Centro de Investigación y de Estudios Avanzados del
Instituto Politécnico Nacional
Unidad Zacatenco

Departamento de Computación

Implementación en FPGAs de Algoritmos de Compresión-Descompresión para Dispositivos Móviles

Tesis que presenta:
Oscar Alvarado Nava

para obtener el Grado de:
Maestro en Ciencias
en la Especialidad de:
Ingeniería Eléctrica
Opción:
Computación

Director de tesis:
Dr. Arturo Díaz Pérez

México, D.F. **Febrero de 2007**

Resumen

La Codificación Fractal de Imágenes (CFI) es una técnica de compresión con pérdidas cuyas características son muy prometedoras para dispositivos móviles de bajos recursos, sin embargo, se ha matenido marginada debido a la gran cantidad operaciones que deben de llevarse a cabo para la codificación, traduciéndose en un alto tiempo de cómputo.

Por otro lado, el desarrollo de la tecnología VLSI ha permitido crear dispositivos programables con mayores prestaciones, los cuales no solo ofrecen una gran densidad de compuertas para la programación de módulos de hardware, si no que además cuentan con uno o más procesadores empotrados, permitiendo con ello la creación de sistemas completos dentro de un solo chip (*System on Chip*, **SoC**). La utilización de componentes hardware y software en un único sistema electrónico permite combinar las características de flexibilidad que el software ofrece y la altas prestaciones de cómputo y paralelismo del hardware.

Por medio del Codiseño Hardware-Software (CHS), es posible desarrollar un sistema completo que mejore el desempeño de una solución puramente en software al llevar parte de la aplicación a un hardware de propósito específico. Inicialmente se realiza un análisis del perfil de rendimiento de la aplicación para localizar las partes más recurrentes y costosas en cómputo. Con clasificación de las partes en base a su costo, recurrencia y complejidad computacional, se lleva a cabo un particionamiento, lo cual implica llevar a hardware algunas de estas partes y mantener el resto en software. Finalmente se desarrollan interfaces o *drivers* para la integración de las nuevas partes de hardware con las partes en software.

En el presente documento se presenta el CHS para el desarrollo de un sistema de tipo **SoC** basado en FPGA, el cual mejora el desempeño de la solución en software de la CFI.

Abstract

The Fractal Image Coding (FIC) is a lossy compression technique with promising features for low-resources mobil devices. However, this technique has been left aside due to the big amount of operations required for the step. Hence, requiring large computing time.

On the other side, VLSI technology has evolved into programmable devices with higher capabilities. The devices not only offer a high gate density for programming hardware modules, and devices now have one or more embedded processors. This allows for complete system development inside a single chip (*System on Chip*, **SoC**). Hardware and software components in a single electronic system allow to exploid the software flexibility and hardware performance, as well as its parallelism.

A complete system can be developed with the Hardware/Software co-design (HSC) in a way that outperforms a pure-software implementation. The performance improvement is obtained when part of the application is implemented a specific-purpose hardware. Initially, an analysis of the performance is done to find the most recurrent and expensive (computationally speaking) parts. Some of the parts are implemented on hardware and some in software. The selection is done by classifying the parts based on cost, recurrence, and computational complexity. Finally, interfaces or *drivers* are developed for the integration of the new hardware parts with those of software.

The present document contains the HSC for the development of a **SoC** system based on FPGA. The system has a better performance than the FIC's software implementation.

Agradecimientos

- Al Departamento de Computación del Centro de Investigación y de Estudios Avanzados del IPN.

- Al Consejo Nacional de Ciencia y Tecnología por el apoyo económico otorgado.

- Al Consejo Nacional de Ciencia y Tecnología por financiar parcialmente este trabajo a través del proyecto 45306: Estudio, Análisis y Desarrollo de Algoritmos de Muy Alto Desempeño para Arquitecturas Hardware/Software

- A la División de Ciencias Básicas e Ingeniería de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Motivaciones

El volumen de datos electrónicos que maneja la humanidad crece de forma continua. A la enorme cantidad de información en formato electrónico que se genera cada día, se deberá de agregar todo el acervo documental e histórico que estará totalmente digitalizado para las primeras décadas del siglo XXI. La manipulación de la información en formato electrónico origina diferentes compromisos, entre los que destacan el almacenamiento y la transmisión. En algunos casos, la información enviada debe ser almacenada en el dispositivo que la recibe para su consulta y tratamiento, como sucede con la información que se maneja en Internet. Mucha de la información que se transmite debe ser almacenada por un intervalo de tiempo grande, mientras otra requiere menos tiempo de almacenamiento. En ambos casos, sin embargo, tienen como punto común el manejo de grandes volúmenes de información.

Por otro lado, los dispositivos móviles han adquirido gran popularidad debido principalmente a dos factores: el continuo decremento del costo de su fabricación y de servicio, y al poderoso atractivo que ofrecen las aplicaciones multimedia combinados con la tecnología inalámbrica. Entre los diferentes tipos de información que manipulan actualmente estos dispositivos, la información visual es la que mayor relevancia ha tenido y de manera específica las imágenes digitales, las cuales requieren de arreglos de pixeles para representar la información, cuando se busca tener imágenes de calidad. Estos arreglos ocupan cantidades importantes de recursos de almacenamiento y de un tiempo considerable para su transmisión a través de un canal de comunicación limitado y reducido.

Si bien es cierto que estos dispositivos móviles cada día son más poderosos, tienen necesidades específicas para la manipulación de imágenes digitales, ya que aún carecen de la capacidad de cómputo y almacenamiento adecuados para su manipulación.

La codificación o compresión de datos, y en especial la codificación de imágenes, ha cobrado gran importancia y continúan surgiendo métodos novedosos para representar la información de manera más eficiente. La compresión de imágenes puede ser sin pérdidas y con pérdidas. Las técnicas sin pérdidas normalmente no logran reducir el tamaño de las imágenes más allá de su tercera parte, pero son necesarias, ya que algunas imágenes (imágenes médicas o con texto, por ejemplo) se vuelven inservibles si se pierde alguna información en el proceso de descompresión. Ya que el ojo humano tiene límites, normalmente se puede tolerar alguna pérdida en la imagen al descomprimirla de forma que la imagen restaurada sea una aproximación cercana a la original, sin que se pierda la semántica de la información.

De entre una gran variedad de métodos de codificación de imágenes con pérdidas [1], la Codificación Fractal de Imágenes (CFI) es una técnica cuya alta tasa de compresión [2, 3, 4], rápida decodificación, independencia de resolución, progresividad de la codificación-decodificación [5] la hacen competitiva para mejorar la representación de la información y la transmisión de imágenes en dispositivos móviles de bajos recursos. El precio a pagar por las características anteriores es un considerable tiempo de codificación cuando se implanta el algoritmo en sistemas de procesamiento convencionales. Sin embargo, a pesar de que la CFI requiere de un gran número de operaciones para la codificación, se pueden aprovechar dos factores: en primer término, es un algoritmo asimétrico en donde el costo computacional de compresión es mucho mayor que el de descompresión, y por otra parte, sus operaciones son altamente paralelizables [6, 7, 8].

Durante los últimos años se han publicado numerosos trabajos que proponen mejoras sobre la CFI [9, 4, 10, 11, 7], prestando especial atención en su complejidad computacional, principalmente la temporal. La mayoría de estos trabajos centran sus esfuerzos sobre aspectos todavía abiertos en el algoritmo de la CFI, entre los que pueden destacarse la forma de particionar la imagen (segmentación) [12], la búsqueda eficiente de bloques, el mecanismo para determinar la correspondencia entre regiones, etc. En general, las propuestas anteriores y otras más ha sido implantadas puramente en software con mejoras de tiempo no muy significativas.

Otras trabajos reportados aprovechan el gran paralelismo del algoritmo, y proponen la implantación de la CFI en un hardware dedicado. Las arquitecturas propuestas se han implantado en circuitos integrados de propósito específico (ASIC,*Application-Specific Integrated Circuit*) [6] o en un dispositivos programables (FPGA, *Field-Programmable Gate Array*) [8, 13, 14, 15, 16]. Los resultados obtenidos con este tipo de soluciones muestran mejoras en los tiempos de codificación y en algunos casos de manera notable.

Analizando los resultados reportados en las dos vertientes de soluciones: únicamente software ó únicamente hardware, se observa por un lado que, la implementación en software de la CFI tiene un costo de desarrollo reducido debido a la gran flexibilidad de diseño, prueba y depuración que el software permite, pero su tiempo de ejecución es considerable y muy difícil de reducir. Para el caso de la implementación del mismo sistema puramente en hardware, se tienen aceleraciones notables para la codificación,

pero debido a que se trata de un sistema de gran complejidad, el tiempo de diseño, implementación y prueba es considerablemente costoso y poco flexible. Los elevados costos de desarrollo en hardware incluye a los realizados en dispositivos programables, debido que los desarrolladores de aplicaciones en estos dispositivos necesitan adquirir nuevas habilidades para la descripción de hardware. Por ejemplo, es común que un desarrollador se enfrente a la gran complejidad de describir en hardware de bajo nivel, funcionalidades de alto nivel que fueron hechas en software. Esta barrera entre únicamente software y únicamente hardware, obliga a los diseñadores a trabajar con parámetros totalmente opuestos a la hora de realizar un diseño que cumpla los requerimientos exigidos.

Ahora bien, el gran avance de la microelectrónica ha permitido que en los últimos dos años surjan novedosos dispositivos programables, FPGA (*Field-Programmable Gate Arrays*), los cuales además de contar con un sistema de compuertas altamente denso para la programación de hardware, contienen uno o más microprocesadores de propósito general, ya sean programados o incrustados. Con la aparición de estos dispositivos y sus herramientas de programación, se ha abierto un gran campo de desarrollo en los sistemas conocidos como sistemas empotrados (*embedded system*) o sistemas en un solo chip (SuC, *System on Chip*). Los SuC permiten, entre otras cosas, la aceleración de aplicaciones por medio del trabajo en conjunto de un microprocesador y hardware dedicado programado, todo en el mismo chip.

El diseño de sistemas cuyo principal objetivo es mejorar el tiempo de ejecución, promete ser ahora más sencillo y práctico para una amplia gama de aplicaciones, los cuales incluyen entre otros al procesamiento de imágenes, al procesamiento de señales y al cifrado de datos.

1.2. Objetivos

En esta tesis, se presenta la implementación de la CFI bajo el Codiseño Hardware-Software (CHS) en un SuC cuya plataforma es el FPGA XC2VP30 de la familia Virtex II Pro. El objetivo primordial de la tesis es el desarrollo de un sistema hardware-software cuyo desempeño sea mejor que la solución puramente en software de la CFI.

Para alcanzar el objetivo principal de la tesis, fue necesario resolver los siguientes objetivos previos, los cuales son parte de la metodología del CHS:

- **Implantar el programa que realiza la CFI en el SuC y evaluar su rendimiento.** Este objetivo se llevó a cabo en dos pasos. El primero fue implantar el algoritmo de la CFI en un lenguaje de alto nivel para su compilación y ejecución en sistema de cómputo convencional. El segundo paso fué crear un sistema de cómputo en el SuC y portar el programa al microprocesador incrustado. En ambos casos se llevaron a cabo experimentos para medir su tiempo de ejecución. Los experimentos incluyen la codificación de imágenes diferentes tanto en resolución como en contenido.

- **Análisis del perfil de rendimiento de la solución en software de la CFI.** El perfil de rendimiento se obtuvo por medio de herramientas de software, las cuales muestran entre otros resultados, el porcentaje de tiempo ocupado en cada función o procedimiento que integra la aplicación. El porcentaje es calculado en base al número de llamadas y al tiempo ocupado por cada llamada de la función. Clasificadas las funciones en base a su porcentaje de tiempo ocupado, se llevó a cabo el análisis de las funciones para determinar su viabilidad de ser candidatas a ser llevadas a hardware.

- **Particionamiento y desarrollo Hardware-Software.** En base al análisis del perfil de rendimiento, se determinó qué partes deberían de ser llevadas a hardware y cuáles deberían de permanecer en software. Hecha la partición, se dió inicio al diseño y desarrollo de las partes de hardware y a las modificaciones necesarias de las partes de software.

- **Desarrollo de un módulo de hardware para la aceleración de la CFI.** Las funcionalidades seleccionadas para ser llevadas a hardware, fueron descritas por medio de un lenguaje de descripción de hardware. La descripción dio como resultado una unidad de procesamiento (coprocesador) de propósito específico, que en conjunto con los microprocesadores incrustrados, acelerarán la aplicación.

- **Desarrollo de la interfaz hardware-software.** Para la integración de la unidad de procesamiento con los microprocesadores incrustados, se desarrolló una interfaz capaz de sincronizar y comunicar dichas unidades por medio de un sistema de buses. La interfaz consiste por un lado, en agregar elementos extras a la unidad de procesamiento y por el lado del software, la creación de funciones especiales para la comunicación y sincronización del hardware, comúnmente conocidos como *drivers*.

- **Validación y experimentos.** Integrado el sistema de sus partes de hardware y de software, se verificó inicialmente que realizara la codificación de las imágenes de forma correcta. Validado el sistema, se llevaron a cabo experimentos similares a lo realizados en la evaluación de la solución puramente en software, es decir, codificar imágenes con resoluciones y contenidos diferentes.

El perfil de rendimiento de la solución en software de la CFI, arrojó como resultado que dos funciones ocupaban en conjunto el 85 % del tiempo total de ejecución. Una de estas funciones es la encargada de aplicar alguna transformación sobre un conjunto de 64 pixeles, mientras que la segunda es la encargada de evaluar la distorsión entre dos conjuntos de 64 pixeles. Ambas funcionalidades, transformación y evaluación de distorsión, fueron llevadas a hardware dando como resultado la Unidad de Transformación y Comparación (UTC).

Al integrar una UTC con los microprocesadores (PowerPC 405) incrustados en el FPGA XC2VP30-FF896, se obtuvieron aceleraciones superiores a 8, cuando se codificaron de forma correcta las mismas imágenes utilizadas en los experimentos donde la solución era puramente en software, mejorando de manera notable el rendimiento de la CFI.

1.3. Organización de la tesis

La tesis está organizada de la siguiente manera: En el capítulo 2 se presenta una introducción a la CFI utilizando el Sistema de Funciones Iteradas Locales o Particionados (SFIL o SFIP). Se lleva a cabo en el mismo capítulo una descripción de la codificación, enumerando los pasos a seguir para llevarla a cabo y presentando los resultados más importantes: tiempos de codificación y decodificación, distorsión de la imagen y la relación de compresión.

En el capítulo 3 se introduce al cómputo reconfigurable y a la Metodología de Codiseño Hardware-Software (MCHS). En el mismo capítulo se describen los pasos que constituyen la MCHS y mediante un ejemplo sencillo se resaltan las consideraciones para el particionamiento hardware-software de una aplicación.

El capítulo 4 inicia con la descripción del algoritmo de la CFI en pseudocódigo y continúa con los resultados del perfil del programa que representa la solución en sofware de la CFI. Del perfil se extrae la información cuantitativa de las partes del programa que son las más costosas y por ello candidatas a ser implementadas en hardware. Hecho el particionamiento de la aplicación, se continua con la implementación de un circuito que lleva a cabo la parte más costosa en cómputo, así como la interfaz de comunicación entre el programa ejecutado en los procesadores y la unidad de hardware. En la última sección del mismo capítulo 4 se presenta el sistema final, el cual es programado en un FPGA.

En el capítulo 5 se presentan resultados comparativos entre las dos soluciones: únicamente software y la solución hardware-software, así como el análisis de los resultados. Finalmente, las conclusiones son presentadas en el capítulo 6.

Capítulo 2

Codificación Fractal de Imágenes

2.1. Introducción

El primer esquema de codificación de imágenes basado en técnicas fractales fue propuesto por Michael Barnsley [2] utilizando el Sistema de Funciones Iteradas propuestos por John Hutchinson en 1980. La Codificación Fractal de Imágenes (CFI) para imágenes en escala de grises fue desarrollada en su tesis doctoral por Arnaud Jacquin en 1989 mediante Sistemas Funciones Iteradas [17], pero actuando de manera particionada sobre la imagen, a éste método lo nombró Sistema de Funciones Iteradas Locales o Particionado (SFIL o SFIP). Un SFIL es el conjunto de funciones que describen partes de un fractal que, una vez juntas, recrean dicho fractal en su totalidad. Si una imagen puede ser descrita por un número pequeño de éstas funciones, el SFIL es una descripción bastante compacta de la imagen. La imagen puede ser rápidamente desplegada a cualquier grado de ampliación con ilimitados niveles de detalle fractal. El SFIL posee la propiedad de contractividad, y al aplicar valores de forma iterada a cada una de las funciones que lo componen, convergen a una imagen denominada atractor del sistema. El concepto básico detrás de CFI, es por tanto, tomar una imagen y expresarla como un SFIL [18]. El método de Jacquin es un método de codificación con pérdidas, lo cual significa que dada una imagen a modelar, el atractor generado por el SFIL no es idéntico a ella. El principal problema detrás de este novedoso método de codificación, es el tiempo para hallar los parámetros de las transformaciones adecuadas del SFIL que se aproxime con un grado deseado a una imagen dada.

2.2. Fractales en todas partes

En los años 70's comenzaron a vislumbrarse las aplicaciones de los fractales. En su tan citada obra *The Fractal Geometry of Nature*, Benoit Mandelbrot razonó que la naturaleza entiende mucho más de geometría fractal que de geometría diferenciable. En 1980 John Hutchinson publicó un trabajo en el que se desarrolla el concepto de conjunto autosemejante, estudiando las propiedades comunes (compacidad, autosemejanza etc.) de los fractales ya conocidos. Elaboró así mismo, una teoría unificada para la obtención de una amplia clase de conjuntos fractales: los conjuntos autosemejantes, de gran trascendencia en el desarrollo posterior de la geometría fractal. A partir de ahí, muchos científicos se han encontrado fractales en sus campos de estudio. El título de uno de los libros sobre el tema es bastante sugerente, *Fractals Everywhere* [19].

Básicamente los fractales se caracterizan por dos propiedades: autosemejanza (o autosimilitud) y autorreferencia. La autorreferencia determina que el propio objeto aparece en la definición de sí mismo, con lo que la forma de generar el fractal necesita algún tipo de algoritmo recurrente o iterativo. La autosemejanza implica invarianza de escala, es decir, el objeto fractal presenta la misma apariencia independientemente del grado de ampliación con que lo se observe, así, por más que se amplíe cualquier zona de un fractal, siempre hay estructura, hasta el infinito, apareciendo muchas veces el objeto fractal inicial, contenido en sí mismo.

Un ejemplo muy intuitivo que ayuda a la comprensión de los conjuntos autosemejantes es la divertida sensación que producen algunos libros en cuya portada un personaje muestra un libro cuya portada es igual a la del primer libro y en la que, por tanto, aparece el mismo personaje sosteniendo un libro con una portada igual a la del libro... Aunque, evidentemente, se trata de un montaje fotográfico y el nivel de profundización no es infinito, no nos resulta complicado imaginar una sucesión interminable del personaje sosteniendo un libro en el que aparece él mismo mostrando la misma portada. La situación anterior posee en cierta forma estructura fractal, ya que la invarianza a escala y la autosemejanza se manifiestan de manera notoria. La matemática de los conjuntos autosemejantes modelizan el comportamiento anterior y son el punto de partida hacia nuestro objetivo en el presente capítulo: la compresión de imágenes mediante Sistemas de Funciones Iteradas.

2.3. Sistema de funciones iteradas

En 1985, Michael F. Barnsley generalizó el método de Hutchinson. Mientras que el método Hutchinson utiliza semejanzas contractivas, Barnsley utiliza aplicaciones contractivas, lo que permite ampliar notablemente la familia de fractales obtenidos, de la que ahora los conjuntos autosemejantes son un subconjunto. Con estas mejoras, Michael Barnsley y su grupo de investigación del Instituto Tecnológico de Georgia fue el primero en desarrollar el potencial de los Sistemas de Funciones Iteradas para modelar y representar objetos como montañas, árboles, hojas y otros [2].

Las ecuaciones 2.1 forman un conjunto de cuatro funciones contractivas, que al aplicarle valores de forma iterada convergen a un punto fijo y en el proceso de convergencia forman una imagen binaria llamada atractor del Sistema de Funciones Iteradas. El atractor es una hoja de maple mostrada en la figura 2.1.

$$\begin{aligned}
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} 0{,}27 & 0{,}01 \\ 0 & 0{,}62 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 25 \\ -2 \end{bmatrix} \\
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} 0{,}18 & 0{,}52 \\ -0{,}40 & 0{,}36 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 56 \end{bmatrix} \\
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} 0{,}18 & -0{,}73 \\ 0{,}50 & 0{,}26 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 88 \\ 8 \end{bmatrix} \\
\begin{bmatrix} x' \\ y' \end{bmatrix} &= \begin{bmatrix} 0{,}04 & -0{,}01 \\ 0{,}50 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 52 \\ 32 \end{bmatrix}
\end{aligned} \tag{2.1}$$

El sistema de funciones iteradas que representa la imagen de la hoja de maple está formado por cuatro funciones, cada una de las cuales está determinada por 6 parámetros. Así, los parámetros de las cuatro funciones iteradas forman un código de 24 parámetros de la imagen de la hoja de maple. La representación binaria de esta imagen es de 32KB y la representación fractal utiliza 24 parámetros de 1 Byte por lo tanto la razón de compresión es de 1365.3. El código de 24 parámetros necesita mucha menos cantidad de datos para la representación de la hoja de maple en cualquiera de los formatos de imagen conocido.

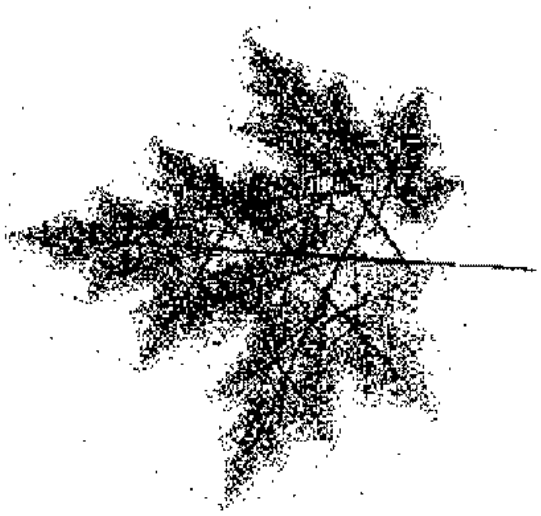

Figura 2.1: Hoja de maple en resolución de 512x512 pixeles generada mediante el sistema de funciones iteradas de la ecuación 2.1.

Resultados como estos motivaron enormemente las investigaciones de Barnsley, de hecho en 1987 Barnsley y sus colegas especularon con relaciones de compresión de 10000:1 y con la posibilidad de transmitir video en tiempo real a través de la línea telefónica convencional.

La codificación de la hoja de maple en unos cuantos coeficientes lleva a preguntarse si no será posible obtener codificaciones reducidas similares para cualquier imagen. Fue necesario el planteamiento en el sentido inverso: dada una imagen, determinar el conjunto de funciones iteradas que la puedan generar. El problema de encontrar para una imagen cualquiera un conjunto de funciones iteradas, que actuando en conjunto reproduzcan la imagen completa, es complejo y permanece sin solución [2, 7, 9, 18].

2.4. Sistema de funciones iteradas locales

El enfoque basado en SFI presenta un problema obvio: los fractales que genera un SFI poseen la propiedad de la autosemejanza, es decir, están formados por copias convenientemente transformadas de sí mismos. En el caso de una imagen de un pato, por ejemplo, uno debería poder observar patitos distorsionados por todo su plumaje, lo cual no es evidentemente una suposición muy natural. Los primeros intentos para adaptarse a esta característica de los SFI no produjeron resultados muy alentadores.

Fue en el año de 1989 que un alumno doctoral de Barnsley, Arnaud Jacquin, propuso en su tesis doctoral un método para codificar imágenes mediante SFI, actuando sobre una imagen particionada, dejando a un lado el rígido enfoque basado en SFI globales. La idea de Jacquin es a primera vista muy simple. En lugar de considerar una imagen conformada por copias de sí misma , ahora una imagen estará formada por copias de partes de sí misma, bajo las transformaciones adecuadas. Por ejemplo, en una postal veraniega es difícil que un trozo de nube se parezca a la postal completa, pero sí parece posible encontrar otra sección de alguna nube o de otro elemento de la imagen que se parezca al trozo de nube. La figura 2.2 muestra las regiones que son similares a diferentes escalas y bajo una transformación apropiada en dos imágenes diferentes. El enfoque general consiste en subdividir la imagen mediante una partición (en el caso más sencillo en regiones cuadradas de tamaño fijo) y encontrar para cada región resultante otra parecida en la imagen. Este esquema se conoce como sistemas de Funciones Iteradas Locales (SFIL) o Particionadas (SFIP). En algunas referencias al proceso de obtener el SFIL asociado a una imagen se le conoce como Transformada Fractal [2, 3, 9, 4].

El método de Jacquin modela muy bien imágenes en escalas de grises y puede adaptarse para trabajar sobre imágenes de color, separando la imagen en tres imágenes monocromáticas, las cuales pueden ser las correspondientes a los colores Rojo, Verde y Azul (Red, Green y Blue en inglés, respectivamente), formando la representación conocida como RGB de la imagen de color. Cada imagen componente del sistema RGB se codifica separadamente, pero los códigos entre imágenes componentes están relacionados y pueden predecirse unos a partir de los otros.

Figura 2.2: La idea clave de Jacquin fue considerar una imagen formada por copias de partes de sí misma. Las regiones R1 y R2 son similares bajo una transformación apropiada en ambas imágenes.

El método de Jacquin es un método de codificación con pérdidas, lo cual significa que dada una imagen a modelar, el atractor generado por el sistema de funciones iteradas particionado no es idéntico a ella. El método de codificación fractal de imágenes por medio de SFIL tiene varias propiedades: **Contractividad**, al aplicar valores de forma iterada a cada una de las funciones que lo componen, convergen a una imagen denominada atractor del sistema. **Compresión**, el conjunto de parámetros del sistema de funciones iteradas particionado tiene una representación más compacta que la representación en pixeles de la imagen. **Independencia de la resolución**, la imagen puede ser desplegada a cualquier escala sin perder detalles de la información.

En general la propiedades de este método de codificación abren un campo fabuloso de aplicaciones. Las aplicaciones posibles van desde la codificación y compresión de imágenes y de video, el cifrado de datos en una imagen, la transmisión progresiva de imágenes, la reconstrucción de imágenes a partir de versiones reducidas de ellas (*thumbnails* en inglés) y muchas más.

Los resultados anteriores significaron el inicio de una prolífica sucesión de investigaciones, que llega hasta nuestros días, para intentar mejorar numerosos aspectos todavía abiertos de la transformada fractal, y de manera específica en **la reducción de la complejidad computacional, principalmente la temporal** [7, 13, 14, 6], la cual es uno de los motivantes principales de esta tesis.

2.5. Transformada fractal

Como se ha comentado, el Sistema de Funciones Iteradas Locales (SFIL) fue propuesto en 1989 por Arnaud Jacquin, la referencia principal sobre este método de codificación es [17]. Además, es posible encontrar exposiciones más o menos profundas sobre el tema en muchas otras de las referencias de la bibliografía. Siguiendo particularmente a [2, 7, 20], describiremos cómo compactar fractalmente una imagen en escala de grises.

Para comprender mejor el proceso de codificación, observemos algunos detalles en la figura 2.3. En ella aparece la imagen Lena[1], la cual se utiliza comúnmente en procesamiento de imágenes. Imaginemos que dividimos la imagen en cuadrados de dos distintos tamaños. Se observa que en diferentes zonas de la imagen hay cuadrados que tienen estructuras muy similares. Por ejemplo, en el hombro tenemos dos cuadrados superpuestos de diferente tamaño, levemente desfasados pero que tienen estructura muy similar (la misma curvatura del hombro). Entonces, podemos pensar que es posible generar el cuadrado pequeño a partir del grande o viceversa, realizando los ajustes de tamaño correspondientes.

Figura 2.3: Porciones autosimilares de la imagen Lena.

Una situación similar tenemos entre los dos cuadrados del sombrero: el cuadro que está situado en la zona media izquierda del sombrero, y el otro de mayor tamaño, ubicado en el reflejo del sombrero en el espejo (en la zona inferior derecha de la imagen). Son de estructura muy similar (solamente tienen contraste y brillo diferente), y por ello, es posible generar el cuadrado pequeño a partir del grande, realizando

[1]Playboy Magazine 1972

los ajustes de contraste, brillo y tamaño correspondientes. Al generalizar esta idea obtenemos el proceso de compresión.

El método de Jacquin establece que una imagen I en escala de grises puede ser aproximada por medio de un sistema de $N_R \times N_R$ funciones iteradas:

$$W = w_1 \cup w_2 \cup w_3 \cdots \cup w_u \cdots \cup w_{NRxN_R} \tag{2.2}$$

las cuales tienen la forma:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = w_u \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a_u & b_u & 0 \\ c_u & d_u & 0 \\ 0 & 0 & s_u \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} e_u \\ f_u \\ o_u \end{bmatrix} \tag{2.3}$$

La imagen es partida en $N_R \times N_R$ bloques rangos (se utilizan bloques cuadrados, pero podrían tener cualquier forma) de $n \times n$ pixeles, siendo cada uno de ellos aproximado por medio de una transformación contractiva afín w_u, aplicada sobre un bloque de dimensiones mayores dentro de la misma imagen. La imagen original dividida en bloques rangos es llamada imagen rango.

Se forma un conjunto de $(N_R - 1) \times (N_R - 1)$ bloques denominados dominios, los cuales tienen dimensiones $2n \times 2n$ pixeles, y se traslapan por la mitad. El conjunto de todos los bloques dominio formarán la llamada imagen dominio. La imagen dominio tiene dimensiones de $2n(N_R - 1) \times 2n(N_R - 1)$ pixeles. La imagen dominio forma un libro de códigos, obtenidos de la misma imagen y está contenido en ella.

Para cada bloque dominio se busca en la imagen rango el bloque que más se le asemeja, cuando el bloque rango es sometido a una transformación contractiva afín del tipo mostrado en la ecuación (2.3). La transformación w_u modifica la posición, el tamaño, la orientación, el brillo, y el contraste del bloque rango para aproximarlo al u-ésimo bloque dominio. Los parámetros a_u,b_u,c_u,d_u,e_u,f_u,s_u y o_u de la transformación, son seleccionados para obtener la mejor aproximación de un bloque rango a un bloque dominio. Los elementos $\frac{1}{2}$ en la matriz diagonal efectúan una contracción espacial reduciendo las dimensiones del bloque rango a la mitad, tanto en la dirección horizontal como en la dirección vertical.

Los parámetros a_u,b_u,c_u y d_u se escogen en conjunto para efectuar una de 8 transformaciones isométricas, las cuales se muestran en la tabla 2.1. Las transformaciones se aplican a los bloques rango y efectúan rotaciones y/o reflexiones sobre ellos. La transformación isométrica seleccionada se representa con el parámetro m_u, esto significa que a través del valor de m_u se especifica el conjunto de parámetros a_u,b_u,c_u y d_u. El valor de la isometría m_u tiene una representación binaria que utiliza tres bits. Los parámetros e_u y f_u realizan la translación del bloque rango desde las coordenadas del bloque dominio hasta las coordenadas del bloque rango, y los parámetros s_u y o_u ajustan el contraste y el brillo del bloque rango para que se asemeje al bloque dominio.

Los parámetros s_u y o_u se pueden calcular de acuerdo a diferentes criterios. Uno de ellos es al minimizar el error cuadrático medio MSE de la aproximación entre el u-ésimo bloque dominio D_u y el v-ésimo bloque rango transformado $R'_v = s_u R_v + o_u$.

m_i	Transformación	$\begin{pmatrix} a_u & b_u \\ c_u & d_u \end{pmatrix}$
0	*identidad*	$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
1	*Reflexión en x, x=-x*	$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$
2	*Reflexión en y, y=-y*	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
3	*Rotación de 180º a la derecha*	$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$
4	*Reflexión Diagonal y=x*	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$
5	*Rotación de 90º a la derecha*	$\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$
6	*Rotación de 270º a la derecha*	$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$
7	*Reflexión Diagonal x + y = 0*	$\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}$

Tabla 2.1: Transformaciones isométricas

Otro más sencillo es calculado el valor promedio de los pixeles de los bloques D_u y R_v y con los resultados calcular su diferencia. El resultado de la diferencia es el valor de cambio de la luminancia, el cual representa en conjunto el valor promedio de brillo y contraste.

Cada bloque rango es aproximado por 5 parámetros (m_u,e_u,f_u,s_u,o_u), los cuales son denominados código fractal del bloque rango o mapa de afinidad del bloque rango. El conjunto de los códigos fractales de todos los bloques rangos que forman la imagen constituyen el código fractal de la imagen.

Para evaluar la semejanza entre un bloque dominio D_u y un bloque rango transformado $R_v^{'}$ se utiliza la distorsión δ. Los criterios de distorsión comunes incluyen la media de las diferencias absolutas (MDA) la raíz del error cuadrático medio (RECM) que se definen según las ecuaciones 2.4 y 2.5, respectivamente.

$$\delta_{MAD}(D_u, R_v^{'}) = \frac{1}{n^2} \sum_{k=1}^{n} \sum_{l=1}^{n} \left| d_u(k,l) - r_v^{'}(k,l) \right| \tag{2.4}$$

$$\delta_{RMSE}(D_u, R_v^{'}) = \frac{1}{n^2} \sqrt{\sum_{k=1}^{n} \sum_{l=1}^{n} (d_u(k,l) - r_v^{'}(k,l))^2} \tag{2.5}$$

El error RMSE proporciona una mejor evaluación de la aproximación de un bloque rango transformado con un bloque dominio, pero la MAD es una medida de distorsión que se calcula con más facilidad.

En resumen, para llevar a cabo al codificación, es necesario la búsqueda de redundancias dentro de la propia imagen a través del mapeado con SFI locales, las cuales son un SFI que en lugar de actuar sobre todo el plano que define a la imagen, actúan solamente sobre una parte limitada de la imagen, de ahí el nombre de locales. Inicialmente se divide la imagen en un conjunto de cuadrados adyacentes disjuntos, llamados bloques rango. Por otra parte, se divide la imagen en cuadrados no disjun-

tos de mayor tamaño que los bloques rango, llamados bloques dominio. Entonces, para cada bloque dominio de la imagen, se buscará entre los bloques rango el que sea más similar (al aplicarle las trasformaciones y cambios de luminancia necesarios) y se almacenarán las coordenadas del bloque rango elegido, la diferencia de brillo y contraste respectivo, y la transformación requerida (en el caso que el bloque dominio y bloque rango tengan distinta orientación), en lo que llamaremos mapa de afinidad. Los mapas de afinidad, se almacenan en un archivo (por ejemplo con extensión `.fif`, de *Fractal Image Format*), y constituirán los patrones para la descompresión y así poder reconstruir la imagen original. Como se puede deducir, este proceso se hace muy lento al tener que trabajar con conjuntos de muchos elementos. En la siguiente sección se presenta un ejemplo cuantitativo.

2.6. Ejemplo de codificación

Con la intensión de mostrar un ejemplo cuantitativo, se enumeran los pasos a seguir para realizar la codificación, para que con ello se pueda deducir la cantidad de operaciones necesarias para codificar una imagen. Además, se hacen varios comentarios para su implementación en un lenguaje de alto nivel.

1. **Bloques rango.**
   Conjunto disjunto de $N_R \times N_R$, cada uno formado de $n \times n$ pixeles.
   Tomando la imagen Lena en resolución 512×512 pixeles, como imagen a codificar, se hace con ella una partición en 64×64 bloques rangos teniendo 8×8 pixeles cada uno de ellos. $N_R = 64$, $n = 8$.

2. **Bloques dominio.**
   Conjunto no disjunto de $N_D \times N_D$, cada uno de ellos conteniendo $2n \times 2n$ pixeles.
   Si consideramos que los bloques podrían estar traslapados por la mitad, se tienen 63×63 bloques dominio, cada uno de ellos con 16×16 pixeles. No es necesario tener físicamente separados al conjunto de bloques dominios o al conjunto de bloques rango en secciones diferentes de memoria, cada bloque se extrae de la misma imagen de entrada al mantener índices y desplazamientos adecuados para cada tipo de bloque.

3. **Extracción de bloques rango y dominio.**
   Se extrae de la imagen un bloque rango y se calcula el valor promedio de sus pixeles. Con el desplazamiento adecuado se extrae de la imagen un bloque dominio, el cual tiene el cuádruple de pixeles que un bloque rango.

4. **Contracción espacial del bloque dominio.**
   El bloque dominio es reducido de tamaño submuestreando o decimando sus columnas y sus renglones en un factor de dos, cada uno de ellos.
   Por ejemplo, puede tomarse el valor promedio de cuatro pixeles contiguos y este

valor promedio los representa a esos cuatro pixeles. La decimación es preferida al submuestreo porque evita el aliasing, que es el efecto producido por el traslape del espectro (en la banda de frecuencia) de la imagen.

5. **Cálculo del valor de luminancia promedio.**
   Contraido espacialmente el bloque dominio, se calcula el valor promedio de sus pixeles. La diferencia entre los promedios de los pixeles del bloque rango y del bloque dominio, será el valor de cambio de lumninancia o intensidad para el bloque rango a transformar.

6. **Cálculo del error de aproximación del bloque dominio al bloque rango mediante un criterio de distorsión δ.**
   Se calcula la semejanza (o distancia) entre el bloque rango transformado y el bloque dominio, utilizando un criterio de distorsión, como la MAD o el error RMSE.
   En este trabajo de tesis se utiliza la MAD, descrita en la ecuación 2.4, para evaluar la semejanza entre los bloques dominios transformados y los bloques rangos; y se utiliza el RMSE y la PNSR descritas en las ecuaciones 2.5 y 2.6 respectivamente, para evaluar la semejanza de la imagen decodificada con la imagen original.

7. **Evaluación de la distorsión y selección del mejor mapa de afinidad**
   Se evalúa si el error de aproximación del bloque rango que se procesa es el menor error de aproximación que se produce entre todos los bloques rango que ya se hayan procesado. Si lo es, entonces se seleccionan los parámetros de la transformación 2.3 del bloque que está bajo consideración. En caso de que el bloque rango actual no produzca el menor error de aproximación, entonces se considera otro bloque rango, y así sucesivamente hasta que se hayan considerado todos los bloques rango.

8. **Selección de otro bloque dominio**
   Una vez que se ha completado la comparación de un bloque dominio, comparándolo con todos los bloques rango, se toma el siguiente bloque dominio, y el proceso de codificación descrito se repite para este nuevo bloque. El proceso entero continúa hasta que ya se hayan codificado todos los bloques dominio.
   El código fractal de la imagen de este ejemplo consta de los 4 parámetros fractales para cada una de las 4096 funciones iteradas que actúan sobre la imagen particionada en $64 \times 64 = 4096$ bloques rangos.

La figura 2.4 muestra de forma esquemática el proceso de comparación de bloques dominio y bloques rango. La imagen de Lena es particionada en 1024 (32×32) bloques rango y 256 (16×16) bloques dominio. Por claridad se muestra un solo bloque dominio, el cual es del doble de tamaño de un bloque rango. El bloque rango seleccionado es sometido, inicialmente, a un cambio de luminancia y posteriormente se le aplica una de las ocho transformaciones, el la figura se muestra la transformación identidad. Por otro lado, al bloque dominio es reducido a la mitad de su tamaño. Una vez tratado los dos bloques, se lleva a cabo el cálculo de la distorsión entre bloques, el cual indica que tan parecidos son los bloques.

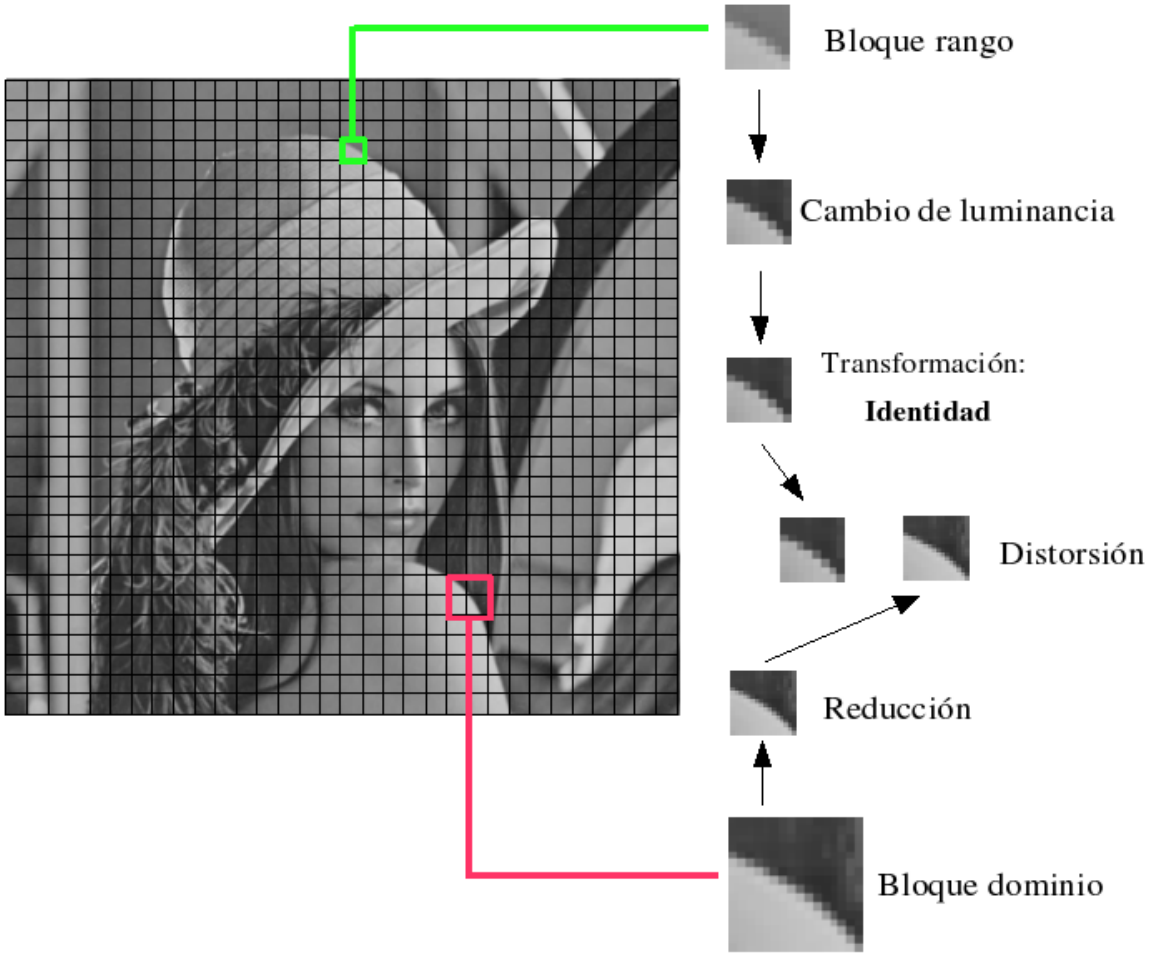

Figura 2.4: de flujo para el proceso de compresión utilizando la Transformada Fractal.

El proceso de codificación continúa al transformar y comparar cada bloque rango con todos los bloques dominio. El proceso completo se ilustra gráficamente por medio del diagrama de flujo mostrado en la figura 2.5.

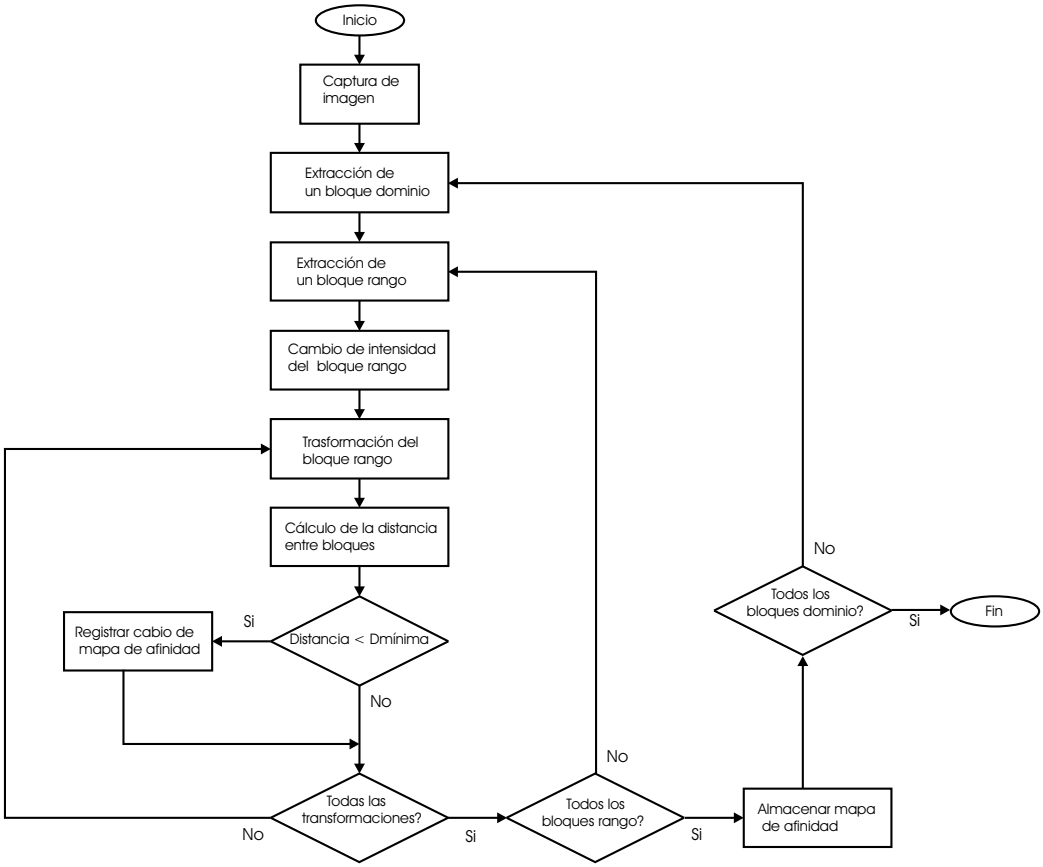

Figura 2.5: Diagrama de flujo para el proceso de compresión utilizando la Transformada Fractal.

En la figura 2.6 muestran las imágenes reconstruidas de la imágenes Lena, Bird y Golden Hill, todas ellas de 512×512 pixeles, pero codificadas con tamaño de bloque rango diferente. Las imágenes representan el atractor de un sistema de Funciones Iteradas Locales que se ha formado para cada una de las imágenes siguiendo los pasos dados en el ejemplo anterior.

2.7. Propiedades de la codificación fractal

2.7.1. Tiempos de codificación y decodificación

El tiempo de codificación depende de varios factores, pero principalmente crece cuando el número de los bloques en los que se ha dividido la imagen aumenta, ya que deben de realizarse un gran número de operaciones entre bloques y entre pixeles. En el ejemplo anterior, para codificar una imagen Lena de 512×512 particionada en 64×64 bloques rangos de 8×8 pixeles y con 63×63 bloques dominios, se realizan $64 \times 64 \times 8 \times 63 \times 63 = 130,056,192$ transformaciones y comparaciones de bloques rango con bloques dominio.

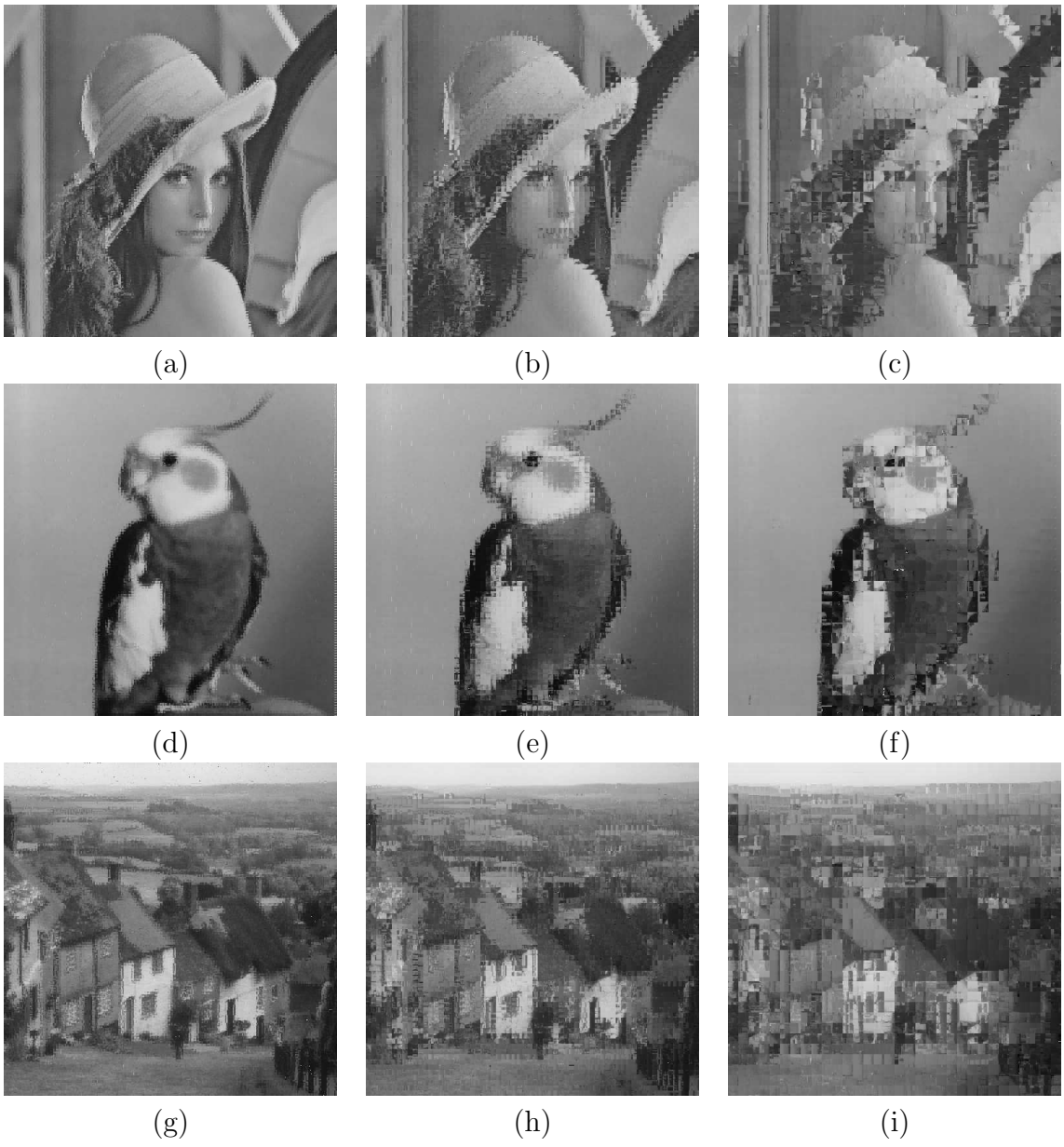

Figura 2.6: Aproximación fractal de varias imágenes utilizando bloques rangos de distinto tamaño. (a) Lena 4×4 pixeles, (b) Lena 8×8 pixeles, (c) Lena 16×16 pixeles; (d) Bird 4×4 pixeles, (e) Bird 8×8 pixeles, (f) Bird 16×16 pixeles; (a) GoldenHill 4×4 pixeles, (b) GoldenHill 8×8 pixeles, (c) GoldenHill 16×16 pixeles;

En [7] se muestra un análisis del número de necesarios para la transformación y comparación de bloques, los cuales dependerán del tamaño del bloque rango y de la arquitectura del procesador en el cual se implemente. El gran número de pasos necesarios para la codificación de la imagen Lena de 512×512 es corroborado al medir el tiempo de ejecución de un programa que implementa la CFI en un sistema

de cómputo convencional. Además, con el experimento se comprueba que los procesos de codificación y decodificación son asimétricos.

El proceso de decodificación se realiza en orden inverso al proceso de codificación, sin embargo, el tipo y número de procesos no es el mismo. La decodificación de la imagen se realiza de manera iterativa y el proceso converge a la formación del atractor de la imagen porque cada una de las transformaciones que forman el Sistema de Funciones Iteradas Locales es una transformación contractiva. Por esta razón el proceso de decodificación se itera hasta que se logra la convergencia de la imagen, la cual se estima por medio del error que se produce al formar la imagen en cada nueva iteración con respecto a la imagen formada en la iteración anterior. Si el error entre estas versiones de la imagen es menor que un valor de umbral preestablecido, entonces se acepta que la imagen ha convergido y se termina el proceso de decodificación. Para propósitos de comparación es conveniente iterar el mayor número de veces, pero en términos visuales, 10 iteraciones son suficientes.

En la tabla 2.2 se presentan los tiempos de codificación y decodificación de las imágenes de la figura 2.6, obtenidos al implantar el algoritmo de la Transformada Fractal propuesto en [2] en un sistema de cómputo convencional con las siguientes características: procesador DoubleCore Pentium IV 3.2 Ghz, 2MB de L2 cache, bus principal de 800Mhz, 1 GB de SDRAM a 800Mhz, Linux 2.6.11-1smp.

Imagen Codificada	**Partición** [pixeles]	**Tiempo de Codificación** [seg]	**Tiempo de Decodificación** [seg]
Lena	4×4	7740	0.39
	8×8	7200	0.36
	16×16	6480	0.34
Bird	4×4	8640	0.39
	8×8	8496	0.36
	16×16	8480	0.34
GoldHill	4×4	8892	0.59
	8×8	8100	0.52
	16×16	7740	0.34

Tabla 2.2: Tiempos de codificación y decodificación.

2.7.2. Calidad de la imagen

La calidad de imagen que se obtiene en el método de codificación fractal presentado depende de los siguientes factores:

- El tamaño de los bloques rangos.
- El contenido de la imagen.

La tabla 2.3 muestra los parámetros PSNR y RMSE (ecuaciones 2.6 y 2.5) obtenidos al comparar las imágenes decodificadas mostradas en la figura 2.6 con las imágenes originales correspondientes. Dichos parámetros nos permiten evaluar la calidad de imagen obtenida en función de la partición utilizada.

Como se menciona, se ha utilizado como medida de distorsión el pico de la relación señal-ruido, PSNR (*Peak Signal-to-Noise Ratio*), como criterio de distorsión entre las imágenes original y reconstruida, ecuación 2.6. El valor del error se calcula en la señal de luminancia solamente, es decir, los valores de los pixeles de las imágenes cuyo rango es de $\left[0, 2^b - 1\right]$ dónde b es el número de bits. Las imágenes codificadas se encuentran en escala de grises, con una representación de ocho bits por pixel, es decir, del negro (0) al blanco (255).

Se utiliza el RMSE descrito en la ecuación 2.5 para calcular el PSNR, el cual se muestra en la ecuación 2.6. Cuando se utiliza el PSNR, las imágenes reconstruidas con valores más altos se juzgan como mejores.

$$\delta_{PNSR} = 20 \log_{10} \left(\frac{255}{RMSE} \right) \tag{2.6}$$

Imagen Codificada	**Partición** [pixeles]	**PSNR** [dB]	**RMSE**
Lena	4×4	81.4820	0.0215
	8×8	77.9736	0.0322
	16×16	75.6451	0.0421
Bird	4×4	84.0484	0.0160
	8×8	79.8648	0.0259
	16×16	77.7605	0.0330
GoldHill	4×4	79.8648	0.0259
	8×8	77.5524	0.0338
	16×16	75.5019	0.0428

Tabla 2.3: Calidad de las imágenes codificadas en función de la partición.

Con base en los datos presentados en la tabla 2.3 y en las imágenes de la figura 2.6 se puede concluir que la codificación que usa los bloques rangos más pequeños logra una mejor aproximación a la imagen original.

2.7.3. La compresión de la imagen y la razón de compresión

Para el caso de la compresión de la imagen y la razón de compresión se parte del análisis presentado en [20, 2, 7]. Debido a que cada bloque de imagen es representado por cinco parámetros fractales, independientemente del tamaño de bloque, la codificación fractal de imágenes mediante SFIL posee la propiedad de compresión de imagen. La compresión de imagen es la propiedad de un sistema codificador de imagen de representar a una imagen utilizando una cantidad menor de bits que la representación original. La capacidad de un sistema codificador para comprimir a una imagen se evalúa mediante la Razón de Compresión (RC). La RC lograda por un método de codificación se define como la razón entre el número de bits que se utilizan para representar un pixel de una imagen y el número de bits que utiliza la representación de un pixel en el método de codificación. Supóngase, como referencia, que un pixel de imagen monocromática se representa utilizando una escala de 256 niveles de intensidad. Para codificar los 256 niveles de intensidad se necesitan 8 bits. Entonces las imágenes monocromáticas se representan usando 8 bits/pixel. En el método de codificación fractal no se codifican pixeles sino bloques de imagen, los cuales son llamados bloques rango.

Si se usa una partición regular, con bloques rango de tamaño fijo, la razón de compresión depende de los siguientes factores:

- El número de bloques que componen el conjunto de bloques dominio.
- El tamaño de bloque rango.
- La cuantificación de los parámetros fractales.
- La representación del código fractal o mapa de afinidad.

Cada bloque rango es especificado por los parámetros fractales $\{m_u, s_u, o_u, e_u, f_u\}$. El parámetro m_u tiene una representación binaria de $N_{bm} = 3$ bits. El parámetro s_u se representa con N_{bs} bits, y el parámetro o_u con N_{bo} bits. El número de pixeles N_{be} y N_{bf} utilizados para representar a los parámetros e_u y f_u depende del número de bloques rangos y de bloques dominios que se consideren en la codificación. Si una imagen se codifica utilizando $N_R \times N_R$ bloques rangos y $N_D \times N_D$ bloques dominio traslapantes por la mitad, entonces el número de bits para representar los parámetros de desplazamiento e_u y f_u está dado por el número de bits que se necesitan para representar N_R. Si $N_R = 2^k$, entonces se utilizan $k = \log_2(N_R)$ pixeles en la representación de N_R, así como para la representación de e_u y de f_u.

El número de bits N_{bF} usados en la representación fractal de un bloque rango estará dado por la ecuación 2.7.

$$N_{bF} = N_{bm} + N_{bs} + N_{bo} + 2\log_2(N_R) \tag{2.7}$$

Para representar los pixeles de un bloque rango que tiene $n \times n$ pixeles se necesitan $8(n \times n)$ bits. Por ello, la razón de compresión está dada por la ecuación 2.8.

$$RC = \frac{8(n \times n)}{N_{bm} + N_{bs} + N_{bo} + 2\log_2(N_R)} \tag{2.8}$$

Los parámetros fractales se pueden cuantizar, cuando se trata de llevar a cabo compresión de imagen. En [21] se aconseja la cuantización para los parámetros fractales s_u y o_u y luego codificar, en oposición a codificar y luego cuantizar los parámetros fractales. Por ejemplo, en [22] se usa $N_{bo} = 6$ bits para el ajuste de brillo y $N_{bs} = 5$ bits para el ajuste de contraste. Inclusive puede asignarse un valor fijo al parámetro de ajuste de contraste, con lo cual ya no necesita incluirse este parámetro en el código fractal, si se conviene en manejar este valor en todos los decodificadores. O puede asignarse a él un valor fijo que tenga una representación de menos de 5 bits. En [2] se lleva a cabo el cambio de **luminancia**, representado con N_{bl} bits, y se almacena dicho parámetro en lugar de los parámetros de brillo y contraste. La luminancia se calcula como la diferencia entre los promedios de los valores de los pixeles de los bloques rango y dominio. Dicho valor puede ser representado con 8 bits con un rango de $[-127, 128]$. La ecuación 2.8 es modificada para incluir el cambio de luminancia y queda como se muestra en la ecuación 2.9.

$$RC = \frac{8(n \times n)}{N_{bm} + N_{bl} + 2\log_2(N_R)} \tag{2.9}$$

Tomando como ejemplo una imagen de resolución 512×512 pixeles, particionada en $N_R = 64$, $n = 8$, y usando $N_{bl} = 8$ bits para el ajuste de la luminancia (brillo y contraste en conjunto), se obtiene una razón de compresión RC = 22.3. En la tabla 2.4 se muestra la razón de compresión que se obtiene al codificar imágenes de diferentes resoluciones y distintos tamaños de bloque.

Tamaño	**Resolución de imagen**		
de bloque	128×128	256×256	512×512
4×4	6.1	5.6	5.1
8×8	26.9	24.4	22.3
16×16	120.5	107.8	97.5

Tabla 2.4: Relación de compresión para imágenes con distintas resoluciones.

Los parámetros de posición e_u y f_u suelen limitarse acotando la búsqueda de similitud a los bloques dominios más cercanos a cada bloque rango [20]. De esta manera se reduce el número de bits en comparación con la búsqueda en la totalidad del conjunto de bloques dominios. El parámetro de ajuste de brillo o_u puede codificarse de modo más eficiente utilizando codificación diferencial, incrementando la razón de compresión [20]. La representación del código fractal puede aún almacenarse de modo más eficiente si se utiliza codificación Huffman [20].

La tabla 2.5 se muestra el tamaño en bytes que ocupa un archivo que contiene diferentes formatos de la imagen de Lena en una resolución de 512×512 pixeles. Los formatos incluyen mapas de bits (TarGA, *TARGA of Truevision*), una codificación sin pérdidas (PNG, *Portable Network Graphics*), una codificación con pérdidas (JPEG, *Joint Photographic Experts Group*) y la CFI (FIF, *Fractal Image Format*). La CFI fue realizada con un tamaño de bloque de 8×8 pixeles y cada uno de los 4 parámetros fractales fueron codificados en 1 byte. El conversión de formatos (excepto la de FIF) se llevó a cabo con el GIMP 2.2.

TarGA	**PNG**	**JPEG**	**FIF**
262,162	109,031	32,458	16,402

Tabla 2.5: Tamaño de archivos con diferente formatos para la imagen de Lena.

2.7.4. Independencia de la resolución

El código fractal o mapas de afinidad de una imagen codificada no hacen referencia alguna al tamaño de la imagen de la cual se obtuvo. Tampoco hace referencia al tamaño de los bloques rangos. El código fractal puede iterarse sobre diferentes resoluciones y siempre convergerá a una imagen llamada atractor del código fractal, o atractor del sistema de funciones iteradas particionado. Ese atractor es una versión de la imagen original de la cual se obtuvo el código, pero con una resolución diferente. La representación del código fractal en resoluciones mayores a la original, da origen a una forma de interpolación fractal, en la cual se añade información concordante con el contexto. Con esta forma de interpolación se puede realizar también la función de acercamiento o zoom fractal, que representa una porción de la imagen en un área mayor. El zoom fractal no permite observar mayor detalle en la imagen, pero presenta una mejor representación que el zoom realizado mediante replicación de pixeles.

2.8. Conclusiones

En este capítulo se ha realizado una introducción a la Codificación Fractal de Imágenes. Se ha realizado un análisis del método de codificación con el objetivo de identificar las principales desventajas del método y poder proponer una solución. En resumen, el método de codificación fractal por medio de Sistemas de Funciones Iteradas Locales presenta las siguientes características.
Ventajas:

1. Codificación por parámetros de transformación.
2. Libro de código (conjunto de bloques dominios) auto-contenido en la imagen a codificar.
3. Posee la propiedad de compresión de imagen.
4. La razón de compresión puede incrementarse mediante cuantización de parámetros.
5. Pueden usarse otros esquemas de compresión como la codificación diferencial y la codificación Huffman para aumentar la razón de compresión del código fractal.
6. El código posee la propiedad de Independencia de la Resolución.
7. Tiempos de codificación-decodificación asimétricos.
8. Algoritmo altamente paralelizable.

Desventajas:

1. Tiempo de codificación muy grande, debido a que requiere un número grande de operaciones en la codificación, las cuales se llevan a cabo de manera serial.
2. Calidad de imagen dependiente del tamaño de bloque de la partición.

En el siguiente capítulo se lleva a cabo una introducción al cómputo reconfigurable y codiseño hardware-software. Ambos temas necesarios para la compresión de los sistemas empotrados o sistemas en un solo chip. El capítulo 4 se presenta un sistema hardware-software que lleva a cabo la Codificación Fractal de manera eficiente. El sistema es implantado en un dispositivo programable, el cual además de contar con una gran cantidad recursos para la programación de hardware, tiene incrustados dos procesadores de propósito general.

Capítulo 3

Codiseño Hardware-Software

3.1. Introducción

El objetivo de las técnicas de Codiseño Hardware-Software (CHS), desarrolladas a lo largo de la década de los 90's, ha sido distribuir una aplicación entre dos o más particiones hardware y software. La inclusión de parte de la funcionalidad en hardware tiene como principal objetivo conseguir acelerar la ejecución de la aplicación. El mantenimiento de partes de la aplicación en software, ejecutado por un microprocesador de propósito general (CPU, *Central Processing Unit*), se debe a que estas partes no son críticas desde el punto de vista de la velocidad de ejecución, y a que la implementación mixta hardware-software resulta más barata que una implementación totalmente en hardware [23]. Ésta ha sido, por tanto, la aproximación del codiseño en los últimos años: sólo llevar a harware lo que resulta imprescindible acelerar.

En los últimos años, la implementación de la partición o particiones de hardware, se ha abordado con dos diferentes tecnologías: circuitos integrados de aplicación específica (ASIC, *Application-Specific Integrated Circuit*) y sobre circuitos programables, mayoritariamente en los FPGA (*Field-Programmable Gate Arrays*). La parte de software, formada por un CPU y la memoria, también se ha implementado de maneras diversas: por medio de los mismos CPUs, Procesadores de Señales Digitales (DSP,*Digital Signal Processing*) y Microcontroladores.

Con el surgimiento en los últimos dos años de nuevos y más complejos FPGAs, ha sido posible que el particionamiento de una aplicación, entre un programa ejecutado y módulos de hardware programados, sea implementado en un solo dispositivo, dando lugar a los sistemas en un solo chip (SuC, *System on a Chip*).

Los SuC basados en FPGA, facilitan de manera especial el rápido desarrollo de prototipos de cómputo, además de constituir una plataforma de pruebas idóneo para el desarrollo de aplicaciones que requieren de un cómputo eficiente al poder implementar de manera sencilla coprocesadores encargados de acelerar partes críticas de diferentes tipos de aplicaciones, como el cifrado de datos o aplicaciones multimedia, por citar algunos ejemplos.

3.2. Inicios

El término Codiseño Hardware-Software (CHS) surgió en los inicios de la última década del siglo pasado, para describir la convergencia de problemas de diseño en un Circuito Integrado (CI). En esos años, los CPU's habían sido exclusivamente utilizados en los sistemas basados en tarjetas. Dichos sistemas integran por medio de una tarjeta principal (MB, *Mother Board*) diversos componentes de hardware de propósito específico con uno o más CPU's. Era claro entonces que el diseño de sistemas basados en MB, se convertiría en una disciplina importante, incluso para los diseñadores de CI. Grandes y complejos CPU's de 16, 32 y 64 bits fueron desarrollados en esos años por los diseñadores de CI, con la firme certeza (basada en la Ley de Moore) de que eventualmente todo un sistema, incluyendo el mismo CPU, cabría en un solo chip [24]. Así mismo, comenzaron a vislumbrarse varios problemas, de entre los cuales destacaron dos: por un lado la necesidad de una metodología de diseño que contemplara la integración de los grandes y complejos microprocesadores previamente diseñados, junto con el desarrollo de hardware; y por otro lado las herramientas capaces de conjuntar el codiseño y enviar todos los componentes del sistema al dispositivo. La contemplación de estos problemas dio origen a las primeras propuestas de diseño.

Uno de los primeros esfuerzos de codiseño fue el sistema SOS desarrollado en la Universidad del Sur de California por Prakash and Parker [25]. Estos sistema podia sintetizar una topología arbitraria de multiprocesadores y planificar una asignación de procesos en el sistema multiprocesador. Debido a que las técnicas de sístesis aún estaban en su parte de maduración, el sistema era lento y no era apto para resolver problemas grandes. Sin embargo, fue un logro importante ya que sentó las bases para que el particionamiento hardware-software surgiera como un paso inicial para la creación de modelos y algoritmos.

La importancia del particionamiento quedó manifestado en el diseño presentados un año después con los sistema Vulcan de la Universidad de Stanford [26] y Cosyma de la Universidad Técnica de Braunschweig [27]. En estos primeros diseños, se asumía que el ASIC y el CPU debían de estar en chips separados para facilitar la síntesis del primero. El CPU y el ASIC se comunicaban por medio de memoria compartida o por medio de registros. Esta arquitectura permitía asignarle al ASIC tareas computacionalmente intensivas mientras el trabajo de asignación y coordinación era orquestado por el CPU.

Los requerimientos para el desarrollo de una aplicación para ambos sistemas, eran especificados inicialmente en un programa de alto nivel, regularmente en lenguaje C. Después, basándose en un análisis de desempeño y de costos de varias implementaciones del programa, se decidía sobre las partes del programa que debían ser llevadas a un ASIC y las partes que permanecerían en software para ser ejecutadas en un CPU de propósito general. Sin embargo, los diseñadores de Vulcan y Cosyma tomaron caminos distintos para la finalización del producto. Por un lado, Vulcan ponía toda la funcionalidad en hardware y movía algunas operaciones al CPU para minimizar costos. Cosyma por su parte, partía con todas las operaciones en el CPU y migraba algunas de ellas a un ASCI para alcanzar las metas de desempeño deseadas.

Los diseñadores de ambos sistemas debían de analizar el desempeño a lo largo de tres dimensiones: hardware, software y sistema en conjunto. De éstos, la medida de desempeño del hardware resultaba ser la más sencilla de evaluar, debido a que se contaba con una infraestructura más accesible. El principal objetivo en estos análisis era determinar la frecuencia máxima a la cual podría operar el hardware diseñado, para determinarlo lo más rápido posible los diseñadores evaluaban varios diseños durante la síntesis. La solución fue el uso de técnicas de síntesis de alto nivel para estimar la ruta más larga a través de la lógica.

El análisis del desempeño del software presenta más de un dilema. El problema a resolver es similar a la formulación presentada en la evaluación del hardware: tiempo de ejecución en el peor caso. Cosyma abordaba el problema al estimar el desempeño al ejecutar casos de prueba sobre un procesador objetivo. Vulcan por su parte llevaba a cabo un análisis en el flujo de control del programa.

El análisis de desempeño del sistema es igualmente complejo. En general, un sistema CPU-ASIC es tanto un sistema de multiprogramación como un sistema de multiprocesamiento, ya que puede incluir multiples procesos ejecutándose de forma concurrente en un mismo CPU. Asi mismo, puede incluir varios elementos de procesamiento ejecutando tareas de manera simultánea. Evidentemente el análisis se complica al aumentar el número de CPU's o de elementos de procesamiento.

Los dos sistemas mencionados, Vulcan y Cosyma, utilizan un modelo computacional sencillo para poder hacer el problema más manejable. Ambos asumen en la implementación un único hilo de ejecución, es decir, el CPU se mantiene en espera ocupada mientras el ASIC realiza la función asignada.

3.3. Maduración

En unos poco años, los investigadores abordaron varios de los problemas del CHS y sus resultados hicieron posible la aplicación del codiseño en un rango amplio de aplicaciones. Algunos trabajos iniciales trataron sobre la cosimulación, y muy pronto fue reconocido como un componente esencial en la metodología de codiseño. El gran reto era llevar a cabo la cosimulación como una combinación de diferentes niveles abstracción, para ejecutar suficientes vectores de entrada que validaran el diseño. Con la simulación formada por una combinación de niveles, los diseñadores pueden

controlar de manera más adecuada la simulación, al poder elegir más detalladamente el nivel de simulación de cada componente del sistema.

En el mismo contexto, Becker, Singh y Tell [28] desarrollaron en 1992 un cosimulador que enlazaba un simulador de hardware con la ejecución de programas en software. El ambiente Ptolemy [29] fue otro de lo primeros sistemas para la simulación de procesamiento de señales y sistemas heterogeneos hardware-software. Este trabajo inicial de cosimulación, estimuló trabajos posteriores para el entendimiento de la relación entre varios modelos computacionales y la unificación de sus entornos de trabajo y desarrollo.

El problema de medir el tiempo de la ejecución en el peor caso para el software, recibió gran atención en la medida en que el codiseño se consolidaba. Li, Malik y Wolf [30] desarrollaron un algoritmo de análisis de rutas, que fue más eficiente que el método de la enumeración de rutas de Park y Shaw. Tiempo después, Li y sus colegas, extendieron su algoritmo para modelar el efecto de las instrucciones en caché.

Otros trabajos centraron su investigación en el problema de la evaluación del desempeño del sistema, adoptando modelos desarrollados para sistemas de tiempo real como una solución parcial. Un sistema de asignación periódica desarrollado por Liu [31], recibió alguna atención como una forma de analizar el desempeño de un conjunto de procesos en un solo CPU. Yen y Wolf [32] desarrollaron un algoritmo para la evaluación de un sistema multiprocesador al ejecutar un conjunto de procesos (incluyendo la dependencia de datos) mapeado en una red de multiprocesadores, en donde cada procesador ejecutaba una planificador de asignación de tipo *Round-Robin*[1].

En la medida de que el CHS maduró, los investigadores iniciaron la exploración de varios modelos computacionales para los sistemas empotrados [33]. El lenguaje C tomó gran ventaja por ser el más ampliamente usado pero no era el ideal para la especificación de sistemas concurrentes. En la parte de control, M. Chiodo y colaboradores construyeron un modelo para el desarrollo del codiseño de máquinas de estados finitos [34], el cual describe la concurrencia y la comunicación de procesos.

Otras investigaciones tenían como fin el desarrollo de métodos para obtener arquitecturas más generales, como las presentadas en 1997 por Kalavade y Lee [35]. Al mismo tiempo Wolf desarrollo un método de síntesis que podía manejar la topologías de interconexión arbitraria y la combinación arbitraria de CPUs y ASICs[36].

El diseño de baja-potencia se convirtió en el tema dominante durante la década de 1990, el cual impulsó la investigación de las técnicas de cosíntesis de baja potencia. Fornaciari y estudiantes [37] desarrollaron un sistema de modelado para estimar el consumo de potencia de un sistema empotrado durante la cosíntesis. Una vez que los métodos arquitectónicos habían madurado, las investigaciones se dirigieron a la características de implementación, una de ellas era la generación de interfaces. Daveau [38] desarrolló modelos y algoritmos para la implementación de protocolos de interfaz.

[1]En este tipo de planificación cada proceso tiene asignado un *quantum* de tiempo para ejecutarse y en el caso de que no pueda terminar la ejecución en su *quantum*, el proceso pasa de nuevo a la cola de procesos para ser ejecutado por otro *quantum* luego de recorrer la cola de procesos.

Un reto en la generación de interfaces fue cómo sintetizar el software para que éste se ejecutara en el sistema empotrado ya que la estructura del software podría influir de manera significativa sobre el desempeño del sistema y el consumo de potencia.

3.4. Sistemas en un chip basados en FPGA

El particionamiento hardware-software es ahora una tarea práctica de diseño, gracias al gran desarrollo que ha tenido el cómputo reconfigurable en la presente década, con la utilización de los dispositivos programables FPGA (*Field-Programmable Gate Arrays*) [24]. Un FPGA es un dispositivo semiconductor que contiene componentes lógicos programables e interconexiones programables entre ellos. Los componentes lógicos programables pueden ser programados para duplicar la funcionalidad de puertas lógicas básicas tales como AND, OR, XOR, NOT o funciones combinacionales más complejas tales como decodificadores o simples funciones matemáticas. En muchos FPGA, estos componentes lógicos programables (o bloques lógicos) también incluyen elementos de memoria, los cuales pueden ser simples flip-flops o bloques de memoria más complejos. Una jerarquía de interconexiones programables permite a los bloques lógicos de un FPGA, ser interconectados según la necesidad del diseñador del sistema. Estos bloques lógicos e interconexiones pueden ser programados por el usuario/diseñador por medio de herramientas especializadas de software, así, el FPGA podría desempeñar cualquier función lógica necesaria. Los FPGA son generalmente más lentos que los ASICs, no pueden soportar diseños muy complejos, y consumen más energía. Sin embargo, ellos tienen muchas ventajas tales como la reducción del tiempo para el desarrollo de aplicaciones, la posibilidad de ser reprogramados cientos de veces a fin de mejorar el diseño y con ello reducir los costos de ingeniería, diseño y prueba de una aplicación.

En los últimos 2 años, las principales compañias fabricantes FPGA, han desarrollado nuevos dispositivos con características muy inovadoras, ya que han combinado dentro de un mismo FPGA lógica programable con uno más procesadores de propósito general, permitiendo a los desarrolladores la creación de sistemas completos dentro de un solo chip.

La plataforma de trabajo basada en FPGA, debido a su arquitectura interna y método de programación, resuelve muchos de los problemas que se venían presentado en la síntesis de hardware, principales retractores en el desarrollo del CHS. De la misma forma, los nuevos FPGA cuentan con herramientas especializadas para llevar a cabo la co-síntesis de hardware-software para programar el FPGA. Gracias a ello, ahora los investigadores centran sus esfuerzos en resolver otro tipo de problemas ántes de que éstas herramientas espcializadas lleven a cabo la cosíntesis del sistema. Un problema muy común a resolver es el uso óptimo de recursos al obtener el máximo desempeño.

Para obtener el uso óptimo de los recursos que brindan los nuevos FPGA, es necesario mapear de manera adecuada una aplicación en el dispositivo. Este mapeo deberá, además resolver los problema tradicionales en las arquitecturas compuestas por un CPU y ASIC, enfrentar un problema fundamental: la comunicación entre ellos. Es claro que el exceso de fuentes de retraso en la comunicación y en la sincronización, anula cualquier ganancia obtenida al agregar un ASIC a un sistema. Una aplicación bien diseñada bajo este tipo de arquitectura implica que, grandes cantidades de operaciones sean realizadas por un ASCI con el mínimo costo de comunicación y bajo la supervisión eficiente por parte de un CPU [24, 39].

Para lograr estos objetivos de diseño, es necesario la creación de interfaces de comunicación entre los módulos de hardware programados y el CPU. Desde el punto de vista del procesador la interfaz, conocida como controlador o *driver*, necesita convertir operaciones y datos de software en señales para el hardware. Desde el punto de vista del módulo de harware, es necesario agregar los circuitos necesarios para que éste pueda ser conectado a uno de los buses del sistema, con el fin de que el módulo pueda responder a las diferentes señales que conforman el protocolo de arbitraje del bus y así poder tener interacción con el sistema y de manera específica con el CPU. Los mecanismos de comunicación más comunes entre un CPU y un ASIC, incluyen el mapeo del dispositivo en el rango de memoria del CPU o el acceso directo a memoria (DMA), el metodo de sincronización más comunmente utilizado es el uso de interrupciones al CPU.

Existe aún cierto debate sobre qué lenguajes son los más indicados para la partición de algoritmos y su posterior implantación en estos sistemas. Por parte del software, el lenguaje de programación C parece ser el más comunmente utilizado ya que se considera como un lenguaje de nivel intermedio; mientras que por parte del lenguaje de descripción de hardware, VHDL y Verilog comparten aún el primer puesto en utilización. Es común que al realizar la partición, sólo unas partes de la aplicación sean descritas con un lenguaje para hardware, dejando el resto en el lenguaje para software. En consecuencia, cuando las operaciones son movidas a través de la partición, sólo una parte relativamente pequeña del total de las especificaciones deberán ser traducidas de un lenguaje a otro.

El desarrollo de sistemas en un solo chip, SuC, es otra de las vertientes de desarrollo de sistemas en donde el CHS y los nuevos FPGAs han tenido una aplicación importante. Debido a que los SuCs no tienen una arquitectura fija, una gran variedad de algoritmos pueden ser implantados en estos sistemas. Los SuC son diseños orientados a módulos con propiedad de diseño (IP, *Intellectual Property*), con ello, los diseñadores pueden utilizar módulos de hardware que han sido pre-programados y se encuentran a disposición del diseñador. Estos módulos van desde dispositivos de comunicación (UART, *Universal Asynchronous Receiver-Transmitter*), dispositivos Ethernet (*Fast Ethernet*,*Gigabit Etehernet*), controladores de memoria, hasta CPU completos. Con la incorporación de estos módulos, uno de los objetivo principales de la cosíntesis, es determinar qué también es integrado un módulo IP en el sistema desarrollado, en términos de tamaño, velocidad y accesibilidad para el diseñador.

Uno de los acercamientos al diseño de SuC que se ha popularizado en los últimos dos años, es el diseño basado en plataforma. Una plataforma es una arquitectura prediseñada que puede ser utilizada por los desarrolladores para la construcción de sistemas en las que puedan implantarse un amplio rango de aplicaciones. Los nuevos FPGA's son un ejemplo de plataforma de trabajo, ya que permiten crear cualquier tipo de arquitectura con la utilización de CPUs (programados o incrustados), lógica programable y la interconexión de hardware.

En algunos aspectos, el diseño basado en plataforma contradice el CHS, ya que los diseñadores deben de adecuarse a la plataforma para sus diseños. Sin embargo, la plataforma puede dar cabida a una gama muy amplia de aplicaciones y con la ayuda de la metodologías de codiseño, es posible explorar el espacio de posibilidades de diseño que brinda dicha plataforma para la creación de arquitecturas.

En la figura 3.1 se muestra un esquema de la plataforma basada en FPGA, el cual presenta sus componente de manera funcional.

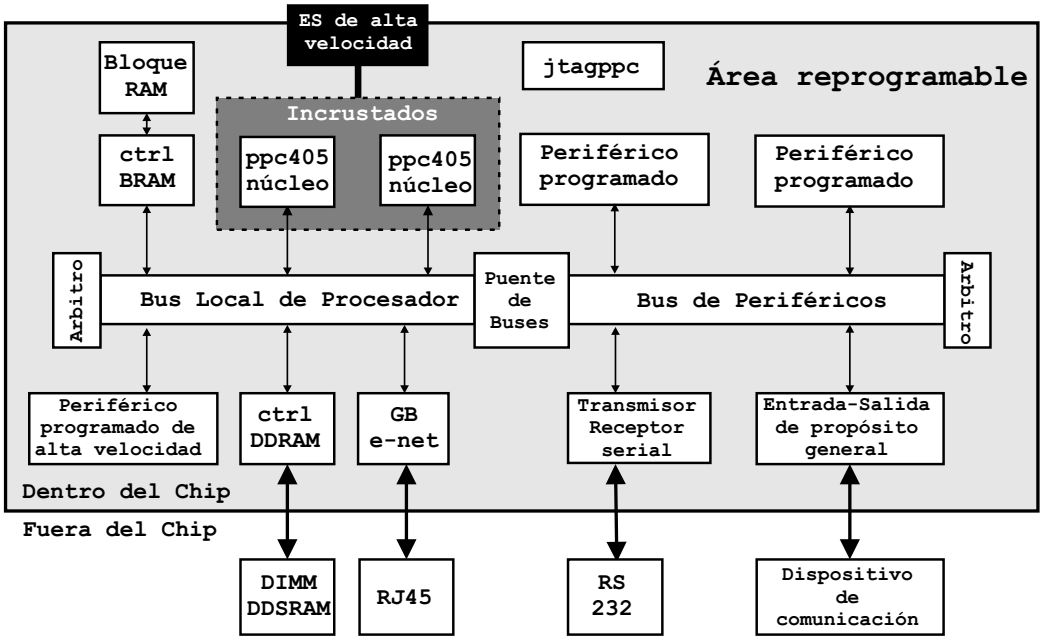

Figura 3.1: Sistema en un solo chip, basado en un FPGA.

Las plataformas basadas en FPGA, como la familia Virtex II Pro de la compañía Xilinx, ofrecen en su dispositivo más denso:

- De 1 a 2 procesadores incrustrados **PowerPC 405** (PPC405) a 400Mhz con 16kB para datos y 16kB para instrucciones en su memoria cache
- La posibilidad de programar varios procesadores **MicroBlaze**[2], de acuerdo a la cantidad de celdas lógicas
- 100,000 celdas lógicas (*Logic Cells*) para la programaciónde hardware
- 8 Mb de bloques RAM de doble vía
- 20 Entradas-Salidas de alta velocidad (*IO Rockets*), con una velocidad máxima de operación de 3.125 Gbps
- 444 multiplicadores de 18×18 bits
- 12 relojes para control de bloques
- 1164 puertos de entrada-salida de usuario (dependiendo del paquete)

Además de lo anterior, Xilinx ha colaborado con IBM para introducir una tecnología de interconexión orientado a buses llamado *CoreConnect*. Este sistema de buses tiene varias características para la interconexión del PPC405 y los dispositivos programados que ayudan al diseño de sistemas. El *CoreConnect* está compuesto por el bus local del procesador, PLB (*Processor Local Bus*), el bus de periféricos en el chip, OPB (*On-chip Peripheral Bus*) y el registro de control de dispositivos, DCR (*Device Control Register*), los buses PLB y OPB pueden operar hasta 100Mhz. Este sistema soporta varios sistemas de control para distintos dispositivos, por ejemplo, para entrada-salida incluye controladores para los UARTs, para memoria, incluye soporte para dispositivos SRAM/FLAH, SDRAM, DDR SDRAM. Incluye además, sistemas para entrada-salida más sofisticados, tales como, controladores para bus PCI, ethernet 10/100, y Gb ethernet. Cada uno de estos módulos puede ser sintetizado por separado.

El conjunto de recursos arriba descritos, abre un campo muy amplio de desarrollo para aplicaciones, además de proveer una gran flexibilidad de desarrollo, ya que se cuenta con herramientas de software que ayudan a la programación, compilación-síntesis, simulación y depuración tanto de hardware como de software, dejando sólo como resposabilidad del diseñador factores propios de todo diseño, como son la utilización de recursos y las frecuencias de operación. Así mismo, para poder obtener el mayor provecho en la utilización de estos recursos es conveniente seguir la metodología de CHS, la cual nos permite el diseño y desarrollo óptimo de hardware y software en forma conjunta.

[2]El MicroBlaze es un procesador de tipo *soft processor core*, el cual es un procesador programado en la lógica configurable de un FPGA. El MicroBlaze está basado en una arquitectura RISC, muy similar al procesador DLX descrita por Patterson y Hennessy.

3.5. Metodología de codiseño hardware software

La Metodología de Codiseño Hardware-Software (MCHS) puede ser definida como el diseño cooperativo de hardware y software, esto es, el desearrollo de sistemas heterogéneos, con el objetivo concreto de crear un sistema capaz de acelerar una aplicación con los mínimos costos de diseño, desarrollo y prueba posibles.

Generalmente, las técnicas de CHS intentan primero localizar las partes de la aplicación que ocupan el mayor tiempo de procesamiento, para después implementarlas en hardware. Sin embargo existen algunas limitaciones que deben de ser consideradas a la hora de seleccionar las partes que serán llevadas a hardware. Una limitante obvia es la cantidad de recursos disponibles para la implementación tanto de las partes de software como de las partes de hardware. La falta de recursos en alguna de las partes puede restringir el particionamiento de hardware y software del problema, siendo necesario la reasignación y segmentación de tareas. Por ejemplo, si se está considerando un circuito programable para la implantación del hardware, se debe de tener en cuenta tanto la densidad de compuertas programables como el número de puertos de entrada-salida disponibles para el usuario. Por el lado del software, la cantidad de memoria principal puede ser una factor decisivo para el flujo de datos del programa. Un factor limitante menos claro y que requiere de análisis más detallados, es la presencia de retroalimentación o *feedbacks* en un algoritmo. Las retroalimentaciones de datos pueden reducir la cantidad de segmentación o paralelismo, decrementando el rendimiento de la solución en hardware. Finalmente, otro factor no menos importante es la correcta definición de las interfaces hardware-software [40], ya que un exceso de comunicaciones entre las partes, puede impactar directamente con el desempeño global del sistema.

Por lo expuesto anteriormente, en el proceso de codiseño de un sistema se deben de analizar cada una de las tareas a realizar, evaluando las repercusiones de las opciones de implementación sobre los parámetros que definen la funcionalidad, el costo y complejidad del sistema global. Normalmente los principales parámetros a considerar son la velocidad de ejecución y el área que conllevaría su implementación hardware. La MCHS es una secuencia de pasos que ayudan al diseño e involucran los análisis anteriormente mencionados.

El flujo de diseño de la MCHS es presentado en la figura 3.2, y es descrito por los los siguientes pasos:

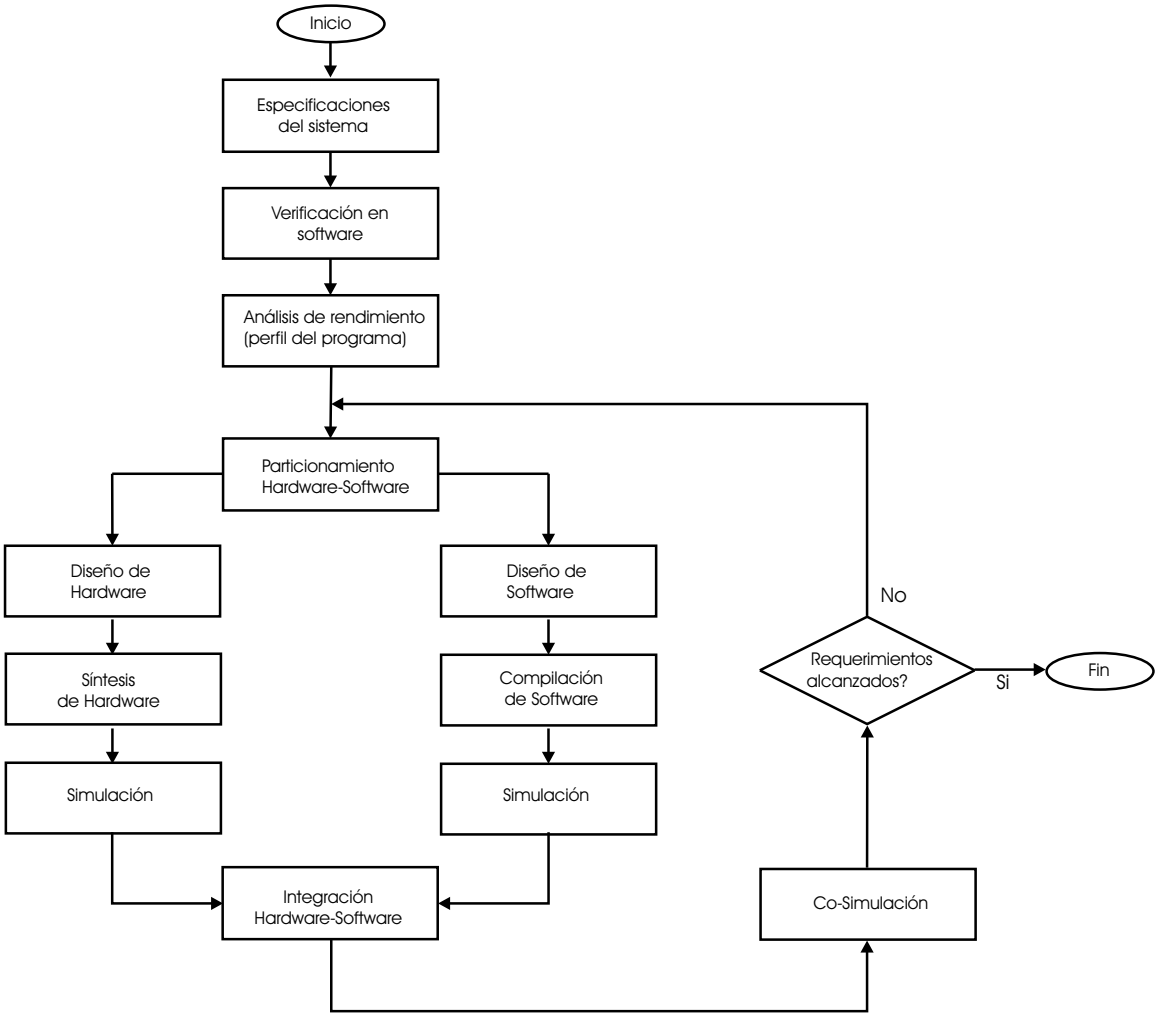

Figura 3.2: Diagrama de flujo para el codiseño hardware-software.

1. El proceso de codiseño inicia con las **especificaciones** del comportamiento del sistema. Se detallan características del sistema, como por ejemplo, el tipo de datos que recibirá para el procesamiento así como la forma de recibirlos.

2. A partir de las especificaciones, un sistema diseñado puramente en software es desarrollado con el propósito de realizar la **verificación** del algoritmo. Esto puede ser llevado a cabo al implantar el algoritmo en un lenguaje de alto nivel.

3. Una vez validado el comportamiento del programa, se lleva a cabo un **Análisis de desempeño** para localizar las partes más costosas en cómputo de la aplicación. Este analisis se puede llevar a cabo de varias formas, las dos más comunes son el análisis del algoritmo y el perfil del programa. La primera es analizar el algoritmo y tratar de localizar las partes que presumiblemente consumen más tiempo en la ejecución del programa. Evidentemente este tipo de análisis puede complicarse cuando la aplicación es muy grande y compleja, además de

que siempre requierirá de una amplia experiencia por parte del diseñador. Una forma más conveniente es obtener el perfil del programa por medio de herramientas de software. Estas herramientas muestran varios parámetros útiles para un análisis, por ejemplo el porcentaje de tiempo ocupado por cada una de las funciones que conforman la apliacación, el número de llamadas por función o procedimiento, el tiempo ocupado por cada llamada, etc.

4. Localizadas las partes más costosas en cómputo, la fase de **particionamiento** Hardware-Software, es la encargada de determinar qué partes serán llevadas a hardware y cuales se mantendran en software. Las partes con mayor consumo de tiempo serán las primeras candidatas a ser llevadas a hardware. Un correcto particionamiento es el que considera no sólo la cantidad de recursos que será ocupados por ambas partes, si no que además prevé las posibles interacciones entre las partes de hardware y software.

5. El hardware y el software son diseñados de forma separada y quizas de forma simultánea.

6. El hardware es sintetizado y el software es compilado.

7. En caso de contar con las herramientas que así lo permitan, será de gran ayuda la simulación de las partes de forma separada para una posterior cosimulación, al integrar ambas partes y realizar un primer análisis de rendimiento.

8. En la fase de integración, será necesario el desarrollo de *drivers* para la sincronización y comunicación de los módulos de hardware por parte del software; y así mismo será necesario agregar los componentes de hardware necesarios a los módulos de hardware para su integración al sistema de buses. Al conjunto de software y hardware agregado para la cooperación de un programa con un módulo de hardware es lo que llamamos interfaz.

9. Integrados las partes de software y hardware por medio de la interfaz, es de gran utilidad llevar a cabo una cosimulación. Su principal objetivo es la verificación de que las señales y datos fluyan adecuadamente a través la interfaz.

10. Si los requerimeintos de desempeño son alcanzados el codiseño es finalizado, en caso contrario, se debe realizar un nuevo particionamiento.

3.5.1. Ejemplo de codiseño hardware software

Con la intención de ejemplificar las posibilidades de diseño utilizando la MCHS, se presenta una propuesta de codiseño para el algoritmo de cifrado IDEA (*International Data Encryption Algorithm*) [41, 42]. Debido a que los algoritmos de criptografía son comunmente implementados en arquitecturas de hardware, ya sea en VLSI o hardware programable, sus principales características han sido reportadas, inclusive algunos de ellos son propuestos como *benchmark* para el cómputo reconfigurable [43]. Valiéndonos de esta información y por simplicidad, la explicación de la propuesta se centrará en los pasos del codiseño.

- **Especificaciones:**

  IDEA es un cifrador por bloques, opera con bloques de 64 bits usando una llave de 128 bits y consiste de ocho transformaciones, llamadas ronda, y una transformación de salida, llamada media ronda. En la figura 3.3 se muestra de forma esquematica una ronda del algoritmo IDEA.

  Gran parte de la seguridad de IDEA deriva del intercalado de operaciones bit a bit de distintos grupos: adición modular ($\boxplus$), multiplicación modular ($\odot$) y OR-Exclusivo ($\oplus$).

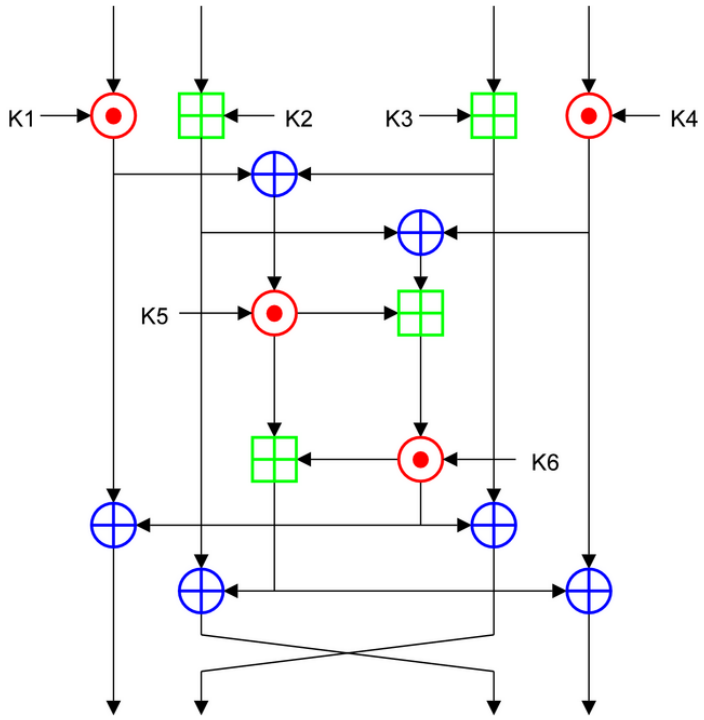

Figura 3.3: Ronda de cifrado de IDEA.

El algoritmo actúa cifrando grupos de 64 bits empleando una llave de 128 bits. Esto quiere decir que con un bloque de 64 bits y una llave de 128 bits obtenemos

un nuevo bloque de 64 bits. Los 128 bits de la llave se dividen en subclaves de 16 bits que se corresponden con las 8 primeras subllaves. Los dígitos de las llaves de 128 bits son rotados 25 veces a la izquierda para obtener la nueva llave, la cual se vuelve a dividir en las siguientes subllaves de 16 bits. Este último paso se repite hasta que se generan las 52 subllaves necesarias, empleando seis subllaves por iteración y cuatro subllaves en la última transformación.

Las subllave para el descifrado se obtienen a partir de la subllaves de cifrado calculando el inverso multiplicativo de la subllaves empleadas en las multiplicaciones y el inverso aditivo de las subllaves empleadas en las sumas. El algoritmo completo se puede consultar en [41, 42].

- **Verificación:**

  Una propuesta de solución en software del algoritmo IDEA puede ser consultada en [41]. El algoritmo es implantado en el lenguaje de alto nivel C. Al compilarlo y ejecutarlo en un sistema de cómputo convencional se puede evaluar su rendimiento. En [41, 42] se presentan algunos resultados. En nuestro estudio, partimos de la necesidad de acelerar la aplicación.

- **Análisis de desempeño:**

  El proceso de expansión de subclaves ocurre sólo cuando una clave es elegida y el resultado puede almacenarse para ser usado después. Por ello, la expansión de subclaves requiere solo una pequeña fracción del total del tiempo de computación. Como se ha mencionado, el algoritmo envuelve tres operaciones: or-exclusiva, suma módulo 2^{16} y multiplicación $2^{16} + 1$, y consiste en 8 rondas computacionalmente idénticas seguidas por una transformación de salida. El componente más crítico de la implementación del algoritmo de IDEA es el multiplicador de 16 bits. La falta de un multiplicador en hardware dentro de la arquitectura del procesador convencional, penaliza la solución en software.

- **Particionamiento:**

  A pesar de que el flujo de datos de IDEA puede ser paralelizado al procesar múltiples bloques de datos a través de un mismo *pipeline*, la presencia de retroalimentación de llaves necesariamente supone un límite superior al paralelismo disponible. Además de lo anterior, como ya se mencionó, el multiplicador de 16 bits es el componente que más procesamiento ocupa. Un multiplicador puede ocupar un espacio prohibitivo dependiendo del tamaño de los operandos. Claramente, el multiplicador $2^{16} + 1$ puede ser implementado tanto en software como en hardware [44].

  Centrándose en los resultados del análisis, en dónde se consideró que una mejora en el desempeño obedece a la necesidad de un multiplicador más eficiente, a continuación se estudian tres propuestas de particionamiento con diferentes codependencias.

1. **Unidad Multiplicadora Combinacional (UMC):** Como primera aproximación se puede implementar un multiplicador como unidad hardware, siendo operado por la parte de software, es decir, el software enviará los operandos y esperará por el resultado. La propuesta incluye además, desarrollar un multiplicador matricial de alto rendimiento, cuyo tiempo de operación sea a lo sumo de un ciclo de reloj. En la figura 3.4 se muestra de forma esquemática la partición. La parte de software es conformada por unidades de memoria y un microprocesador, mientras que la parte de hardware es un arreglo bidimensional de unidades multiplicadoras.

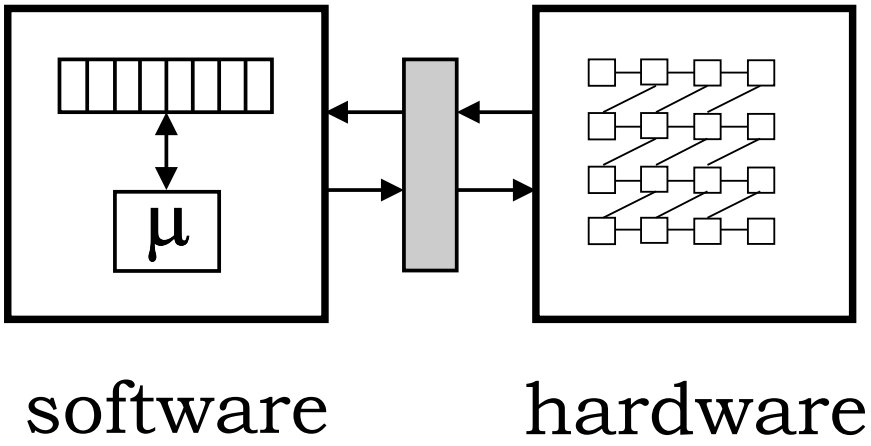

Figura 3.4: Unidad multiplicadora matricial.

Como resultado a esta propuesta se espera una unidad multiplicadora capaz de operar a frecuencias altas, pero con un alto grado de ocupación de hardware, prohibitivo para dispositivos programables pequeños. Un nuevo problema surge con esta solución, y es la de agregar al tiempo computacional una nueva componente consistente en el tiempo de comunicación entre el procesador y la UMC. Si esta componente crece demasiado, cualquier mejora de tiempo obtenida al agregar una unidad en hardware, será opacada.

2. **Unidad Ronda Combinacional (URC):** En esta solución, la implementación en hardware configurable incluye los elementos necesarios para una ronda IDEA, para ello son necesarios 4 multiplicadores, 4 sumadores y 6 xor. En la figura 3.5 se muestra la nueva partición. Esta nueva propuesta puede mejorar el rendimiento de la aplicación, además de, aparentemente solucionar el problema de comunicación entre las partes, ya que sólo requerirá para calcular una ronda de los 64 bits del bloque más los 96 bits de las subclaves.

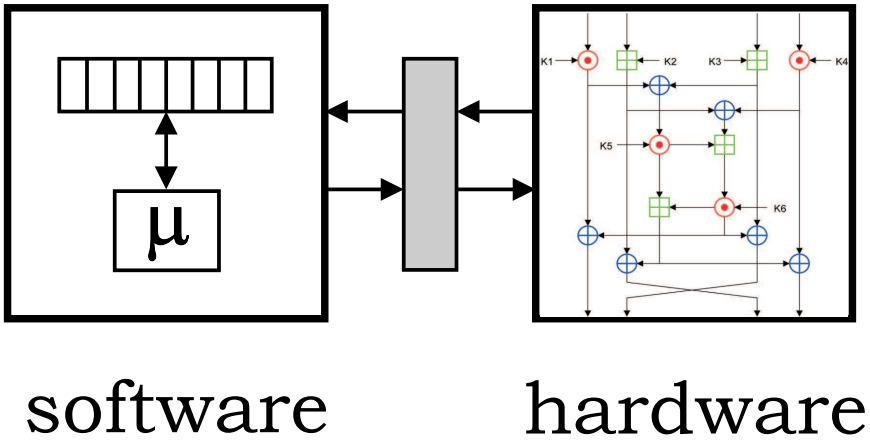

Figura 3.5: Ronda de cifrado de IDEA.

Esta solución evidentemente requerirá de mayor espacio para su programación en hardware y no necesariamente soluciona la comunicación entre las partes. Si bien es cierto que la URC necesita de un bloque y las subclaves para operar, se debe de considerar que el procesador deberá de enviarlo por un medio de comunicación. El medio de comunicación tradicional entre un procesador y unidades perifericas de cómputo, es un sistema de buses, cuya longitud corresponde al tamaño de la palabra en la que opera un procesador, surgiendo la necesidad de un protocolo de comunicación entre la aplicación y la URC.

3. **Unidad Ronda Secuencial (URS):** Las dos soluciones anteriores muestran en común dos problemas de diseño: la gran ocupación en un dispositivo programable y el problema siempre latente de las comunicaciones entre las partes de hardware y software. Todo codiseño de hardware y software debe de enfrentar y resolver de la mejor manera estos dos problemas, además de tener siempre presente el compromiso entre recursos y velocidad de operación.

   Una solución al problema del la ocupación de hardware, es la segmentación de tareas, a costa de un tiempo mayor de operación por parte del hardware. En el ejemplo presentado, una segmentación podría ser el diseño de un multiplicador secuencial. En [43] se presentan algunas soluciones donde el

multiplicador fue sustituido por un simple multiplicador serial, es decir de sumas parciales y rotaciones. El tamaño de este multiplicador es menor en términos de compuertas configurables, sin embargo, necesita de al menos 17 ciclos de reloj para realizar una operación completa.

- **Diseño, Síntesis-Compilación y Simulación:**

  El desarrollo, la síntesis-compilación y la simulación de ambas partes se llevan a cabo con herramientas especializadas. La compañia Xilinx así como otras más, además de distribuir los dispositivos programables, ofrece herramientas de software, las cuales comúnmente están incluidas en un ambiente de desarrollo. Las herramientas además de la síntesis-compilación ofrecen un sofisticado sistema de depuración tanto para software como para hardware.

- **Integración:**

  La integración de las partes se puede llevar al agregar el hardware diseñado a un sistema compuesto de un microprocesador, memoria principal y sistema de buses. Para ello es necesario la creación de una interfaz entre el hardware y el software. La interfaz está compuesta de componentes adicionales a la unidad de hardware y de rutinas especiales en la parte de software, ambas con la finalidad de comunicarse a través de un protocolo de arbitraje bien definido por el sistema de buses. Así, a la unidad de hardware se le agregó un registro de 32 bits para poder comunicarse con el microprocesador, además de la lógica de control. Con ello, el procesador puede leer o escribir en el registro si esté es mapeado al espacio de direcciones del procesador.

3.6. Conclusiones

Las plataformas basadas en FPGA nos permite experimentar con distintas propuestas de diseño de hardware debido principalmente a su flexibilidad de desarrollo, comprobados por el cómputo reconfigurable. La misma plataforma de desarrollo nos da la oportunidad de crear sistema de cómputo completos cuando el FPGA cuenta con procesadores de propósito general, ya que los recursos reconfigurables pueden ser directamente comunicados con ellos.

Por otro lado, con la metodología de CHS podemos explorar las posibles propuestas de particionamiento hardware-software de una aplicación, considerando además sus repercusiones en el sistema global. El ejemplo presentado nos permite concluir que la solución de codiseño con mejor rendimiento puede ser aquella que resuelva la parte más costosa en cómputo de la aplicación en hardware, mantega la mayor parte del control y flujo de datos en software y que emplee la mejor interfaz de comunicación al minimizar el intercambio de datos entre las partes.

Capítulo 4

Codiseño Hardware-Software de la CFI

4.1. Introducción

En el capítulo 2 se mostró que el mayor inconveniente de la Codificación Fractal de Imágenes (CFI) es el gran número de operaciones y comparaciones que deben realizarse para determinar los parámetros de las transformaciones que aproximan a cada uno de los bloques en que se divide la imagen.

Para tratar de resolver este inconveniente, se han propuesto en la literatura especializada algoritmos de procesamiento eficiente, los cuales en general intentan reducir el número de operaciones. En otras propuestas de solución, el método de CFI ha sido llevado a un hardware dedicado, cuyas arquitecturas aprovechan los distintos tipos de ejecución que el hardware permite: procesamientos secuencial, paralelo y en *pipeline*. Al revisar los resultados reportados en estas dos vertientes de soluciones, únicamente software ó únicamente hardware, se puede concluir lo siguiente: la implementación en software de la CFI tiene un costo de desarrollo reducido pero su tiempo de ejecución es considerable y muy difícil de reducir. Para el caso de la implementación de la CFI puramente en hardware, se tienen aceleraciones notables para la codificación, pero debido a que se trata de un sistema de gran complejidad, el tiempo de diseño, implementación y prueba es considerablemente alto y costoso.

En este capítulo es presentado el Codiseño Hardware-Software (CHS) de la CFI para el desarrollo de un sistema en un chip, SuC, en una plataforma basada en FPGA. Este sistema permitirá la aceleración de la CFI por medio del trabajo conjunto de un CPU incrustado y de hardware programado, todo en el mismo chip.

4.2. Algoritmo de la codificación fractal de imágenes

El algoritmo para la CFI propuesto por Michael Barnsley y Lyman Hurd en [2], fué presentado en la figura 2.5 como un diagrama de flujo, ahora con la intensión de tener una mejor visualización del anidamiento de los ciclos que componen el proceso de codificación, se presenta en la figura 4.1 el algoritmo de la CFI en pseudocódigo. Para mayor claridad, el pseudocódigo incluye un formato de llamada a funciones del tipo de lenguaje C.

Algoritmo 4.1. Codificación Fractal de Imágenes
Requiere: Una arreglo de pixeles: imagen

```
01. FOR xD = yD = 0, todos los bloques dominio DO
02.     distanciaMinima = infinito
03.     copiaBloque( imagen, bloqueDominio, xD, yD )
04.     FOR xR = yR = 0, todos los bloques rango DO
05.         copiaBloque( imagen, bloqueRango, xR, yR )
06.         escalamiento( bloqueDominio )
07.         prom = promedio( bloqueRango, bloqueDominio )
08.         cambioLuminosidad( bloqueRango, prom )
09.         FOR simetria = 0, todas las simetrías DO
10.             reflejar( bloque rango, bloqueRangoReflejado, simetria )
11.             dist = distancia( bloqueRangoReflejado, bloqueDominio )
12.             IF dist < distanciaMinima THEN
13.                 distanciaMinima = dist
14.                 registrarMapa( xR, yR, simetria, prom )
15.             END IF
16.         END FOR
17.     END FOR
18.     almacenarMapa( xR, yR, simetria, prom )
19. END FOR
```

Figura 4.1: Parte principal del algoritmo para la CFI.

Para el análisis de pseudocódigo de la figura 4.1, se debe de considerar que los pixeles correspondientes a la imagen a codificar han sido leidos de un archivo y llevados a una zona de memoria, la cual es accedida por la referencia `imagen`. De igual forma, se han creado zonas de memoria para el procesamiento de pixeles, cuyas referencias tienen nombres que indican su propósito: `bloqueRango`, `bloqueDominio`, etc.

El pseudocódigo de la figura 4.1 representa con dos ciclos (línea 1 y línea 3) la comparación de todos los bloques dominio con todos los bloques rango, las coordenadas `(xD,yD)` y `(xR,yR)` establecen la esquina superior izquierda de cada bloque. La función `copiaBloque()`, copia de la imagen los pixeles correspondientes de un bloque dado, ya sea que se trate de un bloque dominio o de un bloque rango. La función `escalamiento()` reduce un bloque dominio al mismo tamaño de un bloque rango. La función `promedio()` calcula el promedio del valor de los pixeles, tanto del bloque dominio como del bloque rango y regresa como resultado la diferencia aritmética de los dos promedios. Dicho valor es almacenado en la variable `prom`. Con el valor promedio de los pixeles de los bloques, se hace una cambio en la luminosidad del bloque rango al sumarle el valor `prom` a cada uno de sus pixeles. Lo anterior lo lleva a cabo la función `cambioLuminosidad()`.

Como se puede observar, las funciones `reflejar()` y `distancia()` se encuentran ubicadas en el ciclo más anidado (linea 9). La función `reflejar()` realiza en cada llamada, una de las ocho transformaciones posibles mostradas en la tabla 2.1 sobre un bloque rango. La función recibe como parámetros dos referencias a memoria, la primera ubica los pixeles que conforman el bloque rango a transformar (`bloqueRango`) y la segunda es la zona de memoria en donde se colocará el bloque rango transformado (`bloqueRangoReflejado`). También es pasado como un argumento la identificación de la trasformación que se deberá de realizar (`simetria`). En general, la función `reflejar()` realiza una reubicación de pixeles dentro de un bloque. En la figura 4.2 se muestra el pseudocódigo de la función `reflejar()`.

Como se puede observar en el pseudocódigo de la figura 4.2, antes de reubicar un pixel del `bloqueRango` al `bloqueRangoReflejado`, se debe calcular un desplazamiento, tanto a lo ancho como a lo alto del bloque. Los desplazamientos dependerán de la transformación que se desee llevar a cabo. La identificación de la transformación a aplicar sobre el bloque rango, es pasada como argumento a la variable local `simetria`, cuyo valor puede estar en el rango de 0 a 7. Para identificar las 8 transformaciones se prueban los 3 bits menos significativos del valor contenido en `simetria` mediante una operación AND lógica. Así por ejemplo, la trasformación 3 (011) se compone de la transformaciones 1 (001) y 2 (010).

Algoritmo 4.2. Función reflejar
Requiere: Dos bloques: bloqueRango, bloqueRangoReflejado; simetria

```
01. FOR j = 1, alto DO
02.     FOR i = 1, ancho DO
03.         IF simetria AND "001" THEN
04.             x = ancho - i
05.         ELSE
06.             x = i
07.         EN IF
08.         IF simetria AND "010" THEN
09.             y = alto -j
10.         ELSE
11.             y = j
12.         EN IF
13.         IF simetria AND "100" THEN
14.             intercambio(i,j)
15.         EN IF
16.         bloqueRangoReflejado[x,y] = bloqueRango[i,j]
17.     END FOR
18. END FOR
```

Figura 4.2: Algoritmo de la función reflejar.

Las operaciones que realiza la función `distancia()` son más sencillas pero también son llevadas pixel a pixel. Ésta función recibe de igual forma dos referencias a memoria, la primera ubica en memoria al bloque rango transformado (`bloqueRangoReflejado`) y la segunda al bloque dominio que se está tratando (`bloqueDominio`). La función calcula la distorsión entre el bloque rango trasformado y el bloque dominio al acumular las diferencias absolutas evaluadas pixel a pixel de acuerdo a su posición espacial en cada bloque Dicho valor es almacenado en la variable `dist`.

Con el valor de la distorsión entre el bloque dominio y el bloque transformado, se evalúa (línea 12) si dicho valor es menor que el valor contenido en `distanciaMinima` (`distanciaMinima` es inicializado (línea 2) con un valor de infinito para cada bloque dominio). Si el valor de `dist` resulta ser menor que `distanciaMinima`, se asigna a `distanciaMinima` el valor de `dist`, garantizando así la menor distorsión entre bloques. Además de reasignar el nuevo valor de `distanciaMinima`, se registra el mapa de afinidad entre los bloques que obtuvieron la menor distorsión. El mapa de afinidad consta de la identificación espacial del bloque rango (`xR,yR`), el cambio de luminancia (`prom`) y la transformación actual (`simetria`). Como se explicó en el capitulo 2, estos valores son conocidos como parámetros fractales.

Tanto las funciones `reflejar()` y `distancia()` así como la condicional que evalúa la menor distorsión, se ejecutan de manera consecutiva dentro de un ciclo (línea 9). El ciclo le indica a la función `reflejar()`, por medio del incremento secuencial de la variable `simetria`, cuál de las transformaciones debe ser aplicado al bloque rango. Aplicada la transformación al bloque rango, la función `distancia()` calcula la distorsión entre bloques y finalmente se evalúa si el valor de la distorsión es la menor que se ha obtendio.

Finalmente la función `almacenarMapa()`, escribe un archivo la información de la codificación, es decir escribe los mapas de afinidad. El archivo podría iniciar con un encabezado para informar acerca del formato de codificación y continuar los parámetros fractales. La información anterior puede ser almacenada en un formato que puedan ser leídos con un editor de texto o se puede almacenar en forma binaria para almacenarlo de forma más compacta.

El pseudocódigo de la figura 4.1 representa la solución en software de la CFI. Con la implantación del mismo pseudocódigo en lenguaje C, se obtuvieron los resultados de los experimentos mostrados en la figura 2.6 (página 19) y de la tabla 2.2 (página 20) del capítulo 2.

4.3. Particionamiento hardware software

En el capítulo 3 se mostró la Metodología de Codiseño Hardware-Software (MCHS), la cual tiene como principal objetivo distribuir una aplicación entre dos o más particiones de hardware y de software, con el fin de mejorar el rendimiento de la aplicación. Se explicó que la intensión de mantener algunas partes de la aplicación en software se debe a que éstas no son críticas desde el punto de vista de la velocidad de ejecución, y que la inclusión de unidades de hardware dedicado tiene como fin acelerar las partes de la aplicación que muestren un tiempo de procesamiento muy alto. La MCHS pretende obtener un compromiso adecuado entre las ventajas e inconvenientes tanto de una solución completamente en software, como de una solución completamente en hardware.

La MCHS inicia con un análisis de una solución en software de la aplicación. Con este análisis se pretende obtener información del tiempo de procesamiento de cada una de las partes que conforman la aplicación, además, se puede ofrecer una visión más clara de cómo se lleva a cabo el flujo de datos en la aplicación. Una herramienta muy útil para realizar el análisis es obtener el perfil de rendimiento de la aplicación. El perfil de rendimiento ofrece resultados cuantitaivos de tiempo y recurrencia de cada parte que conforma la aplicación.

La fase central de la MCHS es la búsqueda de un correcto particionamiento de la aplicación en tareas que serán, ya sea mantenidas en software o llevadas a hardware. Las propuestas de particionamiento deberán de considerar, además del correcto funcionamiento de la aplicación, factores de sincronización y comunicación eficientes, sin que esto implique un aumento en la complejidad de las partes. Como se puede suponer, un análisis previo de la aplicación es fundamental para un distribución de

la partes.

Ubicadas la partes de la aplicación que ocupan el mayor tiempo de procesamiento, se propone por un lado, una solución en hardware de éstas, y por otro lado, se describen las modificaciones de las partes de la aplicación que se mantendrán en software y que deberán de comunicarse con el hardware propuesto. En una primera aproximación, ambas propuestas se describen de manera funcional, con la finalidad de definir la integración de las mismas por medio de una interfaz hardware-software. En base a una primera descripción funcional de ambas partes, se procede a su desarrollo, lo cual puede llevarse a cabo de forma concurrente. Finalmente se procede a la integración de las partes y la verificación del correcto funcionamiento.

4.3.1. Análisis de desempeño

El perfil de rendimiento de un programa involucra analizar dicho programa para detectar cómo se distribuye el tiempo de ejecución en el mismo y así obtener el tiempo que tardan en ejecutarse cada una de las funciones que lo conforman, también es posible conocer, el número de veces que es llamada cada función y así evaluar la recurrencia de cada parte. Una vez que se obtiene el perfil del programa se podrá identificar aquellas funciones o áreas de código dónde se consume más tiempo.

El algoritmo propuesto por Michael Barnsley y Lyman Hurd en [2] y presentado en la figura 4.1, fue implantado en el lenguaje de alto nivel C y ejecutado en un sistema de cómputo convencional, obteniendose los resultados mostrados en la tabla 2.2. De manera funcional el código se dividió en tres grupos: lectura y escritura de archivos, control de bloques y cálculos sobre pixeles. Los nombres de la funciones corresponden a los mostrados en el algoritmo de la figura 4.1

El perfil de rendimiento del programa anteriormente descrito, se llevó a cabo con el comando de GNU `gprof`, disponible para cualquier plataforma UNIX. `gprof` calcula la cantidad de tiempo empleado en cada rutina o función del programa, estos tiempos se propagan a lo largo de los vértices del grafo de llamadas para acumular el tiempo y el número de llamadas totales.

El perfil de rendimiento mostrados en la tabla 4.1 corresponde a la codificación de la imagen Lena de 512×512 pixeles en escala de grises (0 a 255), y con tamaño de bloque rango de 8×8 pixeles.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
65.01	17377.13	17377.13	2031648768	0.00	0.00	reflejar
20.34	22814.36	5437.23	2031648768	0.00	0.00	distancia
7.36	24782.06	1967.69	253960192	0.00	0.00	copiaBloque
3.70	25771.51	989.45	253956096	0.00	0.00	cambioLum
2.75	26507.15	735.65	253960192	0.00	0.00	promedio
0.76	26709.15	202.00				main
0.05	26723.44	14.29	1	14.29	14.29	leerEncabezado
0.02	26728.10	4.66	1	4.66	4.66	reducirImagen

Tabla 4.1: Perfil de desempeño del programa que realiza la codificación fractal de imágenes.

La tabla 4.1 muestra, de izquierda a derecha, en la primera columna el porcentaje de tiempo invertido en la función, en la segunda el tiempo acumulado en ésta función y las listadas anteriormente, en la tercera los segundos empleados en la función, en la cuarta el número de llamadas, en la quinta el tiempo invertido por llamada y en la última el nombre de la función.

Analizando los resultados mostrados en la tabla 4.1, se observa que las funciones `reflejar()` y `distancia()` ocupan el 65.01 % y el 20.34 % respectivamente del tiempo total de ejecución. Ambas en conjunto suman el 85.35 % del tiempo para llevar a cabo la codificación de la imagen. Por otro lado, ambas funciones son llamadas en igual número de veces, pero la función `reflejar()` ocupa más tiempo en su ejecución. El hecho de que ambas funciones se encuentran de manera contigua dentro de un anidamiento de tres ciclos (figura 4.1, líneas 10 y 11), comprueba que son llamadas el mismo número de veces.

Como ya se ha comentado, la función `reflejar()` realiza en cada llamada, una de las ocho transformaciones posibles mostradas en la tabla 2.1. El algoritmo para llevar a cabo cada una de las ocho transformaciones se mostró en la figura 4.2. Las transformaciones se realizan al tomar uno de los 64 pixeles de un bloque rango (para este perfil de 8×8 pixeles) y asignarle un nuevo lugar dentro del bloque rango transformado (del mismo tamaño del bloque rango) de acuerdo a cada transformación. Por ejemplo, la transformación 1 de la tabla 2.1 implica una reflexión en x, entonces el pixel cuyas coordenadas dentro del bloque rango es (0,0) será llevado a las coordenadas (0,7) en el bloque rango transformado.

La función `distancia()`, calcula la distorsión entre el bloque rango trasformado y el bloque dominio, al acumular la diferencia absoluta de cada par de pixeles. La operación se lleva a cabo de la siguiente forma: se toma de manera consecutiva un pixel de cada bloque y se evalúa la diferencia, el resultado de la diferencia se multiplica por sí mismo para obtener un valor positivo. El resultado de la diferencia absoluta de cada par de pixeles es acumulado, obteniéndose al final la suma total de las diferencias, la cual representa la similitud entre los bloques.

A pesar de que las operaciones que involucran la transformación y comparación son sencillas (básicamente operaciones aritméticas de suma y de reubicación de bytes en memoria), éstas se repiten ocho veces para cada uno de los bloques rango en los que está particionada la imagen y todo lo anterior deberá de repetirse para cada uno de los bloques dominio.

Con base en el análisis anterior, podemos concluir lo siguiente:

1. Dos funciones ocupan el 85.35 % del tiempo total de codificación: `reflejar()` y `distancia()`.

2. La función `reflejar()` realiza en cada llamada una de las ocho transformaciones posibles mostradas en la tabla 2.1. La función `distancia()` evalúa la distorsión entre un bloque rango transformado y un bloque dominio.

3. Las operaciones que involucran la transformación y comparación son sencillas, básicamente operaciones aritméticas de suma y de reubicación de bytes en memoria.

4. Ambas funciones actuán en conjunto para determinar cuál de las ocho trasformaciones aplicadas sobre el v-ésimo bloque rango es más parecida al u-ésimo bloque dominio.

Con la ubicación de las partes más costosas en cómputo de la aplicación por medio del análisis anterior, es posible realizar un particionamiento de la aplicación de acuerdo con la MCHS. Como ya se ha mencionado, la MCHS propone llevar a hardware únicamente las partes más costosas en cómputo y mantener el resto de la aplicación en software. El particionamiento de la forma que indica la MCHS debe de considerar otros factores. Por un lado, se debe de realizar un análisis de la funcionalidad que será llevada a hardware ya que impactará directamente en la cantidad de recursos a utilizar y en la velocidad de operación de las unidades en hardware a diseñar. Por otro lado, se debe de considerar el flujo de datos en ambas partes, ya que de encontrarse con una dependencia de datos fuertes entre operaciones, dificultará la segmentación o posible paralelización de las tareas. Además, el particionamiento debe de considerar las formas en que ambas partes se sincronizarán para la comunicación eficiente de datos.

En las siguientes secciones se presenta el particionamiento realizado, así como los requerimientos para su comunicación. Se inicia con la descripción funcional de una unidad de hardware que implementa las operaciones que llevan a cabo en conjunto las funciones `reflejar()` y `distancia()`. La unidad es nombrada Unidad de Transformación y Comparación (UTC). Posteriormente se describen las funcionalidades de la aplicación que permanecieron en software y las modificaciones necesarias para integrar la UTC. Finalmente, se da la descripción de cómo ambas partes se comunicarán a través de una interfaz hardware-software.

4.3.2. Imagen de hardware

Las conclusiones obtenidas en el análisis de la sección anterior, señalan a las funciones `reflejar()` y `distancia()` como las partes del programa que deberían ser llevadas a hardware. Además de ello, se pudo concluir que las funciones `reflejar()` y `distancia()` trabajan en conjunto para determinar cuál de las ocho trasformaciones hechas sobre el v-ésimo bloque rango es más parecida al u-ésimo bloque dominio.

Un análisis sencillo de las funciones `reflejar()` y `distancia()`, muestra que las operaciones que éstas llevan a cabo no son de gran complejidad: reubicación de pixeles con base en desplazamientos y operaciones aritméticas. Tanto el desplazamiento de pixeles, como el diseño de unidades aritméticas, pueden ser implementados por circuitos lógicos, ya se de forma combinacional o de forma secuencial.

Con la información anterior, es posible proponer de manera funcional, una unidad de hardware que realice tanto el proceso de transformación como el proceso de comparación. Su funcionamiento podría ser el siguiente: recibir un bloque rango y un bloque dominio, realizar las transformaciones y comparaciones respectivas y finalmente mostrar como resultado, tanto la identificación de la trasformación aplicada al bloque rango que obtuvo la menor distorsión con respecto al bloque dominio, así como el valor de la misma distorsión mínima. La figura 4.3 muestra un esquema funcional de la unidad de transformación y comparación, UTC.

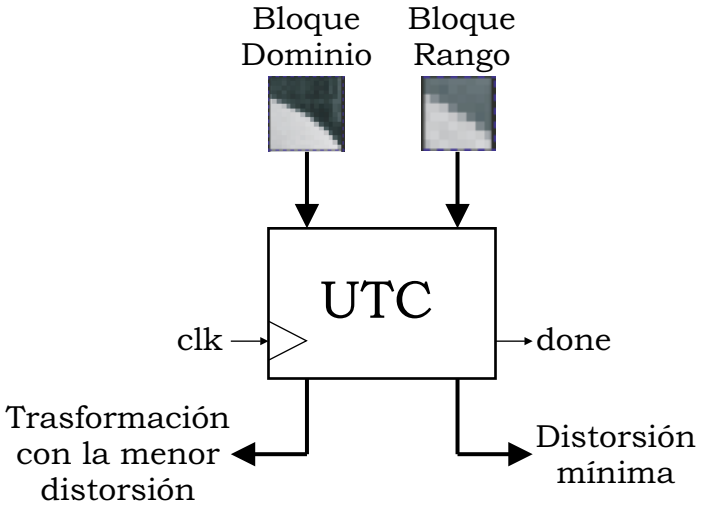

Figura 4.3: Descripción funcional de la unidad de transformación y comparación.

Debido a que el diseño de la UTC podría incluir tanto circuitos secuenciales como circuitos combinacionales, se agrega a la entidad de la UTC un puerto de entrada para una señal de reloj, `clock`. La entidad también presenta un puerto de salida, `done`, el cual puede servir para el envío de una señal de sincronización, muy útil si se pretende que la UTC sea integrado a un sistema de cómputo con un microprocesador de propósito general.

Gracias a las prestaciones de procesamiento que el hardware ofrece, la funcionalidad de transformación y comparación entre bloques descrita anteriormente, puede ser llevada a cabo de forma secuencial, paralela o en un *pipeline*. El desarrollo del circuito estará ahora sujeto a restricciones propias del diseño de hardware, es decir, al compromiso entre la utilización de recursos y la velocidad de operación.

En la sección 4.4, se presentan los detalles de diseño y organización de los circuitos para implementación de la UTC.

4.3.3. Imagen de software

Como se describió en el capitulo 3 (página 35), una de las características más importantes de la MCHS es la premisa de llevar a hardware *únicamente* las partes de la aplicación que ocupen el mayor tiempo de procesamiento, manteniendo las partes restates de la aplicación en software. Evidentemente, serán necesarias modificaciones en algunas de las partes de software original para la correcta integración de las unidades de hardware.

La solución en software original de la CFI, es representado por el pseudocódigo de la figura 4.1. Las modificaciones realizadas a esta solución para la integración de la unidad de hardware descrita en la sección 4.3.2 se muestra en la figura 4.4.

Algoritmo 4.3. Codificación Fractal de Imágenes con UTC
Requiere: Una arreglo de pixeles: imagen

```
01. FOR xD = yD = 0, todos los bloques dominio DO
02.     distanciaMinima = infinito
03.     copiaBloque( imagen, bloqueDominio, xD, yD )
04.     FOR xR = yR = 0, todos los bloques rango DO
05.         copiaBloque( imagen, bloqueRango, xR, yR )
06.         escalamiento( bloqueDominio )
07.         prom = promedio( bloqueRango, bloqueDominio )
08.         cambioLuminosidad( bloqueRango, prom )
09.         dist = reflejarHW( bloqueDominio, bloqueRango, *simetria )
10.         IF dist < distanciaMinima THEN
11.             distanciaMinima = dist
12.             registrarMapa( xR, yR, simetria, prom )
13.         END IF
14.     END FOR
15.     almacenarMapa
16. END FOR
```

Figura 4.4: Parte principal del algoritmo para la CFI modificado para la integración de la UTC.

Como se puede observar, el pseudocódigo mostrado la figura 4.4 es muy similar al pseudocódigo de la figura 4.1. La principal diferencia radica en la sustitución de las funciones `reflejar()` y `distancia()` así como el ciclo que las envolvía, por una única función llamada `reflejarHW()` (línea 9, figura 4.4). Otra diferencia apreciable es que la condicional (línea 12, figura 4.1) contenida originalmente en el ciclo que fue eliminado, se mantiene en el segundo pseudocódigo inmediatamente después de la función `reflejarHW()`. Como ya se ha comentado, por medio de esta condicional se selecciona de todos los bloques rango el que menor distorsión obtuvo y se registra sus parámetros fractales.

Desde el punto de vista del software, la función `reflejarHW()` aparece como una llamada a una función del tipo del lenguaje C. La función recibe tres parámetros y regresa un valor. Recibe dos referencias a memoria, una para un bloque dominio y la segunda para bloque rango. Una tercera referencia (en lenguaje C, un apuntador) es utilizada para que la función `reflejarHW()` regrese la identificación de la mejor transformación. La función regresa de forma normal, el valor de la menor distorsión entre bloques.

Se puede suponer que la función `reflejarHW()` agrupa las operaciones de las funciones `reflejar()` y `distancia()`, cuyo objetivo es indicar cuál de las ocho transformaciones hechas sobre un bloque rango obtuvo la menor distorsión con respecto a un bloque dominio.

Como se verá más adelante, la función `reflejarHW()` sirve como una interfaz para enviar y recibir datos a la unidad de hardware. Los detalles de la definición de la función `reflejarHW()` se muestran en la sección 4.5.

4.4. Unidad de transformación y comparación

Para el desarrollo de la UTC propuesta de manera funcional en la figura 4.3, se utiliza la metodología de diseño *top-down*. La metodología propone la formulación inicial de un modelo sin especificar detalles, siendo el modelo una descripción muy general del sistema. Después, el modelo es dividido en partes, las cuales son redefinidas cada vez con mayor detalle hasta que la especificación completa es lo suficientemente detallada para validar el modelo. El modelo *top-down* se diseña con frecuencia con la ayuda de "cajas negras" las cuales hacen más fácil completar los requerimientos, aunque inicialmente estas cajas negras no expliquen en detalle sus componentes.

Como se especificó en la sección 4.3.2, la funcionalidad de la UTC implica el procesamiento de pixeles ubicados en dos conjuntos disjuntos llamados bloque rango y bloque dominio. El procesamiento se divide en dos partes, la primera parte lleva a cabo la reubicación de pixeles (transformaciones) y la segunda consiste en la aplicación de operaciones aritméticas sobre pixeles (comparación). Las operaciones anteriores deberán de efectuarse ocho veces para cada bloque rango que se reciba. Las secciones siguientes muestran la segmentación de la UTC en una ruta de datos y una unidad de control.

4.4.1. Ruta de datos

Para la creación de una trayectoria de datos, que además de ser eficiente no consuma muchos recursos, se propone que la unidad de procesamiento sea dividida en registros y unidades aritméticas, donde el flujo de datos a través de ellos sería supervisado por una unidad de control. La ruta de datos estará conformada de tres partes que operarán de forma secuencial: recepción de bloques, transformación y comparación.

La figura 4.5 muestra la primera segmentación de la UTC, la cual consiste en tres bloques: transformación, comparación y control. Los bloques llevan el nombre de las operaciones que deben de realizar. En la presente sección se detalla la organización de la ruta de datos y en una sección posterior se describe el bloque de control.

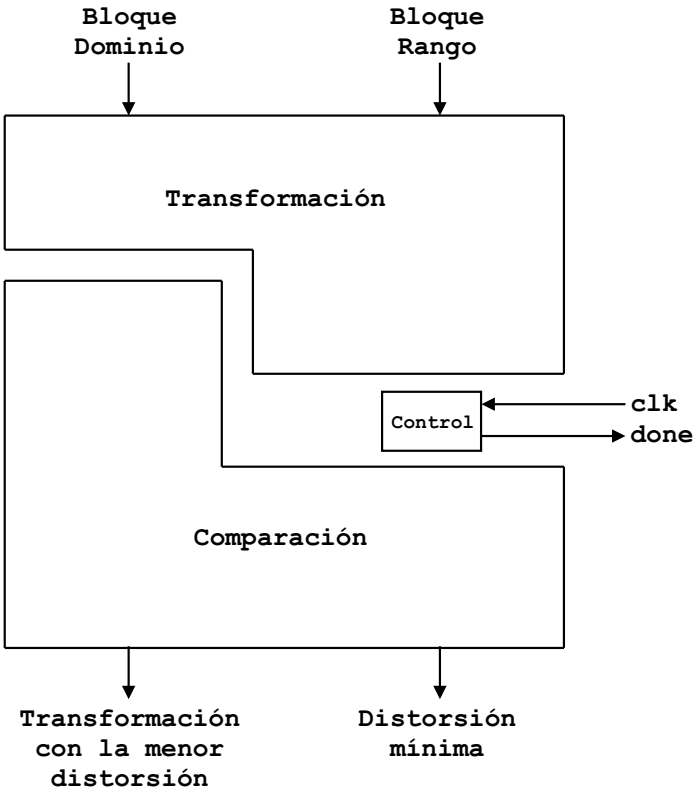

Figura 4.5: Segmentación de la UTC.

En la figura 4.6 se muestra la organización de la parte de la UTC que lleva a cabo las transformaciones sobre un bloque rango. El bloque de transformación lo conforman tres registros de 512 bits cada uno, con la posibilidad de carga en paralelo y corrimientos a la izquierda o a la derecha. El primer registro llamado `BloqueRango`, almacena los pixeles del bloque rango; el segundo llamado `BloqueDominio`, almacena los pixeles del bloque dominio y el tercer registro `BloqueRangoReflejado`, es utilizado para realizar las trasformaciones sobre el bloque rango.

Los registros `BloqueRango` y `BloqueRangoReflejado` están interconectados por medio de tres bloques ruteadores, identificados como `001`, `010` y `100`. Cuando uno de los ruteadores es habilitado, permite la transferencia de un conjunto de pixeles del registro `BloqueRango` a una posición específica dentro del registro `BloqueRangoReflejado`. También es posible la carga completa ya sea del registro `BloqueRango` al registro `BloqueRangoReflejado` o viceversa.

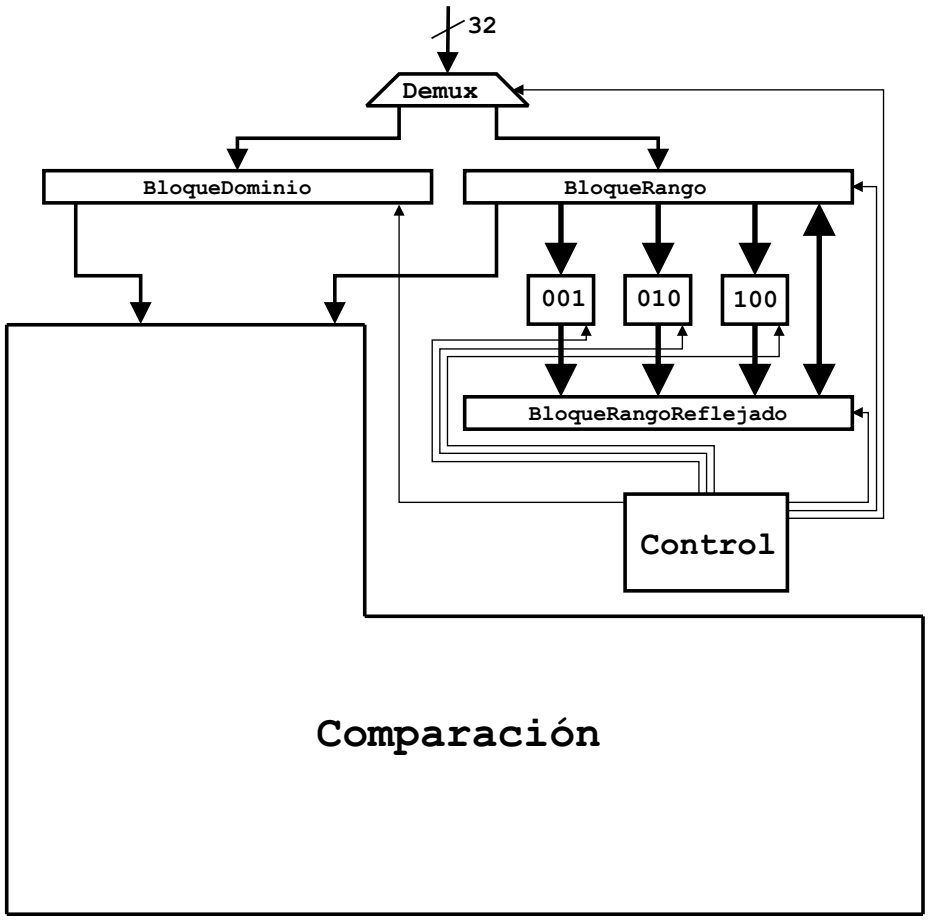

Figura 4.6: Organización para llevar a cabo las transformaciones.

Recepción de bloques

El procesamiento de la UTC inicia con la recepción de los pixeles que conforman el bloques dominio y el bloque rango. Los pixeles deberán ser recibidos y almacenados en los registros para su procesamiento. Debido a que es inviable generar un módulo de hardware con una interfaz capaz de recibir el total de bits de los dos bloques de forma simultánea, es necesario crear un mecanismo para la recepción de los bloques por parte de la UTC. Ya que se ha considerado que la UTC será integrada a un sistema de cómputo interconectado por medio de un sistema de buses, cuya longitud es de 32 bits (el microprocesador tiene una longitud de palabra de 32 bits), se

determinó que los bloques serán recibidos de forma secuencial y además, cada bloque será dividido en palabras o marcos de 32 bits.

Un bloque rango o un bloque dominio lo conforman $8 \times 8 = 64 = 2^6$ pixeles de $2^3 = 8$ bits/pixel, en total $2^{6+3} = 512$ bits. Tanto el bloque rango como el bloque dominio son divididos en 16 marcos de 32 bits, para su transmisión entre el microprocesador y la UTC. La división del bloque se muestra en la figura 4.7. Los 16 marcos que conforman un bloque, son recibidos por la UTC a través de una interfaz de 32 bits y llevados al registro correspondiente. Este proceso es llevado de forma secuencial. Con la inserción de un demultiplexor 1 a 2 (`Demux` en la figura 4.6) entre la interfaz y los registros, la misma entrada de datos es utilizada para la recepción tanto de los marcos que conforman un bloque rango como de los marcos que conforman un bloque dominio.

Cada marco que ingresa a la UTC es llevado a los 32 bits menos significativo del registro que le corresponde. Después de la copia del marco, el registro sufre un corrimiento de 32 bits a la izquierda para así poder recibir el siguiente marco. Serán necesarios 15 corrimientos para almacenar los 16 marcos de un bloque en un registro.

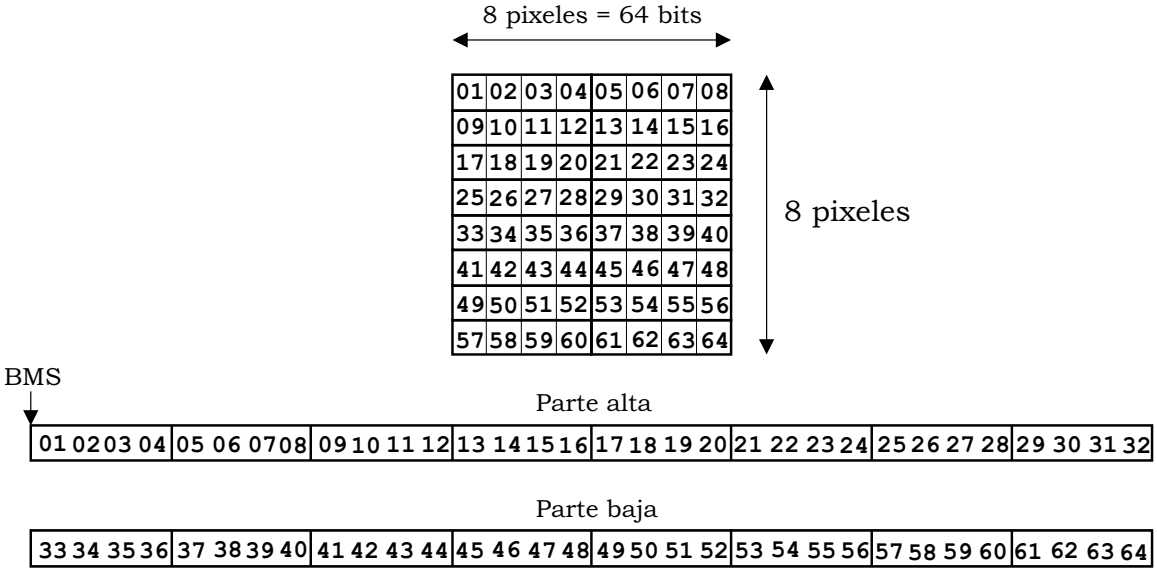

Figura 4.7: División de un bloque en marcos y su distribución en un registro.

Para ejemplificar cómo se distribuyen los pixeles en los registros, considere la figura 4.7. El cuadrado representa un bloque de 8×8 pixeles, cada celda está numerada y puede contener un pixel de 8 bits. Los rectángulos inferiores representan la parte alta y la parte baja de un registro de 512 bits. Cuatro celdas del cuadrado componen un marco de 4 pixeles o 32 bits, obteniendo 16 marcos por bloque. Los marcos del bloque podrían ser transferidos, por ejemplo, de izquierda a derecha y de arriba hacia abajo. Si consideramos que fueron enviados al registro de la forma anterior, el primer marco recibido permanecerá en los 32 bits más significativos del registro después de aplicar los 15 corrimientos a la izquierda.

Transformación

Recibidos tanto el bloque dominio como el bloque rango y almacenados en los registros `BloqueRango` y `BloqueDominio`, inicia el proceso de transformación. Para visualizar cómo se llevan a cabo las transformaciones, considere la transformación 1 de tabla 2.1, la cual implica realizar una reflexión sobre el eje x. La figura 4.8 muestra tanto la imagen original de Lena así como el resultado de la aplicación de la transformación 1. Ambas imágenes han sido divididas en $8 \times 8 = 64$ celdas.

Como se puede observar, la transformación implica el movimiento espacial del contenido de cada una de las celdas. Si hacemos coincidir la numeración de las celdas del bloque mostrado en la figura 4.7, con la división de las imágenes de 4.8, las celdas 01 y 08 intercambiarán contenido, asi como las celdas 02 y 07, 03 y 06 y las celdas 04 y 05 del primer renglon. El mismo proceso de intercambio se llevará a cabo para los 7 renglones restantes, en otras palabras, el proceso se repite cada 64 bits. El movimiento de pixeles descrito anteriormente, está plasmado en el psedocódigo de la figura 4.2.

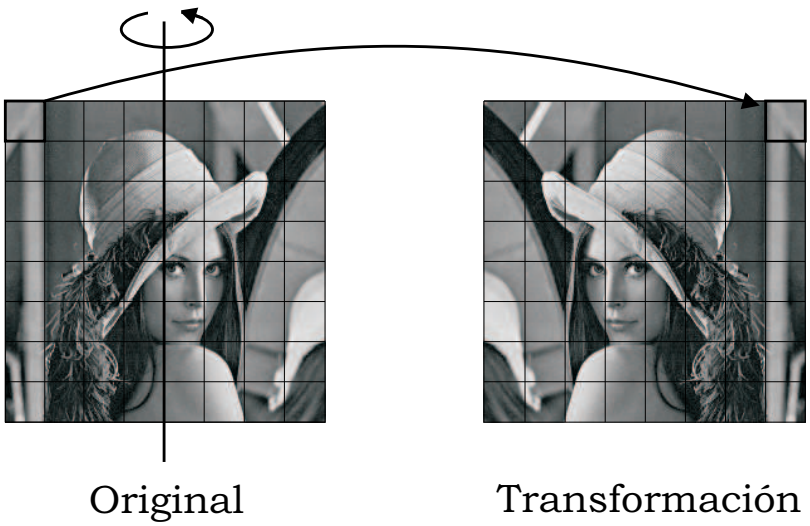

Figura 4.8: Reflexión de la imagen Lena sobre el eje x.

Para que la UTC obtenga el resultado antes descrito, será preciso copiar los pixeles contenidos en el registro `BloqueRango` de una forma particular en el registro `BloqueRangoReflejado` cada 64 bits. Para ello se conectan ambos registros con un bloque ruteador de pixeles, marcado como `001` en la figura 4.6, y se procesa el bloque en forma de lineas, es decir, cada dos marcos o 64 bits a la vez.

El proceso de transformación se realiza en dos partes: copia de pixeles y corrimiento de registros. La copia se lleva a cabo al habilitar el bloque ruteador `001`, el cual tomará los ocho pixeles más significativos del registro `BloqueRango` y los llevará a los ocho pixeles menos significativos de registro `BloqueRangoReflejado`. En la figura 4.9 se puede observar cómo se interconectan ambos registro por medio del bloque ruteador. Después de copiar los pixeles al bloque `BloqueRangoReflejado`, se aplicará un

corrimiento de 64 bits a la izquierda en ambos registros, para que con ello se copien los siguientes 64 bits del bloque del `BloqueRango` al `BloqueRangoReflejado`.

Las transformaciones 2 (reflexión en y) y 4 (reflexión en diagonal) se llevan a cabo de forma similar a la transformación 1. Ambas transformaciones interconectan los pixeles de forma particular por medio de los bloques ruteadores `010` y `100` respectivamente, para después aplicar corrimientos a la izquierda o a la derecha en ambos registros. En todos los casos, el resultado final es copiado del registro `BloqueRangoReflejado` al registro `BloqueRango`. Lo anterior tiene dos propósitos, la primera es tener una sola conexión con las unidades aritméticas que realizarán la comparación entre bloques y la segunda es tener listo los pixeles para al aplicación de transformaciones compuestas, como se explica a continuación.

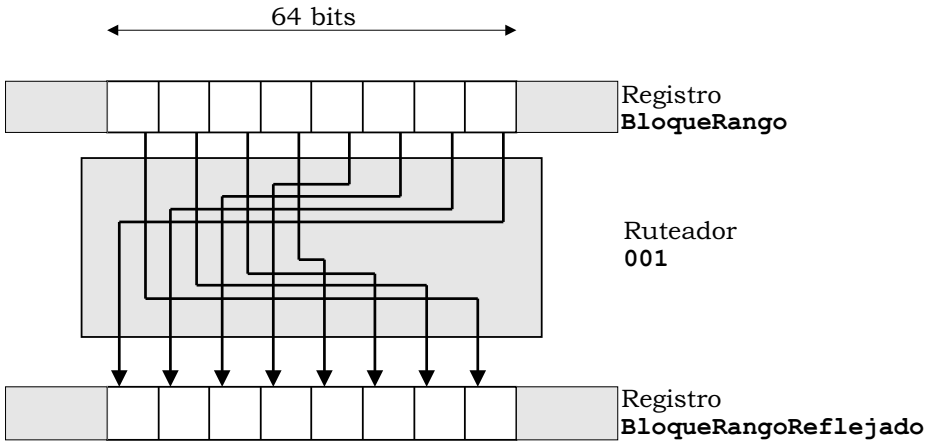

Figura 4.9: Reflexión en x utilizando registros y un bloque ruteador de la UTC.

Para el caso de las transformaciones compuestas: 3 (011), 5 (101) y 7 (111), se deben de realizar dos o hasta tres transformaciones de manera secuencial. Por ejemplo, la transformación 3 la componen la reflexión en x (001) y la reflexión en y (010). En este caso, primero se aplica la transformación en x de manera normal, cuyo resultado es mantenido en el registro `BloqueRangoReflejado`. Para completar la transformación 3, se deberá de aplicar la reflexión en y sobre la reflexión en x, para ello, primero se copia de manera paralela el contenido del registro `BloqueRangoReflejado` al registro `BloqueRango` y después se aplica la reflexión en y de manera normal.

Comparación

Como se ha comentado, el proceso de comparación implica primero, calcular la diferencia entre un pixel del bloque dominio con un pixel del bloque rango cuya posición espacial coincide. Calculada la diferencia, se obtiene su valor absoluto y se acumula junto con los otros 64 resultados. La acumulación de las diferencias absolutas es la distorsión entre los dos bloques.

En la figura 4.10 se muestra las unidades aritméticas que han sido agregadas a la organización de la UTC para llevar a cabo la comparación entre bloques. La `UA`,

toma los pixeles más significativos de los registros `BloqueRango` y `BloqueDominio` y calcula tanto su diferencia como el valor absoluto del resultado. Para ambos cálculos la unidad utiliza aritmética de números con representación en complemento a dos. La unidad marcada como `Acumulador`, es de 32 bits y suma el valor entregado por `UA` con la suma parcial de los resultados previos. Ambos sumadores trabajan de forma serial.

A una señal de inicio, el registro de 32 bits `DistMinima`, es inicializado con un valor de "infinito" (el cuadrado del valor máximo de un pixel, multiplicado por 64). El registro `DistanciaMinima` está conectado junto a la unidad `Acumulador` al comparador `Comp`, el cual al ser habilitado por la unidad de control, realiza la comparación de los valores de ambas unidades y genera una señal que permite el paso a través de los buffers de tres estados. Los registros de 4 bits `Simetria` y `SimetriaFinal` son utilizados para indicar cuál de las transformaciones es la que menor distorsión ha tenido. El control envía al registro `Simetria` la transformación que se está llevando a cabo, y a una señal del comparador `Comp`, el valor es propagado al registro `SimetriaFinal`.

La UTC lleva a cabo el proceso de comparación entre los pixeles contenidos en el registro `BloqueDominio` y los pixeles contenidos en el bloque `BloqueRango` (el resultado de cada transformación es copiada al registro `BloqueRango`) de la siguiente forma: Se toman los pixeles más significativos de los registros `BloqueDominio` y `BloqueRango`, los cuales son enviados a la `UA`. La `UA` calcula el valor absoluto de su diferencia y el resultado es convertido a un número de 32 bits. El resultado es sumado en el `Acumulador` con los valores previamente calculados. Este proceso se repite para todos los pixeles de ambos bloques, cada par de pixeles es obtenido al aplicar corrimientos de 8 bits a la izquierda en los dos registros.

El valor total acumulado en la unidad `Acumulador`, es la distorsión entre dos bloques. Obtenido el valor de la distorsión, se compara con el valor contenido en el registro `DistMinima`, si el valor acumulado es menor que el valor almacenado en `DistMinima`, el registro `DistMinima` es actualizado con el valor contenido en `Acumulador`. Con la actualización del registro `DistMinima`, también lleva a cabo la actualización del registro `SimetriaFinal` con el valor actual del contador `Simetria`.

El proceso anterior se repite hasta completar las ocho transformaciones. Al completar las ocho transformaciones y comparaciones, la UTC coloca tanto el valor de `SimetriaFinal` como el de `DistanciaMinima` en el bus de comunicaciones y envía una señal de terminación.

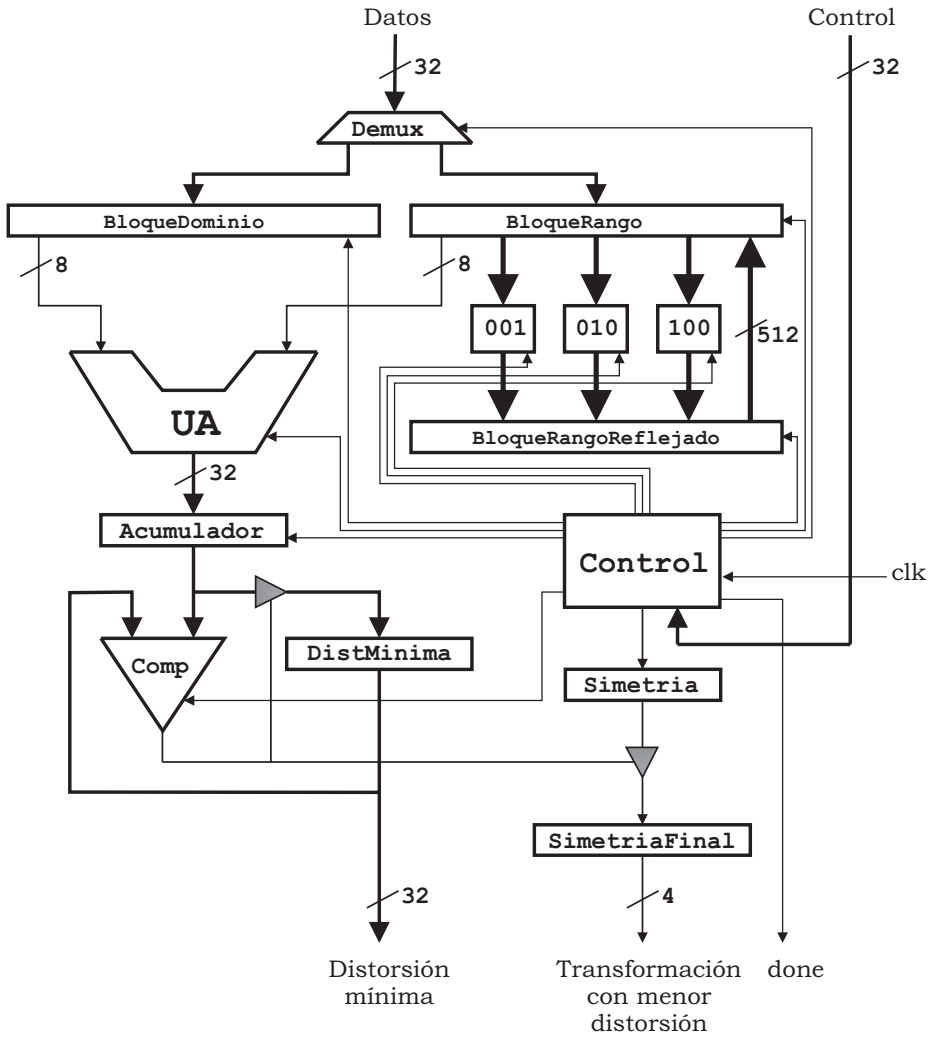

Figura 4.10: Organización completa de la UTC.

4.4.2. Unidad de control

La unidad de control de la UTC está compuesta de un registro de 32 bits, `selecRegistro`, y 2 submódulos de control `cargaBloques` y `TC` (Transformación-Comparación). En el registro `selecRegistro` se recibe una de las palabras de control que son enviada por el microprocesador a través del sistema de buses. Con la decodificación de la palabra de control, la UTC realizará ya sea el proceso de carga de bloques o el proceso de transformación-decodificación, al habilitar alguno de los dos submódulos. Los módulos `cargaBloques` y `TC` son dos máquinas de estados finitos que generan las señales necesarias para la habilitación de las unidades funcionales de la UTC. En la figura 4.11 se muestra la relación del registro con los submódulos, así como las señales que genera cada submódulo. El nombre de cada señal indica a que unidad es conectada.

Las palabras de control o instrucciones que son enviadas por el microprocesador para determinar las operaciones de la UTC son mostradas en la tabla 4.2 junto con la

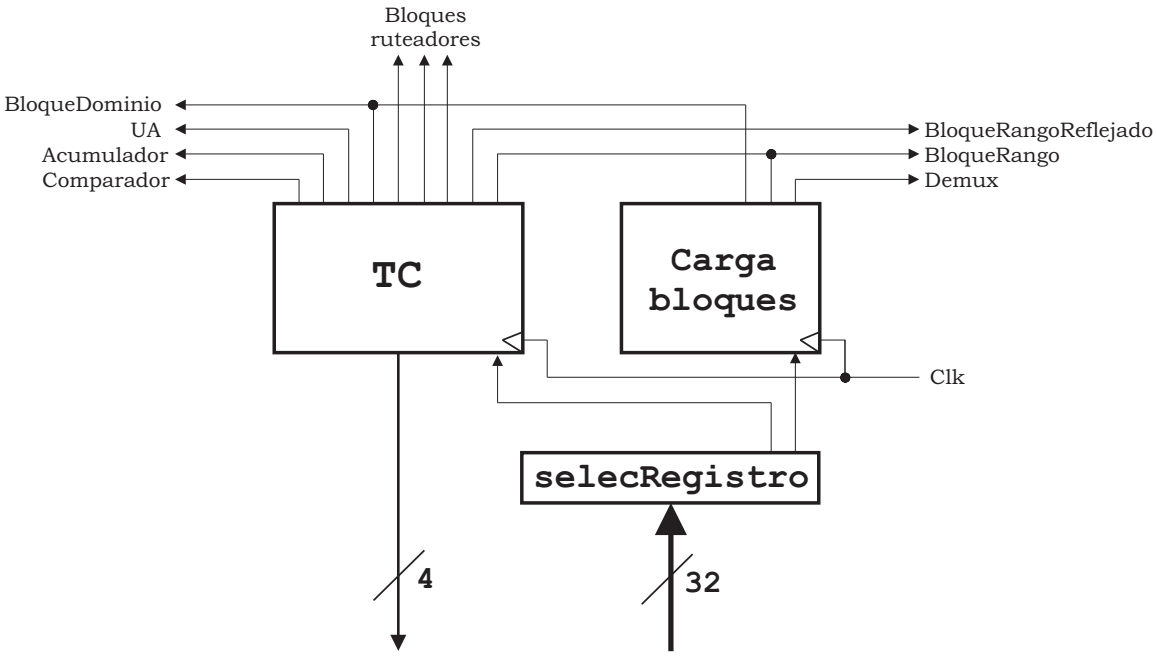

Figura 4.11: Organización de la unidad de control de la UTC.

descripción de su operación. El conjunto de palabras de control o instrucciones, pueden ser representado en un solo byte del registro de control, sin embargo se reciben los 4 bytes para facilitar el protocolo de transmisión de datos entre el microprocesador y la UTC. Lo anterior se describirá en la sección 4.5, la cual datalla la interfaz hardware-software.

Palabra de Control	**Operación**
0x0000	Reinicio
0x0001	Carga bloque dominio
0x0002	Carga bloque rango
0x0004	Transformación-Comparación

Tabla 4.2: Descripción de las palabras de control de la UTC.

Como se ha comentado, por medio de las palabras de control el microprocesador le indicará a la UTC la operación que deberá realizar, siendo ésta una forma de sincronización entre las unidades. La interacción de las unidades se describe a continuación.

Inicialmente el microprocesador enviará a la UTC la palabra de control **0x0000**, con ello la UTC estará lista para la recepción de los bloques. Por medio de una segunda palabra de control, el microprocesador avisará a la UTC del envío de los 16 marcos de uno de los bloques, los cuales son inmediatamente transmitidos de forma secuencial. De la misma forma se envían los 16 marcos del un segundo bloque. Debe notarse que con el uso de una palabra de control para la notificación del envío de un bloque en

particular (**0x0001** para el bloque dominio y **0x0002** para el bloque rango), no hay restricción en el orden del envío de los bloques. En ambos casos, primero se notifica cuál de los dos bloques se va a enviar antes de transmitir sus 16 marcos.

La UTC inicia el proceso de transformación y comparación (considerando que los bloques ya han sido enviados a la UTC) cuando recibe la palabra de control **0x0004**. El microprocesador esperará por la señal de terminación de operaciones por parte de la UTC.

Las palabras de control habilitan alguno de los dos submódulos para realizar ya sea la operación de carga de bloques o las operaciones de transformación y comparación. Cada submódulo está conformado por una parte secuencial y una parte combinacional. Ambas unidades generan las señales de habilitación de las unidades aritméticas y de ruteo de pixeles así como el control de corrimientos de los registros, de acuerdo a la operación que sea llevada cabo. A continuación se describen cada uno de los dos submódulos.

El submódulo `cargaBloques` lleva a cabo la carga de los bloques en los registros correspondientes. Tanto para la carga del bloque dominio como la del bloque rango, el submódulo realiza dos operaciones: lleva los 32 bits del demultiplexor a los 32 bits menos significativos del registro correspondiente y aplica un corrimiento de 32 a la izquierda al mismo registro. Para recibir los 16 marcos de cada bloque, el submódulo deberá de realizar 16 copias y 15 corrimientos. La operaciones anteriores se representan con el diagrama de estado mostrado en la figura 4.12.

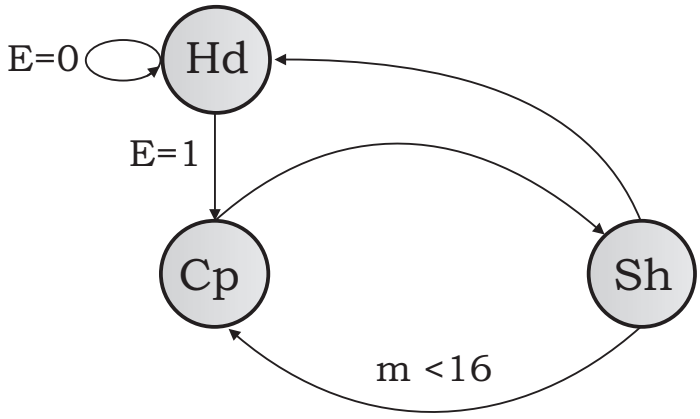

Figura 4.12: Diagrama de estados del submódulo de control para la carga de bloques.

El diagrama lo componen los estados `Hd`, `Cp` y `Sh`. La máquina es inicializada en el estado `Hd` y permanece ahí mientras la señal de habilitación `E` sea cero. Cuando la señal de habilitación es 1, se pasa al estado `Cp`, en el cual se generan las señales para la copia del marco al registro correspondiente y se pasa incondicionalmente al estado `Sh`. El estado `Sh` genera las señales para realizar el corrimiento a la izquierda de alguno de los dos registros. Del estado `Sh` se puede pasar al estado `Cp` para realizar una nueva copia si la variable `m` es menor a 16, si no lo es, se pasa al estado de inacción `Hd`. Como se comentó anteriormente, la transición de estados es válida para la copia

de cualquiera de los dos bloques, ya que la palabra de control además de habilitar el funcionamiento de este submódulo, también seleccionará alguna de las salidas del demultiplexor así como la habilitación del registro adecuado.

El diagrama de estado para el submódulo `TC` es más elaborado debido a que este módulo es el encargado de realizar el total del procesamiento de los bloques. El digarama de estado para el submódulo `TC` se muestra en la figura 4.13.

Para describir el diagrama de estados del submódulo `TC`, es necesario hacer algunas consideraciones previas. Ya que no hay ninguna restricción en el orden de realizar las transformaciones, el submódulo realizará las transformaciones en el orden que se muestran en la tabla 2.1, es decir, iniciará con la transformación 0 y continuará de manera progresiva hasta terminar con la transformación 7. Lo anterior tiene como fin agregar al submódulo un contador ascendente modulo 8, el cuál podrá ser inicializado cada vez que se trabaje con un nuevo bloque rango y además de ello, el contador por si mismo determinará el fin de las operaciones. Así, una sola variable de 4 bits, llamada `sim` determinará tanto el fin de las operaciones como las transiciones entre la mayoría de los estados.

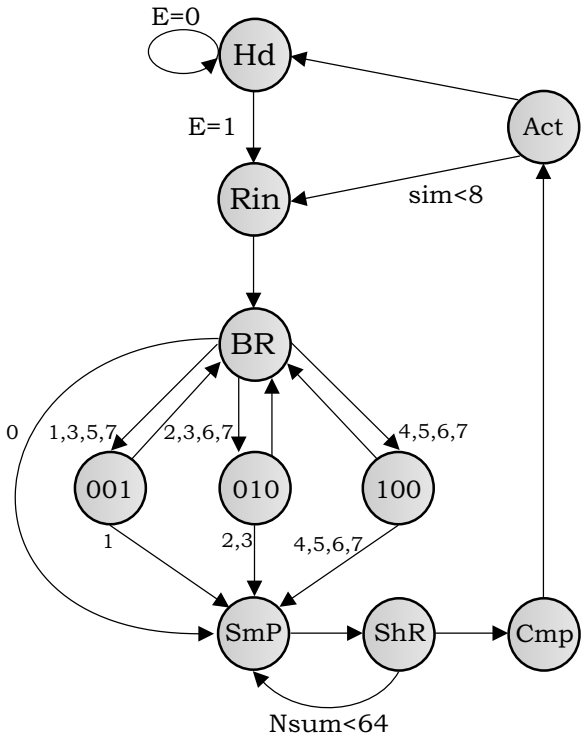

Figura 4.13: Diagrama de estados del submódulo de control para la transformación y comparación de bloques.

El submódulo `TC`, cuenta también con un estado que implica inacción, llamado `Hd` y habilitado por la señal `E`. El estado de `Hd` es aprovechado para la asignación de valores iniciales de algunas variables para el procesamiento de bloques, como por ejemplo, el valor del registro `DistMinima` es puesto a un valor "infinito" y el valor de `sim` es llevado a cero.

En el estado `Rin`, cuyo significado es reinicio, se inicializan valores que son necesarios para el procesamiento de una transformación. Ya que después de cada transformación se realiza la comparación entre el bloque dominio y el bloque rango transformado, es necesario, por ejemplo, llevar a cero los valores de los acumuladores y sumadores.

En el estado `BR` se determina cuál transformación se llevará a cabo, para ello se prueba la variable `sim`. Con la excepción del valor de cero de `sim`, se pasará del estado `BR` a alguno de los estados marcados como `001`, `010` o `100`, los cuales nombraremos en conjunto estados transformadores. Como se puede suponer, los estados transformadores generarán las señales para la habilitación del bloque ruteador correspondiente con su título y así mismo, realizarán los corrimientos necesarios sobre los registros. Como se puede observar en el diagrama, es posible ingresar a alguno de los estados transformadores con valores diferentes de `sim`. También se puede observar que estando en cualquiera de los estados transformadores y de acuerdo con el valor de `sim`, será posible ir al estado `SmP` o regresar al estado `BR`. En la parte inferior de cada estado transformador se marcan los valores de `sim` que hacen posible pasar al estado `SmP`, para cualquier otro caso se retorna al estado `BR`, lo que implicará pasar a otro estado transformador y con ello aplicar una segunda y hasta una tercera transformación.

El estado `SmP` genera las señales de habilitación necesarias para llevar a cabo las suma de los bytes más significativos de los registros `bloqueDominio` y `bloqueRango`. Realizada la suma de estos primeros bytes, se pasa al estado `ShR`, en el cual se aplica un corrimiento de 8 bits a ambos registros y además de ello, se prueba el valor de la variable `Nsum`, si es menor a 64 se regresa al estado `SmP` si no, se pasa al estado `Comp`.

En el estado `Comp` se habilita el comparador `Comp`, el cual compara el contenido de `Acumulador` y con el contenido del registro `DistMinima`. Después de ello se pasa de manera incondicional al estado `Act` en el cual se registran tanto los cambios de los valores de los registros `DistMinima` y `SimetriaFinal` como el incremento del valor de `sim`. Incrementado el valor de `sim`, se prueba si su valor es menor a 8, si es así, se pasa al estado `Rin` de lo contrario termina el procesamiento.

4.5. Interfaz hardware software

Un método común de integrar dispositivos periféricos en un sistema de cómputo, es el **mapeo de memoria**. En el mapeo de memoria, una parte del espacio de direcciones del CPU es interpretada no como un acceso a la memoria principal, sino como un acceso a un registro de entrada-salida del periférico. De esta forma, cuando por medio del programa se deseé enviar o recibir datos al o del periférico, se realizará un proceso de escritura o lectura a un espacio de direcciones específico.

Para la integración de la UTC en el sistema de cómputo conformado por un microprocesador y bloques de memoria e interconectados por un sistema de buses, como el mostrado en la figura 3.1, será necesario por un lado, agregar algunos dispositivos extras a la UTC y por otro lado el desarrollo de funciones especiales capaces de realizar operaciones de escritura y lectura al dispositivo. Tanto los dispositivos de hardware extras como las funciones de software, conforman la interfaz de comunicación entre el hardware y el software. En la figura 4.14 se muestra de forma esquematica la interfaz hardware-software, la cual estará conformada por el sistema de buses, un conjunto de registros de entrada-salida (I/O) y funciones de software especiales para el acceso al dispositivo.

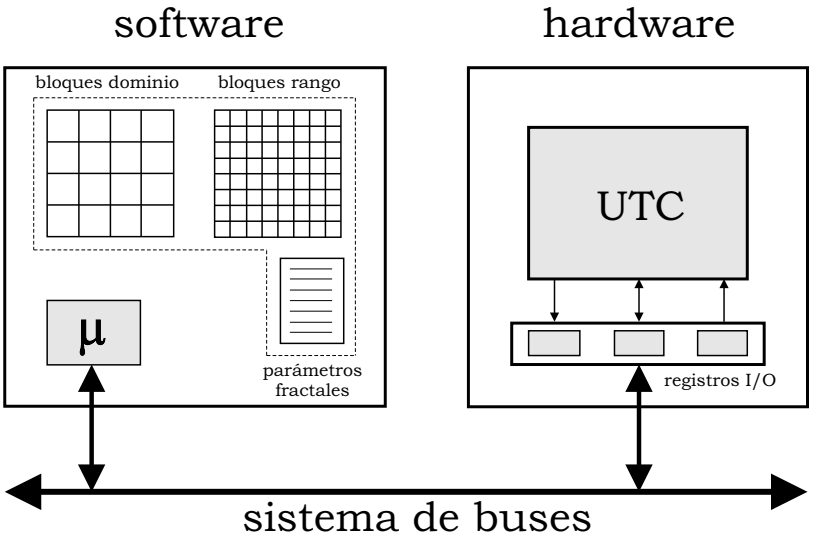

Figura 4.14: Interfaz de hardware par la UTC.

En realidad, los dipositivos de hardware agregados de la UTC y las funciones especiales de software para el microporcesador, servirán de acceso a una interfaz bien establecida: el sistema de buses. Un bus es un camino común que conecta varios dispositivos, así, el sistema de buses tiene como propósito reducir la cantidad de interconexiones entre el microprocesador y los periféricos. La estructura de un bus no solo contiene elementos físicos, si no que incluye un protocolo. El protocolo de bus es un conjunto de reglas bien definidas que todos los dispositivos conectados a él deberán de obedecer, siendo éste un árbitro para la comunicación entre los dispositivos.

Teniendo en cuenta lo anterior, el hardware agregado a la UTC consistió en conjunto de registros de propósito específico, los cuales son utilizados tanto para recibir como para enviar señales de control y datos, de manera bidireccional. A este conjunto de registros extras lo nombraremos registros de entrada-salida o **IO**. Por el lado de software, se desarrollaron funciones en línea (para eficiencientar el tiempo) con el fin de facilitar la minipulación de datos y direcciones. A este conjunto de funciones desarrolladas le llamaremos controlador de hardware o *driver*. Ambas partes son detallas en las secciones siguientes.

4.5.1. Registros IO

En la figura 4.15 se muestra de manera funcional los registros agregados a la UTC, los cuales funcionarán como una interfaz entre la UTC y el bus del sistema. El conjunto de registros **IO**, se dividen tanto en recepción y en envío de señales de control como en registros que reciben y envían datos.

Los registros que reciben señales del sistema de buses, comienzan con la identificación `Bus2IP`, el cual es un acrónimo de *Bus to Intellectual Property*, es decir, la señal va del bus al módulo diseñado. El bus envía los datos a la UTC por medio del registro `Bus2IP_Data` de 32 bits. El registro `Bus2IP_WrCE` de 2 bits, es utilizado para indicar el desplazamiento sobre los registros de usario `slv_reg0` y `slv_reg1`. Las señal de reloj y de reset, son enviadas a la UTC por medio de `Bus2IP_Clk` y `Bus2IP_Rst` respectivamente.

De manera similar, los registros que envían señales de la UTC al bus, comienza con el acrónimo `IP2Bus`. El bus recibe los datos de los registros de usario por medio de `IP2Bus_Data` y también recibe el desplazamiento por medio `IP2Bus_RdCE`. La UTC debe de generar señales para completar el protocolo del bus, estas señales son la de reconocimiento de operación `IP2Bus_Ack` y la señal de error `IP2Bus_Error`. La señal de reconcimiento es generada por la UTC cuando ha recibido un conjunto de bytes del bus.

Los registros cuya identificación inicia ya sea con `Bus2IP` o `IP2Bus`, trabajan en conjunto como una interfaz entre el microprocesador y 8 bytes integrados a la UTC. Cada uno de los 8 bytes pueden ser direccionables para su escritura o lectura desde el microprocesador. Los 8 bytes en realidad son dos registros de 32 bits cada uno, los cuales son marcados en la figura 4.15 como `slv_reg0` y `slv_reg1` y por claridad aparecen tanto en la parte superior (lectura) como en la parte inferior de la figura (escritura). El registro `slv_reg0` contendrá los primeros 4 bytes y el registro `slv_reg1` los últimos 4, cada uno de los cuales podrán ser accesibles al calcular desplazamientos sobre una dirección base.

Desde el punto de vista del hardware el proceso de escritura a uno de los ocho bytes de la UTC sería el siguiente: el microprocesador enviará al bus una solicitud de escritura a una dirección en particular de la memoria principal. El bus recibe la solicitud y reconoce que la dirección solicitada está en el rango de direcciones en donde se encuentra mapeada la UTC. El bus alista a la UTC para escritura y recibe por parte del microprocesador el dato. El bus coloca el dato en el registro `Bus2IP_Data` de la UTC y calcula el desplazamiento, colocando el resultado en `Bus2IP_WrCE`. Finalmente, por medio de un decodificador el dato es colocado en el byte correspondiente ya sea del registro `slv_reg0` o del registro `slv_reg1`.

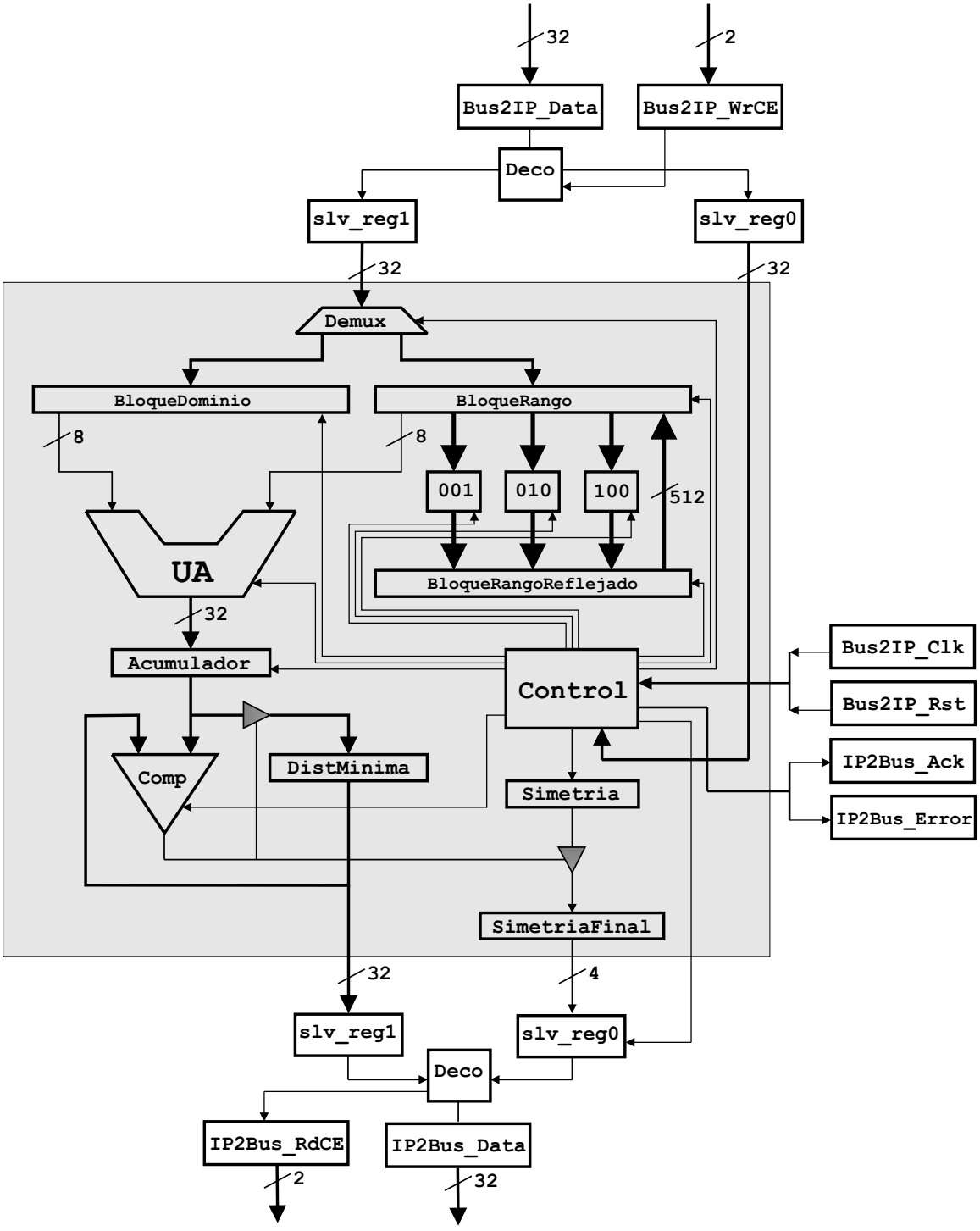

Figura 4.15: Registros de entrada-salida conectados a la UTC.

4.5.2. *Driver* de software

Desde el punto de vista del software, con el método de mapeo de memoria, basta con determinar el rango de direcciones y la dirección base del dispositivo para la escritura y lectura de datos en él, ya que para el microprocesador será la ejecutación de una simple operación de escritura o lectura a memoria. Conocida la dirección base y el rango de direcciones válidas para cada dispositivo, el desarrollo del controlador consistirá en la escritura de funciones que escriban o lean en ese rango direcciones de memoria.

La direcciones base de los dispositivos pueden ser agregadas en un archivo de cabecera con el fin de utilizarlas en el desarrollo de funciones que lean o escriban a un dispositivo en particular. Por ejemplo, las siguientes definiciones en lenguaje C, podrían ser incluidas para la manipulación de la de los parámetros de la UTC en un programa.

```
#define XPAR_UTC_NUM_INSTANCES  1
#define XPAR_UTC_0_DEVICE_ID    0
#define XPAR_UTC_0_BASEADDR     0x87FF0400
#define XPAR_UTC_0_HIGHADDR     0x87FF0408
```

Para la escritura o lectura de datos al dispositivo por medio de un programa escrito en un lenguaje de alto nivel, por ejemplo C, bastará con crear varibles que almacenen direcciones de memoria (apuntadores) y referenciar cada localidad antes de escribir o leer de ella. Al crear apuntadores de cierto tipo de datos (datos multibyte), la aritmética de apuntadores automáticamente realizaran el desplazamiento necesario para su referencia en memoria.

Como es sabido, al utilizar el lenguaje de programación C, es posible agrupar un conjunto de sentencias en una función, con el fin tanto de estructurar las operaciones de escritura y lectura al dispositivo como el reutilizar las operaciones con diferentes parámetros. Así por ejemplo, una función llamada `UTC_mWriteReg()` podría contener todo lo necesario para enviar los datos que le son pasados como parámetros al dispositivo. El prototipo podría ser el siguiente:

```
void UTC_mWriteReg(Xuint32 BaseAddress, unsigned RegOffset, Xuint32
                                                    Data);
```

La función `UTC_mWriteReg()` recibe tres argumentos. El primero es la dirección base del dispositivo almacenada en una variable de tipo entero sin signo de 32 bits, `Xunit32`, el cual es un tipo de dato definido previamente. El segundo argumento es el desplazamiento u *offset*, el cual será sumado a la dirección base, para llegar a los bytes deseados del dispositivo, dicho desplazamiento puede ser almacenado en una variable de 32 bits sin signo. Finalmente, el tercer argumento son los datos que se escribirán, los cuales también pueden ser almacenados previamente en una variable de 32 bits sin signo.

De la misma forma, se podría desarrollar la función `UTC_mReadReg` para la lectura de datos generados por un dispositivo que ha sido mapeado en memoria. Ambas

funciones podría ser integradas en una biblioteca de funciones para su utilización como controlador de hardware.

Las funciones `UTC_mWriteReg()` y `UTC_mReadReg` podrían ser utilizadas para la creación de una función de mayor nivel que tuviera como objetivo la comunicación con la UTC, como por ejemplo la función `reflejarHW()` descrita en la sección 4.3.3.

La función `reflejarHW()` podría preparar a la UTC para un nuevo procesamiento al llamar a la función `UTC_mWriteReg()` con la dirección base de la UTC como primer parámetro, un desplazamiento de cero para indicar que desea escribir al registro de usuario `slv_reg0` (el cual está conectado a la unidad de control) como segundo argumento y finalmente, como tercer argumento la palabra de control `0x0000`.

Una segunda llamada a `UTC_mWriteReg()` pero cambiando el tercer argumento por la palabra `0x0001`, le indicaría a la UTC que le van a ser enviados los marcos correspondientes a un bloque dominio. Para el envío de los 16 marcos, será necesario que la función `reflejarHW()` incluyera la llamada a `UTC_mWriteReg()` dentro de un ciclo que controlara tanto las referencias a los marcos, como los desplazamientos adecuados en los registros de la UTC. Para el envío de los marcos del bloque rango se procedería de la misma forma, atencediendo el envío de la palabra de control adecuada.

Para la lectura del resultado del procesamiento de los bloques por parte de la UTC, la función `reflejarHW()` deberá de incluir la llamada a la función `UTC_mReadReg` con los parámetros adecuados.

4.6. Sistemas adicionales

Para tener una mejor evaluación del tiempo de codificación de una imagen, fue necesario codificar imágenes de mayor resolución y así obtener un mejor panorama de cómo el sistema hardware-software diseñado impactaba en la aceleración. Para el manipulación de imágenes de mayor resolución, fue necesario agregar dos característcas al sistema: el acceso a un medio de almacenamiento mayor y el aumento de la memoria principal. Ambas características tienen en común que deberán de estar fuera del FPGA. El sistema de almacenamiento externo, deberá de contar con un sistema de archivos para su organización. De esta forma será posible almacenar archivos que contendrán imágenes de diferentes resoluciones.

Además de lo anterior, fue necesario contar con una terminal para poder observar el funcionamiento del sistema así como los resultados arrojados por el mismo. Para ello se agregó un módulo UART *Universal Asynchronous Receiver-Transmitter* o transmisor serial, para el envío y recepción de caracteres.

La tarjeta de desarrollo XUPV2P (*Xilinx Virtex-II Pro Development System*) cuenta con un FPGA de la familia Virtex II Pro, cuyos recursos son descritos en la figura 3.1. Además del FPGA, la tarjeta de desarrollo cuenta con varios recursos adicionales que pueden ser utilizados tanto para la programación del FPGA como para el desarrollo de sistemas. De entre todos los recursos que ofrece esta tarjeta de desarrollo, se tienen el sistema ACE para el acceso a memorias flash y un *slot* para la

inserción un módulo de memoria tipo DDR SDRAM. La utilización de ambos recursos se describen en las siguientes secciones.

4.6.1. Sistema ACE

El controlador Ambiente Avanzado para la Configuración del Sistema (SACE, *System Advanced Configuration Environment*) ofrece una interfaz inteligente entre el FPGA y una serie de fuentes de configuración. El controlador tiene varios puertos: un puerto para memorias *Compact Flash*, un puerto para el microprocesador y el puerto de pruebas JTAG. Así, el mismo sistema es utilizado tanto para programar el FPGA de manera directa, como para dar acceso a datos almacenados en las memorias flash. Los datos almacenados pueden ser tanto archivos que contienen una nueva programación para el FPGA, como datos para las aplicaciones.

Para tener acceso a los datos contenidos en una memoria `Compact Flash` desde un programa que se está ejecutando en los procesadores del FPGA, se cuenta con un API de programación similar (en uso y prototipos) al ofrecido en la biblioteca estándar de C, `stdio.h`, con la única diferencia de que el nombre de cada función es precedido por `sysace_`. El proceso de apertura, lectura-escritura y cierre de un archivo contenido en una `Compact Flash`, se puede realizar con las funciones:

```
SYSACE_FILE *sysace_fopen(const char *file, const char *mode );

int sysace_fclose(SYSACE_FILE *stream );

int sysace_fread(void *buffer, int size, int count,
SYSACE_FILE *stream );

int sysace_fwrite(void *buffer, int size, int count,
SYSACE_FILE *stream );
```

Como característica adicional, la memoria *Compact Flash* deberá de tener un sistema de archivos FAT12.

4.6.2. Sistema RAM

El FPGA se encuentra conectado a un *slot* de 184 pines en linea JEDEC-standard, en el cual se puede agregar un DIMM de memoria RAM de tipo DDRSDRAM (*Double Data Rate Synchronous Dynamic RAM memory*), con una capacidad de hasta 2GB o menos, en organizaciones de 64 o 72 bits. Para el sistema, solo será necesario agregar un nuevo controlador de memorias DDRSDRAM y que éste considere el rango de direcciones de acuerdo a la capacidad del DIMM.

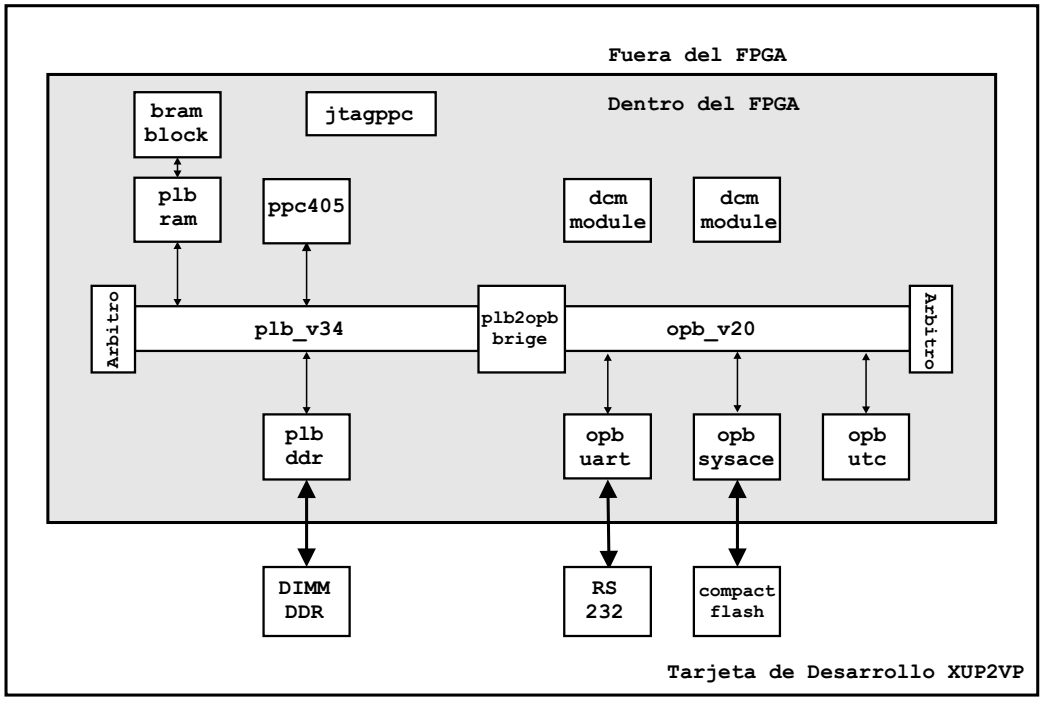

Figura 4.16: Sistema hardware-software en un chip basado en FPGA para la codificación fractal de imágenes.

4.7. Sistema final

En la figura 4.16 se muestra el sistema resultante de forma funcional. Como se ha comentado, el sistema fue implantado en la tarjeta de desarrollo XUP2VP, la cual interconecta el FPGA con otros dispositivos con el fin de utilizarlos como periféricos en el desarrollo de sistemas. El rectángulo central de la figura, representa el FPGA y su configuración funcional, los tres cuadrados representan los periféricos de la terjeta que fueron utilizados por el sistema desarrollado.

El cuadrado marcado como `DIMM DDR` representa el módulo memoria DDR SDRAM incrustado en el *slot* de la tarjeta de desarrollo. De la misma forma el cuadrado `Compact Flash` representa la inserción de una memoria flash en el puerto que ofrece la tarjeta XUP2VP. Finalmente, tanto el dispositivo RS232 para la transmisión serial como su puerto DB9, son representados por el cuadrado marcado como `RS232`. Las necesidades para incluir los tres dipositivos anteriores, fueron descritos en la sección 4.6.

En la siguiente sección se hace una breve descripción de los componentes del sistema que se programará en el FPGA. En otra sección se presentan los detalles para la implementación del sistema en la tarjeta de desarrollo.

4.7.1. Mapa de direcciones

La tabla 4.3 muestra el mapa de dispositivos en memoria para el sistema final.

Inicio-Fin	Dispositivo	Bus
0xfffe0000-0xffffffff	plb_bram	plb
0x00000000-0x0fffffff	DDR_256MB	plb
0x87ff0000-0x87ff00ff	RS232_Uart_1	opb
0x87ff0100-0x87ff01ff	SysACE_CompactFlash	opb
0x87ff0400-0x87ff0408	utc_0	opb

Tabla 4.3: Mapa de dispositivos en memoria.

4.7.2. Sistema en el chip

Descripción del hardware

- `dcm_module` (*Digital Clock Manager*), módulos de reloj digital, los cuales optimizan el manejo de relojes digitales en el FPGA. Emplea la técnica de *Delay-Locked Loop*, el cual es un sistema de control digital que emplea la realimentación de una señal de reloj para mantener parámetros de retraso aceptables en distintas condiciones de temperatura y voltaje.

- `proc_sys_reset`, sistema de reset general para el microprocesador y el sistema.

- `jtagppc_cntlr`, controlador del puerto JTAG (*Joint Test Action Group*) para la programación del FPGA, el microprocesador incrustado y de memorias flash.

- `ppc405`, procesadores PowerPC 405 a 100 Mhz.

- `plb_ram`, controlador para bloques de memoria RAM en chip, para ser agregado al bus local del procesador.

- `bram_block`, bloque de memoria RAM en chip de 128KB. En este bloque de memoria se alojará el programa principal: texto, pila y heap.

- `plb_ddr`, controlador para bloques de memoria DDRSDRAM fuera del chip, agregado al bus local del procesador. En los bloques de la memoria externa (DIMM de 256 MB), se almacenarán los pixeles de las imagenes.

- `plb_v34`, bus local del procesador. Bus síncrono de alta velocidad, utilizado para conectar bloques de memoria interna y externa al FPGA, así como periféricos de alta velocidad. Permite la selección del tamaño de palabra entre 32 y 64 bits.

- `opb_v20`, bus para periféricos en chip. Bus síncrono utilizado para conectar periféricos con tiempos de acceso variables. Soporta varios maestros y la conectividad de muchos periféricos es sencilla gracias a su identificación por multiplexación distribuida. Soporta transferencias de tamaño de palabra dinámico.

- `plb2opb_bridge`, puente para la interconexión del bus local del procesador y el bus de periféricos.

- `opb_uartlite`, controlador para un transmisor serial agregado al bus de periféficos en chip. El dispositivo controlará el puerto serial de la tarjeta para la comunicación del sistema con el puerto serial de una PC. La configuración típica es: 9600 baudios, 8 bits de datos, sin paridad y sin interrupciones. Configurado como entrada y salida estándar.

- `opb_sysace`, controlador para el sistema ACE para el acceso a memorias flash, agregado al bus de periféricos. Descrito en la sección 4.6.1.

- `opb_utc`, Unidad de Transformación y Comparación agregada al bus de periféricos en chip. Dispositivo descrito en la sección 4.4

Descripción del software

- `standalone`, sistema de inicio de programas en el procesador. En este caso, se indica que el sistema será un único programa en ejecución.

- `cpu_ppc405`, indica la arquitectura objetivo al compilador. Debido a que el FPGA es capaz de contener dos tipos de procesadores, el MicroBlaze y el PowerPC 405, se le indica al compilador (el cual puede ser un compilador cruzado[1]) a que arquitectura deberá de compilar el código fuente tanto de los dispositivos como el de los programas de usuario.

- `plbarb`, `uartlite`, `sysace`, `bram`, `ddr` representan el código fuente de los *drivers* de los diferentes dispositivos del sistema, los cuales deberán ser compilados y enlazados en la aplicación por medio de bibliotecas estáticas.

- `utc`, representa el código fuente de las definiciones y funciones para el acceso a la UTC, las cuales fueron descritas en la sección 4.5.2.

- `xilfatfs` (*LibXil FATFile System*). Biblioteca de funciones que permiten realizar operaciones de lectura-escritura sobre archivos almacenados en una memoria flash, a través del sistema *Xilinx SystemACE CompactFlash*.

[1]Un compilador cruzado es un compilador capaz de crear código ejecutable en otra plataforma distinta a aquélla en la que él se ejecuta. Esta herramienta es útil cuando quiere compilarse código para una plataforma a la que no se tiene acceso, o cuando es incómodo o imposible compilar en dicha plataforma, como puede ser el caso de los sistemas empotrados.

4.7.3. Implantación del sistema

El FPGA deberá ser programado para obtener la funcionalidad que se presenta en la figura 4.16. El proceso de programación del FPGA involucra varias etapas, y se llevan a cabo con herramientas especializadas de software. Para el desarrollo de sistemas que contemplan la integración de uno o más microprocesadores, ya sean programados (*soft core*) como el MicroBlaze o incrustados (*hard core*) como el PowerPC 405 de IBM, se utiliza el ambiente de trabajo EDK (*Embedded Development Kit*). El ambiente EDK está compuesto por una serie de aplicaciones para la configuración hardware, la codificación y el desarrollo de periféricos específicos, así como de herramientas para el desarrollo de software, como compiladores, bibliotecas y depuradores de código. El ambiente EDK trabaja de manera conjunta con la herramienta ISE (*Integrated Software Environment*), la cual se encarga de las tareas de síntesis, mapeo y ruteo de los dispositivos en el FPGA.

Configuración del Sistema
La configuración de un sistema por medio de la herramienta EDK, requiere de la creación de varios archivos que describan tanto el hardware como el software del sistema. Un archivo con el nombre del proyecto y con extensión `.mhs` (*Microprocessor Hardware Specification*) contendrá las especificaciones del hardware del sistema. El archivo describe tanto los elementos que conforman el sistema como la configuración de los mismos, como por ejemplo, el rango de memoria en el cual están mapeados dentro del sistema. Las opciones pueden asignarse de manera particular para cada dispositivo en un archivo de opciones hardware `.mpd` (*Microprocessor Peripheral Description*), para después ser descargado al archivo de configuración de hardware. El archivo `.mhs`, también contempla el mapeado de puertos del sistema. Cada puerto ha de ser especificado con un nombre, modo (entrada o salida) y el número de bits que lo componen.

Así por ejemplo, si requerimos que el microprocesador del sistema tenga acceso una memoria externa al FPGA, se especificarán en este archivo las diferentes señales que desde el controlador han de salir del FPGA al exterior para el control de esta memoria. También se especifican las opciones de configuración del procesador, tales como los buses que componen el sistema, su frecuencia de operación, etc.

De forma similar, un archivo con extensión `.mss` (*Microprocessor Software Specification*) contiene el nombre de los controladores de cada uno de los componentes del sistema e incluye también, la especificación del modo de compilación del código, asignación de bibliotecas a periféricos, especificación del método de solución de errores, y otras opciones del compilador. Este archivo depende explícitamente del archivo de descripción de hardware, ya que debe de especificarse el controlador para cada uno de los periféricos instanciados. De igual forma, si el dispositivo donde se va a cargar el diseño utiliza recursos hardware específicos, por ejemplo un multiplicador, es en este archivo donde se debe indicar al compilador que utilice este recurso de hardware. Los controladores de los periféricos se encuentran en otros archivos con extensión `.mdd` (*Microprocessor Driver Definition*) los cuales contienen el código necesario para el

control de cada periférico. En la instanciación del controlador, se debe especificar la versión del mismo, ya que es común que existan varias versiones del mismo.

Los dos archivos anteriores, `.mhs` y `.mss`, son de una sintaxis muy simple, puesto que se trata solamente de especificar opciones tanto de las instancias de periféricos como de sus controladores. El contenido de los archivos `.mhs` y `.mss` para el sistema de la figura 4.16, son mostrados de forma completa en el anexo A.

Especificado los componentes de hardware y software del sistema, sólo resta agregar el código del programa que representa la aplicación que será ejecutado en él. El programa puede ser escrito en lenguaje C, C++, o ensamblador ya que el compilador es el `gcc` del proyecto GNU (el cual es un compilador cruzado) y es capaz de optimizar el código de estos lenguajes a varios niveles. Al ser el PowerPC un procesador de 32 bits, prácticamente no existe ninguna restricción para escribir código, ya que su arquitectura soporta todos los tipos de datos.

En algunos casos, las partes del programa a ejecutar (texto, pila y *heap*), deberán de ser mapeado en diferentes bloques de memoria debido a su tamaño. Para que el programa sea cargado de manera correcta, es necesario especificar el espacio de direcciones que ocupará cada parte por medio de un *Linker Script*.

Finalmente se lleva a cabo tanto la síntesis del proyecto como la inclusión de los archivos ejecutables en la memorias respectivas. El proceso inicia con la sintesis de cada uno de los módulos de hardware incluidos en el archivo `.mhs`. Un segundo paso lleva a cabo la integración de todos los elementos, obteniendo como resultado un archivo de programación con la terminación `.bit`. En ésta parte del proceso, se deberá de estar pendiente de los recursos requeridos para la implementación del sistema, cuidando que no sean excesivos para el dispositivo al que se va a programar. Los recursos a verificar son tanto el número de celdas programables como la frecuencia de operación. Un último paso copia el programa ejecutable en la zona de memoria previamente establecida, al agregar una sección más al archivo de programación `.bit`.

Para el envío del archivo de programación al FPGA, la tarjeta XUP2VP se conecta con otro equipo (regularmente una PC) que contiene el archivo de configuración, por medio de un puerto, comúnmente el puerto USB. A través de una aplicación, se decide si se programa al FPGA de manera directa o si se almacena el programa en una de la memorias flash de la tarjeta XUP2VP para su posterior programación.

El proceso de implantación de un sistema en un FPGA, utilizando las herramientas de desarrollo EDK e ISE, se puede resumir en los siguientes pasos:

1. Describir en los archivos `.mhs` y `.mss` la configuración de un sistema mínimo de cómputo: un microprocesador, un módulo memoria principal (preferentemente bloques de memoria dentro del FPGA), una terminal para visualizar resultados y un sistema de buses para su interconexión.

2. Agregar al sistema mínimo los periféricos que sean necesarios tanto para el procesamiento como para la comunicación, e interconectarlos en alguno de los buses de acuerdo a su velocidad de operación. Por ejemplo, en caso de que la aplicación deba de procesar una cantidad de datos grande, de tal forma que no

se puedan inicializar de forma estática en el código del programa, se deberán de considerar las siguientes opciones para el envio y recepción de datos:

a) Creación de una terminal implementada por un UART y conectada al bus OPB

b) Creación de una terminal implementada por un dispositivo Gigabit ethernet conectada al bus plb

c) Creación de un acceso directo a los procesadores por medio de la implementación de una cola FIFO

La especificaciones y configuraciones de los periféricos a agregar (incluidos los del disarrollador), deberán de ser descritos en los archivos `.mhs` y `.mss`, así mismo, el código que describe tanto el hardware y como el software de cada uno de ellos se agregarán a las rutas respectivas para su compilación y síntesis.

3. El programa que se ejecutará en el sistema configurado (escrito en ensamblador o en un lenguaje de alto nivel) deberá de ser compilado y enlazado con la bibliotecas necesarias. En el mismo proceso, se deberá de verificar las partes del programa (texto, pila y *heap*) para su correcto mapeo en los módulos de memoria. De ser necesario, se deberán de mapear la partes en módulos de memoria separados, incluso en módulos fuera del chip que contiene el FPGA. En caso de no contar con un sistema operativo en el FPGA, el programa deberá de ser colocado en una dirección específica para su inicialización. Tanto el mapeo del programa en los diferentes módulos de memoria, como el tamaño de cada una de sus secciones, deberá de ser descrito en el archivo `Linker Script`.

4. Configurado el hardware del sistema y compilada la aplicación, se deberá de sintetizar primero el hardware y finalmente deberá de agregarse la aplicación de software, dando como resultado el archivo de configuración `download.bit` que es utilizado para la programación del FPGA.

5. Finalmente por medio de una aplicación diseñada para ese propósito, se programa el FPGA o una memoria flash contenida en la terjeta de desarrollo para la eventual programación del FPGA.

Configuración de un Periférico

La creación de un periférico se desarrolla en varias fases bien definidas, y para ello es necesario utilizar un sintetizador VHDL junto con otras herramientas incluidas en la herramienta EDK. Veamos las fases de desrrollo. Una vez diseñados los circuitos que conformarán el periférico, la primera fase consistirá en la descripción del periférico en un lenguaje de descripción de hardware, regularmente VHDL o Verilog. Dependiendo de la forma de comunicación y la velocidad, se elegirá el bus al que se conectará el periférico. Para las conexiones a los buses, se deberá de desarrollar una interfaz que maneje el protocolo necesario para la comunicación con el procesador. A estos bloques se les denomina IPIF (*IP InterFace*).

La siguiente fase es agregar el periférico en el sistema para su posterior compilación. La inclusión de un periférico en el sistema se realiza al describirlo, junto con sus parámetros, en los archivos `.mhs` y `.mss`. Será necesario también, ubicar las rutas en donde se encuentran cada uno de los archivos que conforman el nuevo periférico para llevar a cabo su síntesis.

Para la compilación del periférico, se crea un árbol de dependencias en dónde el módulo aporta sus requerimientos por medio de un archivo *.pao* (*Peripheral Analyze Order*). Una vez compilado satisfactoriamente, se asignan las señales de los buses al que será agregado el periférico y posteriormente se especifican las diferentes opciones de configuración del periférico. Para almacenar los valores por defecto de estas opciones se requiere la creación de un archivo `.mpd` (*Microprocessor Perepheral Definition*). Por medio de estos archivos, el EDK incluye al periférico dentro de la lista de posibles bloques a utilizar por el sistema y sólo habrá que instanciarlo para incluirlo en el diseño.

A continuación se enumeran algunos de los problemas más comunes tanto en la implentación del sistema como en el desarrollo de un periférico.

1. Comprobar que todas las señales de reloj (buses, periféricos y procesador) están conectadas al mismo puerto.

2. Comprobar el nivel de reset activo y cerciorarse de que coincide con el que proporciona la tarjeta de desarrollo.

3. Comprobar la correspondencia entre los puertos del archivo `.mhs` y la asignación de pines en el archivo `.ucf`.

4. Asegurarse que se utiliza el código de las bibliotecas y las versiones adecuadas.

5. Comprobar que todos los archivos necesarios para la síntesis de cada periférico del sistema, tengan la ruta adecuada y asegurarse que estas opciones son las requeridas por el FPGA en de la tarjeta utilizada.

6. Para primeras implementaciones es fundamental el uso de la herramienta de depuración (*debug*), XMD. Esta proporcionara información del estado del sistema, sobre todo si está cargado correctamente en el FPGA.

7. Al utilizar un UART y conectarlo a una PC a través del puerto DB9 de la tarjeta, asegurarse que los parámetros de transmisión y del protocolo RS-232 sean los correctos.

8. A veces, al compilar el código, ocupa más espacio de memoria que el especificado. Aunque no se genera un mensaje de error en el flujo de implementación de la herramienta, el sistema no arranca.

9. A la hora de compilar un periférico, incluir las bibliotecas utilizadas en el diseño en el orden de compilación correcto.

4.8. Conclusiones

La Metodología de Codiseño Hardware-Software (MCHS) nos ha permitido desarrollar una solución hardware-software a partir de una solución únicamente en software de la Codificación Fractal de Imágenes (CFI). Con base en un análisis de rendimiento de la solución en software, fue posible ubicar las partes más costosas en cómputo de dicha solución y llevar a cabo una partición. A partir de las funcionalidades más costosas en cómputo, se diseño un circuito de propósito específico que realiza tanto las trasformaciones sobre un bloque como las comparaciones entre dos bloques. Para la integración de las partes, se desarrolló una interfaz hardware-software, la cuál consistió en registros mapeados en memoria y de funciones de software para la lectura y escritura de datos.

Además del circuito desarrollado para las transformaciones y comparaciones de bloques, fue necesario agregar hardware para completar un sistema de cómputo. El hardawre adicional consistió en módulos para expandir la memoria principal para el procesamiento, así como la expansión de la memoria secundaria para el almacenamiento de datos, además de una terminal.

Todo el sistema de hardware fue compilado, sintetizado y mapeado, dando como resultado un archivo que configurará un dispositivo programable. El software fue compilado y enlazado con las bibliotecas estáticas necesarias, generando un archivo ejecutable. Ambos archivos, hardware y software, son integrados para programar un FPGA.

Capítulo 5

Resultados

5.1. Introducción

En el capítulo 2 se mostró el proceso de la Codificación Fractal de Imágenes (CFI), resaltando el gran número de operaciones que se requieren para codificar una imagen en escala de grises, traduciéndose en un tiempo de codificación considerable. En el capítulo 3 se presentó la Metodología de Codiseño Hardware-Software (MCHS), para el diseño y desarrollo de sistemas en un chip. Dichos sistemas pueden tener como plataforma un circuito programable con microprocesadores incrustados. En el capítulo 4 se llevó a cabo el Codiseño Hardware-Software (CHS) de la CFI. Con base en el análisis de una solución en software de la CFI, se realizó su particionamiento, llevando a hardware únicamente las partes del programa computacionalmente costosas y manteniendo en software las partes restantes. A partir de la funcionalidad de las partes más costosas en cómputo, se diseño una unidad de hardware a nivel de transferencia entre registros. Algunas de las partes de la aplicación que se mantuvieron en software, fueron modificadas para la integración con la unidad de hardware. Las modificaciones al software y la inclusión de algunos componentes extras en la unidad de hardware, dio como resultado una interfaz de comunicación. El sistema final fue el resultado de la integración de las partes de hardware, software e interfaz en un circuto programable con microprocesadores incrustados. Los microprocesadores ejecutan las partes de software, mientras que la unidad de hardware es agregada como un coprocesador especializado. Ambas partes, hardware-software, trabajan en conjunto para la aceleración de la CFI.

En este capítulo se presenta los resultados obtenidos al codificar imágenes por medio del sistema hadware-software diseñado en el capítulo 4.

Al aplicar la MCHS sobre una solución en software de la CFI, se obtuvo como resultado el sistema mostrado en la figura 4.16, el cual es un sistema de cómputo integrado por microprocesadores, memoria RAM y dispositivos periféricos, todos interconectados por un sistema de buses. La característica más sobresaliente del sistema resultante es que la mayoría de las unidades se encuentran integradas en un FPGA, es decir, en un solo chip.

Entre los periféricos incluidos en el sistema anterior, sobresale la unidad de transformación y comparación UTC, la cual es el resultado de llevar partes de la funcionalidad de software a hardware. El proceso de diseño de la UTC se mostró en la sección 4.4.

El sistema programado en el FPGA, junto con los módulos de hardware fuera de él, constituyen la solución hardware-software para llevar a cabo la CFI. A este sistema le llamaremos Codificación Fractal de Imagenes Hardware-Software en un FPGA (CFIHS-FPGA). La implantación del sistema CFIHS-FPGA en una tarjeta de desarrollo, de la forma mostrada en la sección 4.7.3, será nuestra plataforma de pruebas, la cual se describe en la siguiente sección.

5.2. Plataforma de pruebas

Por medio de los pasos descritos en la sección 4.7.3, el sistema CFIHS-FPGA fue implantado en la tarjeta de desarrollo Xilinx XUP Virtex II Pro Development System (XUP2VP), la cual cuenta con un FPGA de familia Virtex II Pro (XC2VP30-FF896) y varios módulos de hardware interconectados al mismo FPGA. Los módulos que rodean al FPGA, pueden ser divididos en tres grupos. El primero grupo lo conforman dispositivos que facilitan la programación del FPGA, en el segundo grupo se tienen módulos para la comunicación de los sistemas programados en el FPGA con dipositivos externos y en el tercero se encuentran dispositivos para asistir a los sistemas programados en el FPGA en un procesamiento específico.

El sistema CFIHS-FPGA requirió, además del FPGA, la utilización de la ranura (*slot* o *socket*) para la inserción de un DIMM de memoria DDRSDRAM de 256MB como memoria extra para el procesamiento; la inserción en el puerto correspondiente de una tarjeta *Compact Flash* de 8MB con un sistema de archivos FAT12 con la finalidad de que el sistema CFIHS-FPGA contara con un dispositivo de almacenamiento; y finalmente, un dispositivo de transmisión serial RS232 utilizado como una terminal del sistema.

Para llevar a cabo la codificación de una imagen por medio del CFIHS-FPGA implantado en la tarjeta XUP2VP, se deberán tener en cuenta las consideraciones siguientes.

- Al ejecutar la parte de software del sistema CFIHS-FPGA como una aplicación única en los procesadores del FPGA (*standalone*), antes de ser cargarda para su ejecución, se deberá de tener previamente almacenadas en la tarjeta *Compact Flash* las imágenes a codificar.

- El nombre de la imagen a codificar podría ser especificado ántes de compilar el programa, o de otra forma, se deberá de desarrollar código para la interacción con el usuario por medio de una terminal.

- Los resultados de la codificación podrían ser mostrados a través de la terminal (escribiendo a la salida estándar) y/o ser escritos en un archivo almacenado en la *Compact Flash*.

- También será necesario tener conectado la salida de la terminal del sistema (puerto DB9 de la XUP2VP) al puerto serial de una PC para recibir y enviar datos, con la configuración adecuada.

Resueltos los requisitos anteriores se procede a la programación del FPGA. El resultado de la descripción, integración, compilación y síntesis del sistema CFIHS-FPGA, es un archivo para la configuración del FPGA (`.bit`), el cuál es generado en conjunto por las herramientas ISE y EDK. Para el envío del archivo al FPGA, la tarjeta XUP2VP se conecta con otro equipo (regularmente una PC), por medio de un puerto, comúnmente el puerto USB. A través de una aplicación, se decide si se programa al FPGA de manera directa o si se almacena el archivo en una de la memorias flash de la tarjeta XUP2VP para su posterior programación.

5.3. Verificación del sistema

La primera prueba a realizar es la verificación del correcto funcionamiento del sistema CFIHS-FPGA implantado en la XUP2VP. Para ello se realizan experimentos consistentes en la codificación de imágenes por parte del sistema CFIHS-FPGA y posteriormente se verifican los parámetros fractales obtenidos, los cuales pueden ser mostrados en la terminal o ser almacenados en un archivo con extensión `.fif`, de *Fractal Image Format*, como se sugiere en [2]

La verificación de los parámetros fractales se puede llevar a cabo de dos formas. La primera es hacer que tanto el sistema CFIHS-FPGA como la solución en software original, realicen la codificación de la misma imagen. Obtenidos los archivos que contienen los parámetros fractales por parte de ambos sistemas, se lleva a cabo la decodificación, también por ambos sistemas. Las imágenes resultantes se comparan por medio de la aplicación de la ecuación 2.5. El segundo método podría ser la comparación de valores de los parámetros fractales obtenidos por ambos sistemas al codificar la misma imagen.

En ambos casos los resultados fueron idénticos, es decir, los parámetros fractales obtenidos tanto por la solución en software original como los obtenidos por el sistema CFIHS-FPGA, tienen los mismos valores. Lo anterior se explica por la forma en que se desarrollo la UTC. Durante el desarrollo de cada parte del circuito, se comparaban los resultados obtenidos por el hardware con los resultados de la solución en software.

Al codificar una imagen con el sistema CFIHS-FPGA y obtener los mismos valores de los parámetros fractales obtenidos con la solución en software original, los análisis sobre la compresión de imágenes presentados en las secciones 2.7.2, 2.7.3 y 2.7.4 son válidos para el sistema CFIHS-FPGA.

En la figura 5.1 se muestran la decodificación de las imágenes Lena, Bird y GoldenHill. Todas ellas con una resolución de 512×512 pixeles. Las imágenes fueron inicialmente codificadas por el sistema CFIHS-FPGA y posteriormente decodificadas al implantar el código de descompresión en el propio sistema CFIHS-FPGA.

(a) (b) (c)

Figura 5.1: Aproximación fractal de varias imágenes utilizando el sistema CFIHS-FPGA. (a) Lena, (b) Bird, (c) GoldenHill.

5.4. Tiempo de codificación

Comprobado el correcto funcionamiento del sistema CFIHS-FPGA, se procedió a realizar experimentos para medir el tiempo de codificación. Los experimentos consistieron en medir el tiempo codificación tanto para una solución puramente en software, como para una solución hardware-software. Con ambos resultados es posible evaluar la aceleración obtenida al utilizar un sistema hardware-software sobre un sistema únicamente en software.

La medición del tiempo de ejecución de una aplicación programado en un FPGA se puede llevar a cabo con la la llamada la función `Xtime_GetTime()`, definida en `xtime_l.h`. La función `Xtime_GetTime()` accede a un registro de 32 bits dentro del PowerPC en el cual se lleva la cuenta del número de ticks del procesador. Debido a que la codificación de las imágenes de mayor resolución puede desbordar el conteo del registro varias veces, la medida de tiempo también se puede realizar al sincronizar la programación del FPGA con el reloj de una terminal.

5.4.1. Solución software

En la sección 2.7.1 se presentaron los tiempos de codificación por medio de una solución en software, al implantar el algoritmo propuesto en [2] en lenguaje C y ejecutarlo en un sistema de cómputo convencional. El mismo código se modificó ligeramente para implantarlo en un sistema de cómputo programado en un FPGA.

El sistema presentado en la figura 4.16 es capaz de ejecutar programas como cualquier sistema de cómputo convencional, siempre y cuando se consideren tanto las restricciones de memoria principal como la de los periféricos agregados al sistema. Si omitimos la unidad UTC de la figura 4.16, se obtiene, guardando las proporciones, el sistema de cómputo convencional de la figura 5.2, el cual nombraremos Codificación Fractal de Imágenes únicamente Software en FPGA (CFISFPGA).

Al codificar la imagen Lena, Bird y GoldenHill en distintas resoluciones por medio de este sistema mostrado en la figura 5.2 , se obtuvieron los tiempos de codificación mostrados en la tabla 5.1.

	Tiempo de Codificación		
Resolución [pixeles]	**Lena** [seg]	**Bird** [seg]	**GoldenHill** [seg]
64×64	52.8	52.5	53.1
128×128	1056	1057	1055
256×256	17408	17416	17421
512×512	303104	303123	303145

Tabla 5.1: Tiempos de codificación de varias imágenes por medio de la solución en software de la CFI, programado en un FPGA.

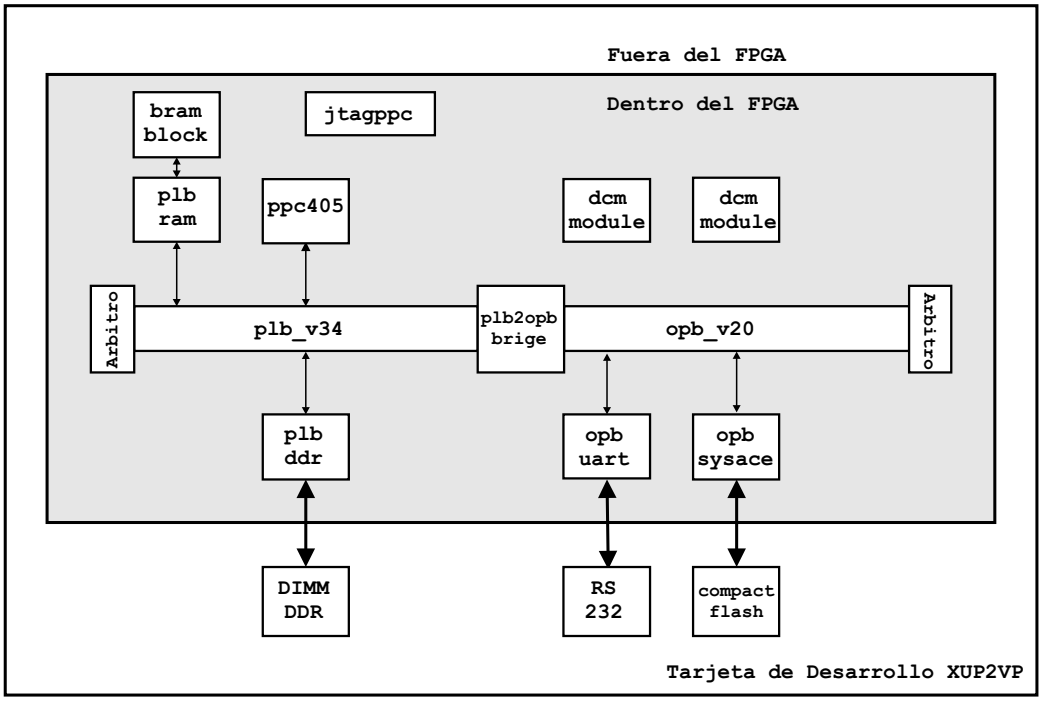

Figura 5.2: Sistema de cómputo en un chip basado en FPGA, para la solución en software de la codificación fractal de imágenes .

5.4.2. Solución hardware-software

En la tabla tabla 5.2 se muestra el tiempo de codificación de las imágenes Lena, Bird y GoldenHill en diferentes resoluciones, al utilizar el sistema CFIHS-FPGA, mostrado en la figura 4.16, página 71.

	Tiempo de Codificación		
Resolución [pixeles]	**Lena** [seg]	**Bird** [seg]	**GoldenHill** [seg]
64×64	7.1	6.8	7.3
128×128	160	163	165
256×256	2048	2056	2059
512×512	45056	45101	45091

Tabla 5.2: Tiempos de codificación de varias imagenes por medio de la solución hardware-software de la CFI, programado en un FPGA.

5.4.3. Aceleración

En base a los resultados obtenidos en las secciones 5.4.1 y 5.4.2, se calculó la aceleración por medio de la ecuación 5.1. En la tabla tabla 5.3 se muestra la aceleración obtenida para cada caso.

$$A = \frac{Tiempo_{Software}}{Tiempo_{HardwareSoftware}} \tag{5.1}$$

	Aceleración		
Resolución	**Lena**	**Bird**	**GoldenHill**
64×64	7.4	7.7	7.3
128×128	6.6	6.5	6.4
256×256	8.5	8.5	8.4
512×512	6.7	6.7	6.7

Tabla 5.3: Tiempos de codificación de varias imagenes por medio de la solución en hardware-software de la CFI, programado en un FPGA.

5.5. Análisis de resultados

Al comparar los resultados de las tablas 5.1 y 5.2, se puede apreciar una mejoría notable en el tiempo de codificación de una imagen al utilizar el sistema CFIHS-FPGA. El sistema CFIHS-FPGA obtiene un mejor desempeño al conjuntar en su procesamiento, un microprocesador y una unidad de hardware dedicada.

Sin duda, la característica de ser un sistema programado en un solo chip, es una contribución importante para el mejoramiento del rendimiento de la aplicación. En las siguientes secciones se muestra un análisis, tanto de los recursos ocupados para la programación de la aplicación en el circuito programable, como de los tiempos de procesamiento. Dicho análisis es tanto para la unidad de hardware diseñada como del sistema en general

5.5.1. Análisis de recursos ocupados

Como ya se ha comentado, el sistema fue programado en un FPGA de la familia Virtex II Pro, específicamente en en el dispositivo 2vp30ff896-7. En la tabla 5.4 se muestra un resumen de los recursos utilizados de este dispositivo

Los IOB sirven como una interfaz entre los pines del chip y la lógica interna programa. El sistema utilizó una cuarta parte de éstos dispositivos, de los cuales, 5 fueron de entrada, 88 de entrada-salida y 46 de salida. La mayoría de estos puertos fueron utilizados para el acceso a los módulos de memoria externos agrupados en un DIMM DDR.

Recurso	Utilizados	Porcentaje
External IOBs	139 de 556	25 %
PPC405	1 de 2	50 %
RAMB16	64 de 136	47 %
SLICE	7269 de 13696	53 %
DCM	2 de 8	25 %
JTAGPPC	1 de 1	100 %

Tabla 5.4: Recursos del FPGA 2vp30ff896-7, utilizados por el sistema.

Fue utilizado uno de los dos procesadores PPC405 y para su programación fue agregado el único JTAGPPC disponible. Se ocupó aproximadamente la mitad de los bloques de memoria RAM de 16 bits programados (RAMB16). Fueron utilizados 2 de los 8 módulos digitales DCM disponibles.

Los reportes muestran una ocupación de aproximadamente la mitad de la lógica programble disponible (SLICEs) en el dispositivo 2vp30ff896-7, al implantar el sistema de la figura 4.16. Como se definieron en la sección 3.4 (página 31), los FPGAs están construídos como arreglos de compuertas estructurados como CLBs (*Configurable Logic Block*), los cuales pueden ser interconectados por medio de líneas de conexión implantadas en el circuito para este propósito. Cada CLB en un FPGA de la serie Virtex de Xilinx, está compuesto por dos slices, y cada slice contiene dos registros y dos generadores de función, que son dos LUT (*Look up Table*) de cuatro entradas.

Por medio de los archivos de reportes, tanto del EDK como del ISE, se pudo comprobar que la UTC por sí sola ocupa aproximadamente 5000 SLICES, lo que equivale 70 % del total del sistema CFIHS-FPGA programado. Al observar la organización de la UTC mostrada en la figura 4.10, se puede suponer que la mayor parte de los recursos son ocupado por los registros de 512 bits, los cuales además de ser de gran longitud, requieren de lógica extra para llevar a cabo la carga de datos, corrimientos y rotaciones. La suposición anterior queda comprobada al revisar los archivos de reportes al implementar unicamente la UTC por medio de la herramienta ISE. Los unidades aritméticas al ser secuenciales, ocupan pocos recursos, así como los registros restantes.

Con base en sus recursos, el dispositivo 2vp30ff896-7 podría contener un sistema similar al mostrado en 4.16 pero con dos instancias de la UTC, para el trabajo paralelo. Evidentemente, para esta nueva organización sería necesario modificar en el programa que se ejecuta en los procesadores para el flujo de datos entre las dos unidades. Si se utilizara el dispositivo 2vp1001164 de la misma familia Virtex II Pro, el cuál es el aproximadamnte 3 veces más denso que el dipositivo utilizado en esta tesis, podría contener de 3 a 4 UTCs para el trabajo paralelo, con lo que muy probablemente se obtendría un desempeño aún mejor. Para estos casos, las componentes de sincronización y comunicaciones comenzarían a tomar valores que deben ser consideras.

5.5.2. Análisis de tiempos de operación

Al examinar los archivos de reportes de las herramientas de diseño, se verificó que el sistema implantado en el dispositivo 2vp30ff896-7 tiene una frecuencia de operación máxima 101.327MHz. La frecuencia de operación máxima del sistema es principalmente dictaminada por la frecuencia de operación del microprocesador y por el reloj del bus del sistema. Debido a que se planeó incluir a la UTC como un periférico conectado al sistema de buses, se determinó que su frecuencia debería de acercarse a la máxima frecuencia de operación del bus.

Debido a lo anterior, el diseño de los circuitos de la UTC estuvo basado en una frecuencia de operación de 100Mhz, esto quiere decir que las operaciones fueron segmentadas de tal forma que las operaciones y transferencias fueran estables en un tiempo máximo de 10ns. Para algunas unidades de la UTC, este tiempo excedía por mucho el tiempo para su operación, como es el caso del corrimiento y rotación de los registros; pero para las unidades secuenciales es un tiempo suficiente, como es el caso de las unidades aritméticas. Aumentar la frecuencia de operación implica, por un lado, crear más unidades combinacionales y menos secuenciales, pero como se sabe, el diseño de unidades combinacionales grandes tiene como consecuencia una mayor utilización de recursos. Lo anterior es el compromiso siempre presente en el diseño de sistemas digitales.

Como se ha comentado, el procesamiento de la UTC está dividido en dos partes, transformación y comparación. Para llevar a cabo las transformaciones, se realizan operaciones de corrimientos sobre registros. Cada corrimento se pueden llevar a cabo en un ciclo de reloj, lo que implica que una transformación puede llevarse a cabo entre 8 y 24 ciclos de reloj. Es el proceso de comparación el que ocupa más tiempo, ya que por un lado debe de efectuar un número mayor de operaciones de forma secuencial y por otro lado las operaciones son de tipo aritméticas y de comparación. Debemos de recordar que las operaciones aritméticas tienen operandos de 8 y de 32 bits y las llevan a cabo en unidades combinacionales que incluyen sumadores con propagación de acarreo. La mediciones hechas dan como resultado que cada transformación se enfectúa en un promedio de 250 microsegundos, de los cuales más del 90 % del tiempo es ocupado por la comparación entre bloques.

El reporte de tiempo muestra también que el máximo retardo entre dispositivos es de 2.184ns y se da entre el controlador de memoria RAM interno al FPGA y el módulo de memoria externo DDR. De nuevo al analizar el diagrama de la figura 4.16, queda claro que los dispositivos periféricos que son ocupados más intensamente son los bloques de memoria RAM, tanto la programada como la externa. De manera intuitiva se puede suponer que son los módulos de memoria externa (ubicadas en el DIMM DDR) y sus controladores internos, los que provocan el mayor retraso entre dispositivos.

Capítulo 6

Conclusiones y Trabajo Futuro

6.1. Conclusiones

La mayor ventaja de un procesador de propósito general es la gran flexibilidad que ofrece al cambiar su funcionalidad en base al software. Sin embargo, la gran flexibilidad tiene como consecuencia un bajo rendimiento en tiempo, comparado con los circuitos diseñados para una aplicación en específico. Los circuitos de propósito especifico son muy eficientes, sin embargo, su planeación, desarrollo, implementación y pruebas son muy costosos y aumentan conforme a la complejidad de la aplicación. El codiseño hardware-software basado en plataformas de circuitos programables que incluyen procesadores incrustados, ayuda por un lado a acelerar aplicaciones mediante la integración de partes de software y partes de hardware y por otro lado, disminuye la complejidad del desrrollo del hardware, al utilizar las herramientas de síntesis modernas desarrolladas en el cómputo programable.

Los resultados presentados en el capítulo 5, muestran que es posible mejorar el rendimiento de de una solución en software de la Codificación Fractal de Imágenes (CFI), al trasladar únicamente las funcionalidades más costosas en tiempo a un hardware dedicado, manteniendo las demás partes en software. Este tipo de particionamiento de la aplicación da como resultado un sistema de colaboración entre el hardware y software mejorando notablemente su rendimiento. Fue posible comprobar también que el diseño, desarrollo e implementación de sistemas hardware-software es ahora más accesible cuando se implanta en un circuito programable que cuente con procesadores programados o incrustados.

El particionamiento de la solución inicial en software de la CFI, se basó en un análisis de rendimiento. Dicho análisis mostró el tiempo que ocupa cada una de las partes en su ejecución así como el número de llamadas de las mismas. Como la mayoría de las aplicaciones, la solución en software de la CFI se compone principalmente de tres partes: lectura de datos, procesamiento de datos y escritura de resultados. La lectura y escritura de datos ocupan un tiempo proporcional a la resolución de la imagen, sin embargo, es el procesamiento de los datos el que ocupa la mayor parte del tiempo de procesamiento aumentando también de forma proporcional a la resolución. Fueron entonces las partes de procesamiento de datos las principales candidatas a ser llevadas a un hardware dedicado, manteniendo las restantes en software.

Fue necesario un análisis más detallado de la funcionalidad de las partes más costosas y recurrentes, ya que por un lado no es sencillo llevar una funcionalidad originalmente diseñada para software a circuitos de hardware de bajo nivel, y por otro lado un diseño mal planeado puede tener como consecuencias un consumo de recursos inecesarios o una disminución de la aceleración consecuencia de agregar una nueva componente de tiempo debido a la comunicación entre las partes involucradas. El análisis de rendimiento señaló que dos funciones ocupaban en conjunto la mayor parte del tiempo de procesamiento, una revisión de sus funcionalidades mostró que sus operaciones consistían tanto en el movimiento espacial de pixeles (transformaciones) como de operaciones aritméticas sencillas (comparaciones). La reasignación espacial de datos y las operaciones aritméticas de suma y multiplicación, pueden ser ejecutadas por unidades funcionales conectadas entre registros y supervisadas por una unidad de control, teniendo como posibles restricciones la cantidad de recursos a ocupar y/o su frecuencia de operación. Como se sabe, los circuitos digitales combinacionales siempre será más rápidos, pero su implantación requiere de grandes consumos de recursos y un tiempo considerable de diseño, en cambio, los circuitos digitales secuenciales aunque más lentos que los combinacionales, ayudan tanto al ahorro de recursos como a posibles segmentaciones de la aplicación. Las restricciones de recursos ocupados y tiempo de operación, son un compromiso en el diseño de sistemas digitales y dependen principalmente de la experiencia e intuición del diseñador.

La metodología de diseño *top-down* permite el diseño del circuitos digitales cuando éstos son descritos a partir de requerimientos funcionales. Esto se logra al dividir la funcionalidad en partes más pequeñas, las cuales son planeadas e implementadas por separado considerando en todo momento la interacción con las demás para su posterior integración. El circuito diseñado en esta tesis, llamado UTC, fue dividido en dos partes: transformación de bloques (reubicación de pixeles) y comparación de bloques (operaciones aritméticas). La reubicación de pixeles se puede llevar a cabo de forma combinacional o de forma secuencial, siendo esta última la que menor cantidad de recursos utiliza pero con un tiempo mayor de operación. Por medio de corrimientos y rotaciones de registros de manera secuencial, fue posible realizar la reubicación espacial de un conjunto de pixeles en unos cuantos ciclos de reloj. La comparación de bloques se implemento con la combinación de circuitos secuenciales y combinacionales, ya que se obtienen de forma secuencial resultados parciales calculados por unidades

totalmente combinacionales, cuyos operandos son de longitud de 8 y 32 bits.

La unidad de control hace posible la ejecución secuencial entre una transformación de un bloque y la posterior comparación entre bloques. La unidad de control envía y recibe señales tanto para la habilitación como para la deshabilitación de las unidades funcionales que conforman la ruta de datos, y así ejecutar una operación en específico. Así por ejemplo, en un momento puede enviar las señales de corrimiento y de habilitación de los bloques ruteadores para realizar una de las ocho transformaciones posibles, y en otro momento puede habilitar las unidades aritméticas para realizar las operaciones de comparación. Las unidades de control son generalmente creadas de dos formas: alambradas y microprogramadas. Las unidades alambradas están compuestas de una parte combinacional y una parte secuencial y son el resultado de la implatanción de una funcionalidad descrita por un diagrama de estado. La microprogramadas es una secuencia de palabras de control almacenadas en una memoria de solo lectura y direccionadas de forma secuencial con posibilidades de saltos al calcular una nueva dirección válida de la misma memoria. Un nuevo compromiso surge al diseñar unidades de control: por un lado tenemos la complejidad de la descripción y la flexibilidad para el cambio de funcionalidad, y por otro lado se tiene la utilización de recursos y su velocidad de operación.

Las posibilidades de diseño de los sistemas de digitales antes mencionadas, dejan abierto el desarrollo de un circuito a partir de una funcionalidad. Por ejemplo, al ser independiente cada transformación y comparación, se podría proponer que la unidad UTC llevara a cabo las ocho transformaciones de forma paralela, a través de submódulos totalmente combinacionales. Las comparaciones entre bloques de pixeles, también podrían ser evaluadas de manera más eficiente si se crearan circuitos que calcularan sumas parciales. La descripción seguramente sería mucho más rápida que la UTC diseñada en esta tesis, sin embargo se puede deducir que su implementación la haría prohibitiva para algunos dispositivos con recursos limitados.

Un diseño inadecuado de la interfaz de comunicación entre hardware y software puede penalizar cualquier mejora de rendimiento que un hardware agregado al sistema pueda aportar. Para evitar estos inconvenientes, es necesario que la planeación de las unidades de hardware consideren desde un inicio, el cómo serán agregadas al sistema. Si por ejemplo, se planea que la unidad de hardware sea agregada a uno de los buses del sistema, la selección del bus deberá ser determinada por la velocidad de operación tanto del bus como de la misma unidad de hardware. Una unidad de hardware agregada a un bus inadecuado, podría aumentar el *overhead* de la operación del bus, si ésta unidad es demasiado lenta para la velocidad de operación del bus. Por otro lado, una unidad muy rápida (y posiblemente muy costosa en recursos) podría verse subutilizada si el microprocesador no es capaz de suministrar adecuadamente los datos para su procesamiento. El diseño de la UTC fue inicialmente proyectada para que fuera agregada al bus de dispositivos periféricos y operara a la frecuencia máxima del mismo bus. A partir de la frecuencia de operación se disañaron las unidades funcionales mantenido un equilibrio entre la velocidad operación y los recursos empleados. Un aumento o disminución en la frecuencia de operación del bus implicaría necesa-

riamente cambios en el diseño de la UTC. En caso de un aumento de la frecuencia, sería necesario segmentar las operaciones que realiza la UTC, especificamente en la unidad de control.

El control de dispositivos de hardware a través de un programa ejecutado en un microprocesador (*drivers*), debe considerar el tiempo de respuesta del dispositivo. Cuando el periférico opera a frecuencias altas, es común que estos programas sean desarrollados totalmente en lenguaje ensamblador o en una combinación con un lenguaje de mediado nivel, como C. Lo anterior tiene como fin mejorar el tiempo de ejecución del programa al minimizar el número de instrucciones que podría generar un compilador. En el lado opuesto está el acceso a los dispositivos cuya frecuencia de operación es baja. En esos casos el programa deberá de ser capaz de detectar cuando el dispositivo ha terminado su procesamiento, esto lo puede hacer aplicando un mecanismo de espera ocupada o por medio de la recepción de señales por parte del hardware interrupiendo el procesamiento del procesador. En el caso del sistema presentado en esta tesis, el procesador puede operar desde una frecuencia igual o mayor al del bus de periféricos en el que se agregó la UTC, siendo una frecuencia igual o mayor, el procesador deberá de esperar por una señal de terminación por parte de la UTC, la cual puede implementarse como una espera ocupada al probar un valor en una localidad de memoria.

Finalmenete, el desarrollo de un controlador de hardware o *driver*, puede complicarse si se considera que el controlador deberá de ser integrado a un sistema operativo. La complejidad del desarrollo dependerá ahora también del modelo del sistema operativo.

6.2. Trabajo Futuro

El problema de cómo implementar una aplicación de forma eficiente en un sistema de cómputo, es aún un problema abierto. Cualquier aplicación se podría implementar en puramente software o puramente hardware o en una combinación de ellas. Ahora incluso, se puede explorar cuál es el porcentaje de la aplicación que debería ser software y cuál porcentaje hardware. Lo anterior podría obedecer no solo a la eficiencia en tiempo, sino también en costos de diseño e implementación.

Con base en lo anterior, explorar las posibilidades de diseño para la CFI son muchas. Comentaremos algunas. La primera alternativa es instanciar más de una UTC y modificar ligeramente el programa para la asignación de bloques entre las unidades y así obtener un trabajo paralelo a nivel de unidades de hardware. Esta posibilidad podría mejorar aún más si el sistema cuenta con más de un procesador, ya que el nivel de paralelismo de la apliacación permite el procesamiento concurrente entre bloques, se podría plantear un sistema de procesamiento multihilo con un mecanismos de de memoria compartida, en dónde tanto el código como los datos sean compartidos. Se puede preever que al involucrar más microprocesadores, ya sea incrustado o programados en un sistema con más de una UTC, implicaría modificaciones mayores tanto en software como en hardware, principalmente en la sincronización y comunicación entre unidades. Otra posibilidad a nivel de hardware es la de proponer un paralelismo de grano menor, al aumentar el número de registros en la UTC y permitir evaluar más de un bloque rango a la vez. Esta segunda opción estaría principalmente restringida por la cantidad de recursos del dispositivo programable y la complejidad de la unidad de control ya que tendría más unidades que supervisar. Con esta propuesta se deberán de considerar el integrar más unidades aritméticas, las cuales ocupan un espacio importante de los recursos, ya que no se gana mucho al realizar las ocho transformaciones de manera paralela, si al final el cuello de botella será la comparación.

En general, el mayor trabajo para una propuesta de codiseño, es encontrar un balance óptimo en el particionamiento de las aplicaciones y esa parte sigue aún estando fundamentada en la experiencia y la intuición del diseñador.

Apéndice A

Archivos de Configuración

Archivo: `sistema.mhs` (*Microprocessor Hardware Specification*)

```
1 # ##############################################################################
2 # Sun May 21 02:40:10 2006
3 # Target Board:  Xilinx XUP Virtex-II Pro Development System Rev C
4 # Family:       virtex2p
5 # Device:       xc2vp30
6 # Package:      ff896
7 # Speed Grade:  -7
8 # Processor: PPC 405
9 # Processor clock frequency: 100.000000 MHz
10 # Bus clock frequency: 100.000000 MHz
11 # Debug interface: FPGA JTAG
12 # On Chip Memory : 128 KB
13 # Total Off Chip Memory : 256 MB
14 # - DDR_SDRAM_32Mx64 Single Rank = 256 MB
15 # ##############################################################################
16
17 PARAMETER VERSION = 2.1.0
18
19 PORT fpga_0_RS232_Uart_1_RX_pin =
20  fpga_0_RS232_Uart_1_RX, DIR = INPUT
21 PORT fpga_0_RS232_Uart_1_TX_pin =
22  fpga_0_RS232_Uart_1_TX, DIR = OUTPUT
23 PORT fpga_0_SysACE_CompactFlash_SysACE_CLK_pin =
24  fpga_0_SysACE_CompactFlash_SysACE_CLK, DIR = INPUT
25 PORT fpga_0_SysACE_CompactFlash_SysACE_MPA_pin =
26  fpga_0_SysACE_CompactFlash_SysACE_MPA, VEC = [6:0], DIR = OUTPUT
27 PORT fpga_0_SysACE_CompactFlash_SysACE_MPD_pin =
28  fpga_0_SysACE_CompactFlash_SysACE_MPD, VEC = [15:0], DIR = INOUT
29 PORT fpga_0_SysACE_CompactFlash_SysACE_CEN_pin =
30  fpga_0_SysACE_CompactFlash_SysACE_CEN, DIR = OUTPUT
31 PORT fpga_0_SysACE_CompactFlash_SysACE_OEN_pin =
32  fpga_0_SysACE_CompactFlash_SysACE_OEN, DIR = OUTPUT
33 PORT fpga_0_SysACE_CompactFlash_SysACE_WEN_pin =
34  fpga_0_SysACE_CompactFlash_SysACE_WEN, DIR = OUTPUT
35 PORT fpga_0_SysACE_CompactFlash_SysACE_MPIRQ_pin =
36  fpga_0_SysACE_CompactFlash_SysACE_MPIRQ, DIR = INPUT
37 PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clk_pin =
38  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clk, VEC = [0:2], DIR = OUTPUT
39 PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clkn_pin =
```

```
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clkn, VEC = [0:2], DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Addr_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Addr, VEC = [0:12], DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_BankAddr_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_BankAddr, VEC = [0:1], DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CASn_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CASn, DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_RASn_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_RASn, DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_WEn_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_WEn, DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DM_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DM, VEC = [0:7], DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQS_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQS, VEC = [0:7], DIR = INOUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQ_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQ, VEC = [0:63], DIR = INOUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CKE_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CKE, DIR = OUTPUT
PORT fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CSn_pin =
fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CSn, DIR = OUTPUT
PORT fpga_0_DDR_CLK_FB = ddr_feedback_s, DIR = INPUT
PORT fpga_0_DDR_CLK_FB_OUT = ddr_clk_feedback_out_s, DIR = OUTPUT
PORT sys_clk_pin = dcm_clk_s, DIR = INPUT, SIGIS = CLK
PORT sys_rst_pin = sys_rst_s, DIR = INPUT


BEGIN proc_sys_reset
 PARAMETER INSTANCE = reset_block
 PARAMETER HW_VER = 1.00.a
 PARAMETER C_EXT_RESET_HIGH = 0
 PORT Ext_Reset_In = sys_rst_s
 PORT Slowest_sync_clk = sys_clk_s
 PORT Chip_Reset_Req = C405RSTCHIPRESETREQ
 PORT Core_Reset_Req = C405RSTCORERESETREQ
 PORT System_Reset_Req = C405RSTSYSRESETREQ
 PORT Rstc405resetchip = RSTC405RESETCHIP
 PORT Rstc405resetcore = RSTC405RESETCORE
 PORT Rstc405resetsys = RSTC405RESETSYS
 PORT Bus_Struct_Reset = sys_bus_reset
 PORT Dcm_locked = dcm_1_lock
END

BEGIN ppc405
 PARAMETER INSTANCE = ppc405_1
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE JTAGPPC = jtagppc_0_1
END

BEGIN ppc405
 PARAMETER INSTANCE = ppc405_0
 PARAMETER HW_VER = 2.00.c
 BUS_INTERFACE JTAGPPC = jtagppc_0_0
 BUS_INTERFACE IPLB = plb
 BUS_INTERFACE DPLB = plb
 PORT PLBCLK = sys_clk_s
 PORT C405RSTCHIPRESETREQ = C405RSTCHIPRESETREQ
 PORT C405RSTCORERESETREQ = C405RSTCORERESETREQ
 PORT C405RSTSYSRESETREQ = C405RSTSYSRESETREQ
 PORT RSTC405RESETCHIP = RSTC405RESETCHIP
 PORT RSTC405RESETCORE = RSTC405RESETCORE
 PORT RSTC405RESETSYS = RSTC405RESETSYS
 PORT CPMC405CLOCK = sys_clk_s
END

BEGIN bram_block
```

```
106  PARAMETER INSTANCE = plb_bram_if_cntlr_1_bram
107  PARAMETER HW_VER = 1.00.a
108  BUS_INTERFACE PORTA = plb_bram_if_cntlr_1_port
109 END
110
111 BEGIN plb_bram_if_cntlr
112  PARAMETER INSTANCE = plb_bram_if_cntlr_1
113  PARAMETER HW_VER = 1.00.b
114  PARAMETER c_plb_clk_period_ps = 10000
115  PARAMETER c_baseaddr = 0xfffe0000
116  PARAMETER c_highaddr = 0xffffffff
117  BUS_INTERFACE SPLB = plb
118  BUS_INTERFACE PORTA = plb_bram_if_cntlr_1_port
119  PORT PLB_Clk = sys_clk_s
120 END
121
122 BEGIN plb2opb_bridge
123  PARAMETER INSTANCE = plb2opb
124  PARAMETER HW_VER = 1.01.a
125  PARAMETER C_DCR_INTFCE = 0
126  PARAMETER C_NUM_ADDR_RNG = 1
127  PARAMETER C_RNG0_BASEADDR = 0x87ff0000
128  PARAMETER C_RNG0_HIGHADDR = 0x87ff0fff
129  BUS_INTERFACE SPLB = plb
130  BUS_INTERFACE MOPB = opb
131  PORT PLB_Clk = sys_clk_s
132  PORT OPB_Clk = sys_clk_s
133 END
134
135 BEGIN plb_v34
136  PARAMETER INSTANCE = plb
137  PARAMETER HW_VER = 1.02.a
138  PARAMETER C_DCR_INTFCE = 0
139  PARAMETER C_EXT_RESET_HIGH = 1
140  PORT SYS_Rst = sys_bus_reset
141  PORT PLB_Clk = sys_clk_s
142 END
143
144 BEGIN opb_v20
145  PARAMETER INSTANCE = opb
146  PARAMETER HW_VER = 1.10.b
147  PARAMETER C_EXT_RESET_HIGH = 1
148  PORT SYS_Rst = sys_bus_reset
149  PORT OPB_Clk = sys_clk_s
150 END
151
152 BEGIN jtagppc_cntlr
153  PARAMETER INSTANCE = jtagppc_0
154  PARAMETER HW_VER = 2.00.a
155  BUS_INTERFACE JTAGPPC0 = jtagppc_0_0
156  BUS_INTERFACE JTAGPPC1 = jtagppc_0_1
157 END
158
159 BEGIN dcm_module
160  PARAMETER INSTANCE = dcm_1
161  PARAMETER HW_VER = 1.00.a
162  PARAMETER C_CLK0_BUF = TRUE
163  PARAMETER C_CLK270_BUF = TRUE
164  PARAMETER C_CLK90_BUF = TRUE
165  PARAMETER C_CLKIN_PERIOD = 10.000000
166  PARAMETER C_CLK_FEEDBACK = 1X
167  PARAMETER C_PHASE_SHIFT = 60
168  PARAMETER C_CLKOUT_PHASE_SHIFT = FIXED
169  PARAMETER C_EXT_RESET_HIGH = 0
170  PORT CLKIN = ddr_feedback_s
171  PORT CLK90 = ddr_clk_90_s
```

```
172 PORT CLK270 = ddr_clk_90_n_s
173 PORT CLK0 = dcm_1_FB
174 PORT CLKFB = dcm_1_FB
175 PORT RST = dcm_0_lock
176 PORT LOCKED = dcm_1_lock
177 END
178
179 BEGIN dcm_module
180 PARAMETER INSTANCE = dcm_0
181 PARAMETER HW_VER = 1.00.a
182 PARAMETER C_CLK0_BUF = TRUE
183 PARAMETER C_CLK180_BUF = TRUE
184 PARAMETER C_CLK270_BUF = TRUE
185 PARAMETER C_CLK90_BUF = TRUE
186 PARAMETER C_CLKIN_PERIOD = 10.000000
187 PARAMETER C_CLK_FEEDBACK = 1X
188 PARAMETER C_EXT_RESET_HIGH = 1
189 PORT CLKIN = dcm_clk_s
190 PORT CLK0 = sys_clk_s
191 PORT CLK90 = clk_90_s
192 PORT CLK180 = sys_clk_n_s
193 PORT CLK270 = clk_90_n_s
194 PORT CLKFB = sys_clk_s
195 PORT RST = net_gnd
196 PORT LOCKED = dcm_0_lock
197 END
198
199 BEGIN carga_utc
200 PARAMETER INSTANCE = carga_utc_0
201 PARAMETER HW_VER = 1.00.a
202 PARAMETER C_BASEADDR = 0x87ff0400
203 PARAMETER C_HIGHADDR = 0x87ff05ff
204 BUS_INTERFACE SOPB = opb
205 PORT OPB_Clk = sys_clk_s
206 END
207
208 BEGIN opb_sysace
209 PARAMETER INSTANCE = SysACE_CompactFlash
210 PARAMETER HW_VER = 1.00.b
211 PARAMETER C_MEM_WIDTH = 16
212 PARAMETER C_BASEADDR = 0x87ff0100
213 PARAMETER C_HIGHADDR = 0x87ff01ff
214 BUS_INTERFACE SOPB = opb
215 PORT OPB_Clk = sys_clk_s
216 PORT SysACE_CLK = fpga_0_SysACE_CompactFlash_SysACE_CLK
217 PORT SysACE_MPA = fpga_0_SysACE_CompactFlash_SysACE_MPA
218 PORT SysACE_MPD = fpga_0_SysACE_CompactFlash_SysACE_MPD
219 PORT SysACE_CEN = fpga_0_SysACE_CompactFlash_SysACE_CEN
220 PORT SysACE_OEN = fpga_0_SysACE_CompactFlash_SysACE_OEN
221 PORT SysACE_WEN = fpga_0_SysACE_CompactFlash_SysACE_WEN
222 PORT SysACE_MPIRQ = fpga_0_SysACE_CompactFlash_SysACE_MPIRQ
223 END
224
225 BEGIN opb_uartlite
226 PARAMETER INSTANCE = RS232_Uart_1
227 PARAMETER HW_VER = 1.00.b
228 PARAMETER C_BAUDRATE = 9600
229 PARAMETER C_DATA_BITS = 8
230 PARAMETER C_ODD_PARITY = 0
231 PARAMETER C_USE_PARITY = 0
232 PARAMETER C_CLK_FREQ = 100000000
233 PARAMETER C_BASEADDR = 0x87ff0000
234 PARAMETER C_HIGHADDR = 0x87ff00ff
235 BUS_INTERFACE SOPB = opb
236 PORT OPB_Clk = sys_clk_s
237 PORT RX = fpga_0_RS232_Uart_1_RX
```

```
238  PORT TX = fpga_0_RS232_Uart_1_TX
239 END
240
241 BEGIN plb_ddr
242  PARAMETER INSTANCE = DDR_256MB_32MX64_rank1_row13_col10_cl2_5
243  PARAMETER HW_VER = 1.11.a
244  PARAMETER C_PLB_CLK_PERIOD_PS = 10000
245  PARAMETER C_NUM_BANKS_MEM = 1
246  PARAMETER C_NUM_CLK_PAIRS = 4
247  PARAMETER C_INCLUDE_BURST_CACHELN_SUPPORT = 1
248  PARAMETER C_REG_DIMM = 0
249  PARAMETER C_DDR_TMRD = 20000
250  PARAMETER C_DDR_TWR = 20000
251  PARAMETER C_DDR_TRAS = 60000
252  PARAMETER C_DDR_TRC = 90000
253  PARAMETER C_DDR_TRFC = 100000
254  PARAMETER C_DDR_TRCD = 30000
255  PARAMETER C_DDR_TRRD = 20000
256  PARAMETER C_DDR_TRP = 30000
257  PARAMETER C_DDR_TREFC = 70300000
258  PARAMETER C_DDR_AWIDTH = 13
259  PARAMETER C_DDR_COL_AWIDTH = 10
260  PARAMETER C_DDR_BANK_AWIDTH = 2
261  PARAMETER C_DDR_DWIDTH = 64
262  PARAMETER C_MEM0_BASEADDR = 0x00000000
263  PARAMETER C_MEM0_HIGHADDR = 0x0fffffff
264  BUS_INTERFACE SPLB = plb
265  PORT PLB_Clk = sys_clk_s
266  PORT DDR_Addr =
267  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Addr
268  PORT DDR_BankAddr =
269  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_BankAddr
270  PORT DDR_CASn =
271  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CASn
272  PORT DDR_CKE =
273  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CKE
274  PORT DDR_CSn =
275  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_CSn
276  PORT DDR_RASn =
277  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_RASn
278  PORT DDR_WEn =
279  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_WEn
280  PORT DDR_DM =
281  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DM
282  PORT DDR_DQS =
283  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQS
284  PORT DDR_DQ =
285  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_DQ
286  PORT DDR_Clk =
287  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clk & ddr_clk_feedback_out_s
288  PORT DDR_Clkn =
289  fpga_0_DDR_256MB_32MX64_rank1_row13_col10_cl2_5_DDR_Clkn & 0b0
290  PORT Clk90_in = clk_90_s
291  PORT Clk90_in_n = clk_90_n_s
292  PORT PLB_Clk_n = sys_clk_n_s
293  PORT DDR_Clk90_in = ddr_clk_90_s
294  PORT DDR_Clk90_in_n = ddr_clk_90_n_s
295 END
```

Apéndice B

Descripción de la UTC en VHDL

Archivo: `utcCore.vhdl`

```
1 UTC_PROC : process( Bus2IP_Clk ) is
2 begin
3   if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
4     selecRegistro <= to_bitvector(slv_reg0);
5     if selecRegistro(3 downto 0 ) = "0000" then
6       cargadoDominio <= '0';
7       cargadoRango   <= '0';
8       sim <= "0000";
9       done <= "00";
10      add  <= "00";
11      op   <= (others => '0');
12      acc  <= (others => '0');
13      uno  <= '0'; dos  <= '0';
14      bloqueDominio <= (others => '0');
15      bloqueRango   <= (others => '0');
16      bloqueRangoReflejado <= (others =>'1');
17      distancia_minima <= x"003f8040";
18    elsif selecRegistro(3 downto 0 ) = "0001" and
19          cargadoDominio = '0' then
20      bloqueDominio  <= to_bitvector( bus_bloqueDominio  )
21      cargadoDominio <= '1';
22    elsif selecRegistro( 3 downto 0 ) = "0010" and
23          cargadoRango = '0' then
24      bloqueRango   <= to_bitvector( busbloqueRango  )
25      cargadoRango  <= '1';
26    elsif selecRegistro(3 downto 0 ) = "0100" and sim < "1000" then
27      if done = "00" then
28        if sim = "0000" then
29          done <= "01";
30        elsif (sim = "0001" or
31               sim = "0011" or
32               sim = "0101" or
33               sim = "0111") and
34               uno = '0' then
35          bloqueRango <= bloqueRango srl 8;
36          bloqueRangoReflejado <= bloqueRangoReflejado sll 8;
37          bloqueRangoReflejado( 7 downto 0)
38         <= bloqueRango( 7 downto 0);
39          bloqueRangoReflejado( 71 downto 64)
40         <= bloqueRango( 71 downto 64);
41          bloqueRangoReflejado( 135 downto 128)
```

```
42          <= bloqueRango( 135 downto 128);
43          bloqueRangoReflejado( 199 downto 192)
44          <= bloqueRango( 199 downto 192);
45          bloqueRangoReflejado( 263 downto 256)
46          <= bloqueRango( 263 downto 256 );
47          bloqueRangoReflejado( 327 downto 320)
48          <= bloqueRango( 327 downto 320 );
49          bloqueRangoReflejado( 391 downto 384)
50          <= bloqueRango( 391 downto 384 );
51          bloqueRangoReflejado( 455 downto 448)
52          <= bloqueRango( 455 downto 448 );
53          if bloqueRangoReflejado( 511 downto 504) /= "11111111" then
54            if sim = "0001" then
55              done <= "01";
56              bloqueRango <= bloqueRangoReflejado;
57            else
58              uno <= '1';
59              bloqueRango <= bloqueRangoReflejado;
60              bloqueRangoReflejado <= (others =>'1');
61            end if;
62          end if;
63        elsif sim="0010" or
64             (sim="0011" and
65              uno='1')   or
66             (sim="0110" and
67              dos='0')   or
68             (sim="0111" and
69              uno='1'    and
70              dos= 0')then
71          bloqueRango <= bloqueRango srl 64;
72          bloqueRangoReflejado
73          <= bloqueRangoReflejado sll 64;
74          bloqueRangoReflejado( 63 downto 0)
75          <= bloqueRango( 63 downto 0);
76          if bloqueRangoReflejado( 511 downto 504) /= "11111111" then
77            if sim = "0010" or sim = "0011" then
78              done <= "01";
79              bloqueRango <= bloqueRangoReflejado;
80            else
81              dos <= '1';
82              bloqueRango <= bloqueRangoReflejado;
83              bloqueRangoReflejado <= (others =>'1');
84            end if;
85          end if;
86        elsif sim="0100" or
87             (sim="0101" and
88              uno='1')   or
89             (sim="0110" and
90              dos='1')   or
91             (sim="0111" and
92              uno='1'    and
93              dos='1')then
94          bloqueRango <= bloqueRango sll 64;
95          bloqueRangoReflejado   <= bloqueRangoReflejado sll 8;
96          bloqueRangoReflejado( 7 downto 0)
97          <= bloqueRango( 455 downto 448 );
98          bloqueRangoReflejado( 71 downto 64)
99          <= bloqueRango( 463 downto 456 );
100         bloqueRangoReflejado( 135 downto 128)
101         <= bloqueRango( 471 downto 464 );
102         bloqueRangoReflejado( 199 downto 192)
103         <= bloqueRango( 479 downto 472 );
104         bloqueRangoReflejado( 263 downto 256)
105         <= bloqueRango( 487 downto 480 );
106         bloqueRangoReflejado( 327 downto 320)
107         <= bloqueRango( 495 downto 488 );
```

```
108           bloqueRangoReflejado( 391 downto 384)
109           <= bloqueRango( 503 downto 496 );
110           bloqueRangoReflejado( 455 downto 448)
111           <= bloqueRango( 511 downto 504 );
112           if bloqueRangoReflejado( 511 downto 504) /= "11111111" then
113             done <= "01";
114             bloqueRango <= bloqueRangoReflejado;
115             bloqueRangoReflejado <= (others =>'1');
116           end if;
117         end if;
118       elsif done = "01" then
119         if add = "00" then
120           bloqueDominio  <= bloqueDominio sll 8;
121           bloqueRango    <= bloqueRango sll 8;
122           op( 7 downto 0 ) <=
123           sumador(
124           bloqueDominio(511 downto 504),
125           bloqueRango(511 downto 504)
126           );
127           add <= "01";
128         elsif add = "01" then
129           acc <= add32( acc, op );
130           add <= "10";
131         elsif add = "10" then
132           if bloqueRango( 511 downto 504) = "00000000" and
133              bloqueDominio( 511 downto 504) = "00000000" then
134             done <= "10";
135           else
136             add <= "00";
137           end if;
138         end if;
139       elsif done = "10" then
140         if acc < distancia_minima then
141           distancia_minima <= acc;
142           sim_final   <= sim;
143         end if;
144         if sim = "0000" or sim = "0001" then
145           salidaRegistro2 <= salidaRegistro2 sll 16;
146           salidaRegistro2(15 downto 0) <= acc(15 downto 0);
147         elsif sim = "0010" or sim = "0011" then
148           salidaRegistro3 <= salidaRegistro3 sll 16;
149           salidaRegistro3(15 downto 0) <= acc(15 downto 0);
150         elsif sim = "0100" or sim = "0101" then
151           salidaRegistro4 <= salidaRegistro4 sll 16;
152           salidaRegistro4(15 downto 0) <= acc(15 downto 0);
153         elsif sim = "0110" or sim = "0111" then
154           salidaRegistro5 <= salidaRegistro5 sll 16;
155           salidaRegistro5(15 downto 0) <= acc(15 downto 0);
156         end if;
157         if sim = "0111" then
158           salidaRegistro6 <= distancia_minima;
159         end if;
160         sim <= inc_bv4(sim,"0001",'0');
161         done <= "00";
162         add  <= "00";
163         op   <= (others => '0');
164         acc  <= (others => '0');
165         uno  <= '0';
166         dos  <= '0';
167         bloqueDominio <= bus_bloqueDominio;
168         bloqueRango   <= bus_bloqueRango;
169         bloqueRangoReflejado <= (others =>'1');
170       end if;
171     end if;
172   end if;
173 end process UTC_PROC;
```

Bibliografía

[1] Wayne E. Carlson. A survey of computer graphics image encoding and storage formats. *SIGGRAPH Comput. Graph.*, 25(2):67–75, 1991.

[2] Michael F. Barnsley and Lyman P. Hurd. *Fractal Image Compression.* A. K. Peters, Ltd., Natick, MA, USA, 1993.

[3] D. Saupe, R. Hamzaoui, and H. Hartenstein. Fractal image compression – an introductory overview. In D. Saupe and J. Hart, editors, *Fractal Models for Image Synthesis, Compression and Analysis*, New Orleans, Louisiana, 1996.

[4] Erjun Zhao and Dan Liu. Fractal image compression methods: A review. In *ICITA '05: Proceedings of the Third International Conference on Information Technology and Applications (ICITA'05) Volume 2*, pages 756–759, Washington, DC, USA, 2005. IEEE Computer Society.

[5] I. Kopilovic, D. Saupe, and R. Hamzaoui. Progressive fractal coding. In *Proc. IEEE ICIP-01*, pages 86–89, Thessaloniki, October 2001.

[6] Kevin P. Acken, Mary Jane Irwin, and Robert M. Owens. A parallel ASIC architecture for efficient fractal image coding. *J. VLSI Signal Process. Syst.*, 19(2):97–113, 1998.

[7] Alejandro Martínez Ramírez. *Diseño de un Procesador Digital de Señales para Aplicaciones Específicas en Comunicaciones.* Tesis doctoral, Instituto Nacional de Astrofísica Óptica y Electrónica, 2005.

[8] A. Martínes-Ramírez A. Díaz-Sánchez M. Linares Aranda J. Vega Pineda. An architecture for fractal image compression using quad-tree multiresolution. *ISCAS, IEEE*, 2:897–900, May 2004.

[9] Dietmar Saupe and Matthias Ruhl. Evolutionary fractal image compression. In *Proceedings ICIP-96 (IEEE International Conference on Image Processing)*, volume I, pages 129–132, Lausanne, Switzerland, 1996.

[10] Brendt Wohlberg and Gerhard de Jager. A review of the fractal image coding literature. *IEEE Transactions on Image Processing*, 8(12):1716–1729, December 1999.

[11] Dietmar Saupe and Raouf Hamzaoui. A review of the fractal image compression literature. *SIGGRAPH Comput. Graph.*, 28(4):268–276, 1994.

[12] Ghim-Hwee Ong, Chorng-Meng Chew, and Yi Cao. A simple partitioning approach to fractal image compression. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 301–305, New York, NY, USA, 2001. ACM Press.

[13] Barry S. Fagin and Pichet Chintrakulchai. A reprogrammable processor for fractal image compression. In *FPL '94: Proceedings of the 4th International Workshop on Field-Programmable Logic and Applications*, pages 129–131, London, UK, 1994. Springer-Verlag.

[14] Hidehisa Nagano, Akihiro Matsuura, and Akira Nagoya. An efficient implementation method of fractal image compression on dynamically reconfigurable architecture. In *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, pages 670–678, London, UK, 1999. Springer-Verlag.

[15] Huaqing Wang, Meiqing Wang, Tom Hintz, Xiangjian He, and Qiang Wu. Fractal image compression on a pseudo spiral architecture. In *CRPIT '38: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 201–207, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[16] Jurgen Becker, Thilo Piontek, and Manfred Glesner. Dream: A dynamically reconfigurable architecture for future mobile communications applications. In *FPL '00: Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications*, pages 312–321, London, UK, 2000. Springer-Verlag.

[17] A.E.Jacquin. Image coding based on a fractal theory of iterated contractive image transformations. *IEEE Transactions*, 1(1):18–30, 1992.

[18] A. E. Jacquin. Fractal image coding: a review. *Proceedings of the IEEE*, 81:1451–1465, 1993.

[19] Michael F. Barnsley. *Fractals Everywhere*. Morgan Kaufmann Pub; 2nd edition, 340 Pine Street, Sn Francisco CA, June 1993.

[20] Ning Lu. *Fractal Imaging*. Academic Press, 525 B Street, Suite 1900, Sn Diego, CA, USA, 1997.

[21] Fisher Yuval. *Fractal Image Compression: Theory and Application to Digital Images*. Springer Verlag, New York, 1995.

[22] John Kominek. Advances in fractal compression for multimedia applications. Technical Report. University of Waterloo, Canada, 1995.

[23] Wayne H. Wolf. Hardware-software co-design of embedded systems. *Proceedings of the IEEE*, 82(7):967–989, July, 1994.

[24] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[25] S. Prakash and A. Parker. Sos: Synthesis of application-specific heterogeneous multiprocessor systems. *Journal of Parallel and Distrib. Comp.*, 16:338–351, 1992.

[26] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.

[27] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[28] David Becker, Raj K. Singh, and Stephen G. Tell. An engineering environment for hardware/software co-simulation. In *Design Automation Conference*, pages 129–134, 1992.

[29] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):155–182, 1994.

[30] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems.*, 4(3):257–279, 1999.

[31] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[32] Ti-Yen Yen and Wayne Wolf. Performance estimation for real-time distributed embedded systems. In *ICCD '95: Proceedings of the 1995 International Conference on Computer Design*, pages 64–71, Washington, DC, USA, 1995. IEEE Computer Society.

[33] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: Formal models, validation, and synthesis. *Proc. of the IEEE*, 85(3):366–390, year 1997.

[34] Massimiliano Chiodo, Paolo Guisto, Attila Jurecska, Harry C. Hsieh, Alberto Sangiovanni-Vincentelli, and Luciano Lavagno. Hardware-software codesign of embedded systems. pages 313–323, 2002.

[35] Asawaree Kalavade and Edward A. Lee. The extended partitioning problem: hardware/software mapping, scheduling, and implementation-bin selection. pages 293–312, 2002.

[36] Wayne H. Wolf. An architectural co-synthesis algorithm for distributed, embedded computing systems. pages 338–349, 2002.

[37] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano. Power estimation of embedded systems: A hardware/software codesign approach. *IEEE Trans*, 6:634–639, 1998.

[38] J. Daveau, G. Marchioro, T. Ben-Ismail, and A. Jerraya. Protocol selection and interface generation for hw-sw codesign, 1997.

[39] Giovanni De Micheli. Computer-aided hardware-software codesign. *IEEE Micro*, 14(4):10–16, 1994.

[40] Dan C. R. Jensen, Jan Madsen, and Steen Pedersen. The importance of interfaces: A HW /SW codesign case study. *Codes CASHE97*, pages 87–91, 1997.

[41] Bruce Schneier. *Applied Cryptography:Protocols, Algorithms and Source Code in C*. John Wiley & Sons, 605 Third Avenue, New York, NY 10158-0012, 1996.

[42] Paul C.van Oorschot Alfred J. Menezes and Scott A. Vanstone. *Hadbook of Applied Cryptography*. CRC Press, 1996.

[43] N. Weaver and E. Caspi. IDEA as a benchmark for reconfigurable computing, 1996.

[44] Reto Zimmermann. Efficient VLSI implementation of modulo $(2^n \pm 1)$ addition and multiplication. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 158–167, Los Alamitos, CA, 1999. IEEE Computer Society Press.