suplantó la declaración de *basisf* dentro del archivo *main* por la declaración de los senos y cosenos usados en la optimización.

Al igual que en la primera implementación, la función que incluye este algoritmo (*dct.c*), toma una unidad de datos almacenada en *bloque* y almacena los coeficientes resultantes en *bloque_t*.

En la figura 4.28 se muestra el diagrama de flujo para el algoritmo optimizado de la Transformada Coseno Inversa.

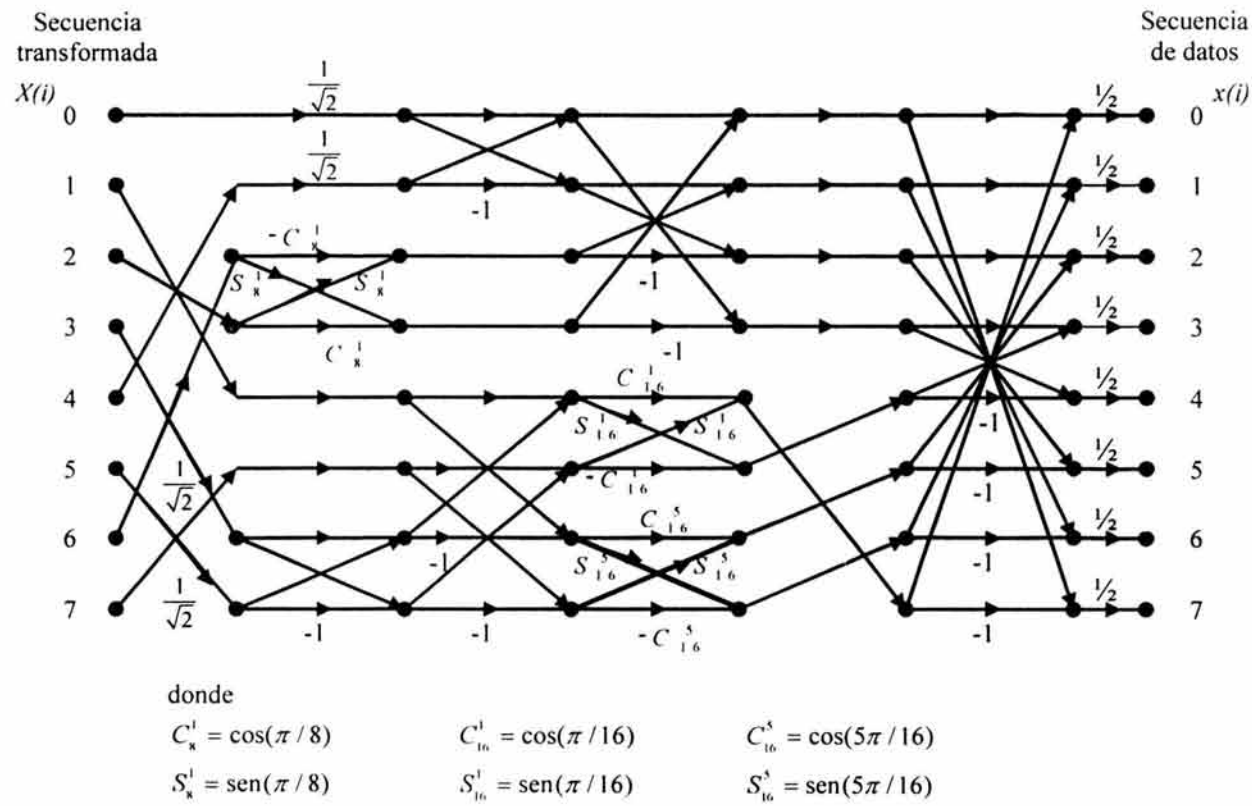


Figura 4.28 Diagrama de flujo para la optimización de Wang, Suehiro y Hatori de la IDCT

En la figura 4.28 entran ocho coeficientes DCT al proceso y salen ocho muestras recuperadas. Notamos que el número de operaciones en este diagrama es el mismo que para la DCT.

La implementación de este diagrama fue también de forma directa. Al igual que en la IDCT basada en la definición 1-D de la transformada directa, la función que ejecuta este algoritmo (*idct.*c) también incluye el redondeo de las muestras recuperadas y la rutina para detección de *overflow*. Asimismo los datos de entrada se toman de *bloque_t* y los datos de salida se almacenan en *bloque*.

4.8 Resumen

La herramienta elegida para procesar los algoritmos de este proyecto es un procesador digital de señales. Los DSP's tienen buen desempeño al realizar operaciones matemáticas intensivas. Se cuenta con un DSP TMS320C6711, que es un dispositivo de punto flotante basado en una arquitectura VLIW y VELOCITI. Dicha arquitectura permite ejecutar hasta ocho operaciones en paralelo y con ello lograr tiempos de cómputo menores que los obtenidos en procesadores que cuentan con un reloj interno más rápido.

La tarjeta DSK C6711 admite programación en lenguaje ensamblador, *C*, *C++* ó ensamblador lineal gracias al CCS. El código fuente se puede optimizar mediante las herramientas del CCS para lograr un proceso más eficiente. La comunicación entre la PC y el DSP es posible gracias al puerto paralelo HPI.

Los algoritmos de codificación y decodificación se programaron en lenguaje *C*, y se implementaron alrededor de una función principal que los llamaba. Tanto para la codificación como para la decodificación se utilizaron cuatro procesos generales aplicados a cada unidad de datos de la imagen. Para la codificación los procesos fueron la DCT, cuantización, codificación de coeficientes DC y codificación de coeficientes AC, mientras que para la decodificación los procesos fueron decodificación de coeficientes DC, decodificación de coeficientes AC, decuantización e IDCT. Finalmente, se describió la implementación de un proceso alternativo para calcular la DCT e IDCT. Dicho proceso se basó en una optimización de la definición de la Transformada Coseno.

En el siguiente capítulo se mostrarán los resultados de todos estos procesos explicados.

Capítulo 5
Resultados

En este capítulo se describen los resultados del algoritmo de compresión de imágenes implementado. Aquí se presentan comparaciones entre la imagen original y la imagen recuperada con diferentes razones de compresión, para evaluar la calidad de los resultados. Asimismo se incluyen imágenes que muestran las diferencias entre la imagen original y la imagen recuperada. Por último se hace un resumen del desempeño del DSP utilizado al ejecutar los algoritmos de la Transformada Coseno en la compresión.

5.1 Compresión y calidad de las imágenes recuperadas

Las imágenes a las que se les realizó el proceso de compresión se consiguieron en diferentes fuentes, una se obtuvo en *Internet* (*Lenna*), otra mediante una cámara digital (*Titania*) y la última mediante la captura de un cuadro en una película (*Jodie*). Originalmente todas las imágenes se encontraban en el espacio de color *RGB*, y mediante su manipulación con *Matlab*, se convirtieron al espacio de color *YCbCr*. Todas estas imágenes constan con una resolución de 256×256 píxeles y cada píxel se representa por 24 bits, 8 bits por cada componente de color.

Cada componente de color de las imágenes a comprimir se almacenó en un arreglo de 8 bits de 65536 elementos. Lo anterior da un total de 1572864bits, o bien, 196608bytes (192kbytes) de información por cada imagen original.

El nivel de compresión depende directamente de la información descartada durante la cuantización de los coeficientes DCT. Como se mencionó en el apartado 3.3.3, al modificarse los coeficientes de cuantización se modificará el nivel de compresión. Las ecuaciones implementadas, que muestran la modificación de todos los coeficientes en una tabla de cuantización mediante un divisor *div*, son la 4.1 y la 4.2. Las pruebas se realizaron para cinco valores distintos del divisor de las tablas de cuantización *div*:

- 0.3
- 0.5
- 1
- 2
- 3

Cada divisor se aplicó durante la compresión tanto a las tablas de luminiscencia como a las de crominancia.

Con base en el diseño de concatenado de información codificada, explicado en el apartado 4.3.5, la información comprimida en *códigos Huffman* y *cantidades en bits variables* se almacenó en tres distintos arreglos, cada elemento de los arreglos contando con 16 bits de precisión, un arreglo por cada componente:

- *cadcod*: almacenó la componente de luminiscencia
- *cadcodcb*: almacenó la componente Cb
- *cadcodcr*: almacenó la componente Cr

En las tablas 5.1, 5.2 y 5.3 se pueden observar los resultados numéricos de la compresión para las imágenes *Lena*, *Titania* y *Jodie* respectivamente. En dichas tablas se almacena el número de elementos de cada arreglo de 16 bits necesario para almacenar la información comprimida.

Lenna

	div				
	0.3	0.5	1	2	3
#elementos de cadcod	1525	2058	3140	4642	5859
#elementos de cadcodcb	357	457	728	1157	1549
#elementos de cadcodcr	343	443	712	1150	1541

Tabla 5.1 # de elementos necesario en cada arreglo para almacenar *Lenna* comprimida

Dentro de la tabla 5.1 se pueden ver el número de elementos necesarios en cada arreglo de salida para almacenar la información comprimida dependiendo del valor de *div*. Para un valor de *div* igual a 0.3 que presenta mayor compresión, encontramos que son necesarios 1525 elementos de *cadcod* para almacenar la información comprimida de la componente de luminiscencia, 357 elementos de *cadcodcb* para almacenar la información de la componente Cb y 343 elementos de *cadcodcr* para almacenar a la componente Cr.

De la misma manera se pueden leer los resultados en las tablas 5.2 y 5.3.

Titania

	div				
	0.3	0.5	1	2	3
#elementos de cadcod	1653	2352	3686	5570	6933
#elementos de cadcodcb	306	377	612	1076	1473
#elementos de cadcodcr	288	332	502	911	1269

Tabla 5.2 # de elementos necesario en cada arreglo para almacenar *Titania* comprimida

Jodie

	div				
	0.3	0.5	1	2	3
#elementos de cadcod	1203	1631	2463	3684	4636
#elementos de cadcodcb	310	382	578	955	1278
#elementos de cadcodcr	311	382	574	924	1231

Tabla 5.3 # de elementos necesario en cada arreglo para almacenar *Jodie* comprimida

Podemos observar en los tres casos que el número de elementos necesarios para almacenar las componentes de crominancia es mucho menor al número de elementos necesarios para almacenar la componente de luminiscencia. Lo anterior significa que la compresión resultante de las componentes de crominancia es mucho mayor a la compresión de la componente de luminiscencia, corroborando los conceptos mencionados en el apartado 3.4.3.

Para obtener un porcentaje de compresión se comparó el espacio utilizado en memoria por la imagen original con respecto al espacio de la imagen comprimida.

Como se mencionó al principio de este apartado, cada imagen original en el DSK ocupa un espacio de 192kbytes en memoria. Si tomamos como ejemplo de nuevo el caso en que *div* es igual a 0.3 para la imagen *Lena*, para almacenar la información comprimida es necesario el siguiente número de elementos de cada arreglo:

- 1525 elementos de *cadcod* para la componente de luminiscencia
- 357 elementos de *cadcodcb* para la componente Cb
- 343 elementos de *cadcodcr* para la componente Cr

Sabemos que cada elemento de los arreglos consta de 16 bits por lo que el total de memoria será igual al resultado de la operación

$$(1525+357+343)\,elementos \times \frac{16bits}{elemento} = 35600bits \tag{5.1}$$

Estos 35600bits ocupados por la imagen comprimida *Lena* se pueden dividir entre 8 para obtener el espacio total en bytes utilizado por la imagen de salida para *div* = 0.3:

$$\frac{35600bits}{8} = 4450bytes \tag{5.2}$$

Finalmente, para obtener el porcentaje de espacio utilizado por la imagen comprimida con respecto a la original se comparan los bytes utilizados por la imagen comprimida y los bytes utilizados por la imagen original:

$$\frac{4450bytes}{196608bytes} \times 100 = 2.26\% \tag{5.3}$$

Lo anterior indica que en 2.26% del espacio ocupado por el original de la imagen *Lena* se puede almacenar la imagen comprimida *Lena* con *div* =0.3.

Siguiendo el procedimiento anterior, con base en las tablas 5.1, 5.2, 5.3 y las ecuaciones 5.1, 5.2 y 5.3, se incluyen a continuación las tablas 5.4, 5.5 y 5.6 con los porcentajes de compresión alcanzados para cada valor de *div* en cada imagen.

Lenna

Div	0.3	0.5	1	2	3
% de espacio utilizado por la imagen comprimida con respecto a la original	2.26%	3.01%	4.66%	7.07%	9.10%

Tabla 5.4 Porcentajes de compresión para *Lena*

Titania

Div	0.3	0.5	1	2	3
% de espacio utilizado por la imagen comprimida con respecto a la original	2.29%	3.11%	4.88%	7.69%	9.84%

Tabla 5.5 Porcentajes de compresión para *Titania*

Jodie

Div	0.3	0.5	1	2	3
% de espacio utilizado por la imagen comprimida con respecto a la original	1.86%	2.44%	3.68%	5.66%	7.27%

Tabla 5.6 Porcentajes de compresión para *Jodie*

Podemos observar dentro de las tablas 5.4 a 5.6 que el porcentaje aproximado de compresión para las tablas de cuantización incluidas en el estándar JPEG es de imágenes comprimidas en promedio a 4% del tamaño original.

En las figuras 5.1 a 5.3 se incluyen las componentes de luminiscencia para las imágenes originales y para las imágenes recuperadas donde *div* toma valores iguales a 0.3, 1 y 3.

La razón para presentar exclusivamente la componente de luminiscencia consiste en que en dicha componente se pueden apreciar mejor los resultados de calidad a simple vista. Es importante mencionar que la mezcla de las tres componentes provoca que la imagen recuperada se asemeje aún más a la original en comparación a cotejar cada componente por separado.

Figura 5.1 Comparación para distintos valores de *div* para la componente de luminiscencia de la imagen *Lenna*[*]

En la figura 5.1 encontramos la componente de luminiscencia de la imagen *Lenna* original junto con los resultados visuales para la misma componente para diferentes valores de *div*:

[*] Imagen obtenida de la página *http://sipi.usc.edu/services/database/Database.html* de USC-SIPI Image Database.

- Para el caso en que *div* = 0.3 notamos que la imagen presenta artefactos en toda la imagen, así como ruido en todos los contornos.
- Para *div* = 1 casi no aparecen artefactos y apenas es perceptible cierto ruido alrededor de los contornos, principalmente el hombro y en el reflejo del espejo.
- Para *div* = 3, la diferencia entre la imagen recuperada y la original es casi imperceptible.

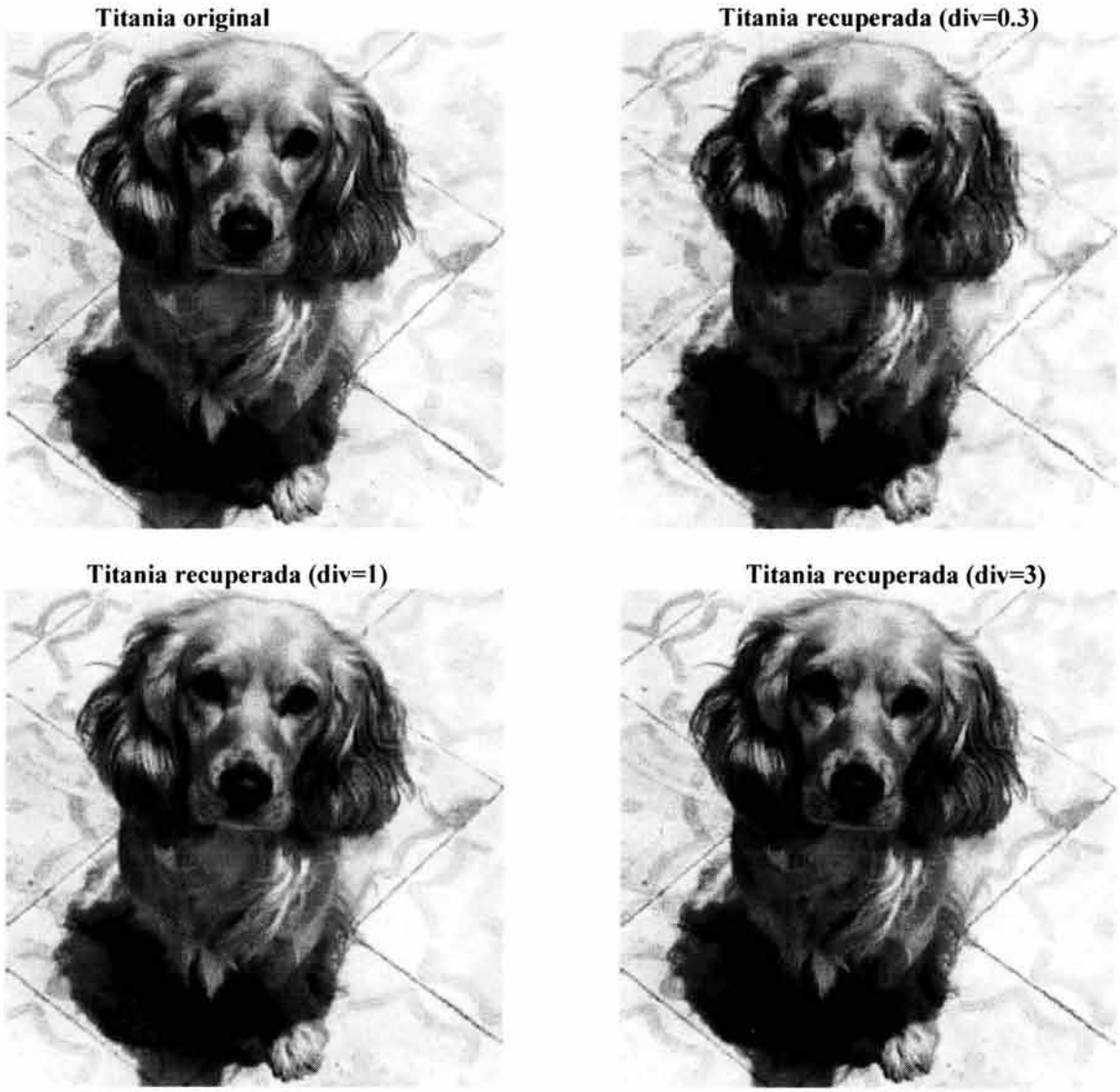


Figura 5.2 Comparación para distintos valores de *div* para la componente de luminiscencia de la imagen *Titania*

En la figura 5.2 notamos que los cambios en calidad dependientes de *div* para la imagen *Titania* se distinguen menos que en el caso de *Lena*:

- Para el caso en que *div* = 0.3 es posible ver artefactos en toda la imagen, sin embargo, aparentan ser más pequeños que en la imagen *Lena*. Adicionalmente también aparece ruido alrededor de los contornos.
- Para *div* = 1 desaparecen todos los artefactos, y sólo mediante una revisión minuciosa es posible ver que el pelaje en el pecho del perro no tiene la definición de la imagen original.
- Para *div* = 3 la imagen recuperada es prácticamente indistinguible de la original.

Figura 5.3 Comparación para distintos valores de *div* para la componente de luminiscencia de la imagen *Jodie*[*]

La última comparación entre imágenes recuperadas y originales se da en la figura 5.3:

[*] Imagen capturada de la película *Taxi Driver* Dir. Martin Scorsese, 1976.

- Para *div* = 0.3 aparecen artefactos en toda la imagen, sin embargo, la distorsión semejante a ruido no aparece en los contornos.
- Para *div* = 1 desaparecen los artefactos y los contornos en la imagen recuperada son diferentes a los de la original, sólo que no aparecen estas diferencias como ruido sino como imagen fuera de foco. Ciertos detalles, como el ojo izquierdo, no se definen bien.
- Para *div* = 3 la imagen recuperada es casi indistinguible de la original.

En todos los ejemplos para las componentes de luminiscencia recuperadas podemos observar que la compresión con *div* = 0.3 es excesiva; la imagen todavía se reconoce pero resulta ser de calidad pésima. Cuando *div* = 1 para algunos casos la calidad es regular y para otros se puede considerar buena calidad. Mientras que para *div* = 3 la calidad pasa por ser excelente hasta casi indistinguible.

La comparación cualitativa es subjetiva, y a pesar de que en ciertos casos es imposible distinguir entre las imágenes originales y las recuperadas, existe una diferencia. Con el propósito de ilustrar la falla en la percepción se incluyen a continuación las figuras 5.4 a 5.6. Dichas figuras muestran las diferencias entre la imagen original y la imagen recuperada *Jodie* para los casos en que *div* es igual a 0.3 y a 3. Asimismo, las figuras ayudan a hacer una mejor comparación entre el caso de mayor y de menor compresión.

Dado que las diferencias entre las componentes originales y recuperadas son tanto positivas como negativas, se les aplicó un *offset* positivo de 128 a las diferencias para que se pudiesen presentar como imágenes en tono de grises con valores desde 0 a 255. Posteriormente, el valor de las diferencias se aumentó 0.5 veces con el fin de incrementar el contraste y mejorar la visualización.

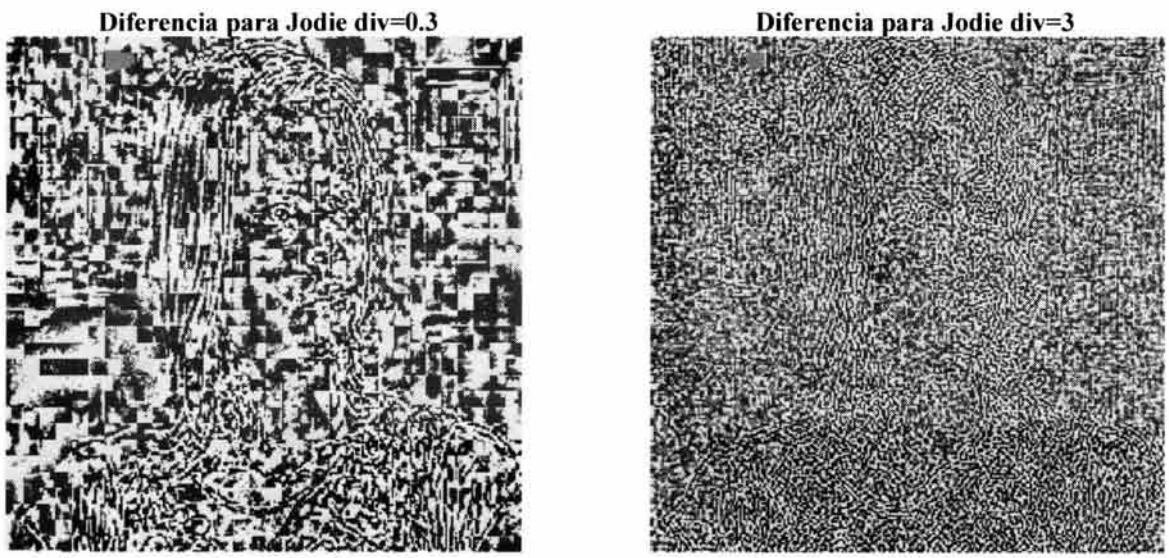


Figura 5.4 Diferencias entre las componentes de luminiscencia original y recuperada para *div* = 0.3 y *div* = 3 en la imagen *Jodie*

La figura 5.4 contiene imágenes que representan las diferencias entre las componentes de luminiscencia original y las componentes de luminiscencia recuperadas para *Jodie*. En la imagen con *div* = 0.3 observamos que al lado izquierdo de la cabeza se puede encontrar un pequeño bloque con tonalidad intermedia entre los tonos más oscuros y los más claros. Este tono corresponde al valor 128 del *offset* y al mismo tiempo a la ausencia de diferencia entre las componentes original y recuperada. Por ende, si no hubiera diferencia entre la imagen original y recuperada, la imagen que representa las diferencias constaría en su totalidad de este tono.

Adicionalmente notamos en la imagen *div* = 0.3 que a partir de las diferencias podemos distinguir la silueta de *Jodie*. Asimismo, se pueden ver las diferencias que generan la creación de artefactos.

En la imagen donde *div* = 3 la silueta es casi indistinguible y los tonos de gris se homogenizan. Adicionalmente los tonos de diferencia se acercan bastante al tono neutro de ausencia de diferencia. Debido a estas características sabemos que la imagen recuperada se asemeja más en el caso en que *div* = 3 que cuando *div* = 0.3.

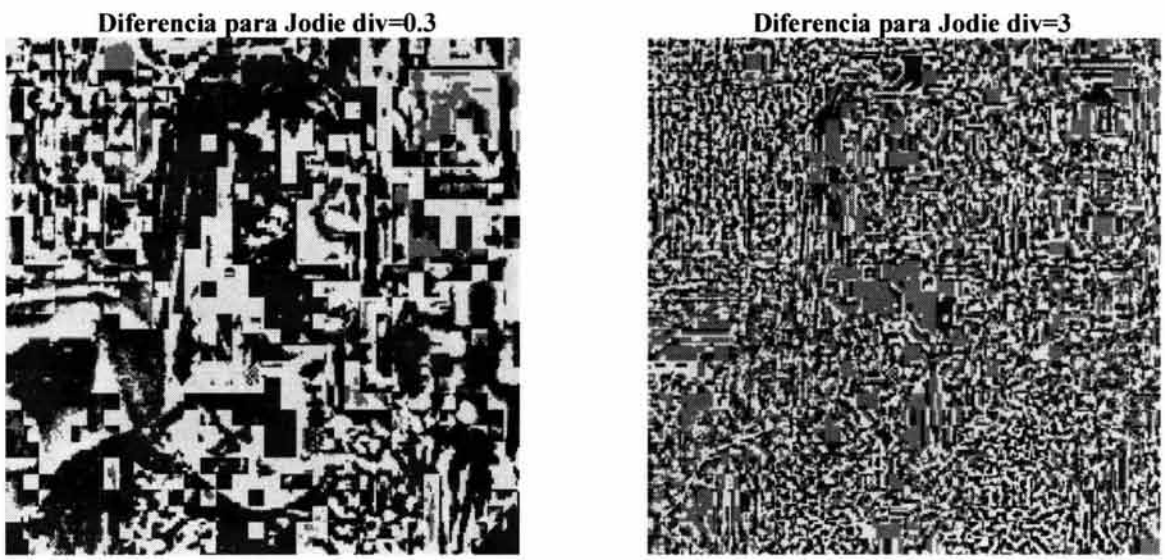


Figura 5.5 Diferencias entre las componentes de crominancia Cb original y recuperada para *div* = 0.3 y *div* = 3 en la imagen *Jodie*

En la imagen 5.5 se encuentran las diferencias para la componente de crominancia Cb para *Jodie*.

En el caso de mayor compresión (*div* = 0.3) la variación en las diferencias disminuye, lo cual se puede observar en el hecho de que existen zonas amplias con el mismo tono. Esto contrasta con el caso de la compresión de la componente de luminiscencia en el cual aparece una gran variedad de diferencias representadas por una multiplicidad de tonos.

En el caso de menor compresión (*div* = 3), las diferencias no se homogenizan tanto como en el caso de la luminiscencia, sin embargo, en esta componente se encuentran bloques con ausencia de diferencia.

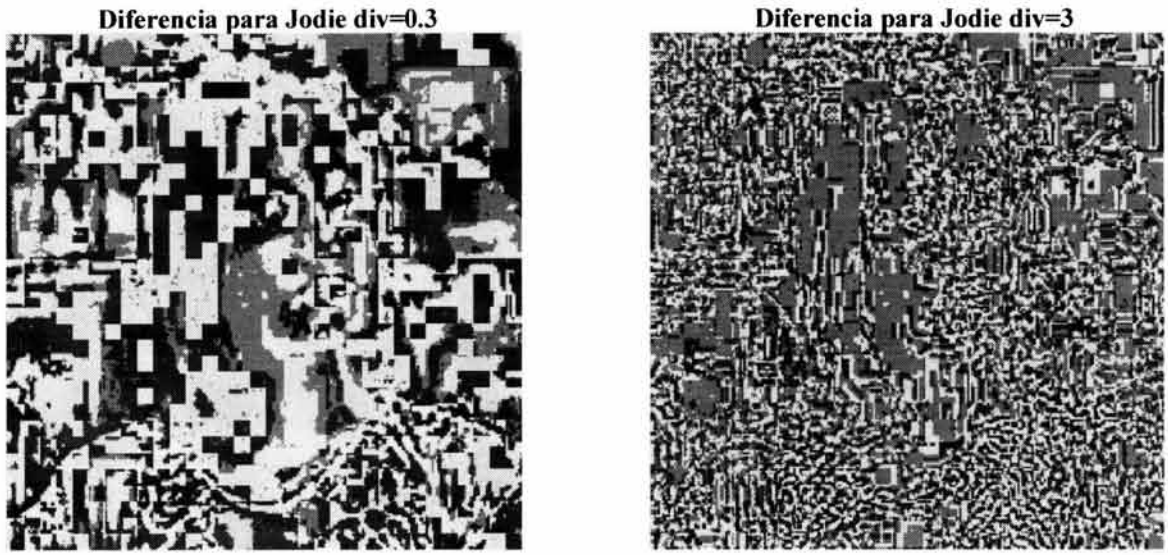


Figura 5.6 Diferencias entre las componentes de crominancia Cr original y recuperada para *div* = 0.3 y *div* = 3 en la imagen *Jodie*

En la figura 5.6 se encuentran las imágenes para el caso de mayor (*div* = 0.3) y de menor (*div* = 3) compresión para la componente de crominancia Cr.

En la imagen que ilustra el caso de mayor compresión, las diferencias se concentran y como resultado aparecen zonas amplias sin diferencias, es decir, se encuentran secciones extensas con el tono equivalente al valor 128. Por lo tanto, en comparación con las otras componentes, la componente Cr es la que presenta menos diferencias entre la imagen original y la recuperada.

De la misma manera, para el caso de menor compresión, aparecen más zonas con el tono 128 en comparación a las otras componentes, a pesar de que los tonos no se homogenizan tanto como en el caso de la componente de luminiscencia.

Con base en los resultados cuantitativos y cualitativos anteriores podemos evaluar las relaciones de compresión:

- 2-4% del tamaño original: desde calidad moderada hasta buena
- 4-7% del tamaño original: desde calidad buena hasta muy buena
- 7-9% del tamaño original: desde calidad muy buena hasta excelente
- 9-25% del tamaño original: indistinguible del original

Estas apreciaciones cumplen con el objetivo de este trabajo y se asemejan a los resultados esperados. Sin embargo, hay que tener en cuenta el hecho de que la información de la imagen sólo podrá ser procesada por sistemas externos si dichos

sistemas esperan este tipo de información. De lo contrario, la información se debe encapsular siguiendo las reglas estipuladas JFIF, y por ende, se deberá incluir la información necesaria para identificar qué tablas Huffman y cuáles tablas de cuantización se utilizaron.

5.2 Desempeño del DSP en la implementación de la Transformada Coseno

A continuación se incluye una tabla de ciclos y tiempos necesarios para procesar una unidad de datos tanto por la DCT como la IDCT en el DSP TMS320C6711. Los resultados corresponden a la implementación directa de la definición de la Transformada Coseno en una dimensión y a la implementación directa del algoritmo desarrollado por Wang, Suehiro y Hatori [8]. La tabla contiene ciclos y tiempos para ambas implementaciones sin optimización del *pipeline* y con optimización del *pipeline*.

	Sin pipeline		Con pipeline	
DCT (definición unidimensional)	125296 ciclos	835.307μseg	98838 ciclos	658.92μseg
IDCT (definición unidimensional)	102098 ciclos	680.653μseg	97368 ciclos	649.12μseg
DCT (Wang, Suehiro y Hatori)	293122 ciclos	1954.147μseg	23619 ciclos	157.46μseg
IDCT (Wang, Suehiro y Hatori)	437398 ciclos	2915.987μseg	38073 ciclos	253.82μseg

Tabla 5.7 Ciclos y tiempos aproximados para la ejecución de implementaciones de la DCT e IDCT a una unidad de datos con y sin *pipeline*

Los ciclos totales de la tabla 5.7 incluyen la llamada de las funciones correspondientes *dct.c* e *idct.c*, así como su retorno a la función *main*.

En la tabla 5.7 podemos observar que los tiempos se reducen cuando el DSP optimiza el uso de sus unidades de ejecución mediante el uso del *pipeline*. Esto es, que las unidades lógicas, aritméticas y multiplicadoras del procesador aprovechen al máximo su capacidad de funcionar en paralelo:

- En la implementación de la definición de la DCT unidimensional, el tiempo de ejecución utilizando *pipeline* es menor al tiempo sin la utilización del *pipeline* aproximadamente en un 21%.
- En la implementación de la definición de la IDCT unidimensional, el tiempo de ejecución utilizando *pipeline* es menor al tiempo sin la utilización del *pipeline* aproximadamente en un 5%.

- En la implementación de la optimización de la DCT realizada por Wang el tiempo de ejecución utilizando *pipeline* es menor al tiempo sin la utilización del *pipeline* aproximadamente en un 92%.
- En la implementación de la optimización de la IDCT realizada por Wang el tiempo de ejecución utilizando *pipeline* es menor al tiempo sin la utilización del *pipeline* aproximadamente en un 91%.

Los tiempos de ejecución no sólo corresponden a las operaciones aritméticas realizadas por el proceso de la DCT e IDCT; sino también existen ciclos utilizados por lectura y escritura de datos. Esto se hace evidente al momento en que comparamos el tiempo de las implementaciones sin el uso de *pipeline*.

A pesar de que las optimizaciones realizadas por Wang cuentan con un número menor operaciones que las definiciones de la DCT e IDCT (apartado 4.7), la ejecución de la optimización de Wang se efectúa en un mayor número de ciclos que la definición de las definiciones de la DCT e IDCT unidimensional.

Lo anterior se debe al poco control que se tiene sobre las unidades de ejecución del DSP en el lenguaje *C*, lo cual repercute en el diseño del programa. Podemos considerar que el DSP funciona como un procesador normal del mismo reloj cuando no se aprovecha su capacidad de realizar operaciones en paralelo, es decir, cuando no se optimiza el algoritmo mediante *pipeline*.

La utilización de las unidades de ejecución mediante el *pipeline* reduce tiempos de acceso a memoria, aproximando el tiempo requerido para realizar las operaciones reales al tiempo que se requeriría exclusivamente para realizar las operaciones teóricas.

Por ende, una forma más adecuada para comparar los diferentes algoritmos implementados de la DCT e IDCT consiste en cotejar los tiempos de los algoritmos de la DCT e IDCT basados en su definición unidimensional utilizando *pipeline*, con los tiempos de las optimizaciones de Wang utilizando *pipeline*.

- La implementación de la DCT utilizando *pipeline* con menor tiempo de ejecución (optimización Wang), redujo los ciclos utilizados en un 81% en comparación a la implementación sin utilizar *pipeline* con menor tiempo de ejecución (definición de la DCT).
- La implementación de la IDCT utilizando *pipeline* con menor tiempo de ejecución (optimización Wang), redujo los ciclos utilizados en un 63% en comparación a la implementación sin utilizar *pipeline* con menor tiempo de ejecución (definición de la IDCT).

Es importante remarcar que los tiempos de ejecución en la IDCT no se optimizan tanto como los tiempos de la DCT. Lo anterior se debe a que dentro de la función que realiza la IDCT no sólo se encuentra el algoritmo que recupera el valor de los píxeles a partir de coeficientes DCT, sino también una comparación para redondear los valores entregados y una pequeña rutina de corrección de *overflow* (apartado 4.5.5).

Los resultados son congruentes a la teoría (apartado 3.3.6), la definición de la Transformada Coseno implica un proceso muy intensivo en operaciones, por lo que es prudente utilizar una optimización teórica para disminuir en gran medida su tiempo de ejecución. Por ende, no es suficiente tener un procesador muy eficiente como el DSP para reducir los ciclos en que se lleva a cabo el algoritmo.

El tiempo aproximado para aplicar la DCT o la IDCT a una imagen de 256×256 píxeles se puede obtener multiplicando el tiempo aproximado en que se le aplica la transformada a una unidad de datos por el número total de unidades de datos (1024), y este resultado a su vez por el número de componentes (3).

Dado lo anterior, sabemos que la **DCT** implementada más rápida para una imagen de 256×256 **se ejecuta** en **483.72 milisegundos**.

A su vez, la **IDCT** implementada más rápida para una imagen con el mismo número de píxeles **se ejecuta** en **779.74 milisegundos**.

En el caso de que se requiriese procesar varias imágenes tan sólo se debe inicializar las estructuras para la compresión y para la descompresión una sola vez, por lo que en las imágenes a partir de la primera se necesitaría descontar el tiempo de dicha inicialización del tiempo de la DCT. El **tiempo de** dicha **inicialización** es de **20.48 milisegundos**.

En la tabla 5.8 se incluyen los tiempos totales en que se realizó el proceso de compresión y descompresión para las tres imágenes presentadas en este capítulo. Se midió la duración de los procesos más largos, es decir, en los que se recupera una imagen con mejor calidad (div = 3).

	Tiempo de compresión	Tiempo de descompresión
Lena	1.51seg	2seg
Titania	1.52seg	2.05seg
Jodie	1.49seg	1.8seg

Tabla 5.8 Tiempos de compresión y descompresión para imágenes

En la tabla 5.8 se observa que el tiempo total de compresión y descompresión varía de imagen a imagen, esto es debido que el tiempo de búsqueda y asignación de códigos es diferente en cada imagen.

Los tiempos de totales de proceso podrían mejorarse si se implementase una optimización de la DCT en donde los tiempos de ejecución de las sumas y las multiplicaciones se consideraran iguales; el DSP puede aprovechar dichos algoritmos en comparación a otros procesadores que ocupan más ciclos para ejecutar una multiplicación que para ejecutar una suma.

Adicionalmente los algoritmos de asignación y detección de códigos se podrían depurar aún más y buscarse alguna optimización.

5.3 Resumen

En este capítulo se describieron los resultados de este trabajo, la calidad de las imágenes obtenidas para diferentes configuraciones del algoritmo de compresión implementado así como el desempeño del DSP ejecutando las operaciones del algoritmo de la DCT y la IDCT. Los resultados en cuanto a la calidad de las imágenes recuperadas fueron satisfactorios. Dentro de las pruebas realizadas para las dos formas de realizar la DCT e IDCT implementadas, se hizo evidente que mientras menos control se tenga sobre las asignaciones de ejecución de un procesador, el tiempo en que se ejecuta el algoritmo puede incrementarse excesivamente. La utilización del *pipeline* en el DSP puede corregir la falta de control sobre las unidades de ejecución y adicionalmente disminuye de forma considerable los tiempos en que se llevan a cabo los algoritmos programados. El *pipeline* aproxima el tiempo en que se realizan los algoritmos al tiempo que se tomaría dicho algoritmo en realizar las operaciones teóricas. La definición de la Transformada Coseno implica un gran número de operaciones por lo cual se debe implementar una optimización teórica de ella con el fin de que nuestro proceso sea eficiente. En el siguiente capítulo se concluye sobre los resultados obtenidos.

Capítulo 6
Conclusiones

Se ha logrado implementar una compresión de imágenes basada en el estándar JPEG utilizando un DSP, con porcentajes de compresión de hasta un 5% con una muy buena calidad en la imagen recuperada.

La compresión obtenida a partir de la Transformada Coseno inmersa en las técnicas propuestas por el estándar JPEG es muy efectiva. La imagen recuperada guarda gran semejanza con la original, a pesar de que el nivel de compresión sea alto.

La eficiencia del algoritmo general de compresión implementado aumenta en gran medida al implementarse una optimización de la DCT y podría incrementarse aún más si se opta por una DCT diseñada especialmente para arquitecturas que utilicen *pipeline*.

En general, la calidad de las imágenes es buena mientras no se comprima por debajo de 5% del tamaño original (apartado 5.1).

El procesador TMS320C6711 mejora en gran medida la implementación de los algoritmos que ocupen la operación MAC (multiplicación y acumulación) repetidas veces (apartado 5.2, tabla 5.7), en nuestro caso en particular la Transformada Coseno.

Debido a la operación en paralelo de las unidades de ejecución del DSP inmersas en un *pipeline* de información, se logró disminuir los tiempos de ejecución del algoritmo de la DCT y de la IDCT en su forma original así como a su optimización implementada: la reducción del 5% de los ciclos de ejecución utilizados en el peor de los casos y 92% en el mejor de los casos (apartado 5.2, tabla 5.7).

Tanto el proceso de compresión como de descompresión se podrían incrustar en una aplicación de transmisión o almacenamiento de imágenes fijas, como una cámara digital o transmisión por protocolo TCP/IP. Sin embargo, si se quisiese utilizar para transmitir video en tiempo real sería necesario disminuir la resolución de las imágenes a procesar y/o optimizar los procesos de codificación dependiendo de la aplicación. En una aplicación en Internet se debe tener en cuenta que la transmisión de video por un protocolo TCP/IP no es estable, por lo que usualmente se transmite con una tasa baja de cuadros por segundo (ej. 9 cuadros por segundo), pero la aplicación puede tener una transmisión estable y posibilitar una imagen de alta resolución a una tasa de 29.97 cuadros por segundo.

Anexos

Anexo A
Tablas para la cuantización y para la codificación Huffman

Tablas de cuantización para las componentes de luminiscencia y crominancia

Tabla A.1 Tabla para la cuantización de la componente de luminiscencia

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Tabla A.2 Tabla para la cuantización de las componentes de crominancia

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

Las tablas A.1 y A.2 se obtuvieron de manera empírica basándose en las curvas características de la receptividad del ojo a las variaciones de luminiscencia y de crominancia. Debido a que la sensibilidad a los cambios de la luminiscencia es mayor que a los cambios en la crominancia, la variación en los coeficientes de cuantización para luminiscencia es paulatina mientras que en el caso de la crominancia la variación es brusca [1].

Tablas Huffman para la codificación de los coeficientes DC

Tabla A.3 Tabla de códigos Huffman para las diferencias DC de luminiscencia

SSSS	Longitud del código	Código
0	2	00
1	3	010
2	3	011
3	3	100
4	3	101
5	3	110
6	4	1110
7	5	11110
8	6	111110
9	7	1111110
10	8	11111110
11	9	111111110

Tabla A.4 Tabla de códigos Huffman para las diferencias DC de crominancia

SSSS	Longitud del código	Código
0	2	00
1	2	01
2	2	10
3	3	110
4	4	1110
5	5	11110
6	6	111110
7	7	1111110
8	8	11111110
9	9	111111110
10	10	1111111110
11	11	11111111110

Tablas Huffman para la codificación de los coeficientes AC

Tabla A.5 Tabla de códigos Huffman para los coeficientes AC de luminiscencia

RRRR/SSSS	Longitud del código	Código
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111111000110
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101
4/1	6	111011
4/2	10	1111111000
4/3	16	1111111110010110
4/4	16	1111111110010111
4/5	16	1111111110011000
4/6	16	1111111110011001
4/7	16	1111111110011010
4/8	16	1111111110011011
4/9	16	1111111110011100
4/A	16	1111111110011101

Tabla A.5 (continuación)

RRRR/SSSS	Longitud del código	Código
5/1	7	1111010
5/2	11	11111110111
5/3	16	1111111110011110
5/4	16	1111111110011111
5/5	16	1111111110100000
5/6	16	1111111110100001
5/7	16	1111111110100010
5/8	16	1111111110100011
5/9	16	1111111110100100
5/A	16	1111111110100101
6/1	7	1111011
6/2	12	111111110110
6/3	16	1111111110100110
6/4	16	1111111110100111
6/5	16	1111111110101000
6/6	16	1111111110101001
6/7	16	1111111110101010
6/8	16	1111111110101011
6/9	16	1111111110101100
6/A	16	1111111110101101
7/1	8	11111010
7/2	12	111111110111
7/3	16	1111111110101110
7/4	16	1111111110101111
7/5	16	1111111110110000
7/6	16	1111111110110001
7/7	16	1111111110110010
7/8	16	1111111110110011
7/9	16	1111111110110100
7/A	16	1111111110110101
8/1	9	111111000
8/2	15	111111111000000
8/3	16	1111111110110110
8/4	16	1111111110110111
8/5	16	1111111110111000
8/6	16	1111111110111001
8/7	16	1111111110111010
8/8	16	1111111110111011
8/9	16	1111111110111100
8/A	16	1111111110111101
9/1	9	111111001
9/2	16	1111111110111110
9/3	16	1111111110111111
9/4	16	1111111111000000
9/5	16	1111111111000001
9/6	16	1111111111000010
9/7	16	1111111111000011
9/8	16	1111111111000100
9/9	16	1111111111000101
9/A	16	1111111111000110
A/1	9	111111010
A/2	16	1111111111000111
A/3	16	1111111111001000
A/4	16	1111111111001001
A/5	16	1111111111001010

Tabla A.5 (continuación)

RRRR/SSSS	Longitud del código	Código
A/6	16	1111111111001011
A/7	16	1111111111001100
A/8	16	1111111111001101
A/9	16	1111111111001110
A/A	16	1111111111001111
B/1	10	1111111001
B/2	16	1111111111010000
B/3	16	1111111111010001
B/4	16	1111111111010010
B/5	16	1111111111010011
B/6	16	1111111111010100
B/7	16	1111111111010101
B/8	16	1111111111010110
B/9	16	1111111111010111
B/A	16	1111111111011000
C/1	10	1111111010
C/2	16	1111111111011001
C/3	16	1111111111011010
C/4	16	1111111111011011
C/5	16	1111111111011100
C/6	16	1111111111011101
C/7	16	1111111111011110
C/8	16	1111111111011111
C/9	16	1111111111100000
C/A	16	1111111111100001
D/1	11	11111111000
D/2	16	1111111111100010
D/3	16	1111111111100011
D/4	16	1111111111100100
D/5	16	1111111111100101
D/6	16	1111111111100110
D/7	16	1111111111100111
D/8	16	1111111111101000
D/9	16	1111111111101001
D/A	16	1111111111101010
E/1	16	1111111111101011
E/2	16	1111111111101100
E/3	16	1111111111101101
E/4	16	1111111111101110
E/5	16	1111111111101111
E/6	16	1111111111110000
E/7	16	1111111111110001
E/8	16	1111111111110010
E/9	16	1111111111110011
E/A	16	1111111111110100
F/0 (ZRL)	11	11111111001
F/1	16	1111111111110101
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101

Tabla A.5 (continuación)

RRRR/SSSS	Longitud del código	Código
F/A	16	1111111111111110

Tabla A.6 Tabla de códigos Huffman para los coeficientes AC de crominancia

RRRR/SSSS	Longitud del código	Código
0/0 (EOB)	2	00
0/1	2	01
0/2	3	100
0/3	4	1010
0/4	5	11000
0/5	5	11001
0/6	6	111000
0/7	7	1111000
0/8	9	111110100
0/9	10	1111110110
0/A	12	111111110100
1/1	4	1011
1/2	6	111001
1/3	8	11110110
1/4	9	111110101
1/5	11	11111110110
1/6	12	111111110101
1/7	16	1111111110001000
1/8	16	1111111110001001
1/9	16	1111111110001010
1/A	16	1111111110001011
2/1	5	11010
2/2	8	11110111
2/3	10	1111110111
2/4	12	111111110110
2/5	15	111111111000010
2/6	16	1111111110001100
2/7	16	1111111110001101
2/8	16	1111111110001110
2/9	16	1111111110001111
2/A	16	1111111110010000
3/1	5	11011
3/2	8	11111000
3/3	10	1111111000
3/4	12	111111110111
3/5	16	1111111110010001
3/6	16	1111111110010010
3/7	16	1111111110010011
3/8	16	1111111110010100
3/9	16	1111111110010101
3/A	16	1111111110010110
4/1	6	111010
4/2	9	111110110
4/3	16	1111111110010111
4/4	16	1111111110011000
4/5	16	1111111110011001
4/6	16	1111111110011010
4/7	16	1111111110011011

Tabla A.6 (continuación)

RRRR/SSSS	Longitud del código	Código
4/8	16	1111111110011100
4/9	16	1111111110011101
4/A	16	1111111110011110
5/1	6	111011
5/2	10	1111111001
5/3	16	1111111110011111
5/4	16	1111111110100000
5/5	16	1111111110100001
5/6	16	1111111110100010
5/7	16	1111111110100011
5/8	16	1111111110100100
5/9	16	1111111110100101
5/A	16	1111111110100110
6/1	7	1111001
6/2	11	11111110111
6/3	16	1111111110100111
6/4	16	1111111110101000
6/5	16	1111111110101001
6/6	16	1111111110101010
6/7	16	1111111110101011
6/8	16	1111111110101100
6/9	16	1111111110101101
6/A	16	1111111110101110
7/1	7	1111010
7/2	11	11111111000
7/3	16	1111111110101111
7/4	16	1111111110110000
7/5	16	1111111110110001
7/6	16	1111111110110010
7/7	16	1111111110110011
7/8	16	1111111110110100
7/9	16	1111111110110101
7/A	16	1111111110110110
8/1	8	11111001
8/2	16	1111111110110111
8/3	16	1111111110111000
8/4	16	1111111110111001
8/5	16	1111111110111010
8/6	16	1111111110111011
8/7	16	1111111110111100
8/8	16	1111111110111101
8/9	16	1111111110111110
8/A	16	1111111110111111
9/1	9	111110111
9/2	16	1111111111000000
9/3	16	1111111111000001
9/4	16	1111111111000010
9/5	16	1111111111000011
9/6	16	1111111111000100
9/7	16	1111111111000101
9/8	16	1111111111000110
9/9	16	1111111111000111
9/A	16	1111111111001000
A/1	9	111111000
A/2	16	1111111111001001

Tabla A.6 (continuación)

RRRR/SSSS	Longitud del código	Código
A/3	16	1111111111001010
A/4	16	1111111111001011
A/5	16	1111111111001100
A/6	16	1111111111001101
A/7	16	1111111111001110
A/8	16	1111111111001111
A/9	16	1111111111010000
A/A	16	1111111111010001
B/1	9	111111001
B/2	16	1111111111010010
B/3	16	1111111111010011
B/4	16	1111111111010100
B/5	16	1111111111010101
B/6	16	1111111111010110
B/7	16	1111111111010111
B/8	16	1111111111011000
B/9	16	1111111111011001
B/A	16	1111111111011010
C/1	9	111111010
C/2	16	1111111111011011
C/3	16	1111111111011100
C/4	16	1111111111011101
C/5	16	1111111111011110
C/6	16	1111111111011111
C/7	16	1111111111100000
C/8	16	1111111111100001
C/9	16	1111111111100010
C/A	16	1111111111100011
D/1	11	11111111001
D/2	16	1111111111100100
D/3	16	1111111111100101
D/4	16	1111111111100110
D/5	16	1111111111100111
D/6	16	1111111111101000
D/7	16	1111111111101001
D/8	16	1111111111101010
D/9	16	1111111111101011
D/A	16	1111111111101100
E/1	14	11111111100000
E/2	16	1111111111101101
E/3	16	1111111111101110
E/4	16	1111111111101111
E/5	16	1111111111110000
E/6	16	1111111111110001
E/7	16	1111111111110010
E/8	16	1111111111110011
E/9	16	1111111111110100
E/A	16	1111111111110101
F/0 (ZRL)	10	1111111010
F/1	15	111111111000011
F/2	16	1111111111110110
F/3	16	1111111111110111

Tabla A.6 (continuación)

RRRR/SSSS	Longitud del código	Código
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Anexo B
Mapa de memoria del TMS320C6711 DSK

En la tabla B.1 se muestra el mapa de memoria del DSK C6711; dentro de ella se puede observar que los 72kB de memoria interna comienzan en la dirección 0x00000000, los 16MB de SDRAM abarcan CE0 direcciones comenzando en 0x80000000, y los 128kB de memoria flash comienzan en 0x90000000.

Incluyendo los cuatro espacios para memoria externa CE0, CE1, CE2 y CE3 el espacio total para memoria es igual a 4GB.

Intervalo de direcciones (Hex)	Tamaño (Bytes)	Descripción del bloque de memoria
0000 0000-0000 FFFF	64K	RAM Interna (L2)
0001 0000-017F FFFF	24M-64K	Reservado
0180 0000-0183 FFFF	256K	Bus de registros de configuración interna EMIF
0184 0000-0187 FFFF	256K	Bus de registros de control de configuración interna L2
0188 0000-018B FFFF	256K	Bus de registros de configuración interna HPI
018C 0000-018F FFFF	256K	Bus de registros de configuración interna McBSP 0
0190 0000-0193 FFFF	256K	Bus de registros de configuración interna McBSP 1
0194 0000-0197 FFFF	256K	Bus de registros de configuración interna temporizador 0
0198 0000-0198 FFFF	256K	Bus de registros de configuración interna temporizador 1
019C 0000-019F FFFF	256K	Bus de registros de configuración interna para selección de interrupciones
01A0 0000-01A3 FFFF	256K	Bus de registros de configuración interna EDMA RAM
01A4 0000-01FF FFFF	6M-256K	Reservado
0200 0000-0200 0033	52	Registros QDMA
0200 0034-2FFF FFFF	736M-52	Reservado
3000 0000-3FFF FFFF	256M	Datos McBSP 0/1
4000 0000-7FFF FFFF	1G	Reservado
8000 0000-BFFF FFFF	256M	Interfaz CE0 de memoria externa
9000 0000-9FFF FFFF	256M	Interfaz CE1 de memoria externa
A000 0000-AFFF FFFF	256M	Interfaz CE2 de memoria externa
B000 000-BFFF FFFF	256M	Interfaz CE3 de memoria externa
C000 0000-FFFF FFFF	1G	Reservado

Tabla B.1 Mapa de memoria del TMS320C6711 DSK

Anexo C
Código fuente de los programas para el DSP

transmision_main.c

```
/*************************************************
*****************/
/* transmision_main.c versión 0.83
                                                  */
/* Codificación y decodificación de los coeficientes DC y AC
para                                              */
/* la componente de luminancia y las componentes de
crominancia.                                      */
/* Se incluye una rutina para detectar los errores de
codificación                                      */
/* o decodificación.
                                                  */
/*************************************************
*****************/

/***************************** Archivos include
*******************/
#pragma DATA_SECTION (original,"varia0")
#pragma DATA_SECTION (recuperada,"varia1")
#pragma DATA_SECTION (cadcod,"varia2")
#pragma DATA_SECTION (bugb,"varia3")
#pragma DATA_SECTION (originalcb,"varia4")
#pragma DATA_SECTION (originalcr,"varia5")
#pragma DATA_SECTION (recuperadacb,"varia6")
#pragma DATA_SECTION (recuperadacr,"varia7")
#pragma DATA_SECTION (cadcodcb,"varia8")
#pragma DATA_SECTION (cadcodcr,"varia9")
#include "valores1.h"
#include "estruhuff.h"
#include "valores2.h"
#include "valores3.h"

/************** Declaración de variables y arreglos
globales ******/

/* Arreglo 1-D para almacenar la imagen de salida */
unsigned char recuperada[65536];
unsigned char recuperadacb[65536];
unsigned char recuperadacr[65536];

char componente=1;

/* Arreglo 1-D para manipular un bloque de números enteros
*/
short bloque[64];

/* Arreglo 1-D para manipular un bloque de números reales
*/
float bloque_t[64];

/* Arreglo 1-D para almacenar la imagen codificada */
short cadcod[32768];
short cadcodcb[32768];
short cadcodcr[32768];

/* Variables para formar una cadena de bits codificada */
unsigned char btsacu = 0; /* Contador de bits acumulados */
unsigned short w=0; /* Contador para el arreglo cadcod */
short valant = 0, vec1 = 0; /* Valor anterior DC y variable
temporal para escribir a cadcod */


/* Funciones base */

const float basisf[8][8] = {

0.353553, 0.353553, 0.353553, 0.353553, 0.353553,
        0.353553, 0.353553, 0.353553,
0.490393, 0.415735, 0.277785, 0.097545, -0.097545, -
0.277785, -0.415735, -0.490393,
0.461940, 0.191342, -0.191342, -0.461940, -0.461940, -
0.191342, 0.191342, 0.461940,
0.415735, -0.097545, -0.490393, -0.277785, 0.277785,
        0.490393, 0.097545, -0.415735,
0.353553, -0.353553, -0.353553, 0.353553, 0.353553, -
0.353553, -0.353553, 0.353553,
0.277785, -0.490393, 0.097545, 0.415735, -0.415735, -
0.097545, 0.490393, -0.277785,
0.191342, -0.461940, 0.461940, -0.191342, -0.191342,
        0.461940, -0.461940, 0.191342,
0.097545, -0.277785, 0.415735, -0.490393, 0.490393, -
0.415735, 0.277785, -0.097545
};

/* Tabla de cuantización de luminancia */
const unsigned char tql[64] = {

16,     11,     10,     16,     24,     40,
        51,     61,
12,     12,     14,     19,     26,     58,
        60,     55,
14,     13,     16,     24,     40,     57,
        69,     56,
14,     17,     22,     29,     51,     87,
        80,     62,
18,     22,     37,     56,     68,     109,
        103,    77,
24,     35,     55,     64,     81,     104,
        113,    92,
49,     64,     78,     87,     103,    121,
        120,    101,
72,     92,     95,     98,     112,    100,
        103,    99
};

/* Tabla de cuantización de crominancia */
const unsigned char tqc[64] = {
17,     18,     24,     47,     99,     99,
        99,     99,
18,     21,     26,     66,     99,     99,
        99,     99,
24,     26,     56,     99,     99,     99,
        99,     99,
47,     66,     99,     99,     99,     99,
        99,     99,
99,     99,     99,     99,     99,     99,
        99,     99,
99,     99,     99,     99,     99,     99,
        99,     99,
99,     99,     99,     99,     99,     99,
        99,     99,
99,     99,     99,     99,     99,     99,
        99,     99
};

const float div = 3;

/***************** Declaración de estructuras globales
*************/
```

```
/* Se declara e inicializa el arreglo codDC para los doce
casos */
/* de código DC          para la luminiscencia
*/
struct chufdc codDC[12]  = {
{2,0},
{3,2},
{3,3},
{3,4},
{3,5},
{3,6},
{4,14},
{5,30},
{6,62},
{7,126},
{8,254},
{9,510}
};

/* Se declara e inicializa el arreglo codDC para los doce
casos */
/* de código AC
*/
struct chufdc codDCc[12] = {
{2,0},
{2,1},
{2,2},
{3,6},
{4,14},
{5,30},
{6,62},
{7,126},
{8,254},
{9,510},
{10,1022},
{11,2046}
};

/* Se declara el arreglo codAC para los 160  casos de código
AC */
/* sin incluir ZRL ni EOB
*/
/* No se inicializaran antes de que el programa corra ya que
*/
/* el compilador no permite tantas de inicializaciones de
*/
/* elementos de estructuras
*/
struct chufac codAC[16][10];
struct chufac codACc[16][10];

/* Se declara e inicializa el arreglo codEOB                */
struct chufeob codEOB = {10};
struct chufeob codEOBc = {0};

/* Se declara e inicializa el arreglo codZRL                */
struct chufzrl codZRL = {2041};
struct chufzrl codZRLc = {1018};

/* Se declara e inicializa la estructura para la decodificación
*/
/* de los coeficientes  de luminiscencia DC
*/
struct dchufdc decodDC[8] = {
{2, 0, 0, 0},
{3, 2, 6, 1},
{4, 14, 14, 6},
{5, 30, 30, 7},
{6, 62, 62, 8},
{7, 126, 126, 9},
{8, 254, 254, 10},
{9, 510, 510, 11}
};

/* Se declara e inicializa la estructura para la decodificación
*/
/* de los coeficientes  de crominancia DC
*/
struct dchufdc decodDCc[10] = {
{2, 0, 2, 0},
{3, 6, 6, 3},
{4, 14, 14, 4},
{5, 30, 30, 5},
{6, 62, 62, 6},
{7, 126, 126, 7},
{8, 254, 254, 8},
{9, 510, 510, 9},
{10, 1022, 1022, 10},
{11, 2046, 2046, 11}
};


/* Se declaran el arreglo de estructuras para la decodificación
*/
/* de los coeficientes AC
*/
struct dchufac decodAC[16][8];
struct dchufac decodACc[16][9];

/********************* Prototipos de funciones
**********************/

/* inichufac.c */
void inichufac(void);

/* dct.c */
void dct(void);

/* idct.c */
void idct(void);

/* qtz.c */
void qtz(void);

/* dqtz.c */
void dqtz(void);

/* CodDC.c */
void CodDC(void);

/* CodAC.c */
void CodAC(unsigned short ext1, unsigned short ext2);

/* zigzag.c */
void zigzag(void);

/* pot2.c */
unsigned int pot2(unsigned char tam);

/* dcodDC.c */
void dcodDC(void);

/* dcodAC.c */
void dcodAC(void);

/* dzigzag.c */
void dzigzag(void);

/***************************** MAIN
*********************************/

void main()
```

```
}
/* k determina el bloque, i la columna y j el renglón */
        unsigned short i,j,k,l,m;

/* Inicializa las estructuras de codificación AC */
        inichufac();

/* for anidados para lectura de las unidades de datos */
        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
                        for(j=0, m=0;j<8;j++)
                        {
                                for(i=0;i<8;i++)
                                {

        bloque[m]=((short)
original[(l*2048)+(8*k)+(256*j)+i])-128;

                        m++;
                                }
                        }

/* Espacio para implementar la dct e idct */
                        dct();
                        qtz();
                        CodIDC();
                        CodAC(l,k);
                }
        }


        /* for anidados para lectura de las unidades de
datos */
        /* Componente Cb */
        btsacu=0;
        w=0;
        valant=0;
        componente=2;
        vec1=0;

        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
                        for(j=0, m=0;j<8;j++)
                        {
                                for(i=0;i<8;i++)
                                {

        bloque[m]=((short)
originalcb[(l*2048)+(8*k)+(256*j)+i])-128;

                        m++;
                                }
                        }

/* Espacio para implementar la dct e idct */
                        dct();
                        qtz();
                        CodIDC();
                        CodAC(l,k);
                }
        }

        /* for anidados para lectura de las unidades de
datos */
        /* Componente Cr */
        btsacu=0;
        w=0;
        valant=0;
        componente=3;
        vec1=0;

        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
                        for(j=0, m=0;j<8;j++)
                        {
                                for(i=0;i<8;i++)
                                {

        bloque[m]=((short)
originalcr[(l*2048)+(8*k)+(256*j)+i])-128;

                        m++;
                                }
                        }

/* Espacio para implementar la dct e idct */
                        dct();
                        qtz();
                        CodIDC();
                        CodAC(l,k);
                }
        }

/* Inicia la decodificación */

        btsacu=0;
        w=0;
        valant=0;
        componente=1;

        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
/* Espacio para decod */
                        dcodIDC();
                        dcodAC();
                        dqtz();
                        idct();

/* Inicia escritura de datos al arreglo de salida */
                        for(j=0, m=0;j<8;j++)
                        {
                                for(i=0;i<8;i++)
                                {

        recuperada[(l*2048)+(8*k)+(256*j)+i]=((unsigne
d char) (bloque[m]+128));
                                m++;
                                }
                        }
                }
        }

/* Decodificación de cb */

        btsacu=0;
        w=0;
        valant=0;
        componente=2;

        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
/* Espacio para decod */
```

```
                                        decodDC();
                                        decodAC();
                                        dqtz();
                                        idct();

/* Inicia escritura de datos al arreglo de salida */
                                        for(j=0, m=0;j<8;j++)
                                        {
                                                for(i=0;i<8;i++)
                                                {

        recuperadacb[(l*2048)+(8*k)+(256*j)+i]=((unsig
ned char) (bloque[m]+128));
                                                m++;
                                                }
                                        }
                                }
        }

        /* Decodificación de cr */

        btsacu=0;
        w=0;
        valant=0;
        componente=3;

        for(l=0;l<32;l++)
        {
                for(k=0;k<32;k++)
                {
/* Espacio para decod */
                                        decodDC();
                                        decodAC();
                                        dqtz();
                                        idct();

/* Inicia escritura de datos al arreglo de salida */
                                        for(j=0, m=0;j<8;j++)
                                        {
                                                for(i=0;i<8;i++)
                                                {

        recuperadacr[(l*2048)+(8*k)+(256*j)+i]=((unsign
ed char) (bloque[m]+128));
                                                m++;
                                                }
                                        }
                                }
        }


        for(;;);                /* Ciclo infinito de espera
*/
}
```

dct.c (optimización Wang)

```
/*************************************************
***/
/* Nombre de archivo: dct.c                     */
/* Función: realiza dos transformadas coseno de 1D */
/* de 64 valores                                */
/* Versión 2, DCT basada en Wang
                                */
/*************************************************
***/

/* Variables externas */
extern short bloque[64];
extern float bloque_t[64];
extern const float c4, c4d, c8d, s8d, c16, s16, c516, s516;

/* Inicia la función dct */

void dct(void)
{
        unsigned short int i,j;
        float x[8], y[8], x2[8], x3[4], x4[2], x5[2], x6[4];

/* DCT en los renglones de bloque */
        for(j=0;j<8;j++)
        {
/* Se trabaja un renglón de la unidad de datos */

                for(i=0;i<8;i++)
                        x[i]=bloque[i+(j*8)];
/* Sigue el código para la DCT */

/* Primera iteración */
                x2[0]=x[0]+x[7];
        x2[1]=x[1]+x[6];
        x2[2]=x[2]+x[5];
        x2[3]=x[3]+x[4];
        x2[4]=x[3]-x[4];
        x2[5]=x[2]-x[5];
        x2[6]=x[1]-x[6];
        x2[7]=x[0]-x[7];

/* Mariposas superiores */
                x3[0]=x2[0]+x2[3];
        x3[1]=x2[1]+x2[2];
        x3[2]=x2[1]-x2[2];
        x3[3]=x2[0]-x2[3];

                y[0]=(x3[0]+x3[1])*c4d;
        y[4]=(x3[0]-x3[1])*c4d;
        y[6]=(x3[2]*(-c8d))+(x3[3]*s8d);
        y[2]=(x3[2]*s8d)+(x3[3]*c8d);

/* Mariposas inferiores */
        x4[0]=x2[6]*c4;
        x4[1]=x2[5]*c4;

        x5[0]=x4[0]+x4[1];
        x5[1]=x4[0]-x4[1];

        x6[0]=x2[7]+x5[0];
        x6[1]=x2[4]+x5[1];
        x6[2]=x2[7]-x5[0];
        x6[3]=x2[4]-x5[1];

        y[1]=.5*(x6[0]*c16+x6[1]*s16);
        y[7]=.5*(x6[0]*s16-x6[1]*c16);
        y[5]=.5*(x6[2]*c516+x6[3]*s516);
        y[3]=.5*(x6[2]*s516-x6[3]*c516);

/* Una vez completada la DCT en renglones se escriben en
bloque_t */
                for(i=0;i<8;i++)
                        bloque_t[i+(j*8)]=y[i];
        }

/* DCT en las columnas de bloque_t */
        for(j=0;j<8;j++)
        {
/* Se trabaja una columna de la unidad de datos */

                for(i=0;i<8;i++)
                        x[i]=bloque_t[(i*8)+j];
/* Sigue el código para la DCT */
```

```
/* Primera iteración */
                    x2[0]=x[0]+x[7];
          x2[1]=x[1]+x[6];
          x2[2]=x[2]+x[5];
          x2[3]=x[3]+x[4];
          x2[4]=x[3]-x[4];
          x2[5]=x[2]-x[5];
          x2[6]=x[1]-x[6];
          x2[7]=x[0]-x[7];

/* Mariposas superiores */
                    x3[0]=x2[0]+x2[3];
          x3[1]=x2[1]+x2[2];
          x3[2]=x2[1]-x2[2];
          x3[3]=x2[0]-x2[3];

                    y[0]=(x3[0]+x3[1])*c4d;
          y[4]=(x3[0]-x3[1])*c4d;
          y[6]=(x3[2]*(-c8d))+(x3[3]*s8d);
          y[2]=(x3[2]*s8d)+(x3[3]*c8d);

/* Mariposas inferiores */
          x4[0]=x2[6]*c4;
          x4[1]=x2[5]*c4;

          x5[0]=x4[0]+x4[1];
          x5[1]=x4[0]-x4[1];

          x6[0]=x2[7]+x5[0];
          x6[1]=x2[4]+x5[1];
          x6[2]=x2[7]-x5[0];
          x6[3]=x2[4]-x5[1];

          y[1]=.5*(x6[0]*c16+x6[1]*s16);
          y[7]=.5*(x6[0]*s16-x6[1]*c16);
          y[5]=.5*(x6[2]*c516+x6[3]*s516);
          y[3]=.5*(x6[2]*s516-x6[3]*c516);

/* Una vez completada la DCT en columnas se escriben en
bloque_t */
                    for(i=0;i<8;i++)
                              bloque_t[(i*8)+j]=y[i];
          }
}
```

idct.c (optimización Wang)

```
/*************************************************
***/
/* Nombre de archivo: idct.c                    */
/* Función: realiza dos antitransformadas coseno  */
/* de 1D de 64 valores                          */
/*************************************************
***/


/*Archivos #include */
#define _TI_ENHANCED_MATH_H 1
#include <math.h>

/* Variables externas */
extern short bloque[64];
extern float bloque_t[64];
extern const float c4, c8, s8, c16, s16, c516, s516;

/* Inicia la función idct */

void idct(void)
{
          unsigned short int i,j;
          float x[8], y[8], x2[4], x3[2], x4[4], x5[2], x6[2],
x7[4], x8[4];

/* IDCT en los renglones de bloque */
          for(j=0;j<8;j++)
          {
/* Se trabaja un renglón de la unidad de datos */

                    for(i=0;i<8;i++)
                              x[i]=bloque_t[i+(j*8)];
/* Sigue el código para la IDCT */

/* Mariposas superiores */
          x2[0]=x[0]*c4;
          x2[1]=x[4]*c4;
          x2[2]=x[2]*s8-x[6]*c8;
          x2[3]=x[2]*c8+x[6]*s8;

                    x3[0]=x2[0]+x2[1];
          x3[1]=x2[0]-x2[1];

          x4[0]=x3[0]+x2[3];
          x4[1]=x3[1]+x2[2];
          x4[2]=x3[1]-x2[2];
          x4[3]=x3[0]-x2[3];

/* Mariposas inferiores */
          x5[0]=x[3]*c4;
          x5[1]=x[5]*c4;

          x6[0]=x5[0]+x5[1];
          x6[1]=x5[0]-x5[1];

          x7[0]=x[1]+x6[0];
          x7[1]=x[7]+x6[1];
          x7[2]=x[1]-x6[0];
          x7[3]=x[7]-x6[1];

          x8[0]=x7[0]*c16+x7[1]*s16;
          x8[1]=x7[0]*s16-x7[1]*c16;
          x8[2]=x7[2]*c516+x7[3]*s516;
          x8[3]=x7[2]*s516-x7[3]*c516;

/* Mariposa Final */
          y[0]=.5*(x4[0]+x8[0]);
          y[1]=.5*(x4[1]+x8[3]);
          y[2]=.5*(x4[2]+x8[2]);
          y[3]=.5*(x4[3]+x8[1]);
          y[4]=.5*(x4[3]-x8[1]);
          y[5]=.5*(x4[2]-x8[2]);
          y[6]=.5*(x4[1]-x8[3]);
          y[7]=.5*(x4[0]-x8[0]);

/* Una vez completada la IDCT en renglones se escriben en
bloque_t */
                    for(i=0;i<8;i++)
                              bloque_t[i+(j*8)]=y[i];
          }

/* IDCT en las columnas de bloque_t */
          for(j=0;j<8;j++)
          {
/* Se trabaja una columna de la unidad de datos */

                    for(i=0;i<8;i++)
                              x[i]=bloque_t[(i*8)+j];
/* Sigue el código para la IDCT */

/* Mariposas superiores */
          x2[0]=x[0]*c4;
          x2[1]=x[4]*c4;
```

```
        x2[2]=x[2]*s8-x[6]*c8;
        x2[3]=x[2]*c8+x[6]*s8;

                x3[0]=x2[0]+x2[1];
        x3[1]=x2[0]-x2[1];

        x4[0]=x3[0]+x2[3];
        x4[1]=x3[1]+x2[2];
        x4[2]=x3[1]-x2[2];
        x4[3]=x3[0]-x2[3];

/* Mariposas inferiores */
        x5[0]=x[3]*c4;
        x5[1]=x[5]*c4;

        x6[0]=x5[0]+x5[1];
        x6[1]=x5[0]-x5[1];

        x7[0]=x[1]+x6[0];
        x7[1]=x[7]+x6[1];
        x7[2]=x[1]-x6[0];
        x7[3]=x[7]-x6[1];

        x8[0]=x7[0]*c16+x7[1]*s16;
        x8[1]=x7[0]*s16-x7[1]*c16;
        x8[2]=x7[2]*c516+x7[3]*s516;
        x8[3]=x7[2]*s516-x7[3]*c516;

/* Mariposa Final */
        y[0]=.5*(x4[0]+x8[0]);
        y[1]=.5*(x4[1]+x8[3]);
        y[2]=.5*(x4[2]+x8[2]);
        y[3]=.5*(x4[3]+x8[1]);
        y[4]=.5*(x4[3]-x8[1]);
        y[5]=.5*(x4[2]-x8[2]);
        y[6]=.5*(x4[1]-x8[3]);
        y[7]=.5*(x4[0]-x8[0]);

/* Una vez completada la IDCT en columnas se escriben en
bloque_t */
                for(i=0;i<8;i++)
                        bloque_t[(i*8)+j]=y[i];
        }

/* Pasa los valores flotantes de bloque_t a los valores enteros
de bloque */
  for(i=0;i<64;i++)
  {
        bloque[i]=(short)(roundf(bloque_t[i]));
        if((bloque[i]>127)||(bloque[i]<-128))
        {
                if(bloque[i]>127)
                        bloque[i]=127;
                else
                        bloque[i]=-128;
        }
  }

}
```

Glosario

A/D: abreviatura de "analógico(a) a digital".
C6711: diminutivo del procesador digital de señales modelo TMS320C6711 de Texas Instruments.
C6X: diminutivo de la familia de procesadores digitales de señales de Texas Instruments pertenecientes a las serie 6000.
CCS: siglas de las palabras en inglés *Code Composer Studio*, que dan nombre al ambiente integral que provee Texas Instruments para algunas familias de procesadores digitales de señales.
CIV: siglas de *campo instantáneo de vista*. Es el nombre que toma la unidad de un conjunto de muestras de una imagen discreta.
Cuantizar: procedimiento propuesto por el estándar JPEG para descartar coeficientes de una unidad de datos a la que se le haya aplicado la Transformada Coseno con el propósito de generar compresión con pérdidas.
D/A: abreviatura de "digital a analógico(a)".
DCT: siglas de las palabras en inglés *Discrete Cosine Transform*, que se traduce como *Transformada Coseno Discreta* y es el nombre completo de la Transformada Coseno.
Decuantizar: proceso inverso a la cuantización mediante el cual se recupera el valor aproximado de los coeficientes de una unidad de datos que no se hayan descartado durante el proceso de compresión.
DFT: siglas de las palabras en inglés *Discrete Fourier Transform*, que se traduce como *Transformada Discreta de Fourier*.
DHT: siglas de las palabras en inglés *define Huffman table(s)*, que se traduce como *definir tabla Huffman*. Las siglas dan nombre a una bandera utilizado en JPEG que marca el inicio de información que permite saber qué tablas Huffman fueron utilizadas en la compresión.
DMA: siglas de las palabras en inglés *Direct Memory Access*, que se traduce como *Acceso Directo a la Memoria* y consiste en una técnica que permite la transferencia de datos entre la memoria principal y un dispositivo sin pasar por la unidad de procesamiento central.
DPCM: siglas de las palabras en inglés *Differential Pulse Code Modulation*, que se traduce como *Modulación por Codificación Diferencial de Pulsos*. Dan nombre a un modelo de codificación en el cual se codifica la diferencia de valores entre dos muestras consecutivas de una señal en lugar de codificar el valor de cada muestra.
DQT: siglas de las palabras en inglés *define quantization table(s)*, que se traduce como *definir tabla de cuantización*. Las siglas dan nombre a una bandera utilizado en JPEG que marca el inicio de información que permite saber qué tablas de cuantización fueron utilizadas en la compresión.
DSK: siglas de las palabras en inglés *Digital Signal Processor Starter Kit*, que se puede traducir como *paquete para principiantes del procesador digital de señales*, y da nombre a una tarjeta en donde reside un procesador digital de señales (incluyendo el procesador) y que contiene memorias, periféricos, etc. para realizar pruebas.
DSP: siglas de las palabras en inglés *Digital Signal Processor*, que se traduce como *Procesador Digital de Señales*.

EMIF: siglas de las palabras en inglés *External Memory Interface*, cuya traducción es *Interfaz para Memoria Externa.*
EOB: siglas de las palabras en inglés *End of Block*, cuya traducción es *Fin de Bloque* y que da nombre a una bandera dentro del estándar JPEG que indica que los coeficientes restantes a partir de dicha bandera en una unidad de datos tienen un valor igual a cero.
EOI: siglas de las palabras en inglés *End of Image*, cuya traducción es *Fin de la Imagen* y dan nombre a una bandera del estándar JPEG que indica el fin de la información de la imagen.
FDCT: siglas de las palabras en inglés *Forward Discrete Cosine Transform*, que se puede traducir como *Transformada Discreta Coseno Directa*, y dan nombre al proceso de aplicar la Transformada Discreta Coseno a un conjunto de muestras para obtener un conjunto de coeficientes o amplitudes de las funciones base de dicha transformada.
FFT: siglas de las palabras en inglés *Fast Fourier Transform*, cuya traducción es *Transformada Rápida de Fourier* y dan nombre a un algoritmo optimizado que realiza la Transformada Discreta de Fourier a un conjunto de muestras en menos operaciones.
GIF: siglas de las palabras en inglés *Graphics Interchange Format*, cuya traducción es *Formato para Intercambio de Gráficos* y que dan nombre a un formato de imágenes comprimidas.
HPI: siglas de las palabras en inglés *Host-Port Interface*, cuya traducción es *Puerto Interfaz para el huésped.* Este puerto sirve para conectar y permitir el intercambio de información entre el procesador digital de señales y un dispositivo huésped (como una PC).
IDCT: siglas de las palabras en inglés *Inverse Discrete Cosine Transform*, cuya traducción es *Transformada Discreta Coseno Inversa.* Se refiere al proceso mediante el cual se recupera un conjunto de muestras a partir de un conjunto de coeficientes o amplitudes de funciones base. Es el proceso inverso a la Transformada Discreta Coseno Directa.
JFIF: siglas de las palabras en inglés *JPEG File Interchange Format*, que se traduce como *Formato de Intercambio de Archivos JPEG.* Es el formato de imágenes basado en el estándar JPEG más utilizado en el mundo.
JPEG: siglas de las palabras en inglés *Joint Photographic Experts Group* que dan nombre al grupo que desarrolló un estándar de compresión con el mismo nombre.
KLT: siglas de las palabras en inglés *Karhunen-Loeve Transform*, que se traduce como *Transformada Karhunen-Loeve.* Esta transformada decorrelaciona completamente una función aleatoria y nos entrega coeficientes dentro del dominio de la transformada
LZW: siglas de *Lempel-Ziv-Welch.* Da nombre a una variación del algoritmo de compresión LZ creada por las personas del mismo nombre.
MAC: siglas de las palabras en inglés *Multiply and Accumulate*, que se traduce como *Multiplicación y Acumulación.* Nombre de un proceso muy utilizado en los procesadores digitales de señales que consta de la multiplicación entre dos valores y su suma al resultado de la multiplicación entre dos valores anteriores.
McBSPs: siglas de las palabras en inglés *Multichannel Buffered Serial Port.* Da nombre a un puerto serial multicanal con la capacidad de mandar a memoria muestras que se almacenen temporalmente.
MCU: siglas de las palabras en inglés *Minimum Coded Unit*, que se traduce como *Unidad Mínima Codificada* y se refiere en el estándar JPEG al conjunto más pequeño de unidades de datos de una imagen que se pueden codificar.

MFLOPS: siglas de las palabras en inglés *Million Floating-Point Operations per Second*, que se traduce como *Millones de Operaciones de Punto Flotante por Segundo* y es una medida para generar una prueba patrón que evalúe la rapidez de un microprocesador.
MIPS siglas de las palabras en inglés *Million Instructions per Second*, que se traduce como *Millones de Instrucciones por Segundo* y es una medida del número de instrucciones que puede realizar un procesador en un segundo.
PDS: procesamiento digital de señales.
PNG: siglas de las palabras en inglés *Portable Network Graphics*, que se traduce como *Gráficos Portátiles de Redes* y dan nombre a un formato de imágenes comprimidas.
RGB: siglas de las palabras en inglés *Red-Green-Blue*, que se traduce como *Rojo-Verde-Azul* y sirven para hacer referencia al espacio o modelo de color que cuente con esas componentes.
RISC: siglas de las palabras en inglés *Reduced Instruction Set Computer*, que se traduce como *Procesadores de Conjunto Reducido de Instrucciones*. Este grupo de procesadores tiene como meta aumentar la rapidez de sus procesos mediante el uso exclusivo de un pequeño conjunto de instrucciones sencillas.
RST: diminutivo de la palabra en inglés *Restart*, cuya traducción es *Reinicio*. Dentro del estándar JPEG da nombre a una bandera cuyo objetivo es detectar los errores de corrupción de datos en una imagen recuperada.
SDRAM: siglas de las palabras en inglés *Synchronous Dynamic Random Access Memory*, que se traduce como *Memoria Síncrona de Acceso Aleatorio y Dinámico*.
SOI: siglas de las palabras en inglés *Start of Image*, cuya traducción es *Inicio de la Imagen* y da nombre a una bandera que marca propiamente el inicio de la información comprimida de una imagen en el estándar JPEG.
SOF: siglas de las palabras en inglés *Start of File*, cuya traducción es *Inicio del Archivo* y da nombre a una bandera que marca el inicio de la información de un archivo que contiene una imagen así como a los elementos necesarios para recuperarla en el estándar JPEG.
SOS: siglas de las palabras en inglés *Start of Scan*, que se puede traducir como *Inicio de Lectura* y da nombre a una bandera que marca el inicio de una lectura a una componente de la imagen (en el caso de que no estén entrelazadas las componentes) o de una lectura a todas las componentes de la imagen (en el caso de las componentes estén entrelazadas).
TC: diminutivo de *Transformada Coseno*.
TI: siglas de la marca *Texas Instruments*.
VLIW: siglas de las palabras en inglés *Very Long Instruction Word*, y da nombre a un diseño de microprocesadores donde cada instrucción cuenta dentro de sí con varias operaciones simples que no son interdependientes y se ejecutan en un ciclo de reloj; el procesador divide el trabajo dirigiendo cada operación a diferentes *unidades de ejecución*.
YCbCr: Componentes de un espacio o modelo de color con el mismo nombre. La Componente *Y* representa una componente de luminiscencia y las componentes *Cb* y *Cr* representan componentes de crominancia.

Bibliografía

[1] Pennebaker, William B. Mitchel, Joan L. **JPEG** Still image data compression standard
Van Nostrand Reinhold, New York, E.U., 1993

[2] García-Pelayo, R. Gross Pequeño Larousse Ilustrado
Ediciones Larousse, México, 1976

[3] Lira, Jorge La percepción remota: nuestros ojos desde el espacio
Fondo de Cultura Económica, México, 1987, 23p

[4] Jain, Anil K. Fundamentals of Digital Image Processing
Prentice Hall, New Jersey, E.U., 1989

[5] Miano, John **Compressed Image File Formats** JPEG, PNG, GIF, XBM, BMP
Addison Wesley, New York, E.U., 2000

[6] Wallace, Gregory K. The JPEG Still Picture Compression Standard
Communications of the ACM, Abril 1991

[7] Gutmann, Peter compression-faq/part2
ftp://rtfm.mit.edu/pub/usenet/news.answers/compression-faq/, 1999

[8] Rao, K.R. Yip, P. **Discrete Cosine Transform** Algorithms, Advantages, Applications
Academic Press, San Diego, E.U., 1990

[9] Webopedia What is RISC?
http://www.webopedia.com

[10] Application Report SPRA522 How to Begin Development with the TMS320C6711 DSP
Texas Instruments, Marzo 1999

[11] Chassaing, Rulph DSP Applications Using C and the TMS320C6x DSK
John Wiley & Sons, Inc., New York, E.U., 2002

[12] Reference Guide SPRU189F TMS320C6000 CPU and Instruction Set
Texas Instruments, Octubre 2000

[13] Dahnoun, Naim **DSP Implementation** using the TMS320C6000™ DSP Platform
Pearson Education Limited, Harlow, Inglaterra, 2000

[14] Liberty, Jesse **Teach Yourself C++ in 21 Days**
SAMS, 2da ed., Mayo 1997