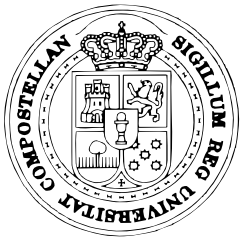
UNIVERSIDAD DE SANTIAGO DE COMPOSTELA

Departamento de Electrónica y Computación



ALGORITMOS Y ARQUITECTURAS PARA LA CODIFICACIÓN ARITMÉTICA: EXPLOTACIÓN DE LA LOCALIDAD UTILIZANDO MEMORIAS CACHE

Roberto Rodríguez Osorio

Julio, 1999

Dr. **Javier Díaz Bruguera**, Catedrático del Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela.

CERTIFICA:

Que la memoria titulada "**ALGORITMOS Y ARQUITECTURAS PARA LA CODIFICACIÓN ARITMÉTICA: EXPLOTACIÓN DE LA LOCALIDAD UTILIZANDO MEMORIAS CACHE**", ha sido realizada por D. **Roberto Rodríguez Osorio** bajo mi dirección en el Departamento de Electrónica y Computación de la Universidad de Santiago de Compostela y concluye la Tesis que presenta para optar al grado de Doctor en Ciencias Físicas.

Santiago, Julio de 1999

Fdo. **Javier Díaz Bruguera**
Director de la tesis

Fdo. Dr. **Senén Barro Ameneiro**,
Director del Departamento de Electrónica y Computación.

A mis padres
y hermanos

Agradecimientos

Después de haber invertido tantos años en la realización de este trabajo es justo hacer memoria y recordar a todas aquellas personas que me han ayudado de una u otra forma en el desarrollo del mismo o en hacer soportable el paso del tiempo.

En primer lugar al director de esta tesis, Javier Díaz Bruguera, en quien he encontrado siempre apoyo y asesoramiento. A él quiero agradecer el tiempo y esfuerzo dedicados así como la confianza depositada. Valoro especialmente la independencia de que he disfrutado, apoyándome siempre en las decisiones que tomaba y aportando ideas en todo momento.

A mis compañeros del Grupo de Arquitectura de Computadores, tanto por la ayuda prestada como por la calidad humana que atesoran. Nunca me ha faltado a quien acudir para resolver un problema o cargar con alguno nuevo; compartir una alegría o una inquietud; gastar una broma o padecerla. Gracias especialmente a mis compañeros de despacho durante todo este tiempo: Antonio, David, Dora, Inma, José, Juan, Margarita, Merchy y Patricia.

De modo especial a Montse, tanto por su contribución en la primera parte del trabajo como por toda la ayuda que he recibido en forma de consejos o peleando con el software de desarrollo.

A todos los miembros del Departamento de Electrónica y Computación por su apoyo y amistad.

A mis amigos, de quienes no sabría que características destacar: la fidelidad de algunos, la alegría de otros, la increíble capacidad para sorprenderme siempre o confirmar cuan bien los conozco.

Finalmente, agradecer a la CICYT por el soporte económico facilitado por medio del proyecto TIC95-1125-C03-02 y especialmente a la Xunta de Galicia por la beca predoctoral de la que he disfrutado durante buena parte de la realización de este trabajo.

Julio de 1999

Snap, snap,
grin, grin,
wing, wing,
nudge, nudge,
say no more!

Monty Python

Índice General

Índice de Figuras

Índice de Tablas

Introducción

El advenimiento de la era digital ha supuesto la aparición de nuevas formas de tratar la información que aportan una flexibilidad y variedad de posibilidades impensables con la tecnología analógica. Los textos se ha convertido en secuencias de códigos, las imágenes en grupos de pixels y la voz en patrones de frecuencias. Las ventajas de la codificación digital son fáciles de encontrar en la vida corriente. Así la televisión digital es menos sensible a las interferencias que la analógica, la búsqueda de información en grandes bases de datos es una operación que se resuelve con facilidad, y muchos minutos de sonido e imágenes pueden ser almacenados en un disco de superficie muy pequeña.

Tal es la cantidad de datos almacenados en la actualidad que uno de las principales líneas de investigación en el campo de la codificación es la compresión. Ésta consiste en eliminar de una muestra de datos la parte que no aporta información, que es redundante y cuya pérdida no disminuye el conocimiento que extraemos de los datos. La mayor parte de las bases de datos utilizan algún método de compresión, y también lo hacen muchas fuentes de audio, e imágenes. Incluso dentro de los computadores de propósito general se comprime la información, para aumentar la capacidad de los dispositivos de almacenamiento masivo e incluso, como se ha propuesto recientemente, para aumentar la capacidad de la memoria RAM.

Estas técnicas son particularmente útiles en uno de los campos que mayor desarrollo ha experimentado en los últimos años, las aplicaciones de imagen y video digital. La implantación de la televisión y el video digitales son cada vez mayores, y se abre un nuevo mercado de consumo de esta tecnología en forma de cámaras fotográficas y de video. La compresión de imágenes tiene sus propias técnicas, que explotan la naturaleza de las imágenes, pero el último eslabón del proceso es siempre un codificador entrópico.

Este trabajo versa sobre una de las muchas técnicas de compresión entrópica existentes, la codificación aritmética adaptativa para alfabetos multinivel. Este método constituye el estado del arte en compresión de información, pero su desarrollo no ha ído paralelo con su eficacia debido a la complejidad de su implementación. El

modelo probabilístico en que se basa el algoritmo necesita gran cantidad de memoria y su mantenimiento es costoso. Por otro lado, la descodificación es una operación complicada basada en búsquedas y comparaciones dentro del modelo. En esta memoria presentaremos los algoritmos de compresión y descompresión, mostrando sus puntos débiles, y llegando a nuevos algoritmos más sencillos y eficicentes orientados a implementaciónes en hardware. Almacenamos el modelo utilizando una memoria con dos niveles de jerarquía. El nivel más alto es una cache que nos permite reducir de forma considerable la complejidad y alcanzar mejores relaciones de compresión que cualquier arquitectura para codificación aritmética multinivel que conozcamos.

El trabajo que presentamos en esta memoria se enmarca dentro de una de las líneas de investigación [1] del grupo de Arquitectura de Computadores del Departamento de Electrónica y Computación de la Universidad de Santiago, sobre diseño e implementación de arquitecturas VLSI de aplicación específica en el campo del procesamiento digital de imágenes.

La memoria está estructurada en cinco capítulos. En el primer capítulo se presentan los conceptos básicos de compresión de datos, definiciones y clasificación de métodos. Se comentan diversos algoritmos de compresión que presentan interés por estar implementados en diversas aplicaciones. La codificación aritmética merece especial atención. Se presenta de forma general y se diferencia a continuación entre las dos variantes existentes: codificación aritmética binaria y multinivel, centrándonos en ésta última por ser el tema de este trabajo. Describiremos arquitecturas y algoritmos publicados en la literatura presentando sus principales ventajas e inconvenientes. Finalmente haremos un examen exhaustivo de una arquitectura para codificación y descodificación relativamente rápida y eficiente que tomaremos como punto de partida para las nuevas arquitecturas que presentaremos.

El segundo capítulo está dedicado a la idea central de este trabajo: la introducción de una memoria cache para simplificar la gestión del modelo probabilístico del codificador. Mostraremos la génesis de la idea partiendo de un análisis de los aspectos más costosos de una arquitectura convencional, introduciendo mejoras que nos llevarán a implementaciones sencillas y rápidas. La memoria cache permite reducir el número de datos involucrados en la operación del modelo, facilita la actualización del mismo y las tareas de descodificación. Como resultado llegaremos a dos arquitecturas básicas con cache. La primera de ellas tiene aplicación en compresión de datos en general y mejora en compresión de imágenes al algoritmo original utilizando un modelo con una jerarquía en dos niveles: memoria RAM y cache. La segunda es adecuada para datos cuyo histograma tiene un perfil conocido, y permite prescindir de la memoria RAM con lo que el modelo se reduce a la cache.

La forma más completa de arquitectura con memoria cache se estudia y desa-

[1] Investigación financiada a través del proyecto concedido por la CICYT, TIC95-1125-C03-02 y por la Xunta de Galicia a través de una beca predoctoral.

rrolla en el tercer capítulo. Describimos los elementos involucrados desde el modelo al cálculo de la iteración y la salida de datos. El modelo se describe en detalle: la estructura de la cache, el cálculo de las probabilidades acumulativas en el codificador y el descodificador, la actualización y corrección de las probabilidades y los reemplazos. La iteración es implementada con un coste inferior al de otras implementaciones. La descodificación experimenta una notable simplificación con la introducción de un divisor previo a las comparaciónes que abarata y acelera las operaciones. Se estudia también el impacto de introducir segmentación. Las ventajas obtenidas en términos de velocidad y área son evaluadas al final del capítulo.

La arquitectura más interesante de las consideradas es aquella que prescinde de la memoria principal. En el capítulo cuarto nos centraremos en el desarrollo de esta arquitectura, a partir de la cual obtendremos otras con características mejoradas. Tras evaluar las ventajas de la arquitectura sin RAM, afrontaremos el diseño de nuevas arquitecturas enfocadas hacia la mejora en la relación de compresión. Llegaremos de esta forma a una arquitectura sin multiplicaciones que aumenta el rendimiento de la cache sin aumentar su tamaño y que permite, al igual que había sucedido con la arquitectura con RAM, mejorar los resultados del codificador convencional. Finalmente, proponemos también una arquitectura no adaptativa.

El quinto y último capítulo está dedicado a estudiar la aplicación de las nuevas arquitecturas a métodos conocidos de compresión de imágenes. Comprimimos imágenes con y sin pérdidas, comparándolas con métodos como GIF, JPEG y los nuevos compresores como CALIC o LOCO-I. Se realiza también una comparación a nivel de arquitectura con dos codificadores descritos en la literatura, concluyendo que nuestra implementación es más conveniente en distintos apartados.

Finalmente hacemos un balance presentando nuestras arquitecturas como una solución poco costosa, rápida y fácilmente adaptable a varios problemas de compresión de información, que nos permite abordar la compresión de distintos datos con escaso tiempo de desarrollo.

Capítulo 1

Compresión de datos

En este primer capítulo introductorio presentamos el problema de la compresión de datos, definiéndolo y describiendo diversas técnicas en uso, para a continuación fijar nuestra atención en el objeto del presente trabajo, la codificación aritmética de imágenes multinivel. Haremos una descripción detallada de la metodología de compresión, mostrando las distintas alternativas de forma comparativa. Finalmente describiremos en detalle una arquitectura que tomaremos como base para mostrar las debilidades de una implementación convencional y desarrollar las mejoras que se proponen en los siguientes capítulos.

1.1 Compresión

Comenzamos esta sección dedicada a presentar los conceptos básicos de la compresión de datos. Definiremos la compresión como un tipo de codificación, incluyendo una taxonomía de las distintas categorías.

Llamamos compresión de datos a la operación consistente en extraer la información contenida en estos datos eliminando la parte redundante. Por información entendemos la aportación de conocimiento que no obraba ya en nuestro poder. Los datos que obtenemos y procesamos contienen habitualmente gran cantidad de redundancias que no aportan nueva información, y que por tanto pueden ser eliminadas sin que por ello disminuya nuestro conocimiento.

La teoría de la información estudia la codificación eficiente para el almacenamiento y transmisión. Cuando medimos la eficiencia en términos de la cantidad de código resultante, nos encontramos ante la compresión. La compresión de datos permite reducir las necesidades de almacenamiento de información y aumentar la velocidad de transmisión de la misma, con los beneficios que ello conlleva. La dificultad de interpretar y manipular datos comprimidos hace necesaria la operación

inversa: la descompresión.

La compresión se basa en la existencia de datos más frecuentes que otros, y que por tanto deben ser codificados utilizando códigos más pequeños, que ocupen pocos bits. Un símil útil para definir este concepto es que los datos más frecuentes son los más esperados, y por tanto el hecho de que ocurran no aporta demasiada información. Por el contrario, los datos con menor probabilidad de ocurrir sí aportan gran cantidad de información.

Otra forma de acercarnos al concepto de la compresión es a través de la **entropía** o desorden de una muestra de datos. La redundancia contenida en los datos es aquella información que ya nos es conocida y que podemos prever en función del conocimiento que obra en nuestro poder. Podemos entonces hablar de un orden interno dentro de los datos, de un conjunto de reglas que a partir de la información nos permite conocer también la parte redundante. En contraposición con esto, la información contenida en los datos carece por completo de orden, es totalmente impredecible.

Shanon demostró que para una muestra de datos formada por símbolos cuya probabilidad conocemos, la codificación más eficiente es aquella que asigna a cada símbolo de probabilidad p un código de longitud $-log_2\ p$. Esta afirmación, si bien es cierta y muy sencilla, no resuelve el problema de la compresión, sino que en su mismo enunciado deja la puerta abierta a uno de los aspectos más importantes de la compresión: la estimación de las propiedades estadísticas de los datos. La elección del modelo estadístico adecuado define en gran medida la eficiencia de la compresión.

1.1.1 Definiciones

Llamamos **Código** a la asignación de **mensajes-fuente** (palabras de un alfabeto origen) en **palabras-código** del alfabeto destino.

Por mensajes-fuente entendemos los elementos que utilizamos como unidades básicas de nuestros mensajes, que pueden estar compuestas por uno o más símbolos del alfabeto origen. Las posibilidades son muchas y definen varios tipos de códigos.

Según sea la naturaleza de las palabras de los alfabetos origen y destino hablaremos de códigos **bloque-bloque**, **bloque-variable**, **variable-bloque** y **variable-variable**. El primer caso es el más sencillo y habitual. Las palabras de ambos alfabetos tienen longitud fija (bloques), lo que facilita las operaciones con ellas: búsqueda, inserción, comparación, etc. Un caso muy conocido que podemos tomar como ejemplo es la codificación ASCII, que hace corresponder a cada letra del alfabeto romano un código de 7 bits.

Los restantes casos son más interesantes desde la perspectiva de nuestro trabajo ya que en ellos se puede conseguir comprimir. Un ejemplo de codificación bloque-variable que estudiaremos más adelante en mayor detalle es la codificación Huffman

aplicada, por ejemplo, a la codificación de texto, obteniendo palabras-código muy cortas para las letras más frecuentes, y más largas para las menos frecuentes.

Los dos últimos casos son los más flexibles y permiten una mejor aproximación a las características estadísticas de los datos. La redundancia no sólo existe a nivel de palabras del alfabeto, sino que la disposición de unas con respecto a otras en los mensajes también es en gran parte redundante. Como ejemplo podemos tomar el lenguaje humano, en el cual no sólo existen letras más abundantes que otras, sino que ciertas combinaciones se producen con mayor frecuencia (sílabas y palabras). Por otra parte, la complejidad de su implementación es mayor que en los casos anteriores [LST92]. La codificación variable-bloque añade robustez a la codificación [LBS98], lo que puede ser un valor añadido.

La codificación variable-variable abre la puerta a otra cuestión relacionada con la construcción de los mensajes-fuente. La forma en que las palabras del alfabeto origen se combinan para formar mensajes puede estar predefinida desde antes de comenzar la codificación. Hablaremos entonces de esquemas de codificación **palabra-definida**. Esto implica tener un cierto conocimiento sobre la naturaleza de los datos con el fin de que la elección de las combinaciones posibles permita una eficiencia cercana a la óptima.

El caso contrario se denomina de **análisis-libre** y es poco habitual entre los esquemas de codificación. Es el algoritmo el que construye los mensajes durante la codificación, basándose en la historia previa para decidir como fragmentar la secuencia de símbolos para formar los mensajes. Este esquema es en principio más complicado que el palabra-definida si bien es más flexible, demostrando su utilidad en algoritmos que veremos en las siguientes secciones (sección 1.2.3).

Un código se dice **distinto** si cada palabra-código es distinguible de cualquier otra. Esto es, no existen dos mensajes distintos a los que corresponda la misma palabra-código. De un código distinto se dice que es **descodificable de forma única** si las palabras-código son identificables dentro de la secuencia codificada.

Ambas propiedades son imprescindibles para los códigos que nos ocupan, de lo contrario no estaría garantizada la reversibilidad del proceso de codificación y este no tendría utilidad. En adelante sólo nos ocuparemos de este tipo de códigos.

Dado un código descodificable de forma única, diremos que es un **código-prefijo** (prefix-code o prefix-free code) si ninguna palabra-código es un prefijo de otra. Todos los códigos con palabras-código de tipo bloque tienen esta propiedad, como es obvio, pero esto no tiene por que ser cierto cuando son del tipo variable. Un código cuyas palabras fuesen los siguientes 1, 100000, 00 es descodificable de forma única pero la primera palabra-código es prefijo de la segunda.

El ejemplo anterior es descodificable de forma única ya que si el símbolo 1 va seguido de un número impar de ceros entonces sabemos que se trata de la segunda palabra-código, mientras que si este número es par se trata del primero seguido del

tercero un cierto número de veces. En este caso para que la descodificación sea posible no es suficiente con haber procesado toda la palabra-código, sino que puede ser necesario inspeccionar más símbolos. Esta posibilidad complica de forma notable la descodificación y por tanto no es deseable.

De un código-prefijo diremos que es **instantáneamente descodificable** si las palabras del mensaje codificado pueden ser identificadas sin necesidad de inspeccionar el resto del mensaje.

Otra propiedad deseable en los códigos-prefijo es que los prefijos sean **mínimos**. Esta propiedad hace que las palabras-código no sean más largos de lo debido y su enunciado es el siguiente: dado un prefijo x de alguna palabra-código, entonces xy será una palabra-código o bien un prefijo de otra palabra-código para cualquier y del alfabeto destino. De lo contrario, de existir posibles combinaciones sin utilizar esto implicaría la existencia de redundancia en la construcción del alfabeto destino.

1.1.2 Clasificación de los métodos

Una de las características más importantes de los métodos de compresión es su evolución en el tiempo. Aquellos métodos en los cuales la traslación de mensajes-fuente a palabras-código no depende del tiempo, sino que está prefijada desde antes de comenzar la codificación, diremos que son **estáticos**. Todos los ejemplos de codificación que hemos visto hasta ahora con estáticos y los hemos escogido así por su sencillez y facilidad de comprensión. Esta sencillez se refleja en la implementación de algoritmos basados en estos métodos. El ejemplo clásico de método estático es la codificación de Huffman. Este es un método que, basándose en las probabilidades que existen de que ocurra cada mensaje-fuente, construye un conjunto de palabras-código cuya longitud depende de la probabilidad de cada mensaje. A mayor probabilidad menor será la longitud. La probabilidad de cada mensaje se estima antes de comenzar la codificación, y por tanto no cambia durante el transcurso de la misma. Esta estimación puede ser fija para todas las distintas muestras, o calcularse para cada una mediante una pasada previa sobre los datos. En este último caso hablaremos de métodos semi-adaptativos.

En contraposición con los anteriores tenemos los métodos **dinámicos**. Decimos que un método es dinámico (o **adaptativo**) cuando la asignación de mensajes-fuente a palabras-código cambia a lo largo del tiempo. La razón de ser de estos métodos es evidente si se considera la naturaleza de ciertos tipos de datos. Consideremos por ejemplo un algoritmo de compresión implementado para comprimir las temperaturas máximas y mínimas diarias registradas en una estación meteorológica. Conociendo el clima de la región donde se va a implantar es posible utilizar un método estático con cierta eficiencia. Sin embargo este método no contemplaría los cambios estacionales, olas de frío, etc. En su lugar consideremos un método dinámico, como puede ser la

versión dinámica de la codificación de Huffman. Este método funciona igual que su equivalente estático con la salvedad de que las probabilidades de cada mensaje-fuente se evalúan durante la codificación y las asignaciones mensaje-código se redistribuyen cada cierto tiempo. Dado un algoritmo de estas características las temperaturas elevadas tendrían asignadas palabras-código cortas en verano y largos en invierno y viceversa. Esta mejor adecuación a las propiedades estadísticas de los datos puede mejorar eficiencia de la compresión en gran medida dependiendo del tipo de datos. Además del ejemplo que hemos propuesto existen muchos otros de gran interés, el más importante de los cuales es la compresión de imágenes, a la que está dedicada parte de esta memoria.

A la mayor eficiencia de los métodos dinámicos se une el hecho de que la codificación se efectúa en una sola pasada sobre los datos, sin un cálculo previo de las probabilidades de toda la muestra.

Por último, existen métodos **híbridos** entre los estáticos y los dinámicos. Partiendo de la premisa de que codificador y descodificador deben tener la misma información sobre el modelo probabilístico de los datos es posible desarrollar esquemas con la sencillez de los estáticos y un cierto grado de adaptabilidad. Codificador y descodificador pueden disponer de varios modelos estáticos almacenados en tablas y escoger el más apropiado en cada momento de la transmisión. El modelo utilizado por el codificador puede ser comunicado al descodificador como información adicional o puede ser inferido por ambos sobre la base de los datos ya procesados.

1.1.3 Eficiencia de un codificador

A la hora de hacer consideraciones sobre la eficiencia de un codificador son varios los factores que han de tenerse en cuenta. Dependiendo de las características de las aplicaciones convendrá optimizar unos aspectos más que otros, dando lugar así a que existan distintos algoritmos cuya eficiencia se estimará atendiendo a conceptos distintos.

Los dos factores más importantes a la hora de evaluar la eficiencia de un codificador son su complejidad y la cantidad de compresión que permiten obtener. A estos podemos añadir otros que tienen interés aplicaciones más específicas, como puede ser la robustez frente al ruido.

La complejidad permite, habitualmente, mejorar la cantidad de compresión gracias a un tratamiento más concienzudo de los datos. Teniendo en cuenta que un compresor utiliza un cierto conocimiento sobre la naturaleza de los datos para estimar la probabilidad que tiene cada mensaje de suceder, será posible incrementar la compresión si ese conocimiento se obtiene por medios más sofisticados. Así pues, el análisis de la historia y la historia reciente del codificador es una herramienta poderosa, tanto para predecir el futuro inmediato como para prever tendencias.

Existe además una complejidad intrínseca al algoritmo aún en su implementación más básica y que se traduce en la mayor popularidad de unos algoritmos respecto a los otros. A su vez la complejidad es relativa dependiendo del tipo de implementación. En una implementación software la complejidad se mide en tiempo de computación y en memoria consumida, pero en una computadora moderna este último aspecto tiene escasa relevancia. En cambio en una implementación hardware la memoria puede ser un parámetro crucial como también lo es el tipo de operaciones a realizar. Una computadora de propósito general es capaz de realizar casi cualquier tipo de operación aritmética sin más coste que el tiempo empleado. Además, se pueden implementar algoritmos con un diagrama de flujo muy irregular. En cambio, en un sistema hardware de aplicación específica el coste de implementar operaciones complejas es muy alto ya que en general este hardware no será utilizado por ninguna otra aplicación. A favor de las implementaciones hardware está el hecho de que las operaciones sencillas se ejecutan con rapidez, es relativamente sencillo introducir paralelismo y se pueden implementar operaciones no existentes en computadoras convencionales.

La medida de la cantidad de compresión que un algoritmo puede conseguir se puede expresar de varias formas. Inicialmente se propuso como medida la cantidad de redundancia existente en el código tras la compresión [SW49]. La redundancia se estima de acuerdo con la compresión que teóricamente se debería alcanzar conocidas las probabilidades que tienen de ocurrir los mensajes-fuente. Por tanto se suponen conocidas las probabilidades de los símbolos y que estas son continuas a través de toda la muestra. No se considera el caso en el que la estadística de los datos sufra fluctuaciones a lo largo de la muestra, y por tanto no es una medida adecuada para evaluar métodos adaptativos.

La **longitud promedio** de los mensajes fue introducida por Huffman [Huf52] y se define como $\sum p(a_i)l_i$, esto es, la suma de la longitud del código asociado a cada mensaje fuente multiplicado por la probabilidad de ese mensaje. En realidad es la longitud promedio del código resultante de la compresión.

Finalmente la medida más utilizada es la **relación de compresión**, que será la que con más frecuencia utilizaremos en esta memoria y que definiremos como el cociente entre la longitud promedio del código y la longitud promedio del mensaje. A menudo, y por motivos de comodidad, la expresaremos en tanto por ciento. Así, si al comprimir un texto con caracteres ASCII de 7 bits, obtenemos un palabras-código con longitud promedio de 5 bits, la relación de compresión será de 0.7143 ó del 71.43%.

1.1.4 Redundancia local

En ocasiones la redundancia en los datos a comprimir ocurre a nivel local de forma que no es detectada directamente por el algoritmo de compresión ni aun cuando éste sea adaptativo. Esto ocurre cuando el valor de un dato de la muestra está fuertemente condicionado por otros datos de su vecindad, lo cual ocurre con frecuencia. Pensemos en dos casos muy comunes: la compresión de imágenes y de texto. El valor de luminosidad (o cualquier otro parámetro de color) de un punto de una imagen (pixel) es muy similar al de los pixeles que lo rodean a excepción de los contornos. En un texto, dado el comienzo de una palabra, puede ser muy fácil para un ser humano completarla. Existen además sílabas características de cada idioma y reglas de combinaciones entre vocales y consonantes. De esta manera es posible construir un esquema que condicione la probabilidad de cada símbolo a la historia más reciente.

La forma inmediata de hacer esto es introducir **codificación de orden superior**. Esta se puede implementar de dos formas [LST92]. La primera consiste en construir mensajes más largos combinando los mensajes originales. Por ejemplo, de un esquema en el que los mensajes serían las letras del alfabeto pasaríamos a otro con mensajes aa, ab, ac, ..., zz. La segunda forma consiste en cambiar el esquema de codificación dependiendo de las entradas anteriores. Esta forma es más habitual ya que tiene una implementación más sencilla. Por ejemplo, podría existir un esquema de codificación para letras precedidas por una vocal y otro para letras precedidas por consonante, etc.

Sin embargo, se suelen utilizar otros métodos de implementación más sencilla. En el caso de imágenes resulta más fácil codificar la diferencia entre un pixel y el anterior, o predecir un valor posible en función de un cierto número de vecinos y codificar el error con respecto a este valor predicho. Esto se denomina **codificación diferencial**. La eficiencia suele ir asociada a la complejidad. Así, por ejemplo, algunos métodos de compresión de imágenes tienen en cuenta un número considerable de vecinos. En compresión de vídeo la redundancia existe también a nivel temporal, entre un cuadro de imagen y los demás. Se puede codificar la diferencia con el cuadro anterior, aunque en esquemas más complejos se puede hacer una predicción bidireccional tomando como referencia cuadros anteriores y posteriores a uno dado.

Otros casos de redundancia local aparecen en al aplicar transformaciones a los datos. Al aplicar la DCT a una imagen suelen aparecer secuencias largas de coeficientes con valor nulo que se comprimen fácilmente indicando un código especial y la longitud de la *carrera* (run-length). En la transformada *wavelet* existe redundancia entre coeficientes de la misma banda y distinta jerarquía [Sha93].

Otro método, muy útil en compresión de textos, y que se ha extendido a otros ámbitos es el de los diccionarios. Consiste en crear (de forma estática o dinámica)

conjuntos de mensajes que se repiten con frecuencia y que llamamos diccionarios (o codebooks). Durante la codificación se sustituyen los mensajes por referencias a los elementos del diccionario. El método de compresión de texto basado en diccionarios más conocido en el LZW [ZL77]. Para compresión de imágenes existen muchos métodos para obtener los diccionarios más adecuados [Ger79, MF90].

1.2 Métodos de compresión

Los primeros estudios sobre compresión de información dieron lugar a los primeros métodos para conseguir esta tarea [SW49] [Huf52]. Algunos de ellos han sido claramente superados por otros en términos de sencillez o prestaciones sin embargo la gran variedad de aplicaciones de distinta naturaleza que demandan algoritmos de compresión permite que hoy en día coexistan muchos de ellos. En su mayor parte los métodos se definen como estáticos, por sencillez, pero comentaremos las versiones adaptativas o semiadaptativas en los casos en que presenten algún interés.

El método clásico es el de Huffman, al que prestaremos gran atención, incluyendo la versión adaptativa. Éste es un código con prefijo instantáneamente descodificable y mínimo, por tanto reúne todas las características deseables. Se muestran también otros códigos con prefijo, con características similares al de Huffman pero con una eficiencia inferior. Finalmente, el código de Lempel y Ziv es un ejemplo de un código de análisis-libre ampliamente utilizado. La codificación aritmética, el objeto de este trabajo, será ampliamente descrita en la siguiente sección.

1.2.1 Huffman

Es, con diferencia, el método de compresión entrópica más extendido, si bien parece que empieza a perder popularidad frente a otros métodos más eficientes o más sencillos. Además, ha servido como base para otras aplicaciones de naturaleza muy distinta, tales como construcción de árboles de búsqueda, generación de árboles de evaluación en compilación de expresiones, localización de fuentes de polución, etc.

La asignación de códigos a los distintos mensajes comienza por la construcción de un árbol binario basado en las probabilidades de los símbolos (figura 1.1). Éste se construye de la manera que se presenta a continuación. Los dos mensajes con menor probabilidad de ocurrir se combinan formando un nuevo elemento que será un árbol binario con dos elementos. Este nuevo elemento tendrá una probabilidad asociada que será igual a la suma de las probabilidades de los dos elementos que lo componen. Nuevamente se combinan los dos elementos con menor probabilidad, uno de los cuales puede ser, si su probabilidad es lo bastante baja, un árbol combinación de mensajes originales. El proceso continúa de forma iterativa hasta que queda un

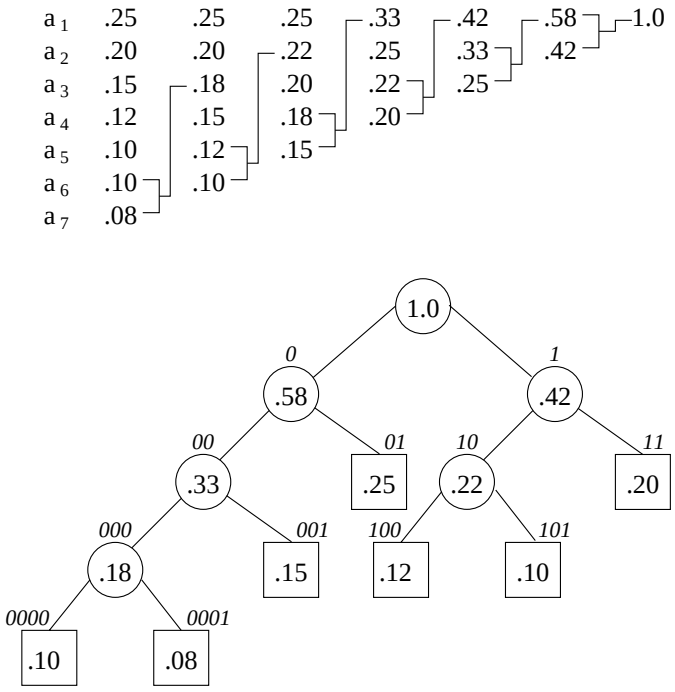


Figura 1.1: Ejemplo de construcción de un código de Huffman

sólo elemento que será un árbol binario de varios niveles y que contiene a todos los mensajes originales. En este momento se elige un criterio cualquiera para realizar la asignación de bits. Por ejemplo, se asigna un cero a la hoja a la izquierda de un nodo y un uno a la hoja de la derecha.

Este método, que es relativamente sencillo, garantiza la mejor asignación posible para una codificación bloque-bloque (o variable-bloque). Existen diversos estudios sobre la eficiencia del método [Gal78] en los que se ha estimado el valor máximo de la redundancia existente en el código resultante, y se ha demostrado que ésta es claramente inferior a la de otros métodos.

Huffman adaptativo

La posibilidad de diseñar una versión adaptativa del algoritmo de Huffman es muy atractiva si bien llevarla a la práctica no carece de dificultad. Existen diversas implementaciones que permiten codificar de forma adaptativa haciendo una única pasada sobre los datos y que describiremos esta sección.

Antes de ello consideraremos la versión semiadaptativa. Consiste en hacer una pasada previa sobre la muestra de datos y construir los códigos a partir de la información obtenida. Los inconvenientes principales de ésta estrategia residen en

que son necesarias dos pasadas sobre los datos, es necesario transmitir la tabla de códigos al descodificador, y que la adaptabilidad está muy restringida ya que no detecta la redundancia local. Sin embargo puede dar buenos resultados y de hecho está implementada en muchos compresores de propósito general y es una opción dentro del estándar JPEG [Gro94].

La dificultad de implementar un algoritmo adaptativo reside en actualizar el modelo estadístico a un coste razonable. La base del modelo es un árbol binario en el que la posición de cada mensaje dentro de él define el código que le corresponde, y la profundidad su longitud. Inicialmente el árbol constará de un único nodo hoja que engloba todos los mensajes. Al procesar un nuevo mensaje se codifica con este nodo único y a continuación se especifica cual es. El nodo que engloba a todos los mensaje que todavía no han aparecido (nodo *cero*) da lugar a dos nuevas hojas. Una para el nuevo mensaje y otra para el nodo *cero*. De esta forma la construcción del árbol se hace de tal forma que cada nodo tiene un nodo hermano. Esta propiedad (sibling) es fundamental en la construcción de árboles óptimos [Gal78].

El proceso de construcción del árbol continúa y convive con el de redistribución de los nodos de forma que los mensajes más probables ascienden por la estructura del árbol y les son asignados códigos más cortos. La forma en que se realiza la reestructuración influye en el tiempo de computación y aun en la eficiencia de la codificación.

El algoritmo conocido como FGK fue desarrollado y mejorado por Faller [Fal73], Gallager [Gal78] y Knuth [Knu85]. Su funcionamiento se ajusta a lo descrito en los párrafos anteriores. Cada vez que un mensaje es referenciado se incrementa su contador y el de los nodos de los que desciende. Cuando un nodo ve crecer su contador por encima de otro que tiene asociado un código más corto, el primero promociona en el árbol por medio de intercambios, cumpliéndose que los códigos más cortos sean asignados a los mensajes más probables.

Una mejora del algoritmo anterior es el conocido como algoritmo V [Vit87]. Las diferencias son dos. En primer lugar las sustituciones dentro del árbol se limitan a una por ciclo, con lo cual el proceso es más conservador. En segundo lugar se busca minimizar la altura del árbol, en lugar de priorizar a los mensajes más probables sin más. El algoritmo FGK tiene el inconveniente de que crece demasiado en altura y los mensajes que se encuentran en la parte baja generarán códigos muy largos en el tiempo que inviertan en mejorar su posición en el árbol. En aquellos casos en los que la probabilidad que tienen los distintos mensajes de aparecer no está excesivamente polarizada, el algoritmo V arroja mejores resultados.

1.2.2 Otros códigos con prefijo

El método de Huffman relegó al olvido un método anterior que fue desarrollado de forma simultánea e independiente por Shanon y Weaver [SW49] y Fano [Fan49] en 1949. Suponiendo, sin pérdida de generalidad que el alfabeto destino es un alfabeto binario, los códigos se construyen agrupando los mensajes en dos grupos de forma que ambos sumen probabilidades lo más parejas posible. A un grupo se le asigna como prefijo el bit '0' y al segundo el bit '1'. Ambos grupo se vuelven a dividir siguiendo el mismo procedimiento hasta que todos los grupos tienen un único elemento.

Existe una categoría adicional de códigos que se caracterizan por su sencillez. El objetivo de estos códigos no es obtener la máxima eficiencia, sino simplificar la codificación y descodificación. Se suele hablar de ellos como códigos universales que son aquellos en los que la longitud promedio del código es una función lineal de la entropía. Esta definición es bastante vaga y preferimos tratarlos dentro del problema de representación de enteros.

Los códigos de Elias [Eli75], Golomb [Gol66] y Rice [Ric79, Ric83, Ric91] entran dentro de esta categoría. Al contrario que otros sistemas de compresión no necesitan conocer con precisión las probabilidades de los mensajes. En su lugar se ordenan en orden decreciente de probabilidades y se codifica el puesto que ocupan en la lista en lugar del mensaje en si. Entonces el problema se reduce a encontrar la forma más eficiente de codificar una lista de enteros de probabilidad decreciente.

Códigos de Elias

Los códigos de Elias son sencillos si bien están lejos de ser óptimos. El primer código se conoce como γ y se construye de la siguiente manera: para cada entero z se introduce una secuencia de $\lfloor logz \rfloor$ 0's, a continuación un 1 como separador, y finalmente el valor de z expresado con la menor cantidad de bits posible. La descodificación es sencilla porque el número de ceros identifica inmediatamente la longitud del código.

A partir de este primer código es posible construir otros más eficientes de forma recursiva. El segundo código se denomina δ y se obtiene como una carrera de $\gamma(\lfloor logz \rfloor + 1)$ 0's y a continuación el valor de z sin utilizar ningún separador. Se pueden obtener más códigos siguiendo este procedimiento pero las mejoras son cada vez menores.

Códigos de Golomb y Rice

Estos códigos se caracterizan por su gran sencillez y es por ello que están siendo muy utilizados en la actualidad a pesar de no ser óptimos. Fueron desarrollados inicialmente por Golomb [Gol66] y responden a la misma necesidad de representar

Golomb	**m=1**	**m=2**	**m=3**	**m=4**	**m=5**	**m=6**	**m=7**	**m=8**
Rice	**k=0**	**k=1**		**k=2**				**k=3**
0	0	00	00	000	000	000	000	0000
1	10	01	010	001	001	001	0010	0001
2	110	100	011	010	010	0100	0011	0010
3	1110	101	100	011	0110	0101	0100	0011
4	11110	1100	1010	1000	0111	0110	0101	0100
5	11...10	1101	1011	1001	1000	0111	0110	0101
6	11...10	11100	1100	1010	1001	1000	0111	0110
7	11...10	11101	11010	1011	1010	1001	1000	0111
8	11...10	111100	11011	11000	10110	10100	10010	10000

Tabla 1.1: Códigos de Golomb y Rice

enteros de forma compacta que los códigos de Elias. Sin embargo la forma de obtener los distintos códigos de Golomb es más directa.

Dada una lista de mensajes se ordenan estos de mayor a menor probabilidad y se les asigna un índice comenzando por el 0, que es el que se ha de codificar. Los distintos códigos de Golomb se identifican por un parámetro m que toma valores enteros mayores que 0. Un entero n se codifica para un código de Golomb m como una secuencia de $\lfloor n/m \rfloor$ 1's (código unario) seguido de $n \ mod \ m$ expresado con la menor cantidad de bits tal y como se hace con los códigos de Elias.

Los códigos de Rice fueron desarrollados de forma independiente [Ric79, Ric83, Ric91] y son un caso particular de los códigos de Golomb, ya que los posibles valores del parámetro se reducen a potencias de 2. El nuevo parámetro se llama k de tal forma que $m = 2^k$ $(m = 0, 1, ...)$. En la tabla 1.1 se muestran los códigos de Golomb para los 8 primeros valores de m y los 4 primeros códigos de Rice. El procedimiento para obtenerlos es el descrito en el párrafo anterior. La columna de la izquierda corresponde al índice asignado a cada símbolo. El 0 se asigna al símbolo más probable y así sucesivamente.

Si bien los códigos de Rice son más limitados que los de Golomb, también es cierto que se restringen a los casos más interesantes para su implementación en una computadora, ya que todas las operaciones con potencias de 2 son triviales y se pueden resolver con aritmética de enteros.

Aunque los códigos de Golomb y Rice están lejos de ser óptimos, funcionan bien para muchas distribuciones de datos del tipo exponencial siempre y cuando se escoja adecuadamente el parámetro.

La dificultad de implementar tales códigos no depende de la construcción de los mismos ni de la descodificación, que son triviales, sino de la construcción de la lista ordenada de probabilidades y de la estimación del valor adecuado del parámetro

m o k. Para la primera tarea tendremos dos casos distintos. Es muy común que ciertos tipos de datos a comprimir presentan un histograma con caída exponencial. En este caso los mensajes ya están ordenados por probabilidades decrecientes y no es necesario realizar ninguna ordenación. Si el histograma es simétrico se pueden intercalar los mensajes de ambos lados y se mantiene una distribución similar. En el caso de que la distribución no sea conocida a priori se puede implementar un esquema de reordenación. Una posibilidad es hacer un seguimiento de las probabilidades de los mensajes y cada cierto tiempo reordenar la lista tanto en el codificador como en el descodificador. Otros métodos son más directos y se basan modificar la posición en la lista del último elemento referenciado de acuerdo con algún criterio, que puede ser hacerle avanzar una posición, ponerlo en primer lugar, etc [BSTW86].

La estimación del parámetro óptimo es un aspecto importante ya que la eficiencia cambia de forma drástica dependiendo de la elección del parámetro. Una primera aproximación consistiría en hacer una pasada previa por los datos, contando el número de ocurrencias y de esta manera hacer el cálculo del coste final para distintos valores de m o k. El valor óptimo se transmite junto con la secuencia de código. Esto tiene el inconveniente de que es necesario dar dos pasadas sobre los datos, y además es necesario estimar el coste para varios valores de los parámetros. Una solución similar, que puede sacar partido de variaciones locales de las probabilidades, consiste en implementar el mismo esquema por bloques. Los datos se dividen en bloques y se estima el parámetro óptimo para cada uno. Al igual que el anterior es necesaria una pasada previa, pero la latencia antes de comenzar la codificación es menor, lo que puede ser interesante en aplicaciones de transmisión de imágenes. La estimación del parámetro ha de hacerse varias veces y el coste computacional aumenta.

Otro tipo de solución, que implica menor coste en computaciones y reduce la latencia, consiste en estimar el contexto actual de los datos y utilizar el valor óptimo asociado a ese contexto. La estimación del contexto se hace de acuerdo con el entorno del dato que se está procesando, y en caso de utilizar únicamente datos que ya se han procesado no será necesario comunicarlo al descodificador sino que éste podrá estimarlo también de forma independiente.

1.2.3 Códigos de Lempel y Ziv

Los códigos de Lempel y Ziv [ZL77] son el caso más conocido de método análisis-libre y están ampliamente implementados en diversos tipos de compresores de propósito general. Mientras que todos los métodos que hemos visto hasta ahora trabajan sobre un conjunto definido de mensajes, al algoritmo Lempel-Ziv define los mensajes en tiempo de codificación. El caso de los algoritmos adaptativos, que incorporan nuevos mensajes durante el proceso, no es muy distinto al caso estático ya que aun cuando

Cadena de símbolos	*Código*	*Salida producida*
A	1	A
L	2	L
LA	3	2 + 1
LO	4	2 + O
LAS	5	3 + S
LOS	6	4 + S
LOSA	7	6 + 1
P	8	P
A	9	1
AA	10	1 + 1
AAA	11	10 + 1
####	4095	4095

Figura 1.2: Ejemplo de codificación de Lempel y Ziv

no todos los mensajes estén incorporados al modelo, se sabe cuales son.

El algoritmo original utiliza un código de longitud fija para codificar cada mensaje, por ejemplo 12 bits. El primer símbolo que se procesa constituye un mensaje por si mismo, se le asocia un código y se introduce en la tabla de mensajes reconocidos. El proceso sigue mientras no se repita ningún símbolo. Cuando esto ocurre se crea un nuevo mensaje que representará a ese símbolo y al siguiente. Entonces se codifican dos símbolos con un solo código. El proceso sigue, añadiendo mensajes, de forma que habrá mensajes que comprenderán secuencias largas de símbolos. Un ejemplo de este procedimiento se muestra en la figura 1.2.

Resulta obvio que la eficiencia de este método depende mucho de los datos. En primer lugar no es adecuado para secuencias cortas, ya que el coste de los primeros mensajes es alto. En realidad, este es un inconveniente de muchos métodos adaptativos. Además, la cantidad de mensajes posibles está limitada por la longitud de las palabras código. En algunos casos las cadenas largas pueden no ser lo suficientemente abundantes para compensar la longitud de los códigos.

En cuanto a su implementación es un algoritmo irregular que debe ser implementado en software. El proceso de descodificación es relativamente sencillo, pero la codificación supone búsquedas complejas en tablas. Por todo ello, el algoritmo ha sido objeto de diversos rediseños y optimizaciones [Ris83, SS82].

1.3 Codificación aritmética

En esta sección introduciremos la codificación aritmética, el método de compresión objeto del presente trabajo. Expondremos los trabajos originales de los que surgió

el concepto y haremos una revisión de todas sus variantes principales, codificación aritmética binaria y codificación quasi-aritmética, para centrarnos a continuación en la codificación aritmética multinivel.

La codificación aritmética fue introducida por Abramson [Abr63] en 1963. Es un método radicalmente diferente de los vistos hasta ahora, tanto por el procedimiento de codificación como por el hecho de que su eficiencia es, en teoría, óptima.

1.3.1 Introducción al algoritmo

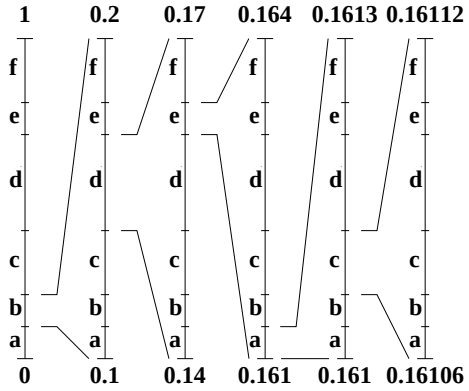
Mientras que todos los métodos anteriores asignaban un código definido a cada posible mensaje, la codificación aritmética funciona de una forma totalmente diferente. En su lugar el codificador genera un único código que representa la secuencia completa de mensajes fuente.

La asignación mensaje-código hace que el modelo estadístico y la codificación estén íntimamente unidos y sea difícil distinguirlos. En cambio, la codificación aritmética establece una clara separación entre el modelo y la codificación. Al igual que los códigos de Huffman y de Shanon-Fano, es un método basado en las probabilidades de los mensajes, que deben ser conocidas, y estar reflejadas en el modelo, con la mayor precisión posible. La diferencia estriba en que la probabilidad de un mensaje define el código que le será asignado en el contexto actual del codificador, en lugar de estar en función de las probabilidades de los restantes mensajes, como era el caso de Huffman. Dado que el modelo se limita a asignar una probabilidad a cada mensaje, y la codificación opera a partir de la información facilitada por el modelo, existe una gran libertad para implementar ambas partes, el modelo y el codificador, atendiendo a distintas necesidades.

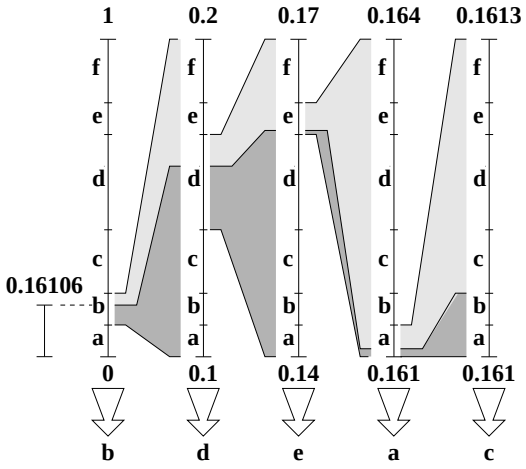
A cada secuencia de mensajes la codificación aritmética asigna un número real comprendido entre 0 y 1 que la identifica. Existe un simil geométrico (figura 1.3) que facilita la comprensión del algoritmo y que se explica a continuación. Dado que las probabilidades de todos los mensajes suman la unidad, suponemos un segmento de longitud unidad que está dividido en tantos subintervalos como mensajes haya, cada uno de los cuales de longitud proporcional a su probabilidad. Cuando se procesa un mensaje su subintervalo asociado es seleccionado y dividido tal y como se hizo inicialmente con el intervalo [0,1). Al procesar un nuevo mensaje se selecciona su subintervalo y se repite el proceso de división. La repetición iterativa de este proceso nos lleva a un último subintervalo que está contenido en todos los anteriores y que por tanto representa a todos los mensajes que se han procesado. En realidad cualquier punto dentro de este intervalo lo hace, así que un único número cumple esta función. La razón de compresión depende la precisión necesaria para representar este número real.

El proceso de descodificación consiste en comparar el valor transmitido con las

símbolo	prob.	prob. acum.	intervalo asociado
a	0.1	0.0	[0.0, 0.1)
b	0.1	0.1	[0.1, 0.2)
c	0.2	0.2	[0.2, 0.4)
d	0.3	0.4	[0.4, 0.7)
e	0.1	0.7	[0.7, 0.8)
f	0.2	0.8	[0.8, 1.0)



(a) Codificación



(b) Descodificación

Figura 1.3: Ejemplo de codificación y descodificación de una secuencia de símbolos.

subintervalos asociados a cada mensaje. Los mensajes se obtienen en el mismo orden en que fueron codificados [Jon81], aunque inicialmente este proceso era ineficiente y los mensajes se descodificaban en orden inverso al de codificación.

Los procedimientos que se han descrito se pueden aplicar al ejemplo que se muestra en la figura 1.3. Para un alfabeto y una distribución de probabilidades dados se codifica la secuencia *b, d, e, a, c*, obteniendo un subintervalo dentro del original que representa la secuencia completa. Cualquier punto dentro del mismo (el extremo inferior en este caso) es transmitido al descodificador, donde la secuencia original es obtenida en el mismo orden en que fue codificada. En la figura 1.3.(b) se muestra este proceso. El código recibido se compara con los subintervalos, descodificándose símbolo por símbolo. Las regiones con distintos sombreados corresponden a las partes del nuevo intervalo que quedan por encima y por debajo del nuevo valor que se compara.

A continuación introduciremos algunos conceptos relacionados con el algoritmo con el objetivo de facilitar la comprensión de las explicaciones.

- símbolo: hasta el momento hemos utilizado la notación más general de mensaje para referirnos a un parte de la secuencia a comprimir. Dado que la codificación aritmética es un algoritmo del tipo bloque-variable, en adelante utilizaremos el término símbolo para referirnos a un elemento del alfabeto origen.

- alfabeto destino: siempre es posible, sin pérdida de generalidad, considerar que se utiliza el alfabeto binario. Dado que esta memoria versa sobre el diseño de hardware esto se puede asumir con mayor motivo.

- rango: longitud del subintervalo que se procesa en cada momento. Su valor en el ciclo de procesamiento i lo denotaremos por A_i.

- punto bajo del rango: Es el extremo inferior del subintervalo que se procesa. El subintervalo está definido entonces por su extremo inferior y su longitud. Lo denotaremos por C_i.

- probabilidad: hablaremos de probabilidad como la estimación que nuestro modelo hace de la probabilidad real del símbolo, esto es, la posibilidad que existe de que el próximo símbolo que se procese sea uno dado. Denotaremos por $P_i(k)$ a la probabilidad del símbolo k en el ciclo i.

- probabilidad acumulativa: la división de un intervalo en subintervalos supone implicitamente la existencia de un orden en el conjunto de símbolos. Siguiendo con el simil geométrico, si la longitud de un subintervalo representa su probabilidad, su posición dentro del intervalo represente su probabilidad acumulativa, esto es, la suma de las probabilidades de los símbolos situados antes de una

dado. La probabilidad acumulativa de un símbolo k en el ciclo i viene dada por la ecuación (1.1), siendo $P_i(j)$ la probabilidad del símbolo j en el ciclo i.

$$S_i(k) = \sum_{j=0}^{j<k} P_i(j) \tag{1.1}$$

Las ecuaciones del algoritmo son las siguientes [Jia95]:

$$\begin{aligned} A_{i+1} &= A_i \cdot P_i(k) \\ C_{i+1} &= C_i + A_i \cdot S_i(k) \end{aligned} \tag{1.2}$$

Evidentemente este es un algoritmo recursivo, lo cual limita en gran medida la velocidad de procesamiento al no poder introducir paralelismo de forma directa.

1.3.2 Implementación básica del algoritmo de codificación

Naturalmente, la aplicación de las ecuaciones (1.2) es inviable tanto en software como en hardware. Al cabo de unos pocos ciclos la precisión necesaria para seguir con los cálculos habría sobrepasado la capacidad de cualquier procesador aritmético. Por tanto se impone introducir un esquema que permita mantener la precisión dentro de un rango razonable. El método que presentamos a continuación fue introducido por Witten, Neal y Cleary [WNC87] y permite no sólo utilizar precisión finita, sino transmisión incremental, esto es, parte del código es transmitido durante la codificación tan pronto como ésta deja de tener influencia sobre los cálculos futuros.

En lugar de trabajar con el rango del subintervalo, en [WNC87] se prefiere considerar los puntos bajo y alto del subintervalo, que llamaremos *bajo* y *alto*, y considerar el rango A como un elemento intermedio.

En caso de que el valor de *alto* sea menor que 1/2, entonces el código final también lo será, y por tanto conocemos un bit de este valor, que será cero. Podemos entonces transmitir un bit '0' y multiplicar el valor de *bajo* y *alto* por 2, de forma que el rango sigue en el intervalo [0,1) pero su valor no disminuya en demasía.

Cuando el rango está contenido totalmente en la segunda mitad del intervalo se transmite un bit '1', el rango se traslada a la primera mitad del intervalo y se multiplican *bajo* y *alto* por 2.

Cuando el rango se encuentra comprendido entre el primer y tercer cuartos, de *bajo* y *alto* se resta un cuarto de intervalo y se multiplican por 2. No se transmite ningún bit, pero se incrementa un contador así, cuando en los siguientes ciclos haya una transmisión, esta será seguida de tantos bits opuestos al transmitido como indique el contador.

Las operaciones anteriores se repiten si es necesario. Así, para cada símbolo se transmitirá una cierta cantidad de bits. Nótese que no existe una relación exacta entre el número de bits que se transmiten y la probabilidad del símbolo. Dependiendo de los últimos valores que se hayan procesado, la cantidad de bits que se transmiten puede variar, aunque no más de una unidad.

La eficiencia de la codificación aritmética se debe a que la longitud del código que se transmite en proporcional a las probabilidades de los símbolos, pero en su conjunto, no símbolo a símbolo. Una determinada probabilidad puede tener asociado un código de 3.4 bits, pero se transmitirá una cantidad entera de bits, que será de 3 y dejará el resto para siguientes ciclos, o bien será de 4 debido a restos de ciclos anteriores. Esta es una diferencia fundamental con los códigos bloque-bloque o variable-bloque que hemos visto hasta ahora.

1.3.3 Algoritmo de descodificación

Si bien el proceso de codificación es sencillo y únicamente implica unas pocas operaciones matemáticas, la descodificación es un proceso costoso. Es éste un inconveniente de la mayor parte de los algoritmos vistos hasta ahora (Huffman especialmente). El proceso consiste en encontrar en cada ciclo el símbolo cuyo subintervalo asociado contenga al rango (figura 1.3.(b)). Esto se traduce en encontrar el mayor k para el cual se cumpla que

$$C_i \geq A_i \cdot S_i(k) \tag{1.3}$$

Cuando el alfabeto fuente es grande, el número de operaciones involucradas: multiplicaciones, comparaciones y actualización, limita mucho la velocidad de procesamiento. Afortunadamente las probabilidades acumulativas de los símbolos forman una lista ordenada, y por tanto se puede introducir descodificación en varios niveles. Una comparación con el símbolo situado a mitad de la tabla nos dirá si el símbolo correcto se encuentra en la mitad superior o en la inferior. Siendo n el número de símbolos del alfabeto, la descodificación necesitará $log_2 n$ comparaciones. Otra solución para acelerar la descodificación, en hardware, será introducir paralelismo, aun cuando el coste aumenta por lo que la solución más adecuada es combinar ambas estrategias [OB97].

Una vez que se conoce el símbolo k, es necesario actualizar los valores de A y C de forma acorde a como se hizo en el codificador. Las operaciones son simétricas de las mostradas en (1.2)

$$\begin{aligned} A_{i+1} &= A_i \cdot P_i(k) \\ C_{i+1} &= C_i - A_i \cdot S_i(k) \cup nuevos\ bits \end{aligned} \tag{1.4}$$

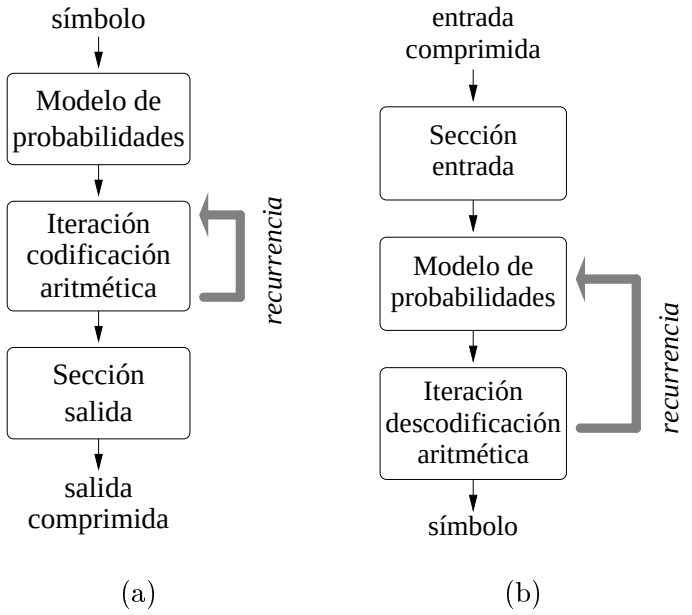


Figura 1.4: Distribución por bloques. (a) Codificador. (b) Descodificador.

Al igual que en el codificador también es necesario mantener la precisión dentro de unos límites manejables, así que el rango se escala de forma conveniente. De forma simétrica al codificador, ahora se introducen nuevos bits desde la corriente de entrada que se añaden al valor de C_{i+1} concatenándolos ($\cup$).

1.3.4 Codificación aritmética adaptativa

La separación que existe entre el modelo y la codificación-descodificación permite modificar ambas partes según convenga. Esta es una de las principales ventajas de la codificación aritmética y que contrasta con los métodos que hemos visto anteriormente. La distribución de los elementos que configuran un codificador y un descodificador aritméticos se puede ver en la figura 1.4. En el descodificador la recurrencia abarca también al modelo dado que no se puede actualizar el modelo hasta haber descodificado el símbolo. En cualquier caso hacer que el modelo sea adaptativo es relativamente sencillo, dado que la única función del modelo es asignar a cada símbolo una probabilidad y una probabilidad acumulativa.

Para cada símbolo se contabiliza el número de veces que ha sido referenciado y su probabilidad será proporcional a esta cuenta. Su probabilidad acumulativa será la suma de las probabilidades de los símbolos que lo preceden en la tabla. Surgen entonces tres inconvenientes. En primer lugar, el valor de los contadores no es

realmente equivalente a las probabilidades, estos deben ser escalados de forma que se cumpla que la suma de todos ellos sea igual a la unidad. En segundo lugar, las probabilidades acumulativas deben ser actualizadas cada ciclo.

El tercer inconveniente es el hecho de que tras procesar una cierta cantidad de símbolos el valor contenido en los contadores excederá la precisión utilizada. La solución comúnmente utilizada consiste en dividir las probabilidades por 2 cuando estas alcanzan el valor máximo.

Se han realizado análisis acerca de como afectan los escalamientos a la compresión, así como otros detalles más sutiles como es el hecho de utilizar aritmética entera, o en punto flotante con precisión finita, demostrando que su influencia es muy pequeña.

El escalado de los contadores para limitar su crecimiento tiene un efecto de influir sobre la adaptabilidad del modelo, al disminuir la importancia relativa de los símbolos que han caído en desuso. De esta forma el modelo puede adaptarse mejor a los cambios de contexto.

1.3.5 Uso de aritmética de precisión reducida

Cuando se trata de una aplicación software el coste es básicamente de consumo de tiempo, y en una computadora moderna esto está más relacionado con el número de instrucciones que con la naturaleza de las mismas. En una implementación hardware sucede lo contrario. El número de operaciones no siempre es crítico ya que pueden ser triviales o ejecutarse en paralelo. En cambio, las operaciones complejas, tales como cálculos en punto flotante, pueden tener un coste demasiado grande. Por ello, las versiones hardware de la codificación aritmética presentan importantes diferencias con respecto a los algoritmos descritos en las secciones 1.3.2 y 1.3.3. Es habitual referirse a estas implementaciones con precisión reducida como codificación quasi-aritmética.

Existen dos aspectos en los que se reduce la precisión de la aritmética: la gestión del modelo de probabilidades y las iteraciones sobre el rango. En el primero las probabilidades son sustituidas por el número de veces que han sido referenciados los símbolos, y estos valores no se escalan para que sumen la unidad, ya que esto implicaría la operación $P_i(k) = m_i(k) * S_{max}/S_i(n)$. Donde $S_i(n)$ es la probabilidad acumulativa del hipotético símbolo n (realmente es $S_i(n-1) + P_i(n-1)$), $m_i(k)$ es la cuenta del número de veces que ha sido referenciado k y S_{max} es el mayor valor posible de $S_i(n)$. Naturalmente el coste de estas operaciones excede la complejidad que se supone a un codificador entrópico implementado en hardware.

Las operaciones de las iteraciones del codificador (1.2) y el descodificador (1.4) también se ven afectadas por la reducción en la precisión. Todas las operaciones se realizan con enteros, si bien esto no es mayor problema ya que manteniendo el

valor de A en el rango adecuado no hay pérdida sensible de precisión. Los productos $A_i \cdot P_i(k)$ y $A_i \cdot S_i(k)$ se realizan con precisión reducida, utilizando únicamente algunos bits de A_i. Existen distintas alternativas para escoger que bits se utilizan. A mayor número de bits mayor precisión y mayor coste. Algunas implementaciones optan por reducir coste y se consideran tan sólo unos pocos bits, o incluso se asume un valor de A constante, redondeando a la unidad [PMJA88]. En otras se optimiza el coste multiplicando por una cantidad b de bits no nulos [CKW91]. Otras posibilidades son realizar una conversión a signed-digit con un radix alto [BBL97], tabular valores [ATL+88, HV94], etc.

Para mantener el rango A dentro de los límites de la precisión y al mismo tiempo efectuar una transmisión incremental utilizaremos un esquema distinto al visto en la sección 1.3.2. En su lugar, las ecuaciones (1.2) se convierten en:

$$\begin{aligned} a_{i+1} &= A_i \cdot P_i(k) \\ c_{i+1} &= C_i + A_i \cdot S_i(k) \\ A_{i+1} &= 2^t \cdot a_{i+1} \\ C_{i+1} &= 2^t \cdot c_{i+1} \end{aligned} \tag{1.5}$$

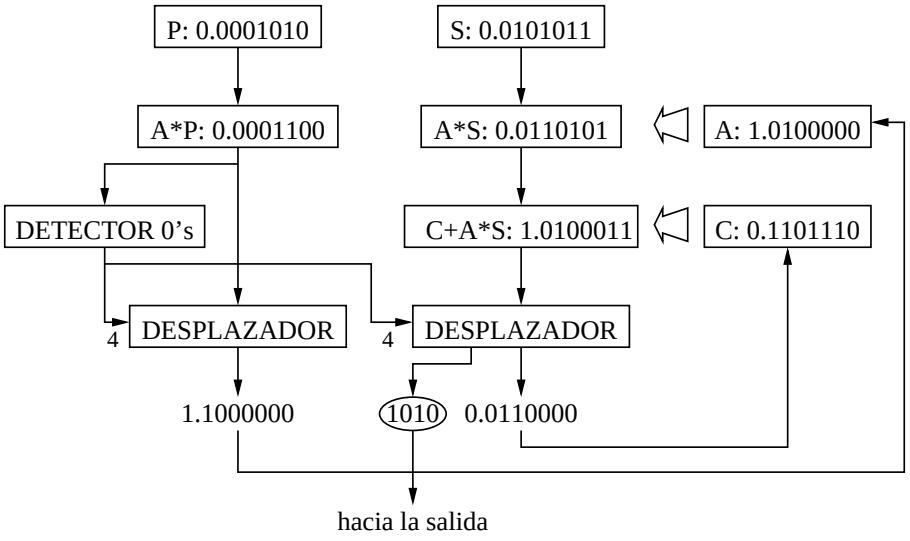
donde t es la cantidad de bits que hay que desplazar a_{i+1} para normalizar su valor. Los bits que son desplazados a la izquierda de C_{i+1} son enviados a la sección de salida para su transmisión. Un ejemplo de este procedimiento se muestra en la figura 1.5.(a). Este esquema genera el código para un símbolo en su único ciclo, al contrario que el visto anteriormente, que se repetía hasta que el rango estaba convenientemente normalizado. Tiene, sin embargo, la desventaja de que es posible que la iteración sobre C genere acarreos hacia la parte que ya ha sido enviada a la salida. Por ello es necesario asimilar estos acarreos antes de transmitir. Para ello la solución más utilizada es un esquema denominado bit-stuffing [LR81] que comentaremos dentro de este mismo capítulo.

Las ecuaciones para la descodificación (1.6) también cambian, incluyendo la entrada de nuevos bits desde la corriente de entrada, que se produce como una simple concatenación ($\cup$). Un ejemplo numérico se puede observar en la figura 1.5.(b).

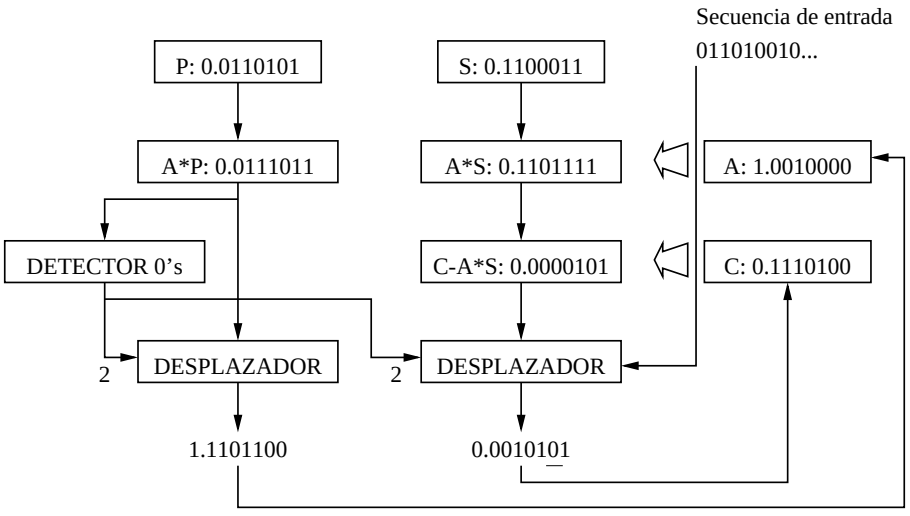
$$\begin{aligned} a_{i+1} &= A_i \cdot P_i(k) \\ c_{i+1} &= C_i - A_i \cdot S_i(k) \\ A_{i+1} &= 2^t \cdot a_{i+1} \\ C_{i+1} &= 2^t \cdot c_{i+1} \ \cup \ (t \ bits) \end{aligned} \tag{1.6}$$

1.3.6 Codificación aritmética binaria

Dadas las dificultades para implementar un modelo para un alfabeto multinivel, los codificadores binivel o binarios han surgido como una alternativa sencilla y eficiente.



(a)



(b)

Figura 1.5: Ejemplos numéricos. (a) Codificador. (b) Descodificador.

Los datos propiamente binarios no son particularmente abundantes, salvo imágenes en blanco y negro para transmisión de fax y archivado de documentos, pero cualquier tipo de fuente puede ser convertida a una secuencia de ceros y unos mediante un árbol binario de decisiones. Existe una penalización asociada a esta transformación, pero con las técnicas adecuadas se pueden obtener buenas relaciones de compresión.

Cuando se trabaja con un alfabeto fuente binario la codificación aritmética es la opción idónea ya que permite trabajar con códigos cuya longitud no es un número entero de bits, y funciona bien cuando la probabilidad de un símbolo es muy próxima a la unidad, lo cual no es cierto para ningún otro codificador.

Por otra parte, las ventajas a nivel de implementación son muchas. Sólo es necesario mantener un modelo de 2 símbolos, la probabilidad acumulativa del primero de ellos es cero y la del segundo es la probabilidad del primero. Dada la simplicidad del modelo, no es costoso introducir compresión de orden superior. Eligiendo distintos contextos de forma adecuada un codificador binario puede superar de forma clara las prestaciones de los codificadores vistos hasta el momento. Por otra parte, y dado que habitualmente la muestra que se comprime será una traslación desde un alfabeto multinivel, el compresor tendrá una velocidad de procesamiento baja, al necesitar varios ciclos para codificar un símbolo del alfabeto original.

Las referencias obligadas al hablar de codificación aritmética binaria son el *Q-Coder* [PMJA88] y el *QM-Coder* [PM93] de IBM. Basados en una implementación anterior, el *Skew-Coder* [LR81], fueron diseñados para ser eficientes y veloces tanto en software como en hardware. No se codifican los dos símbolos como tales, sino el símbolo más probable (MPS) y el menos probable (LPS). La probabilidad del MPS se denota por P_e y la del LPS por Q_e. Ambos métodos sustituyen las multiplicaciones por sumas y desplazamientos sin pérdida apreciable de precisión. Manteniendo el rango A normalizado entre 0.75 y 1.5 las operaciones son tan sencillas como

$$\begin{aligned} A \cdot Q_e &\simeq Q_e \\ A \cdot P_e &= A \cdot (1 - Q_e) \simeq A - Q_e \end{aligned} \tag{1.7}$$

Las principales diferencias entre el *Q-Coder* y el *QM-Coder* son las siguientes. El *QM-Coder* incorpora un esquema de intercambio condicional en el orden de los símbolos que mejora ligeramente la compresión. El control de la propagación de acarreos es también distinto. Mientras el *Q-Coder* hace uso de la técnica de *bit-stuffing*, el *QM-Coder* sacrifica compresión en favor de un sistema más sencillo de *byte-stuffing*. Finalmente, la estimación de probabilidades es también distinta. Ambos utilizan una máquina de estados que pasa de un estado (valor de Q_e) a otro en función de los símbolos que se procesan. En el *QM-Coder* se introduce una evolución más agresiva al inicio para alcanzar la distribución óptima de probabilidades en poco tiempo.

1.3.7 Revisión de arquitecturas para la codificación aritmética multinivel

Haremos en esta sección una revisión de arquitecturas para codificación multinivel descritos en la literatura, comentando sus ventajas e inconvenientes, dejando para la sección siguiente la que tomamos como punto de partida para las nuevas arquitecturas que proponemos.

En [BRM88] se describe una implementación hardware de un codificador aritmético multinivel no adaptativo (figura 1.6). Los autores, introducen codificación de orden superior para mejorar la compresión sin aumentar el coste de almacenamiento ya que no se multiplica el número de tablas. El sistema propuesto consiste en dividir el alfabeto origen en varios grupos cada uno de ellos asociado a un contexto (grupos 1, 2 y 3 de la memoria RAM en la figura). A cada grupo se añaden símbolos que indican cambio de contexto. Los cambios de contexto implican un ciclo extra de procesamiento, y reducen la eficiencia, pero esto último se ve compensado por la codificación de orden superior. La circuitería que controla los cambios de contexto se muestra en la parte izquierda de la figura. No se aplica ningún esquema de reducción de complejidad en las operaciones aritméticas y, tal y como se dijo anteriormente, no es adaptativo. El proceso necesita dos ciclos para completarse ya que se utiliza el mismo multiplicador para computar el punto bajo del intervalo y el rango. La iteración completa y los multiplexores para compartir el multiplicador se pueden ver en la parte inferior de la figura. Los autores aseguran obtener buenos resultados en compresión de ciertos textos e imágenes, si bien éstos no se tabulan. Finalmente, no se apuntan soluciones para disminuir la complejidad de la descodificación.

Ventajas:

- Introduce codificación de orden superior
- Reutiliza hardware en la iteración

Inconvenientes:

- No es adaptativo
- La aritmética es compleja
- Necesita dos ciclos para procesar cada símbolo
- El modelo de orden superior aumenta las necesidades de almacenamiento
- No se describe el descodificador

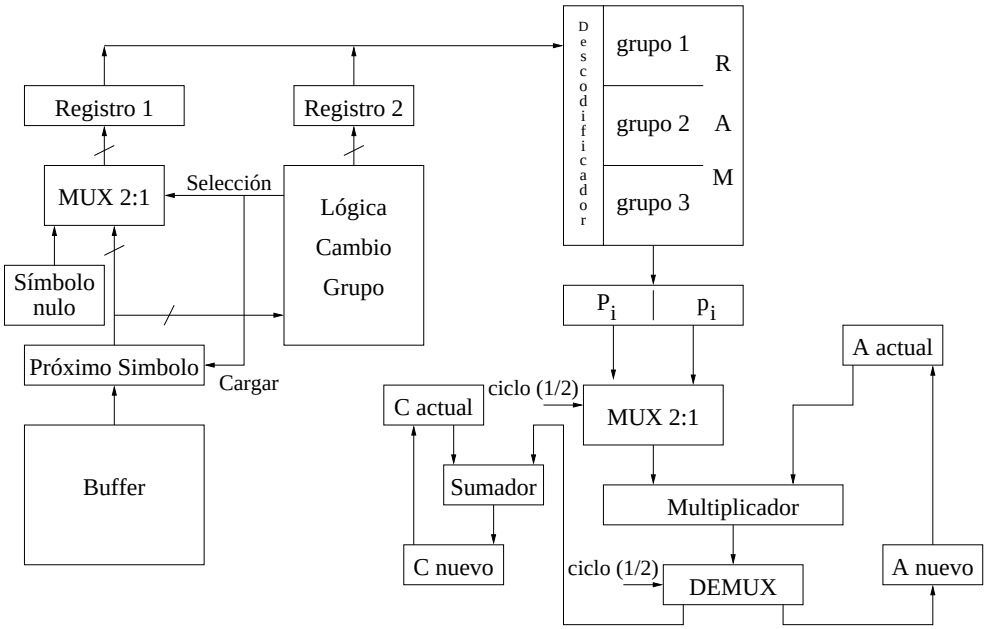


Figura 1.6: Estructura del codificador aritmético con cambio de contexto descrito en [BRM88].

Una descripción de un codificador y un descodificador se muestra en [Jia95]. Las mejoras se proponen en la iteración, simplificándola, pero no se simplifica el modelo. En realidad se minimiza su importancia ya que no es adaptativo. La descodificación se realiza de forma secuencial y se almacenan en memoria las probabilidades y las probabilidades acumulativas. Este trabajo supone una ruptura con las implementaciones con precisión completa y un acercamiento a la realidad del diseño hardware. Sustituye la iteración considerando los puntos alto y bajo del intervalo por la más conveniente de utilizar la longitud del intervalo.

Ventajas:

- Utiliza aritmética simplificada

Inconvenientes:

- No es adaptativo
- El descodificador es muy ineficiente

En [CY91] se presenta un codificador multinivel orientado a comprimir los coeficientes de una DCT. En un algoritmo sin multiplicaciones que se mejora ligeramente

el rendimiento introduciendo ciertas correcciones en las operaciones. Éste es un caso muy común, en el que se pretende eliminar las multiplicaciones y se introducen operaciones adicionales de un coste similar a las que se pretenden evitar. Para conseguir buenos resultados se introducen gran cantidad de contextos que disparan el coste. La implementación no es adaptativa y no se menciona el descodificador. Se muestran resultados comparándolos de forma favorable con la versión base de JPEG, utilizando Huffman.

Ventajas:

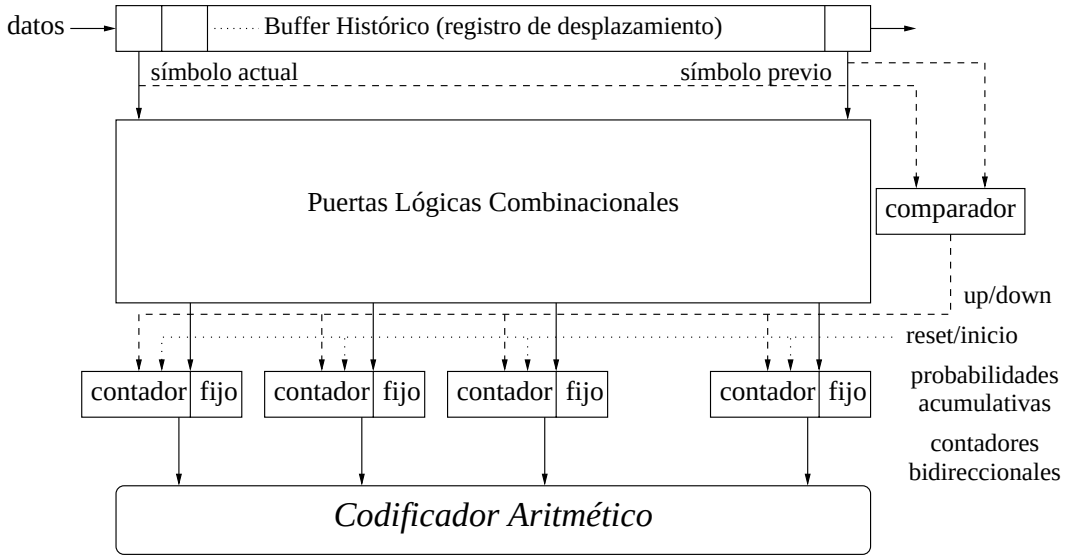
- Aritmética simplificada
- Buenos resultados

Inconvenientes:

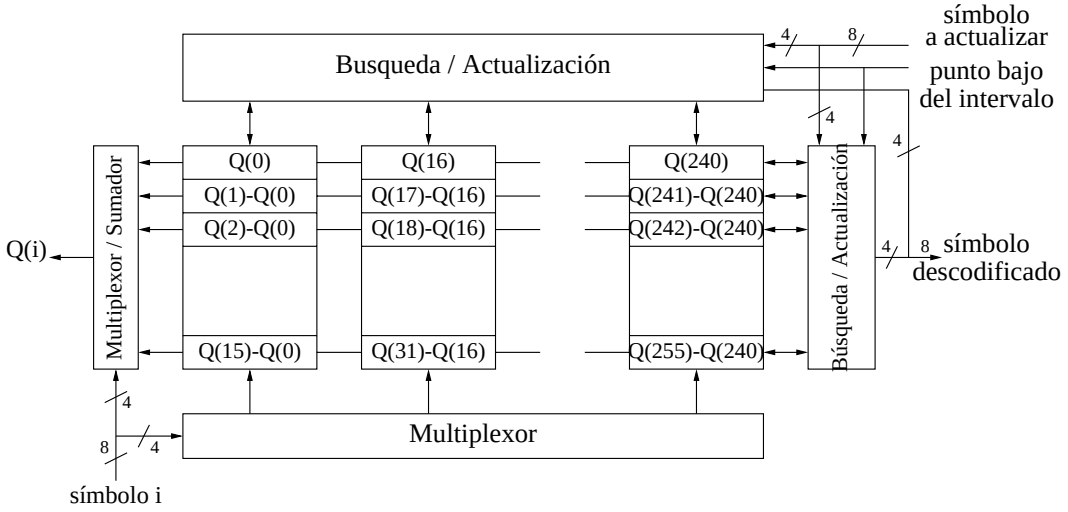
- No es adaptativo
- No se describe un descodificador
- El coste es grande debido a los distintos contextos

Finalmente comentaremos un trabajo reciente [HW98] que basa su eficiencia en un modelo de historia limitada (figura 1.7.(a)). Los últimos símbolos procesados son primados durante la codificación al aumentar su probabilidad. De esta manera se aprovecha la localidad de los datos. Es un sistema adaptativo, pero no necesita escalamiento de probabilidades, ya que cuando unas cuentas se incrementan otras son decrementadas y nunca se produce saturación. El control se realiza mediante un buffer (registro de desplazamiento de la figura). Los símbolos que entran en el buffer ven ponderada su probabilidad en una cantidad mayor que la unidad (contador). Cuando un símbolo es expulsado el contador se decrementa. Por otra parte, utiliza un tamaño de palabra reducido y un esquema *sin* multiplicaciones similar al de [BRM88]. Para facilitar el acceso al modelo para la codificación y la descodificación se utiliza un esquema (figura 1.7.(b)) similar al descrito en [BBL97]. En su contra, este esquema necesita dos ciclos para completar la codificación de cada símbolo, y no logra disminuir la cantidad de datos que debe manejar el modelo.

Las iteraciones de codificación y descodificación se realizan sin multiplicaciones tal y como se puede ver en la figura 1.8. En realidad se introducen operaciones adicionales para compensar la ineficiencia de utilizar un esquema tan sencillo. El codificador y el descodificador son muy similares tal y como se puede ver. En realidad los bloques etiquetados como *unidad de modelado* y *tabla de búsqueda* suponen la parte más compleja, el acceso al modelo y su actualización. Estos últimos tienen la



(a)



(b)

Figura 1.7: Codificador con modelo de historia limitada [HW98]. (a) Modelo. (b) Estructura de cálculo y actualización de probabilidades acumulativas.

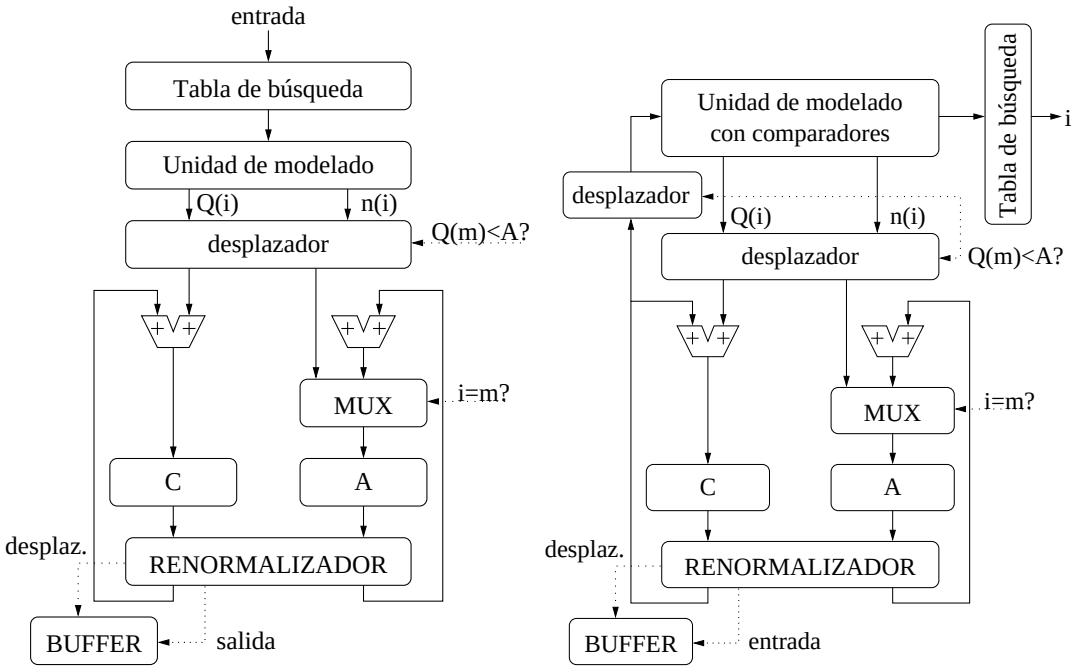


Figura 1.8: Codificador con modelo de historia limitada. (a) Codificador. (b) Descodificador.

estructura que se muestra en la figura 1.7. La iteración en sí utiliza la probabilidad de cada símbolo, $n(i)$ y su probabilidad acumulativa $Q(i)$, pero son ponderados si el valor del tope de las probabilidades acumulativas es menor que el rango A. Por este motivo el último símbolo del alfabeto, m, es codificado de forma diferente. En realidad este esquema enmascara una multiplicación simplificada que también se podría haber conseguido utilizando aritmética de baja precisión y aritmética redundante. Se utilizan sumadores de 16 bits para A y C, que es un tamaño bastante grande. Además, C necesita un registro adicional de 48 bits para compensar la propagación de acarreos. En este aspecto si bien la implementación cuida aspectos como las multiplicaciones, descuida otros utilizar un tamaño de palabra demasiado grande y no introducir aritmética redundante.

Ventajas:

- Es adaptativo
- Utiliza un tamaño de palabra reducido

Módulo	Sistema de memoria	Actualización del intervalo	Global
Área mm^2)	25.7	5.3	31.0
Frecuencia (MHz)	46	39	39

Tabla 1.2: Resultados de la implementación del codificador que se describe en [POB97].

- El modelo no necesita corregir las probabilidades
- Implica codificación de orden superior sin coste adicional

Inconvenientes:

- Necesita dos ciclos por símbolo
- No consigue reducir la cantidad de datos implicados en las operaciones
- El diseño del descodificador no es lo bastante eficiente

1.4 Una arquitectura multinivel

En esta sección describiremos una arquitectura eficiente para la codificación aritmética multinivel [BBL97] [POB97] (tabla 1.2). Ésta nos servirá para mostrar las dificultades que conlleva la implementación de una arquitectura de estas características y será tomada como punto de partida para los nuevos diseños que se proponen en la presente memoria.

De ella podemos decir que es una implementación con precisión reducida, en la que la iteración sigue las ecuaciones (1.5). Se utiliza aritmética redundante tanto en el modelo como en la iteración, en dos modalidades, acarreo almacenado (carry-save) [Nol91] y dígitos con signo (signed-digit) [Avi61] para las multiplicaciones. Dado el volumen de operaciones aritméticas involucrado, esta solución se hace imprescindible para obtener un ciclo de procesamiento con una longitud razonable.

El modelo es adaptativo, para 256 símbolos si bien es generalizable a otros tamaños, y en él se almacenan las probabilidades de todos los símbolos. Las probabilidades acumulativas se calculan con un esquema híbrido que almacena una pequeña parte y calcula las restantes cuando es necesario. Para asimilar los acarreos se utiliza *bit-stuffing*. Una implementación del descodificador basada en esta arquitectura aparece en [OB97].

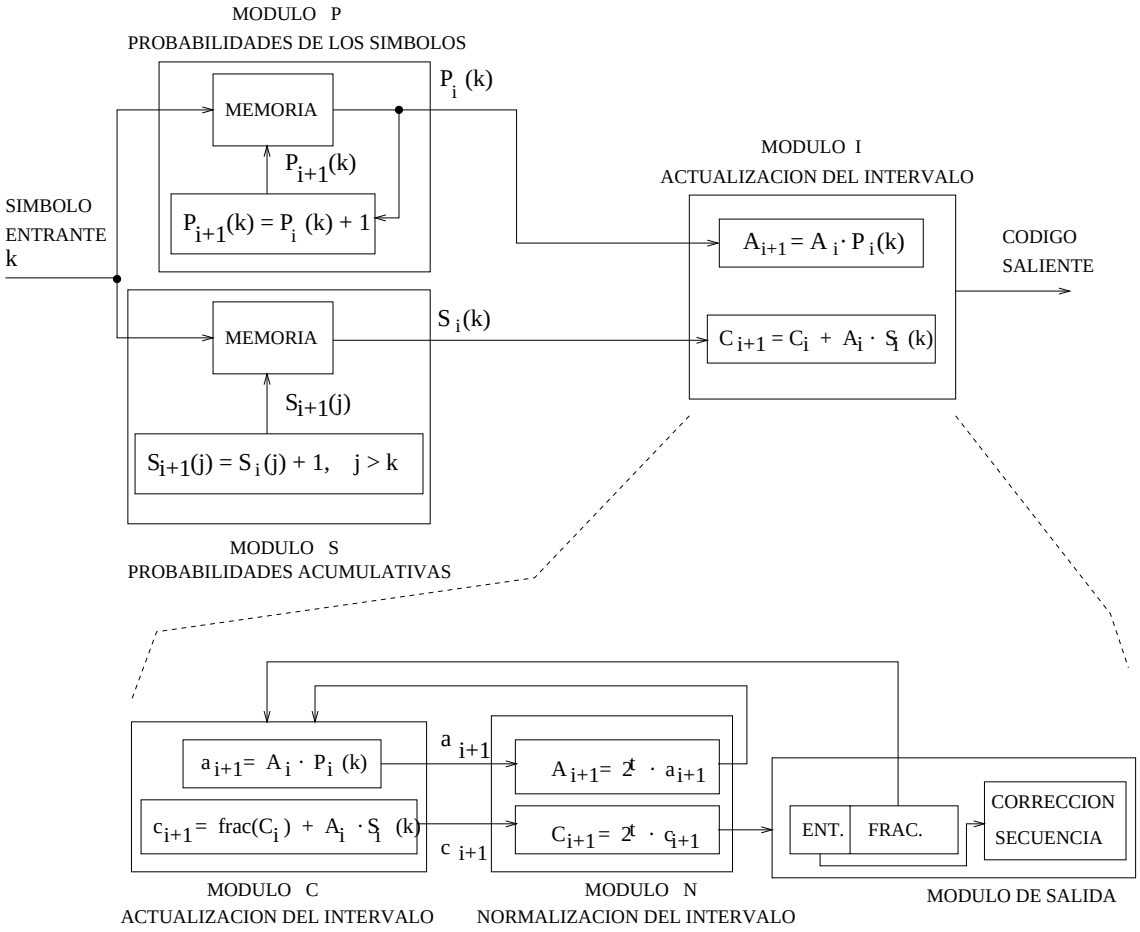


Figura 1.9: Estructura general de un codificador aritmético

En la figura 1.9 se muestra una estructura general de codificador aritmético, en el cual no se aplica ninguna optimización al modelo (módulo P). Las probabilidades y probabilidades acumulativas se almacenan en memoria y se actualizan cada ciclo. El módulo I actualiza el intervalo, dividido en la iteración y normalización de los valores del rango y el punto bajo del intervalo (módulos C u N). Finalmente, el resultado se empaqueta en bytes, y se corrige la propagación de acarreos en el módulo de salida.

1.4.1 Gestión de las probabilidades acumulativas

El almacenamiento y la actualización de las probabilidades acumulativas son operaciones muy complejas en términos de hardware y tiempo dado que es necesario almacenar 255 probabilidades acumulativas y, en el peor caso actualizarlas, todo ello

en un ciclo. Para ello sería necesario utilizar un número elevado de incrementadores.

Para reducir el área de almacenamiento y mejorar el proceso de actualización se ha desarrollado un esquema en el que sólo un 1/16 de las probabilidades acumulativas (que llamaremos de referencia) son almacenadas y actualizadas. En la figura 1.10.(a) se muestra este esquema. Sólo las probabilidades acumulativas de los símbolos $h = 16 \cdot j$, $j = 0, 1, ..., 15$, marcadas como rectángulos sombreados en la figura, son almacenadas. Además, se considera una probabilidad acumulativa adicional $S(256)$ para simplificar la actualización.

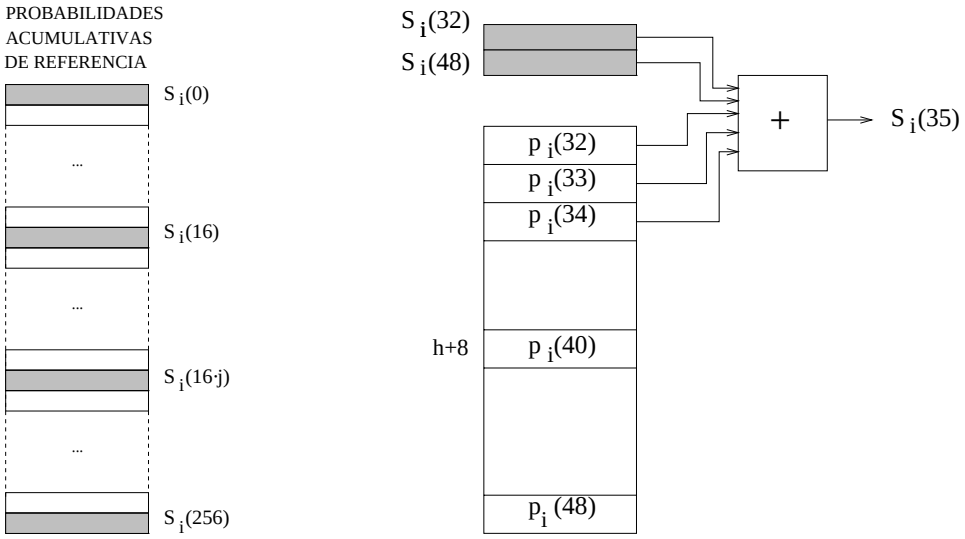
La probabilidad acumulativa del símbolo $S_i(k)$ $(h < k < h + 16)$ es calculada a partir de una probabilidad acumulativa de referencia y un conjunto de 8 probabilidades de símbolos tal y como se indica:

$$S_i(k) = \begin{cases} S_i(h) + p_i(h) + p_i(h+1) + \cdots + p_i(k-1) & \text{si } k < h+8 \\ S_i(h+16) - p_i(h+15) - p_i(h+14) - \cdots - p_i(k) & \text{si } k \geq h+8 \end{cases} \quad (1.8)$$

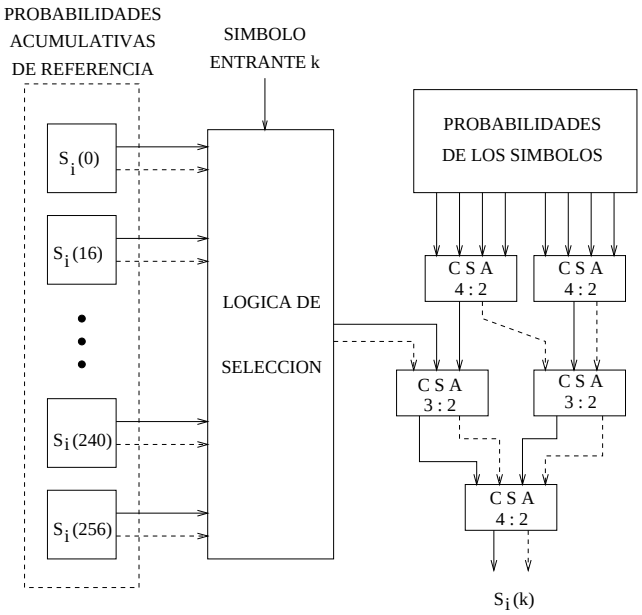
La figura 1.10.(b) ilustra un ejemplo del cálculo de la probabilidad acumulativa del símbolo $k = 35$. En este caso, $h = 32$. Como $k < h+8$, se utilizan la probabilidad acumulativa $S_i(32)$ y las probabilidades $P_i(32)$, $P_i(33)$ y $P_i(34)$.

La figura 1.10.(c) muestra la implementación hardware para computar la probabilidad acumulativa de cualquier símbolo. A lo sumo son necesarias 8 probabilidades de símbolo y una probabilidad acumulativa. Para acelerar la actualización las probabilidades acumulativas de referencia están almacenadas en formato de acarreo almacenado (palabras de semi-suma y acarreo en líneas continua y discontinua respectivamente). Por otra parte, las probabilidades se mantienen en formato no redundante. Además, se incluye lógica de selección para acceder a la probabilidad acumulativa adecuada, $S_i(h)$ o $S_i(h + 16)$. Denotando por $I_7(k), \ldots, I_0(k)$ la representación binaria del símbolo entrante k, los 4 bits más significativos $I_7(k), \ldots, I_4(k)$ seleccionan el valor de h que se ha de utilizar. Además, $I_3(k)$ indica cuando el cálculo de $S_i(k)$ se realiza usando $S_i(h)$ $(I_3(k) = 0)$ o $S_i(h + 16)$ $(I_3(k) = 1)$.

Cada vez que un nuevo símbolo es procesado, las probabilidades acumulativas de referencia han de ser actualizadas. La figura 1.11.(a) muestra la implementación hardware usando 8 sumadores CSA 3:2, tal que en cada CSA se actualizan dos probabilidades acumulativas en el mismo ciclo. Consecuentemente, esta unidad necesita una señal de reloj de frecuencia dos veces la frecuencia del reloj del sistema. Como se verá mas adelante el periodo de reloj es lo suficientemente grande para permitir esta operación. Por otro lado, dado que no es necesario actualizar todas las probabilidades acumulativas de referencia, se ha introducido una lógica de selección (módulo de control) para seleccionar aquellas que se ven afectadas por la actualización. Éstas son aquellas almacenadas en registros cuya dirección es mayor que $I_7(k)I_6(k)I_5(k)I_4(k)0000$.

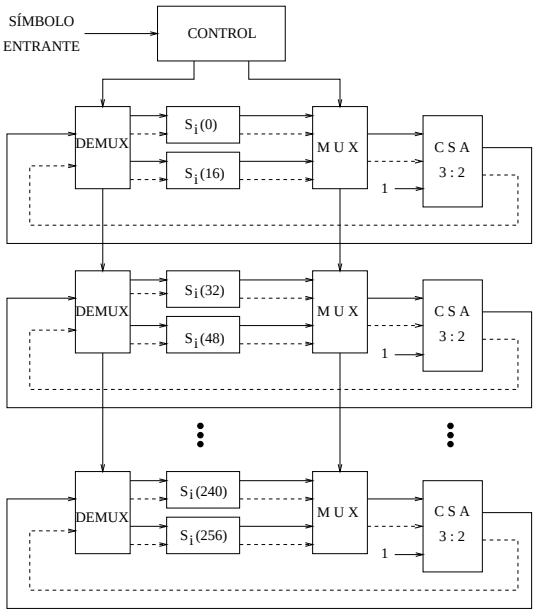


(a) (b)

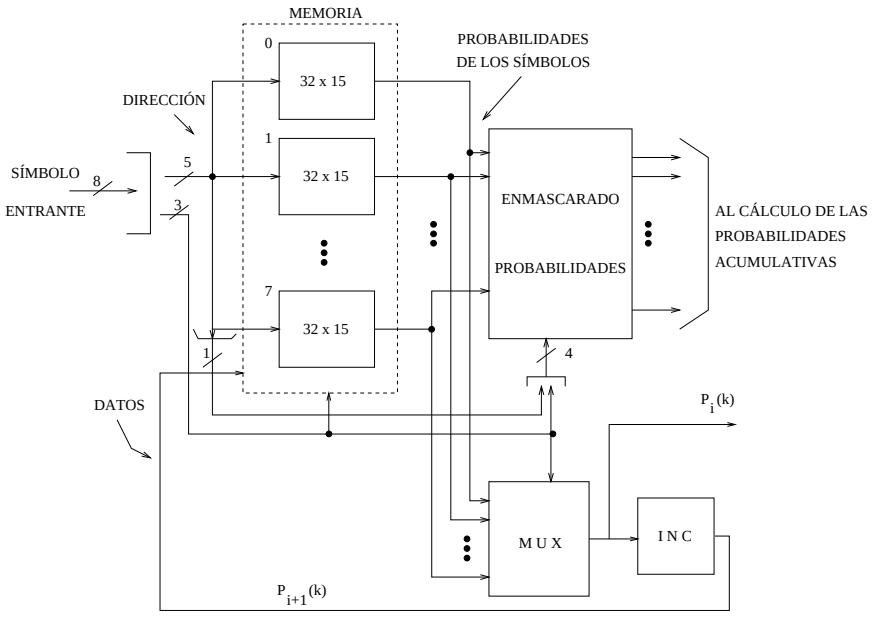


(c)

Figura 1.10: (a) Almacenamiento de las probabilidades acumulativas. (b) Cálculo de la probabilidad acumulativa del símbolo $k = 35$. (c) Cálculo de las probabilidades acumulativas



(a)



(b)

Figura 1.11: (a) Actualización de las probabilidades acumulativas. (b) Actualización de las probabilidades.

1.4.2 Gestión de las probabilidades de los símbolos

Consiste en el almacenamiento de las probabilidades, la lectura de $P_i(k)$ y su actualización, y la lectura de las probabilidades necesarias para calcular la probabilidad acumulativa. Un diagrama de bloques de la implementación se muestra en la figura 1.11.(b).

Las 256 probabilidades están almacenadas en formato no redundante para reducir el espacio de almacenamiento y simplificar la actualización del intervalo. Esta elección no afecta al ciclo de reloj ya que las operaciones con la probabilidad no están en la vía crítica. Ya que se necesitan hasta 8 probabilidades consecutivas para calcular la probabilidad acumulativa, el espacio de almacenamiento se divide en 8 bloques (memoria entrelazada) a los que se accede en paralelo. Se utiliza un multiplexor para seleccionar $P_i(k)$ y el módulo de enmascaramiento cancela las probabilidades que no son necesarias para obtener $S_i(k)$.

Los tres bits más significativos del símbolo entrante especifican el módulo en el que está almacenada la probabilidad del mismo, y los cinco bits menos significativos forman la dirección de cada módulo. Los cuatro menos significativos se usan para controlar el enmascarado de las probabilidades que no se necesitan en el bloque de gestión de las probabilidades acumulativas. La actualización de la probabilidad del símbolo se realiza con un incrementador de acarreo adelantado.

1.4.3 Actualización del intervalo

La actualización del intervalo se realiza usando aritmética de acarreo almacenado. Esta misma solución se ha utilizado recientemente en un codificador aritmético para imágenes binarias [FP95], pero únicamente para actualizar el rango del intervalo, no el punto bajo del mismo. La figura 1.12 muestra el esquema para la actualización del intervalo y su normalización. A_c y A_s representan, respectivamente, las palabras de acarreo y pseudo-suma del rango. La misma notación se utiliza para la representación carry-save del punto bajo del intervalo C y la probabilidad acumulativa $S_i(k)$.

Para simplificar las operaciones de multiplicación (ver figura 1.9), se propone un nuevo esquema: el multiplicador A_i se recodifica a representación signed-digit (dígitos con signo) radix-4, con dígitos dentro del conjunto -2, -1, 0, 1, 2 y se trunca a un número pequeño de dígitos. Por ello el número de sumas se reduce significativamente y la compresión se ve afectada sólo ligeramente. La selección del número de dígitos establece un compromiso entre la relación de compresión y la complejidad hardware. De esta manera, simulando el algoritmo de codificación sobre un conjunto de imágenes, se ha encontrado que utilizando un dígito entero y 2 de la parte fraccional la pérdida en compresión está alrededor del 1% con respecto

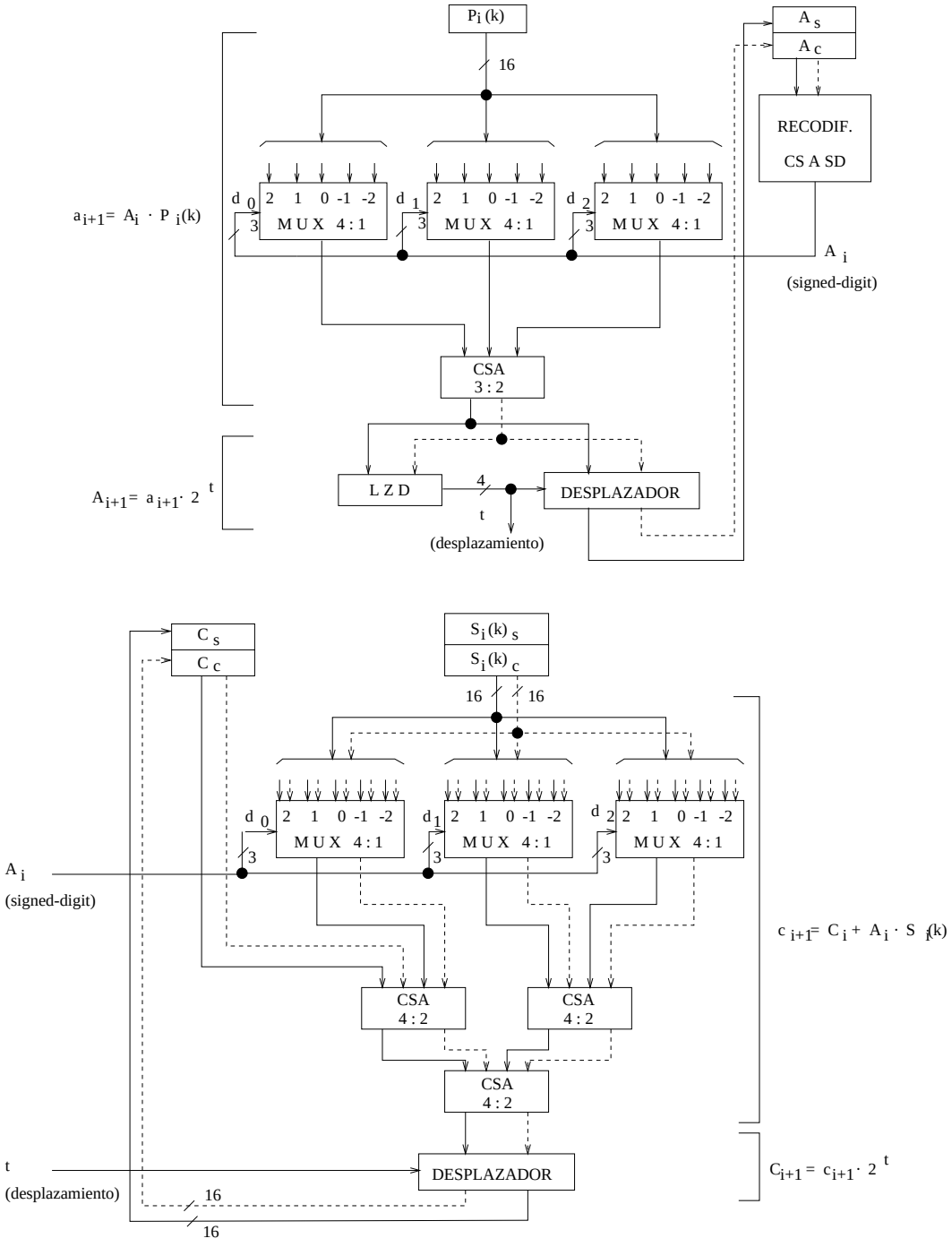


Figura 1.12: Actualización del intervalo

a utilizar todos los dígitos.

Como se ve en la figura, el rango A_i y el punto bajo C_i se mantienen en formato de acarreo almacenado durante todo el proceso. De esta manera la actualización del intervalo se acelera dado que se evita la propagación del acarreo en las sumas.

Una vez que el rango ha sido actualizado, es preciso normalizarlo. La normalización consiste en un desplazamiento a la izquierda tal que el primer dígito no nulo (en la palabra de suma o la de acarreo) esté situado en la posición más a la izquierda. El número de dígitos nulo t se obtiene con un LZD (leading zero detector) [Okl94] que detecta el número de ceros que preceden al primer dígito no nulo. Como se utiliza representación de acarreo almacenado, el rango normalizado estará en el intervalo [0.5, 2) ya que el valor global es la suma de las dos palabras de pseudo-suma y acarreo. El mismo desplazamiento se realiza sobre el punto bajo del intervalo.

1.4.4 Salida del codificador

Una vez que el intervalo ha sido normalizado, los bits de la parte entera del punto bajo del rango son convertidos a formato no redundante y se incorporan a la secuencia codificada, mientras que las partes fraccionales del rango y el punto bajo del intervalo se mantienen en formato redundante para procesar el siguiente símbolo.

La secuencia de salida ha de ser corregida para evitar la propagación de un acarreo a la parte que ya se ha transmitido. Para ello se implementa la técnica de *bit-stuffing*, que consiste en insertar bits adicionales para amortiguar la propagación de acarreos cuando un byte del código tiene todos sus bits a '1' (FF hexadecimal) [LR81]. La figura 1.13 ilustra la implementación de esta técnica. Los bits que son desplazados, durante la normalización, a la izquierda del registro que contiene el punto bajo del intervalo son introducidos en una prolongación del registro. Dado que el máximo desplazamiento posible es de 15 bits, se necesitan 2 bytes para acoger estos bits. Además, durante la actualización del intervalo, y antes de la normalización, se puede producir hasta 2 bits de acarreo, y estos han de ser considerados en la asimilación. Para realizar la asimilación se utiliza un sumador rápido (CLA) que suma 9 o 17 bits de la extensión del registro, dependiendo de si se asimilan 1 o 2 bytes. El resultado no redundante de la asimilación se almacena en el registro llamado *code byte*. El bit de acarreo y el bit más significativo de la asimilación se usan para corregir el *code byte* asimilado el ciclo anterior. De esta manera, se puede producir la salida de la codificación en formato no redundante. El *bit-stuffing* detecta la posibilidad de que un acarreo provoque desbordamiento. Cuando esta situación se produce, se transmite el *code byte*, y en el siguiente ciclo se inserta un bit con el acarreo que se produzca. En el descodificador, se detecta la situación, y se lee el bit de stuffing para sumarlo al último byte leído.

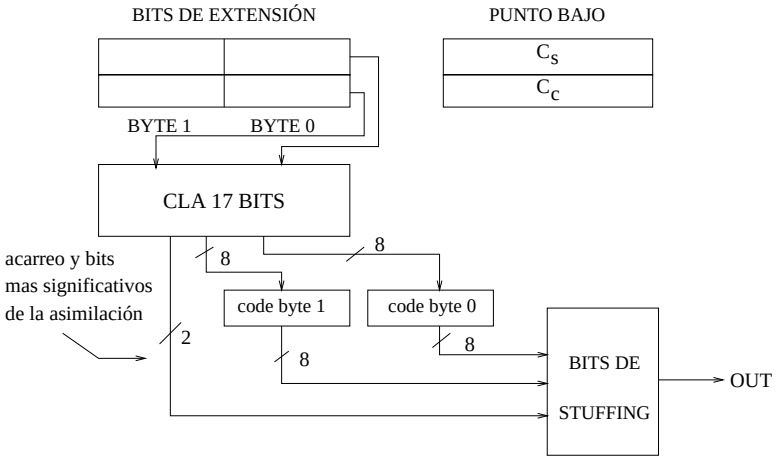


Figura 1.13: Implementación de la técnica de bit-stuffing

1.4.5 El descodificador

Un diseño de descodificador basado en la arquitectura que estamos describiendo se presenta en [OB97]. Aunque conceptualmente la descodificación es un espejo de la codificación, la implementación práctica es considerablemente más compleja. El objetivo es conocer que símbolo verifica la condición de la ecuación (1.3). La arquitectura, que se muestra en la figura 1.14, está dividida en tres partes. En la primera se realiza la comparación en paralelo del punto bajo del rango, C_i con los productos $A'_i \cdot S_i(h)$ para cada probabilidad acumulativa de referencia, siendo A'_i igual a A_i en formato signed-digit. Los productos se calculan en paralelo, utilizando 16 multiplicadores simplificados con sólo tres dígitos radix-4 del multiplicador A_i. La comparación se realiza como una resta seguida de un control de los signos. De esta forma se selecciona un rango de 16 símbolos como posibles candidatos, comprendidos entre dos probabilidades acumulativas de referencia. Las probabilidades acumulativas de referencia situadas por encima del rango seleccionado son incrementadas para actualizarlas. Al la siguiente etapa se pasan las dos S_i de referencia necearias para calcular las S_i de los 16 candidatos.

A continuación se compara C_i en un segundo nivel con los 16 candidatos. Las probabilidades acumulativas de estos se calculan como en el codificador. Para esto se utilizan las probabilidades acumulativas seleccionadas en la etapa anterior y las probabilidades almacenadas en la RAM. Las operaciones se realizan con una árbol de sumadores. Las comparaciones se implementan, al igual que en la primera etapa, restando los productos $A_i \cdot S_i$ (ver ecuación 1.8) obtenidos en paralelo del valor de C_i. Una vez la comparación ha finalizado se conocen el símbolo, su probabilidad y su probabilidad acumulativa. El primero es el resultado del proceso de descodificación.

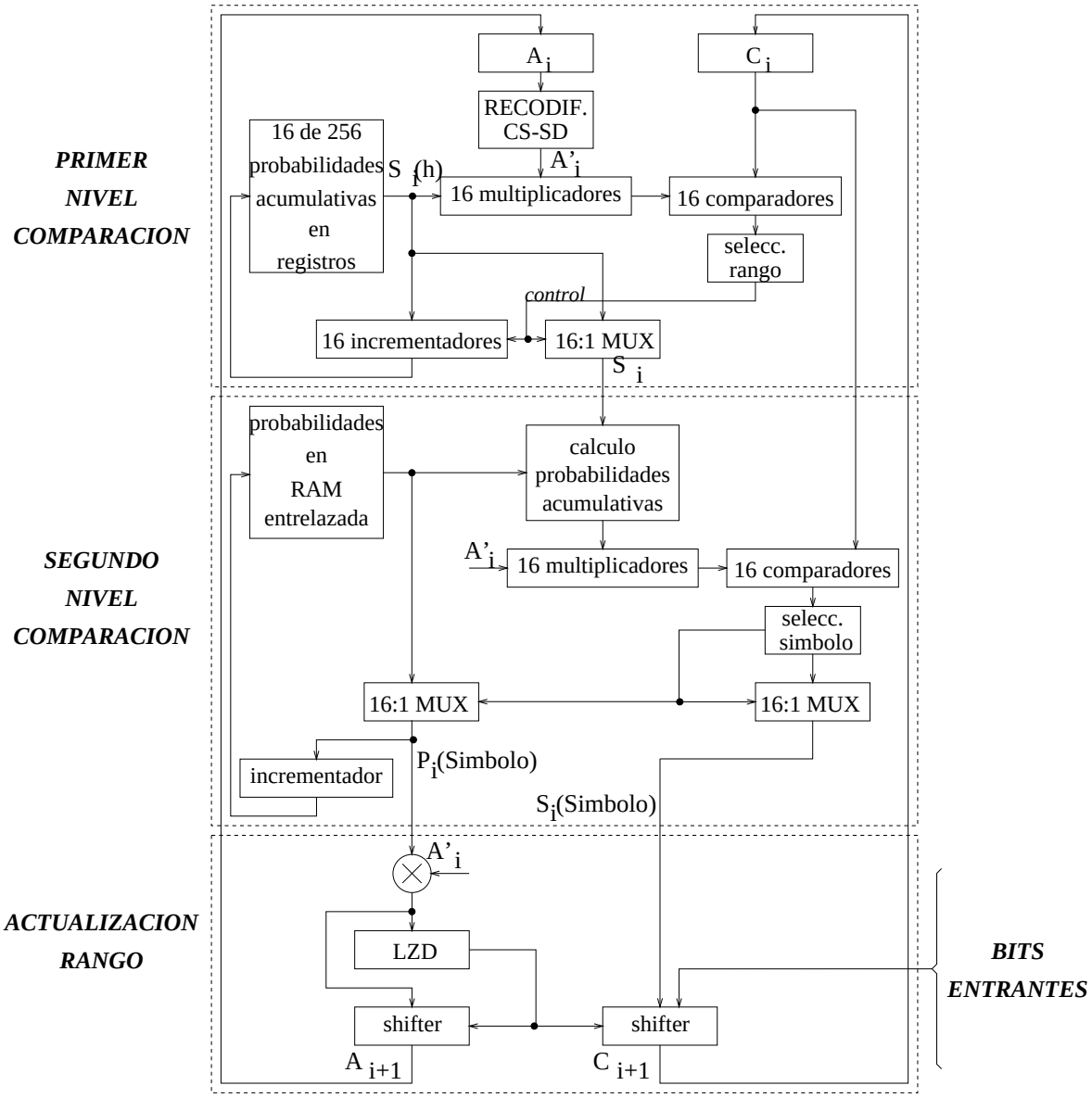


Figura 1.14: Estructura del descodificador.

En la tercera parte se aplican las ecuaciones (1.6) de forma simétrica a como se hizo en el codificador. Como resultado se obtienen los nuevos valores C_i y A_i convenientemente normalizados. Desde la corriente de entrada se introducen en C_i la misma cantidad de bits que es necesario desplazar el rango del intervalo para normalizarlo. Esta última operación es una concatenación trivial que se realiza en el mismo normalizador.

La principal diferencia entre la arquitectura del codificador y el descodificador reside en que ahora el acceso a las tablas de las probabilidades se realiza dentro de la recurrencia. Esta operación de acceso es muy costosa en tiempo, ya que implica leer y modificar el contenido de una memoria RAM. Por otro lado, es imposible comenzar una nueva iteración sin haber completado la anterior, ya que es necesaria información que sólo se conoce al final de la iteración previa. Así, el ciclo de reloj es considerablemente más largo que en el codificador.

Las ventajas de este esquema residen en que la descodificación se hace en sólo dos pasos. El paralelismo que se introduce tiene un coste reducido, ya que en cada nivel sólo se utilizan 16 multiplicadores y comparadores. Así mismo, el uso de aritmética redundante reduce el coste de las multiplicaciones sin perder precisión.

1.4.6 Segmentación

Las diferentes etapas del codificador pueden ser separadas mediante registros para disminuir la duración del ciclo, pero existe la limitación de que no es posible segmentar la parte recursiva del algoritmo, esto es la iteración. Esto afecta especialmente al descodificador, ya que todo el procesamiento es recursivo, al contrario que en el codificador en el que la iteración sólo es una parte.

La única posibilidad de superar esta restricción es modificar la salida del codificador y la entrada del descodificador. Si asumimos que el codificador genere dos secuencias de salida, y el descodificador acepte dos corrientes de entrada, se puede reducir la duración del ciclo de reloj segmentando el hardware. En la figura 1.15 se muestra como un conjunto de datos es dividido en dos partes y codificado alternando el hardware del codificador. Se producen dos corrientes de salida que se transmiten al descodificador que recupera los datos originales.

La forma de conseguir esto sin grandes modificaciones se describe en [OB97], donde se consigue una mejora importante, sobre todo desde el punto de vista del descodificador. Los esquemas segmentados del codificador y el descodificador se muestran en la figura 1.16. Se puede llegar a reducir el tiempo necesario para completar la iteración en el codificador hasta la mitad. Basta para ello con situar los registros de segmentación convenientemente. Durante cada ciclo cada parte del hardware está siendo utilizada por una ejecución distinta del algoritmo. Habrá entonces unos A_i^a, C_i^a y unos A_i^b, C_i^b. Lo mismo sucede con el descodificador, en

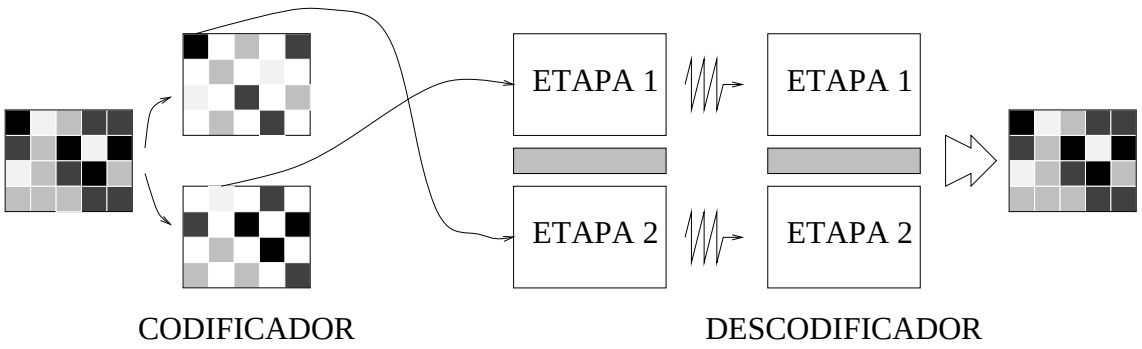


Figura 1.15: Ejemplo de segmentación.

	Codificador	**Descodificador**
No segmentado	39	141
Segmentado	22	93

Tabla 1.3: Comparación de tiempos para el codificador y el descodificador. Versiones básica y segmentada.

el que se redistribuyen los elementos intentando que los tiempos de computación de las 2 etapas sean los más parejos posible. Además es necesario cumplir con la condición de que las probabilidades estén actualizadas en todo momento. Por ello se actualizan las probabilidades acumulativas de referencia en el mismo ciclo en que son utilizadas, y se procede de igual manera con las probabilidades almacenadas en la memoria.

Los tiempos de computación estimados para las iteraciones del codificador y el descodificador en sus versiones básica y segmentada se muestran en la tabla 1.3. Las unidades son retardos de puertas *nand* de dos entradas con una carga de tres puertas del mismo tipo a la salida [Str92].

Los resultados obtenidos con estas implementaciones han de ser mejorados, llegando a arquitecturas más rápidas, sencillas y eficientes. Este será el objetivo que perseguiremos en los siguientes capítulos. Reducir el tiempo de acceso y actualización del modelo es parte importante de esta tarea, especialmente en lo que compete al descodificador. Por otro lado buscaremos también la forma de simplificar la iteración, de lo que se beneficiaran tanto el codificador como el descodificador.

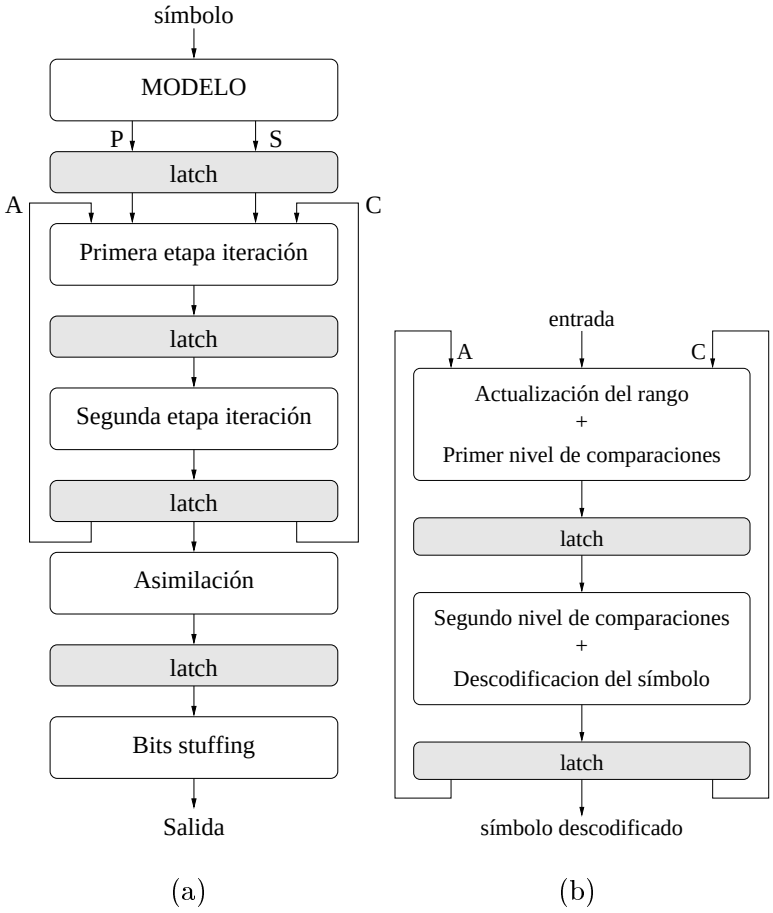


Figura 1.16: (a) Codificador segmentado. (b) Descodificador segmentado.

1.5 Aplicaciones

Las técnicas de compresión de datos han encontrado aplicación en casi todos los campos de la informática y comunicaciones. Existen compresores de uso general muy extendidos en informática personal. Programas como *PKARC* y *compress* utilizan variaciones del algoritmo de Lempel y Ziv que es el más popular en este tipo de compresores. Otros como *compact*, utilidad de UNIX, implementan el algoritmo Huffman adaptativo FGK. Estas son aplicaciones para compresión sin pérdidas de textos, archivos binarios, comunicaciones en redes de área local, etc.

El campo en el que tiene mayor aplicación la compresión es el multimedia (JPEG [Gro94], MPEG [Gro93], AC3 [MH95]). Éste no es tan solo un caso particular de compresión de datos, sino que posee técnicas propias que son las que realmente

permiten obtener altas relaciones de compresión. El tema que surge inmediatamente al hablar de compresión multimedia es la distinción entre los términos con o sin pérdidas. Ambos coinciden en el último paso, la compresión entrópica de una serie de coeficientes que representan la imagen, que es el tema que nos ocupa. Sin embargo, ningún compresor eficiente puede dejar de lado la naturaleza de los datos que ha de comprimir. Cada tipo tiene unas características de las que se puede sacar partido para mejorar la compresión. Es importante conocer si el modelo óptimo es dinámico o estático, la forma del histograma, la existencia de redundancia de orden superior que no ha sido eliminada por las etapas anteriores, etc.

De las imágenes y sonidos digitales se suele decir que todas tienen un cierto nivel de pérdidas derivadas de los procesos de adquisición y digitalización. En ocasiones se apunta que las imágenes de alta calidad (12 bits por componente y pixel o más) contienen una componente de ruido del orden de la precisión que se pretende tener. En cualquier caso, la compresión sin pérdidas es demandada por muchas aplicaciones, como son el almacenamiento y transmisión de imágenes de satélite y médicas, el almacenamiento de datos destinados a edición o a ser procesados en plataformas sin capacidad de descompresión, imágenes de color reducido, etc.

La compresión con pérdidas cubre el resto de las aplicaciones posibles, imagen, sonido y vídeo digital de consumo en todas sus variedades. La pérdida de calidad es graduable, y en ocasiones es indistinguible del original permitiendo sin embargo una relación de compresión muy superior. Sin embargo, pueden producirse deformaciones en las imágenes (artifacts) o aparición de sonidos molestos que son inaceptables. En el caso de edición, continuas compresiones y descompresiones conducen a una paulatina degradación de la calidad.

Algunos de los métodos más extendidos utilizan el método de Huffman, sin embargo las nuevas tecnologías implementan métodos más eficientes o bien más sencillos. El estándar para compresión de imágenes fotográficas JPEG [Gro94] utiliza Huffman o codificación aritmética, si bien esta última está sujeta a patentes de IBM. También se utiliza Huffman en compresión de vídeo MPEG [Gro93], aunque la última versión del estándar, MPEG4 prefiere codificación aritmética.

Capítulo 2

Algoritmo con memoria cache

En este capítulo desarrollamos un nuevo modelo para la compresión aritmética multinivel, que supera en gran medida muchas de las limitaciones de los modelos presentados hasta ahora en la literatura y revisados en el capítulo 1.

Introduciremos un nuevo nivel en la jerarquía de memoria, una pequeña memoria cache. Esta solución es novedosa y se ajusta a nuestros requerimientos en la búsqueda de una arquitectura de alta velocidad y bajo coste que no introduzca modificaciones en el algoritmo tales que la relación de compresión se vea afectada de forma negativa. Veremos como estos objetivos se cumplen dando lugar a una familia de codificadores-descodificadores orientados a distintas aplicaciones, adaptándose a cada una de ellas y optimizando en cada caso aspectos particulares.

2.1 Limitaciones e inconvenientes de una arquitectura convencional

Si entendemos por convencional una arquitectura que es una traslación directa o casi directa del algoritmo descrito en [WNC87], entonces nos encontramos ante implementaciones que ocupan un área considerable y que tienen una velocidad de procesamiento baja y fuertemente comprometida con el área.

Como punto de partida tomaremos la arquitectura descrita en la sección 1.4 e iremos modificando los elementos que suponen un coste excesivo en el codificador o en el descodificador. Esta arquitectura se basa en un modelo que almacena las probabilidades en RAM y calcula las probabilidades acumulativas a partir de éstas y de un conjunto de probabilidades acumulativas de referencia precalculadas. Este sistema híbrido es razonablemente rápido pero tiene serias limitaciones de velocidad y no resuelve los problemas de complejidad en la descodificación y corrección del modelo.

El codificador se puede separar directamente en tres partes: el modelo, la iteración y la etapa de salida. Esta última puede ser relegada ya que no influye en el ciclo de reloj. La iteración es más lenta que el acceso al modelo, pero se puede segmentar en caso de que tener dos secuencias de salida no sea un problema.

Por último, la etapa del modelo contiene acceso a memoria RAM y operaciones de actualización de las probabilidades. En principio la duración mínima estaría fijada por un ciclo de lectura-modificación-escritura en la memoria, sin embargo buscaremos soluciones para reducir el tiempo de acceso.

Desde el punto de vista del descodificador, al acceso al modelo se encuentra ligado a la iteración, de forma que la reducción del ciclo de procesamiento puede venir de mejoras en cualquiera de las dos partes. La segmentación reduce la duración del ciclo, pero no es la solución definitiva en una arquitectura de gran complejidad. Al igual que en codificador el ciclo de memoria es largo, pero el verdadero responsable de la lentitud de la arquitectura es la búsqueda del símbolo. Son necesarios dos niveles de comparaciones, que es una solución costosa en tiempo y área pero preferible a primar la velocidad a costa de aumentar el área o viceversa.

En principio el codificador y el descodificador necesitan soluciones distintas para sus problemas. El codificador necesita un acceso a memoria más rápido, mientras que para el descodificador esto no es tan importante, sino que sería preferible una estrategia de búsqueda menos costosa. En las siguientes secciones veremos como una modificación en el modelo nos permite solucionar ambos problemas.

2.2 Modelo con cache

Dada la gran cantidad de información que es necesario almacenar en un modelo multinivel con un alfabeto grande (probabilidades y probabilidades acumulativas), el concurso de memoria RAM de bajo coste es imprescindible. Ésta tiene la ventaja adicional de incorporar hardware de direccionamiento eficiente, por lo que una solución basada en utilizar registros no sería competitiva. Éstos tienen otras ventajas, como son la velocidad y la versatilidad. Leer y escribir en un registro en el mismo ciclo no tiene ningún coste adicional. Por tanto, una implementación del modelo que nos permitiese eliminar el acceso a la memoria RAM de la vía crítica de la arquitectura nos llevaría a un ciclo de reloj más reducido. Por otro lado no es posible prescindir de ella, ya que su densidad de almacenamiento es muy superior a la que podemos obtener con registros.

En el descodificador la solución ideal sería encontrar una estrategia eficiente para la búsqueda dentro de las probabilidades de todos los símbolos. Pero ésta está ya muy optimizada, es una sencilla búsqueda jerárquica dentro de una lista ordenada. Dado que el modelo es adaptativo y cambiante, y que la descodificación

no consiste en la encontrar un valor exacto sino en una acotación dentro de un rango, la estrategia conocida como *hashing* [Knu98] no es aplicable a este caso. Por tanto, la única solución para reducir las comparaciones pasa por reducir el número de símbolos involucrados en ellas.

Las consideraciones hechas en los párrafos anteriores: memoria rápida y disminución del número de datos a considerar son características de las memorias cache utilizadas en los microprocesadores actuales [PH94] [Han93]. La idea de introducir una memoria cache para gestionar el modelo es atractiva siempre y cuando seamos capaces de implementarla de forma eficiente y la compresión no se deteriore.

En caso de introducir una memoria cache las probabilidades y probabilidades acumulativas se calcularían utilizando únicamente la cache, y el ciclo de memoria sería más corto que el de una memoria RAM. Habría sin embargo que introducir un mecanismo para gestionar los reemplazos en la cache, resolviendo dos problemas: codificar el reemplazo y efectuarlo.

2.2.1 Estrategia para la cache

Serán necesarias unas disposiciones mínimas para definir el comportamiento de la cache antes de afrontar la tarea de encontrar la configuración óptima.

Haremos primeramente un glosario de términos de uso común en memorias cache [PH94]

- línea: es la unidad mínima que se puede direccionar en una memoria cache. Cada línea puede contener una cantidad de datos que depende de la implementación.

- tamaño de línea: es la cantidad de datos contenidos en cada línea de la cache. La cache se basa en el principio de localidad temporal para suponer que los datos contenidos en ella serán referenciados nuevamente en un futuro próximo. Además, dado el principio de localidad espacial, se supone que un entorno espacial de los datos referenciados será requerido también.

- etiqueta (tag): es el elemento que permita identificar el contenido de una línea. La cache se direcciona dando como argumento una etiqueta, y se obtiene el acceso a los datos de la línea a la que está asociada esa etiqueta. La etiqueta está asociada a una porción de la memoria global, no a la línea física de la cache ni al valor particular de los datos en un momento dado.

- acierto (hit): se produce cuando al direccionar la cache se encuentra una línea con la etiqueta requerida.

- fallo (miss): es el caso contrario al acierto. Dada que la cache es un subconjunto de la memoria global, no todos los datos están contenidos en ella, y es posible que ninguna línea presente la etiqueta requerida.

- reemplazo: es la operación de sustitución de un símbolo de la cache por otro que viene de otro nivel de la jerarquía de memoria.

- política de una cache: es el conjunto de reglas que rigen la disposición de los datos en la cache, decidiendo en que lugar o lugares puede ser alojado un dato.

- algoritmos de reemplazo: son las normas que determinan las condiciones bajo las cuales se producen reemplazos y que línea será eliminada de la cache para alojar una nueva. De ellos y de la política escogida dependen en gran medida las prestaciones de la cache.

- tasa de aciertos (hit-ratio): es el porcentaje de aciertos en la cache sobre el número de referencias efectuadas. Una gestión de reemplazos estudiada y compleja puede ayudar a mejorarla.

- tiempo por acierto (hit time): es el tiempo necesario para encontrar un dato y disponer de él cuando se produce un acierto. Cuando la gestión es sencilla este tiempo se reduce mientras que aumenta para gestiones complejas. El tiempo por acierto y la tasa de aciertos son magnitudes enfrentadas si bien ciertas implementaciones intentan compatibilizarlas.

- penalización por fallo: es el tiempo necesario para obtener un dato que no está presente en la cache. En una jerarquía de memoria con muchos niveles dependerá del nivel en que se encuentre.

- asignación directa (direct-mapped): política de reemplazo consistente en establecer un criterio que asigna a cada dirección de la memoria global un espacio fijo dentro de la cache. Localizar un dato en la cache es entonces una operación trivial, similar a localizar un dato en una memoria de acceso aleatorio.

- asociatividad total (fully-associativity): política de reemplazo que permite que un dato se aloje en cualquier línea de la cache. La localización del dato en la cache necesitará entonces una búsqueda exhaustiva entre los tags de todas las líneas.

- asociatividad por conjuntos (set-associativity): híbrida de las dos anteriores. A cada dirección global le corresponde un conjunto de líneas de la cache que a su vez es totalmente asociativa.

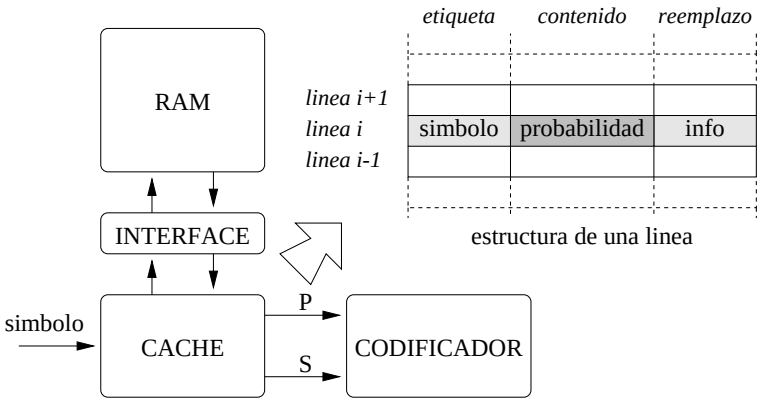


Figura 2.1: Esquema general de la cache: su situación dentro del codificador y la estructura de una línea.

- LRU (Least Recently Used): es uno de los algoritmos de reemplazo más utilizados. La línea que es expulsada de la cache es aquella que fue referenciada hace más tiempo. Se supone que ciertas líneas son utilizadas mucho durante un tiempo y paulatinamente caen en desuso.

- FIFO (first in first out): es una estrategia de colas que también se utiliza en caches como algoritmo de reemplazo. Se elimina la línea más antigua de la cache. No importa si sigue en uso o no. Se supone que las líneas están en uso un tiempo tras ser referenciadas, y que este tiempo es aproximadamente igual para todos los símbolos.

La cache será pequeña, ya que de lo contrario no sería competitiva con la implementación que hemos tomado como base. Consideraremos inicialmente tamaños de 8, 16 y 32 líneas, potencias de 2 por simplicidad. Cada línea de la cache contendrá un sólo símbolo, si bien consideraremos más adelante implementaciones con más de un símbolo por línea. La probabilidad acumulativa de cada símbolo se calculará cuando su valor sea necesario, utilizando únicamente los símbolos de la cache. Su valor no estará precalculado por interferir esta estrategia con los reemplazos. Dado que una de nuestras premisas es la velocidad de procesamiento todos los símbolos serán codificados en un único ciclo. Un esquema general de la cache sería el que se muestra en la figura 2.1. La cache se sitúa entre la memoria principal y el codificador (o el descodificador) facilitando una probabilidad y una probabilidad acumulativa para cada símbolo. La cache estará formada por un conjunto de líneas, cada una de las cuales almacena la probabilidad de un símbolo e información para el funcionamiento de la cache: una etiqueta e información para los reemplazos.

2.2.2 Codificación de los fallos

El siguiente problema que hemos de considerar es cómo se codificarán los fallos. Es necesario encontrar una estrategia que permita informar al descodificador que un símbolo ha sido sustituido por otro en la cache de forma unívoca sin que esto suponga una penalización en compresión. Dado que el codificador y el descodificador mantendrán en todo momento el mismo modelo, y que el algoritmo de reemplazo será totalmente determinístico, bastará con codificar el símbolo que se incorpora a la cache y el que será sustituido se dará por supuesto.

Con estos elementos, y sin definir todavía cual será el algoritmo de reemplazo de la cache, sabemos que el codificador debe emitir un mensaje al descodificador cada vez que se produzca un fallo en la cache. Este mensaje debería integrarse dentro de la secuencia codificada, lo cual no es sencillo habida cuenta de que la codificación aritmética no genera un código de longitud entera para cada símbolo que procesa. Así pues, la solución más inmediata es crear un símbolo especial que sirva para codificar los fallos, asignarle una probabilidad y una probabilidad acumulativa y codificarlo como un símbolo normal. Sin embargo, sería necesario codificar primero el símbolo de fallo, y a continuación especificar el símbolo particular que lo ha provocado, con lo cual se vuelve al mismo problema.

La solución que proponemos es utilizar tantos símbolos para especificar fallos como símbolos hay en el alfabeto. Cuando un símbolo k es referenciado y no está en la cache, su símbolo gemelo k' es codificado en su lugar y así el descodificador será informado del suceso. Esto no supone duplicar el tamaño del alfabeto ya que utilizaremos una forma especial de tratar el alfabeto gemelo.

Los fallos que se produzcan serán codificados por medio de lo llamaremos **tabla virtual** y cuya estructura se muestra en la figura 2.2. Consiste en una tabla que contiene los símbolos del alfabeto gemelo, los cuales tienen una probabilidad constante común para todos ellos. Por tanto, no es necesario almacenar esta tabla físicamente ya que toda la información que contiene es conocida. Llamemos P_{fallo} a la probabilidad asignada a los símbolos de la tabla virtual, entonces el nuevo modelo estará compuesto por 256 símbolos (para un alfabeto de ese tamaño) de la tabla virtual más tantos símbolos como contenga la cache. La probabilidad de los primeros será P_{fallo} y la de los segundos la almacenada en la cache. A la hora de calcular la probabilidad acumulativa es necesario decidir una posición para los símbolos. Pondremos en la base la tabla virtual y la cache inmediatamente a continuación. Así para un fallo provocado por el símbolo k

$$\begin{aligned} P_i(k) &= P_{fallo} \\ S_i(k) &= k \cdot P_{fallo} \end{aligned} \tag{2.1}$$

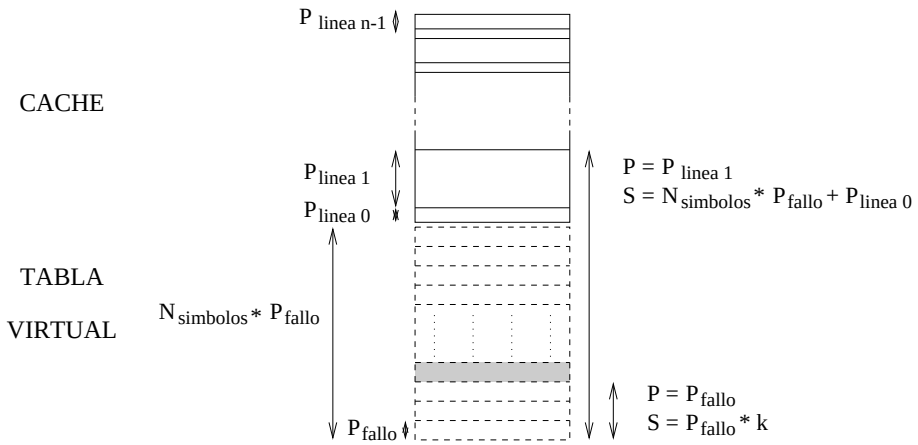


Figura 2.2: Estructura de la jerarquía de memoria incluyendo la tabla virtual

y para un acierto en la línea j, siendo N el número de símbolos del alfabeto

$$\begin{aligned} P_i(k) &= P_{cache}(j) \\ S_i(k) &= N \cdot P_{fallo} + \sum_{i=0}^{i<j} P_{cache}(i) \end{aligned} \tag{2.2}$$

Las operaciones involucradas son triviales. Escogiendo P_{fallo} igual a una potencia de 2 los productos se convierten en un simple desplazamiento, y la cantidad de sumas para obtener $S_i(k)$ es pequeña si la cache lo es también.

La descodificación también es sencilla. Dado un valor de C_i, se compara con $N \cdot P_{fallo}$, la probabilidad acumulativa del primer elemento de la cache. Si el valor de C_i es más pequeño, entonces a ocurrido un fallo, y el símbolo k que lo ha provocado se puede obtener como

$$k = \lfloor \frac{C_i}{A_i} \cdot \frac{1}{P_{fallo}} \rfloor \tag{2.3}$$

Vemos que es necesario dividir el valor de C_i entre A_i para deshacer el producto de la iteración del codificador. Acerca de este punto volveremos más adelante en la sección 2.9.1. Por otro lado el cociente entre P_{fallo} es trivial si se utilizan potencias de 2.

En caso de que C_i sea mayor o igual a $N \cdot P_{fallo}$ se compara con las probabilidades acumulativas de los símbolos de la cache para determinar cual fue codificado. El número de comparaciones es claramente inferior al caso en que la búsqueda se ha de efectuar sobre el alfabeto completo, pudiendo incluso hacerse en un solo nivel, de forma que el ciclo del descodificador se hará notablemente más corto.

CODIFICACIÓN

```
1   iniciar_modelo()
2   iniciar_salida()

3   lazo_principal{

4     leer_simbolo()
5     if(acierto en linea i){
6       Prob=P(linea i)
7       S=N_simb *P_fallo + Σ_{j=0}^{i-1} P(j)
8       actualizar_modelo()
      }
      else{
9       Prob=P_fallo
10      S=k*P_fallo
11      reemplazo()
      }

12    if(tabla satura)
13      escalar_probabilidades()

14    actualizar_intervalo()
15    salida_bits()
    }
16  terminar_salida()
```

(a)

DESCODIFICACIÓN

```
1   iniciar_modelo()

2   lazo_principal{
3     switch(parte_baja_del_rango){
4       if(es un fallo){
5         k=descodificar_simbolo()
6         Prob=P_fallo
7         S=k*P_fallo
8         reemplazo()
        }
        else{
9         i=buscar_en_la_cache(C_i)
10        Prob=P(linea i)
11        S=N_simb *P_fallo + Σ_{j=0}^{i-1} P(j)
12        actualizar_modelo()
        }

13    if(tabla_satura)
14      escalar_probabilidades()

15    actualizar_intervalo()
16    leer_nuevos_bits()
    }
```

(b)

Figura 2.3: Algoritmos de (a) codificación y (b)descodificación

Los algoritmos de codificación y descodificación se pueden resumir tal y como se indica a continuación en la figura 2.3. La principal diferencia con el algoritmo convencional reside en los distintos tratamientos para fallos y aciertos. Las restantes operaciones con el modelo: actualización y escalamiento de las probabilidades también se simplifican gracias a la cache tal y como veremos. La actualización del intervalo no se ve afectada directamente por el modelo, ya que se le limita a procesar los datos que le facilita ésta, pero también llegaremos a importantes reducciones en la complejidad.

La relación entre la memoria principal y la cache influirá forzosamente en las prestaciones de este esquema, pero este es un tema que dejaremos de lado por el momento hasta que tengamos una política de reemplazos adecuada en funcionamiento. Mientras tanto consideraremos que al producirse un fallo se inician las acciones de reemplazo de forma que al siguiente ciclo se haya producido la sustitución. En posteriores capítulos veremos en detalle el tratamiento de los reemplazos.

2.3 El algoritmo de reemplazo

El diseño de memorias cache es una línea de investigación importante en el diseño de computadoras. En los últimos años han sido muchas las novedades introducidas, y también ha crecido su importancia dentro de los microprocesadores, hasta el punto de llegar a ocupar más del 50% del área del chip. La caches sencillas de las primeras implementaciones han sido sustituidas por jerarquías en dos niveles, caches de múltiple acceso, caches separadas para código y datos y otras muchas mejoras.

A la hora de acotar cuales de las alternativas disponibles son más adecuadas para nuestro caso debemos considerar las características que ha de tener el modelo final. Trabajamos con una arquitectura de propósito específico, con la que esperamos procesar un símbolo por cada ciclo. El reemplazo ha de ser totalmente determinístico, de forma que su comportamiento pueda ser fielmente reproducido en el descodificador. La duración del ciclo será muy reducida. En este punto es importante resaltar que con el sistema de codificación de fallos que proponemos la penalización por fallo no toma la forma de consumo de tiempo, sino de degradación de la relación de compresión. En la cache de un procesador, los símbolos pueden ser accedidos en sólo una fracción del tiempo que se necesitaría para buscarlos en memoria, sin embargo en nuestro caso la diferencia no es tan acusada y no se justifican políticas de reemplazo excesivamente complejas.

A la vista de estas condiciones desecharemos de forma casi inmediata muchas de las configuraciones de cache más modernas, que implementan estrategias agresivas. En primer lugar, no resulta interesante una cache de 2 o más niveles [Spa78] ya que necesitamos resolver la codificación de los fallos en el mismo ciclo en que se detecta el problema.

Estrategias como las caches desacopladas [Hil88] tampoco son adecuadas para nuestro caso. Consisten básicamente en utilizar una cache grande asociativa por conjuntos en la cual se predice que elemento tiene mayor probabilidad de ser referenciado. Se produce una primera búsqueda muy rápida sobre la cabeza de cada conjunto (asignación directa) y otra más costosa sobre el resto de los datos, totalmente asociativa.

Las caches de acceso múltiple [AHH88] son caches asociativas por conjuntos a las que se accede como a una de asignación directa. En caso de fallo, en el siguiente ciclo se repite la búsqueda desplazada a otros elementos del conjunto y así tantas veces como líneas tenga cada conjunto. La secuencialidad de esta operación la desaconseja totalmente para nuestra implementación.

Las caches aumentadas utilizan una cache de asignación directa por su sencillez, y añaden una pequeña cache totalmente asociativa, que también tiene una gestión sencilla debido a su pequeño tamaño. Existen tres implementaciones básicas. La primera de ellas es la cache víctima [Jou90], en la cual la cache totalmente asociativa

recoge las líneas expulsadas de la cache de asignación directa. Otra modalidad es conocida como cache asistente [KCZ+94] en la que la cache pequeña oficia de cache principal, y los datos expulsados de esta tienen una segunda oportunidad en la cache de asignación directa. La última posibilidad es utilizar la cache de asignación directa para almacenar datos con gran probabilidad de uso, y dejar la cache pequeña para los restantes. Cualquiera de estas configuraciones responde a nuestras necesidades por cuanto que cualquiera de ellas se puede implementar en un sólo ciclo.

Finalmente, una cache sencilla, con una política y un algoritmo de reemplazo compatible con ella, es la solución básica a considerar.

El conjunto de datos sobre el que probaremos las distintas posibilidades son una serie de imágenes en escala de grises con 8 bits por pixel. Tendremos en cuenta dos casos, con y sin predicción del valor de los pixeles. En el primer caso se aplica un predictor sencillo, codificar la diferencia con el pixel anterior, mientras que en el segundo codificaremos la imagen tal cual, pero no línea a línea, sino por bloques de 8 por 8 pixeles con el fin de aumentar todavía más la localidad de los datos. Las imágenes de test son algunas de las más utilizadas en compresión (Lena, Barbara, Boats, Peppers) y radiografías como ejemplo de imagen que necesita alta calidad sin pérdidas.

La figura de mérito a considerar será el número de fallos que se producen. Pretendemos minimizar esta cantidad, por cuanto que los fallos nos alejan de la correcta codificación de los datos. No hacemos estimaciones de relación de compresión por el momento ya que esta depende de muchos otros factores tal y como veremos más adelante.

2.3.1 Cache de asignación directa

La política de asignación directa impone su propio algoritmo de reemplazo. Al estar perfectamente definida la posición de cada símbolo en la cache, no existe libertad de alojamiento. A lo sumo se puede vetar el acceso a la cache a ciertos datos en función de ellos mismos o para preservar aquellos que ocupan la cache. Es por tanto un algoritmo de implementación trivial, rápido y poco costoso y que se utiliza en muchas caches de microprocesadores.

La ventaja de las caches de asignación directa es doble. Por una parte es inmediato saber si un símbolo está en la cache. Por otro lado, en caso de fallo, también es trivial decidir que dato ha de ser sustituido.

2.3.2 Cache totalmente asociativa

Es la alternativa a las caches de asignación directa. Las restantes posibilidades son combinaciones de estos dos tipos. Por tanto, todas ellas heredan en cierta medida

las ventajas e inconvenientes de estas. En una cache totalmente asociativa, los datos pueden ser alojados en cualquier línea de la cache. De esta manera, la operación de búsqueda de un dato se convierte en una comparación en paralelo con las etiquetas de todas las líneas. Tendremos así una complejidad dependiente del número de líneas de la cache. Esto hace que sea desaconsejable utilizar grandes caches.

En caso de fallo la línea que ha de ser sustituida vendrá dada por el algoritmo de reemplazo. Existe un amplio abanico de posibilidades jugando con la complejidad y la eficiencia. Asumiendo que el coste de la localización de un dato en una cache totalmente asociativa tiene un coste inevitable, el acento debe ser puesto en encontrar el algoritmo de reemplazo más conveniente. Consideraremos cuatro posibilidades, el algoritmo FIFO, dos implementaciones del algoritmo LRU y otro aleatorio. El algoritmo aleatorio no tiene utilidad para nosotros ya que necesitamos un algoritmo determinístico a fin de que el comportamiento del codificador y el descodificador sea el mismo. Sin embargo resulta útil en tanto que al compararlo con otros, nos permite apreciar la importancia que tiene disponer de un buen algoritmo.

El algoritmo FIFO (first-in-first-out) tiene una implementación muy sencilla. Cada vez que se produce un fallo, el contenido de la cache se desplaza desde cada línea a la siguiente. De ésta forma quedará una línea vacía que será ocupada por los nuevos datos que vienen de la memoria principal. De igual manera, el contenido de la última línea será expulsado, y deberá introducirse en la memoria para completar el reemplazo. La ventaja evidente de este método es su simplicidad, y el único inconveniente que se le puede achacar es la necesidad de introducir buses de interconexión entre las líneas para desplazar los datos.

El algoritmo LRU (least-recently-used) es más elaborado. Cada línea de la cache contiene un campo en el que se almacena un contador. Este contador toma un valor máximo al entrar los datos en la cache y cada vez que son referenciados. A cada ciclo que no son referenciados, el contador decrementa su valor. Cuando se produce un fallo, la línea reemplazada es aquella que contiene el valor más bajo en su contador. Los detalles de la implementación son importantes en cuanto al coste hardware que suponen. Una operación que en software se realiza con facilidad resulta muchas veces sumamente costosa en hardware. En este sentido no resulta práctico comparar entre sí todos los contadores. En todo caso sería más práctico crear una cola con los índices de las líneas en la que las líneas referenciadas promocionasen a la primera posición (lista move-to-front). De esta forma la línea que ocupe el fondo de la lista será la que sufra el reemplazo. Esta solución es bastante costosa y no aporta mejoras que justifiquen su uso (no se muestran resultados pero se han realizado simulaciones que corroboran esta afirmación). La opción más sencilla consiste en que el valor al que se inicializa el contador sea igual al número de líneas de la cache. De esta forma es seguro que siempre existe al menos una línea con el contador a cero. Esta línea será una candidata a ser sustituida, pero puede haber más líneas en la misma

situación. Escogeremos la primera que encontremos, pero consideraremos dos casos. Llamaremos algoritmo LRU a aquel que busca la primera línea con el contador a cero comenzando la búsqueda por la primera línea de la cache, y llamaremos LRU* al que comienza la búsqueda una posición más adelante a cada fallo.

Para todos estos casos, y para la cache de asignación directa, mostramos los siguientes resultados en las figuras 2.4 a 2.10. Probamos distintas imágenes con y sin predicción con caches de 8, 16 y 32 líneas. En cada gráfica se muestra además una tabla con los mismos datos.

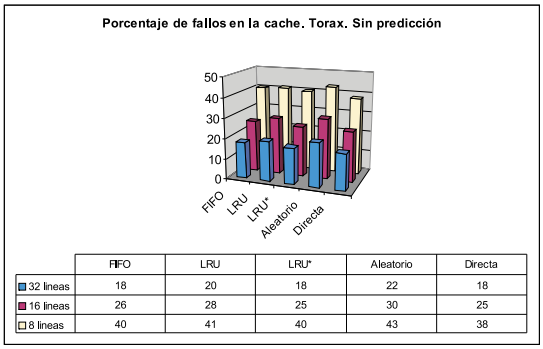
Como era de esperar la cache de 32 líneas produce el menor número de fallos, mientras que la de 8 líneas genera un número excesivo de fallos. Se aprecian dos tendencias claras. Las imágenes de radiografías, que son fácilmente compresibles, producen un bajo número de fallos, especialmente cuando se utiliza predicción. En cambio para imágenes fotográficas el número de fallos se mantiene a un nivel más alto y no existe gran disparidad entre los resultados con y sin predicción.

Los mejores resultados se obtienen con una cache de asignación directa en prácticamente todos los casos. Las diferencias pueden llegar a ser significativas para algunas imágenes y son pocos los casos en que están en desventaja respecto a las demás configuraciones. Se trata pues de una firme candidata para ser utilizada. A continuación, el mejor método es el que hemos llamado LRU* con resultados bastante próximos a la de asignación directa. Por ello será éste el algoritmo que utilizaremos para implementar caches asociativas en las siguientes secciones. Por último resaltar los malos resultados que se obtienen con un algoritmo de reemplazo aleatorio, lo que confirma la importancia de elegir un buen algoritmo de reemplazo.

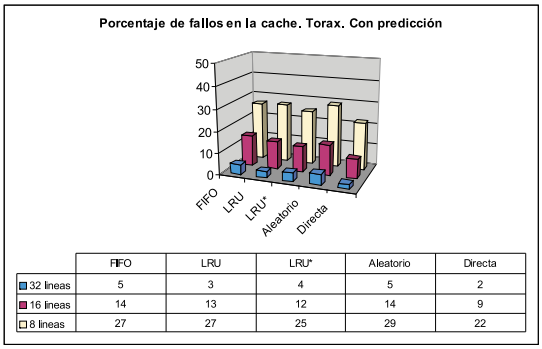
2.3.3 Caches asociativas por conjuntos

Limitar la asociatividad es una forma de reducir la complejidad de gestionar los reemplazos en una cache asociativa. Dividimos la cache en un conjunto de n caches asociativas, cada una de ellas con capacidad para un número de datos m. Diremos entonces que es una cache asociativa por conjuntos de m vías. Como algoritmo de reemplazo consideraremos el que mejores resultados arrojó en la sección anterior, LRU*.

En las siguientes figuras (2.11 a 2.17) presentamos una comparativa del porcentaje de fallos que se producen en nuestro conjunto de imágenes cuando se varía la asociatividad. Como puntos de referencia tenemos una cache totalmente asociativa y en el otro extremo una de asignación directa. El número de vías está limitado por el tamaño de la cache, de manera que sólo para la cache más grande se presentan todos los valores. Cuando sólo existe una vía nos encontramos con una cache de asignación directa, y esta es la posición que ocupa en el eje de las gráficas. Cuando el número de vías es igual al de líneas se trata de una cache totalmente asociativa.

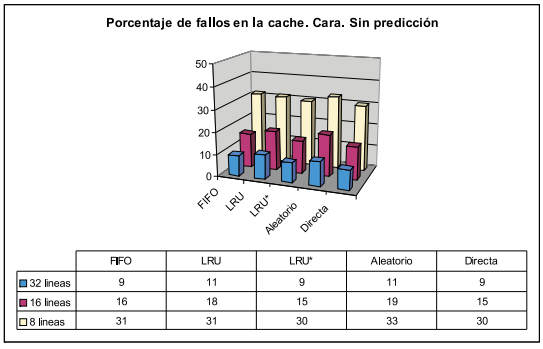


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	18	20	18	22	18
16 lineas	26	28	25	30	25
8 lineas	40	41	40	43	38

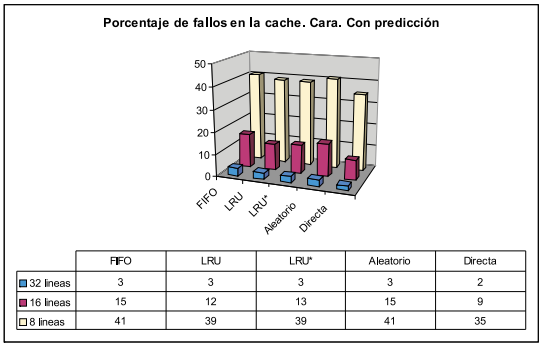


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	5	3	4	5	2
16 lineas	14	13	12	14	9
8 lineas	27	27	25	29	22

Figura 2.4: Porcentaje de fallos. Radiografía de torax

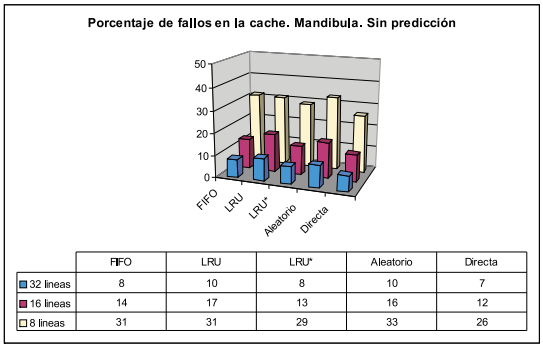


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	9	11	9	11	9
16 lineas	16	18	15	19	15
8 lineas	31	31	30	33	30

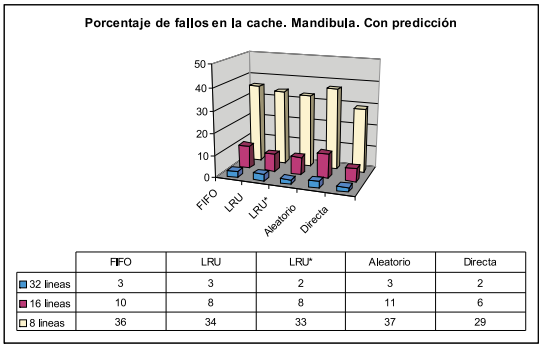


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	3	3	3	3	2
16 lineas	15	12	13	15	9
8 lineas	41	39	39	41	35

Figura 2.5: Porcentaje de fallos. Radiografía de cara

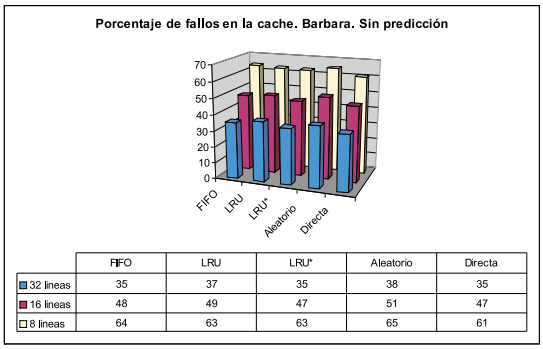


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	8	10	8	10	7
16 lineas	14	17	13	16	12
8 lineas	31	31	29	33	26

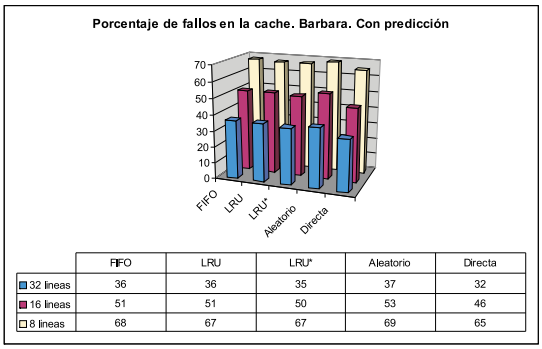


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	3	3	2	3	2
16 lineas	10	8	8	11	6
8 lineas	36	34	33	37	29

Figura 2.6: Porcentaje de fallos. Radiografía de mandíbula

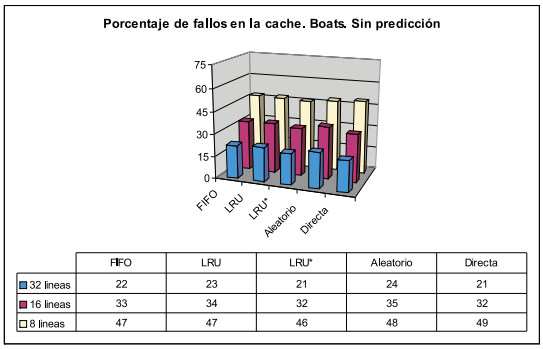


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	35	37	35	38	35
16 lineas	48	49	47	51	47
8 lineas	64	63	63	65	61

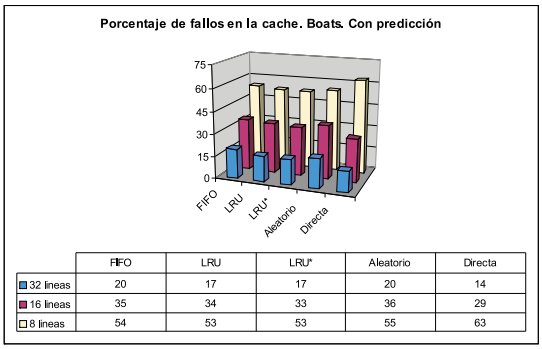


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	36	36	35	37	32
16 lineas	51	51	50	53	46
8 lineas	68	67	67	69	65

Figura 2.7: Porcentaje de fallos. Imagen barbara

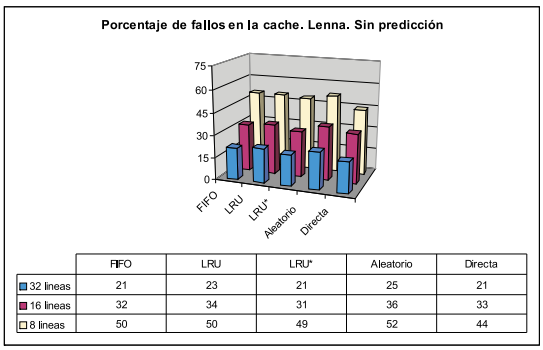


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	22	23	21	24	21
16 lineas	33	34	32	35	32
8 lineas	47	47	46	48	49

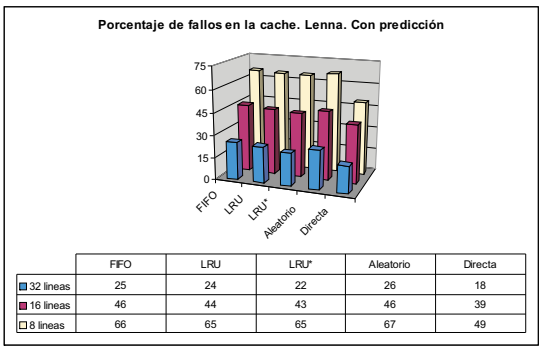


	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	20	17	17	20	14
16 lineas	35	34	33	36	29
8 lineas	54	53	53	55	63

Figura 2.8: Porcentaje de fallos. Imagen boats



	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	21	23	21	25	21
16 lineas	32	34	31	36	33
8 lineas	50	50	49	52	44



	FIFO	LRU	LRU*	Aleatorio	Directa
32 lineas	25	24	22	26	18
16 lineas	46	44	43	46	39
8 lineas	66	65	65	67	49

Figura 2.9: Porcentaje de fallos. Imagen lena

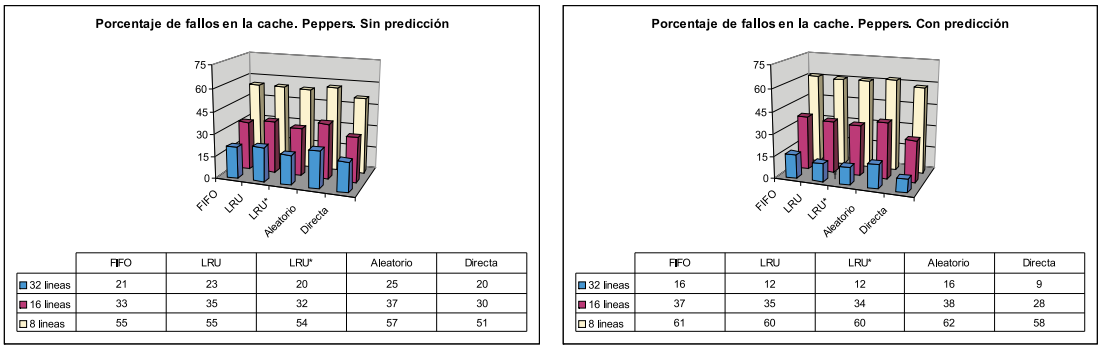


Figura 2.10: Porcentaje de fallos. Imagen peppers

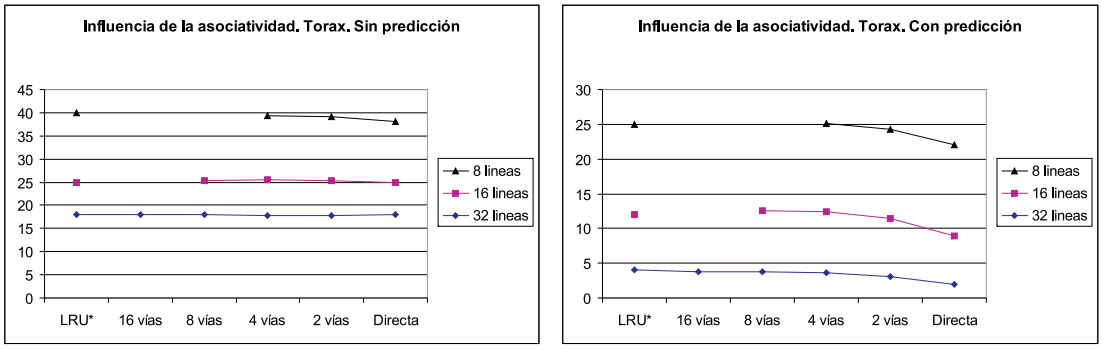


Figura 2.11: Porcentaje de fallos. Radiografía de torax

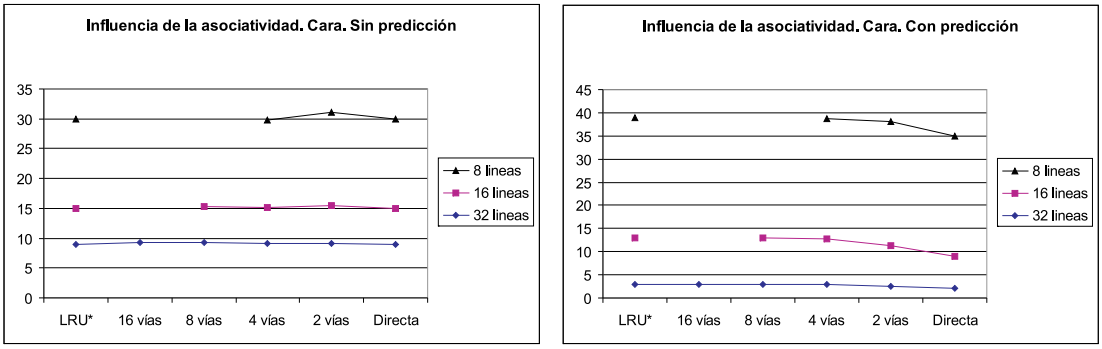


Figura 2.12: Porcentaje de fallos. Radiografía de cara

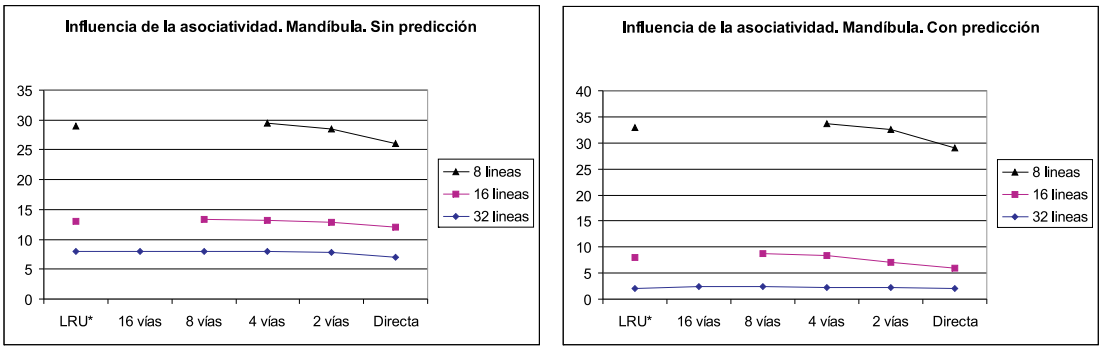


Figura 2.13: Porcentaje de fallos. Radiografía de mandíbula

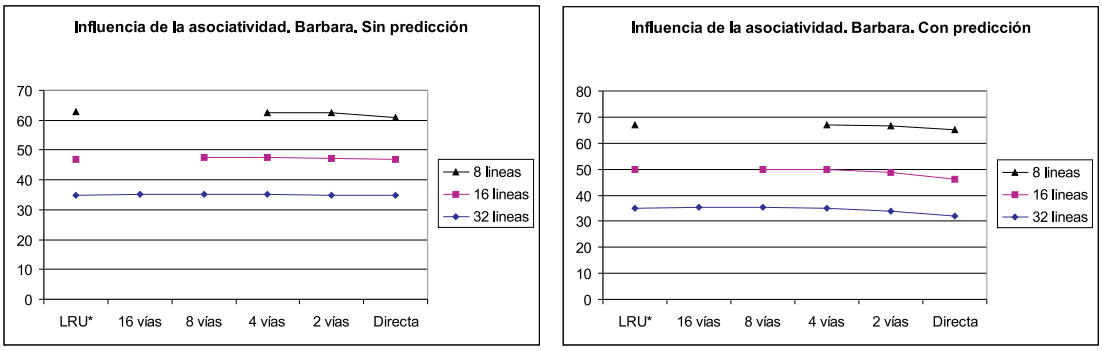


Figura 2.14: Porcentaje de fallos. Imagen barbara

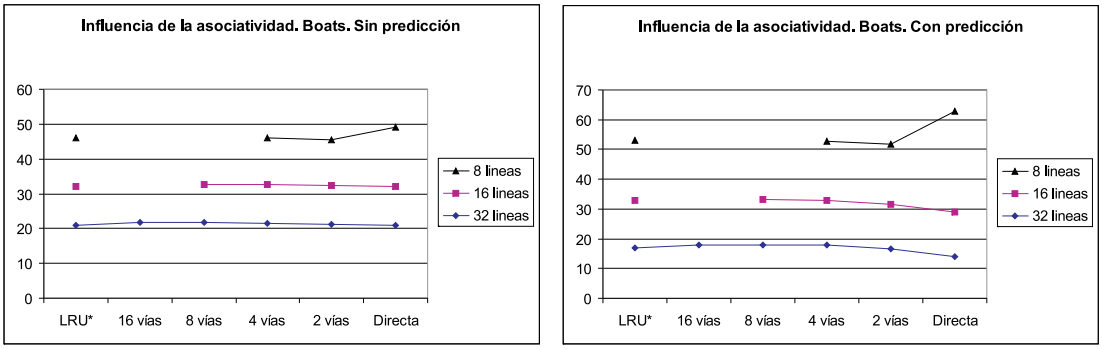


Figura 2.15: Porcentaje de fallos. Imagen boats

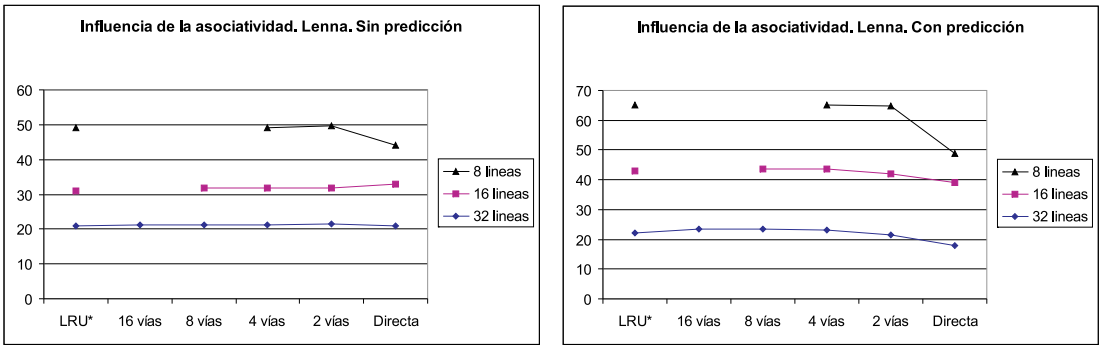


Figura 2.16: Porcentaje de fallos. Imagen lena

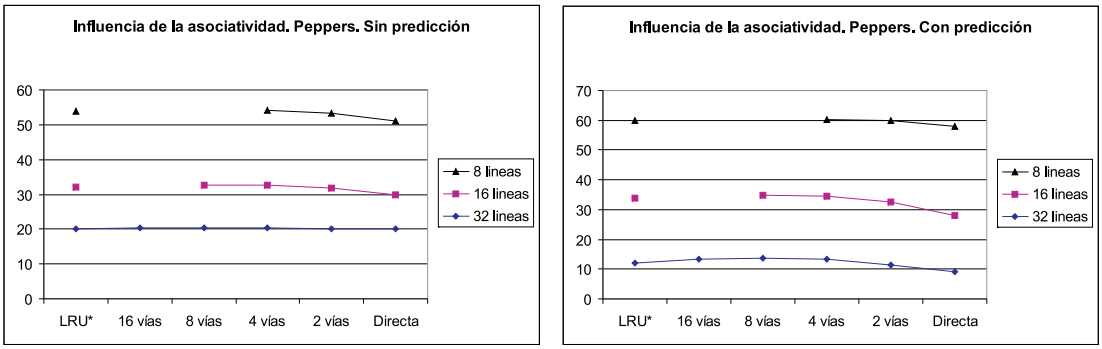


Figura 2.17: Porcentaje de fallos. Imagen peppers

Salvo en algunas excepciones existe una evolución de resultados desde la asociatividad total a la asignación directa pasando por tamaños de vía cada vez más pequeños. Por tanto la asignación directa sigue siendo una opción al menos tan buena o mejor que cualquiera de las otras en la mayoría de los casos.

2.3.4 Caches víctima y asistente

Estas dos configuraciones [Jou90, KCZ+94] utilizan dos caches separadas, una totalmente asociativa pequeña y otra mayor de asignación directa. Las posibilidades van más allá de la simple reducción de la complejidad al reducir el tamaño de la cache asociativa, ya que se establece un sistema de segundas oportunidades para los símbolos que son expulsados de la cache que ejerce de primaria. En un esquema de cache víctima la cache primaria es de asignación directa, y los símbolos expulsados de esta tienen una segunda oportunidad en la cache asociativa. En la configuración de cache asistente los papeles se invierten.

En primer lugar consideraremos tres configuraciones distintas de cache víctima para un total de 16 y 32 líneas. Para 16 líneas las combinaciones posibles son 12+4, 8+8 y 4+12. Para 32 hemos probado con 24+8, 16+16 y 8+24. Dado que la cache de asignación directa representa la complejidad mínima no resulta interesante aumentar el tamaño de la cache. Si una cache víctima es capaz de mejorar la compresión entonces debe demostrarlo sin aumentar el número de líneas. Los resultados se expresan como diferencia en el porcentaje de fallos tomando como referencia el obtenido para una cache de asignación directa, se muestran en la figura 2.18 y a mayor diferencia peores resultados. Al ser una medida relativa se ha promediado para todas las imágenes, y se puede observar que no se mejora la cache de asignación directa, si bien si se puede mejorar en algún caso particular. Se muestran también los resultados para LRU*.

El mismo sistema para comparar resultados se aplica a una configuración con cache asistente. Los tamaños de la cache asistente y principal siguen la misma regla aplicada para la cache víctima. Los resultados que se muestran en la figura 2.19 siguen la misma tónica que en la cache víctima, y en ningún caso se mejora a la cache de asignación directa.

2.3.5 Configuración óptima: asignación directa

A la vista de los resultados de la sección anterior nos decantaremos por la configuración en la que coinciden las dos características deseadas: simplicidad y alta tasa de aciertos. Esta es la cache de **asignación directa**. Los motivos por los cuales este tipo de cache funciona mejor se pueden justificar si tenemos en cuenta la disposición natural de los datos. Cuando no se aplica predicción el principio de localidad

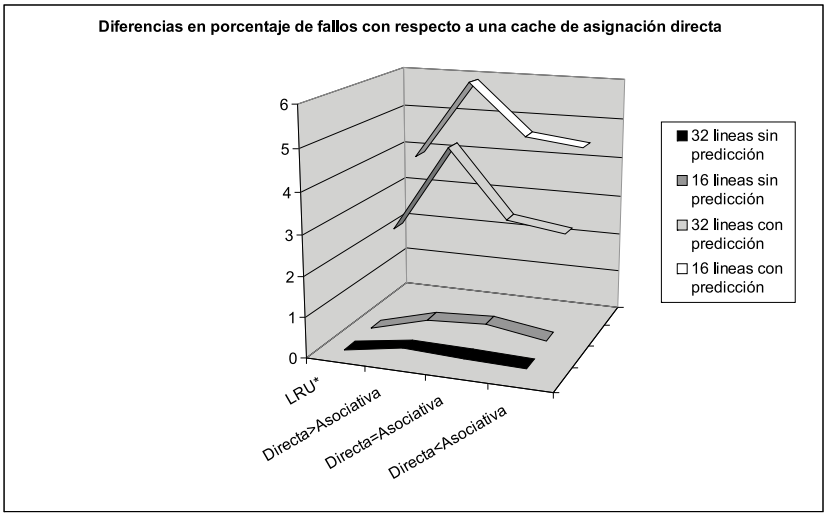


Figura 2.18: Resultados utilizando una cache víctima. Menos es mejor.

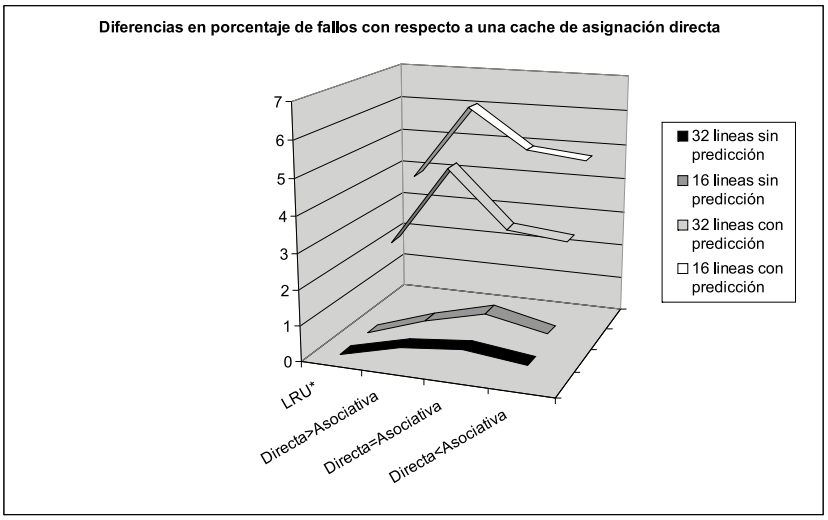


Figura 2.19: Resultados utilizando una cache asistente. Menos es mejor.

espacial (redundancia visual en la imagen) hace que la mayor parte de los pixeles de una zona de la imagen pertenezcan a una misma región del histograma. En la figura 2.20 se ve como una región de la imagen coincide en cierta medida con una región del histograma, y ésta con la cache (parte inferior). Cuando el contexto de la imagen cambia el contenido de la cache se renueva casi por completo. La eficiencia es tal que otros métodos más complejos podrían a lo sumo igualarla.

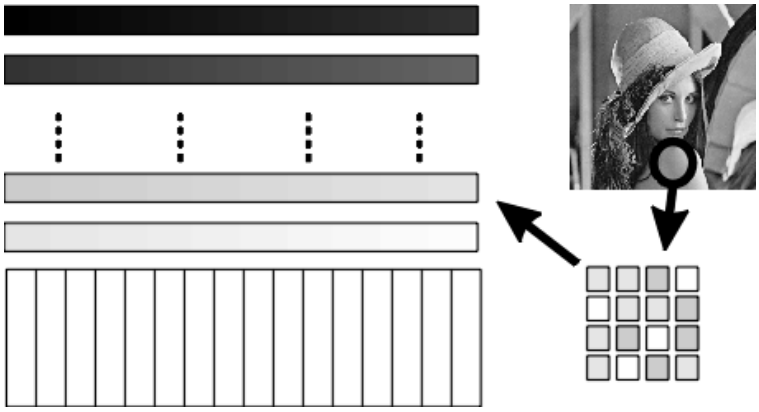
Cuando se aplica predicción tenemos el caso de la figura 2.20. En este caso el razonamiento es distinto. Dado que el histograma de la imagen tiene una forma de pico centrado (si bien algunas imágenes se apartan de esta tendencia), la parte central del mismo ocupara la cache la mayor parte del tiempo. La probabilidad de que se produzca un fallo es baja, y cuando se produce el símbolo entrante permanece poco tiempo en la cache por lo general. En cambio en un algoritmo tipo FIFO o LRU, este símbolo poco probable permanecerá durante un número de ciclos mínimo, igual al número de líneas.

2.4 Eficiencia del codificador con cache

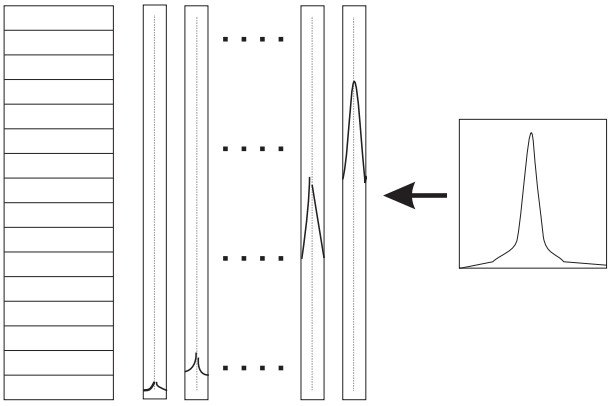
Hasta este punto la introducción de la cache no ha acreditado mejores resultados de compresión. Tan sólo nos permite reducir la cantidad de datos con que se opera y suponemos que esto se traducirá en una implementación más sencilla y rápida. Los resultados reales de compresión que se pueden obtener con este sistema no se han presentado todavía y esto es debido a que implica introducir nuevos parámetros para comparar configuraciones.

Consideremos en primer lugar el algoritmo básico de Witten, Neal y Cleary [WNC87]. Se establece un tamaño de palabra para operar, y este tamaño fija también el valor máximo que pueden tomar las probabilidades acumulativas y el coste de la aritmética si se traslada al hardware. El tamaño de palabra tiene una gran importancia no sólo por el coste, sino porque al limitar el crecimiento de las probabilidades acumulativas fija el intervalo entre reescalados de las probabilidades y de esta forma influye en el grado de adaptabilidad del modelo.

Ya que trabajamos con un algoritmo con precisión reducida debemos prestar atención también a la precisión utilizada en las operaciones, particularmente en los productos. El coste de calcular un producto es muy alto cuando la precisión es alta, pero reducir la precisión acarrea importantes pérdidas en compresión. Éstas se añadirán al déficit que de por sí supone utilizar un algoritmo que no escala los intervalos con total precisión. Sobre los distintos métodos de implementar los productos ya se trató en la sección anterior, y nosotros nos decantaremos por la forma más sencilla: seleccionar una cierta cantidad fija de los bits más significativos del multiplicador. Por ejemplo, el número 1.345, que tiene por representación binaria



(a) Sin predicción



(b) Con predicción

Figura 2.20: Distribución de los datos de una imagen en una cache de asignación directa.

1.010110001, se truncaría a 1.01, 1.010, 1.0101 y 1.01011 para precisiones de 2, 3, 4 y 5 bits fraccionales.

El último parámetro que hemos de considerar es el valor que toma la probabilidad asignada a los fallos, P_{fallo}. Un valor muy alto supone que la suma de las probabilidades de la tabla virtual sea una parte importante del valor máximo de la suma de las probabilidades, con lo que se limitaría mucha la evolución de las mismas. Por otro lado, un valor bajo supone que el coste de codificar cada fallo es muy alto, y la compresión se vería afectada.

Nuevamente recurriremos a la simulación como herramienta para decantarnos por una u otra configuración, si bien por el momento nuestro mayor interés es estimar las relaciones de compresión que se pueden obtener con este método.

2.4.1 Influencia del tamaño de palabra utilizado

Consideraremos en primer lugar un codificador aritmético sin cache. Su funcionamiento es como el descrito para el codificador aritmético del capítulo 1, salvo que los productos no se implementan utilizando dígitos con signo, sino considerando 8 bits fraccionales. Esto es, utilizaremos una precisión alta en los productos para separar ambos efectos.

A continuación se muestran los resultados de compresión (%) para varias imágenes sin y con predicción para distintos tamaños de la palabra utilizada para expresar el rango (figuras 2.21 y 2.22). De ellas extrae una característica muy importante en la cual difieren de forma radical. Cuando no se aplica predicción los resultados mejoran (relaciones de compresión mas bajas) para precisiones bajas. Cuando se aplica predicción la tendencia es la contraria.

Los motivos de este comportamiento son evidentes si atendemos a la naturaleza de los datos implicados. En la figura 2.23 se muestran los histogramas de las imágenes boats y lena con y sin predicción. La predicción iguala mucho los histogramas, y en general podemos suponer que este comportamiento se mantiene a lo largo del procesamiento de la imagen. En cambio el histograma original es muy variable, depende mucho de la imagen y por tanto dependerá también de la zona de la imagen que se esté procesando. El efecto principal de utilizar un tamaño de palabra pequeño es que la actualización de las probabilidades conduce pronto a una saturación de las mismas y por tanto es necesario corregirlas con frecuencia. Cuando no se aplica predicción, la corrección continua de las probabilidades tiene el efecto de facilitar los cambios de contexto, el modelo evoluciona con facilidad, y la relación de compresión mejora. En cambio, al aplicar predicción, los cambios de contexto no son bruscos, y las correcciones tienen el efecto de decrementar las probabilidades continuamente. Por ello comprime mejor con tamaños de palabra grandes, para los cuales se reduce el número de correcciones.

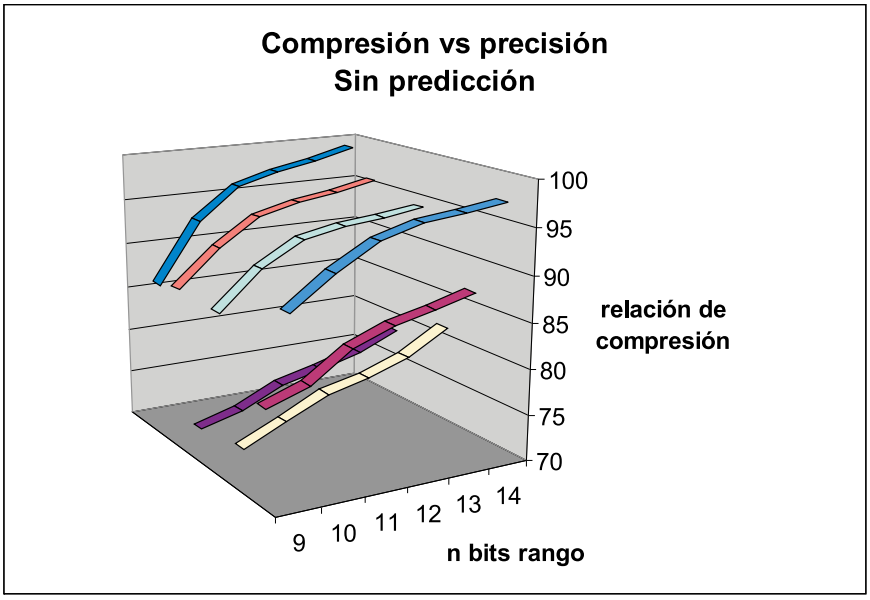


Figura 2.21: Evolución de la compresión para distintos tamaños de palabra. Codificador sin cache sin aplicar predicción.

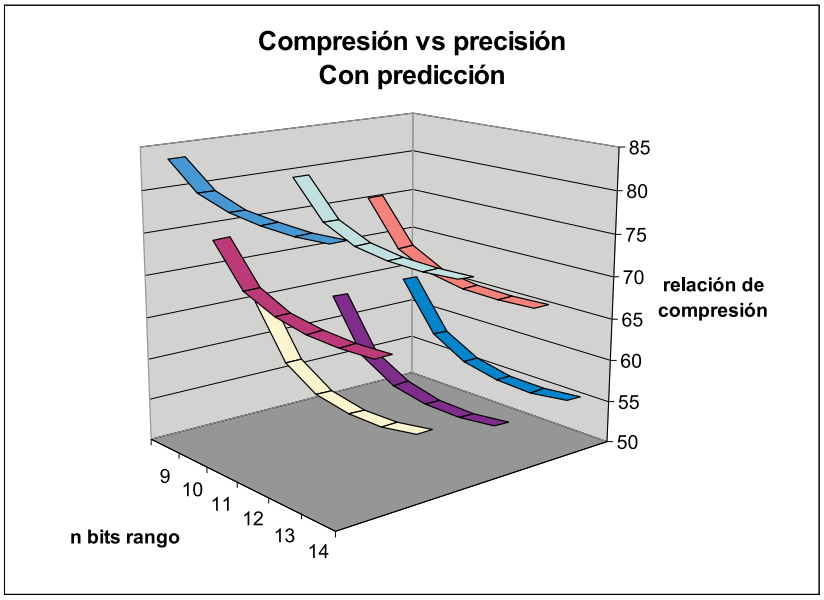


Figura 2.22: Evolución de la compresión para distintos tamaños de palabra. Codificador sin cache aplicando predicción.

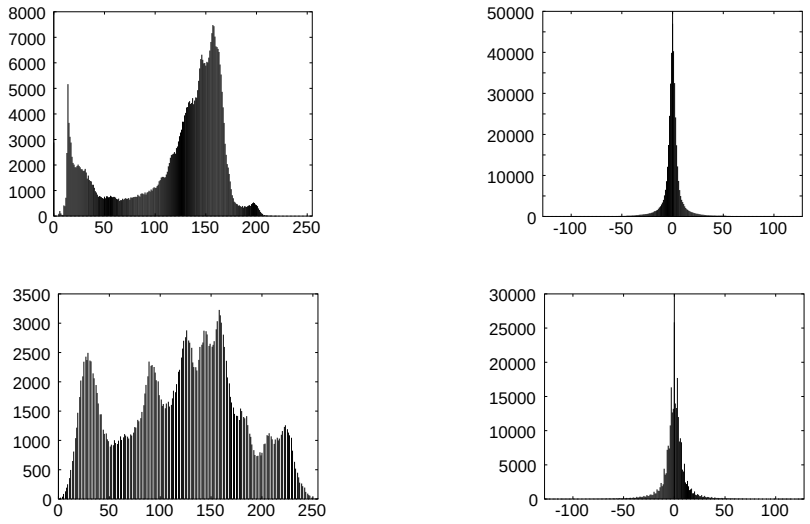


Figura 2.23: Histogramas para boats y lena. A la izquierda sin aplicar predicción, y a la derecha tras aplicarla.

Consideraremos ahora el caso en que se utiliza una memoria cache. Mostraremos como varía la relación de compresión con la precisión utilizada en el rango. Al igual que antes se considera que los productos se implementan con una precisión de 8 bits para no incluir un parámetro más en este momento de la comparación. Sin embargo es inevitable considerar la influencia de la probabilidad que se asigna a los fallos. Por una parte influye directamente en el coste de codificar los fallos y por otra parte influye en la parte del rango que se dedica a codificar estos y la que se dedica a la cache, y por tanto en el número de correcciones.

En las tablas 2.1 y 2.2 se encuentran tabulados los resultados para distintas imágenes. La cache utilizada es de 32 líneas. Variamos el número de bits de longitud del rango y el valor de P_{fallo}, que se expresa como potencias negativas de 2 tomando el valor del rango como la unidad. Al final se resumen los mejores valores obtenidos y se comparan con los obtenidos sin cache. Las diferencias no son grandes ni en uno ni en otro sentido, pero todavía debemos llegar a mejores resultados, especialmente cuando se utiliza predicción. Nótese que en el caso sin predicción ya hemos llegado a resultados comparables con la implementación sin cache, por lo que las mejoras futuras conseguirán superarla de forma clara.

De estas tablas deducimos que la elección de los distintos parámetros involucrados es de suma importancia. Por otra parte, los mejores resultados para cada imagen se encuentran bastante localizados. Dado que la dispersión es relativamente baja, podemos hacer una análisis conjunto para todas las imágenes a fin de facilitar las com-

rango	fallo	barbara	boats	cara	lena	mand.	pepper	torax
9 bits	-9	**90.34**	**80.02**	72.96	**85.78**	72.31	**85.88**	**84.88**
10 bits	-10	92.84	81.33	**72.56**	86.5	**71.29**	86.22	86.64
	-9	92.25	82.6	76.18	88.5	75.25	88.54	88.82
11bits	-11	96.16	84.27	73.73	87.84	72.26	87.22	88.77
	-10	93.06	83.3	74.58	87.06	73.5	86.76	88.22
	-9	93.05	85.17	78.99	89.43	78.16	89.46	91.14
12 bits	-12	100.36	87.29	75.05	90.41	73.55	89.7	91.16
	-11	96.41	85.3	74.88	88.26	73.36	87.68	89.82
	-10	93.41	84.52	76.12	87.5	74.8	87.29	89.43
	-9	93.5	87.06	80.73	89.82	79.97	90.03	92.39
13 bits	-13	104.89	90.72	77.58	93.6	75.16	92.85	94.61
	-12	100.32	88.24	76.66	90.95	74.63	90.66	92.19
	-11	96.84	86.4	76.72	89.11	74.66	88.66	90.72
	-10	94.03	85.54	77.42	88.28	75.95	88.13	90.55
	-9	93.95	88.09	82.29	90.87	81.17	90.85	93.62
14	-14	109.74	94.84	80.95	97.74	78.05	96.74	98.45
	-13	105.33	92	79.82	95.04	77.31	94.17	96
	-12	100.92	89.7	78.87	92.44	76.49	91.87	93.95
	-11	97.21	87.72	78.32	90.61	76.42	90.61	92.65
	-10	94.63	86.97	79.21	89.46	77.59	89.39	92.06
	-9	94.69	89.44	83.76	91.88	82.53	92.08	94.95
Mínimo		90.34	80.02	72.56	85.78	71.29	85.88	84.88
Sin cache		89.5	78.48	72.7	85.88	71.7	86.17	85.59
Diferen.		0.84	1.54	-0.14	-0.1	-0.41	-0.29	-0.71

Tabla 2.1: Resultados de compresión para una cache de 32 líneas de asignación directa sin aplicar predicción. Se varía el número de bits de precisión del rango y el valor de la probabilidad de fallo($-n \equiv 2^{-n}$). Se resume para cada imagen el mejor valor y se compara con el mejor valor posible sin cache.

rango	fallo	barbara	boats	cara	lena	mand.	pepper	torax
9 bits	-9	86.44	76.13	67.29	82.21	66.12	77.92	66.24
10 bits	-10	85.7	71.28	59.76	78.27	58.38	71.94	58.88
	-9	86.25	75.59	66.75	81.8	65.48	77.4	65.87
11bits	-11	88.03	70.68	57.12	78.4	55.72	70.41	56.29
	-10	**85.5**	71.01	59.55	78.01	58.19	71.64	58.69
	-9	86.27	75.34	66.48	81.61	65.21	77.16	65.62
12 bits	-12	91.43	71.49	**56.29**	79.76	54.8	70.47	55.42
	-11	87.89	**70.51**	57.19	78.2	55.75	**70.29**	56.26
	-10	85.53	70.91	59.58	**77.99**	58.21	71.57	58.66
	-9	86.33	75.29	66.41	81.56	65.13	77.04	65.49
13 bits	-13	95.32	72.99	56.39	81.82	**54.76**	71.29	**55.39**
	-12	91.35	71.48	56.66	79.69	55.09	70.48	55.64
	-11	87.95	70.57	57.49	78.27	56.02	70.33	56.51
	-10	85.64	71	59.85	78.13	58.44	71.64	58.85
	-9	86.44	75.35	66.56	81.61	65.22	77.07	65.55
14	-14	99.5	74.93	57.35	84.36	55.44	72.59	56
	-13	95.32	73.1	57.25	81.99	55.49	71.5	55.99
	-12	91.49	71.67	57.54	80.04	55.75	70.76	56.17
	-11	88.2	70.86	58.34	78.57	56.66	70.71	57.04
	-10	85.92	71.3	60.61	78.41	58.97	71.95	59.3
	-9	86.7	75.52	67.02	81.86	65.57	77.24	65.86
Mínimo		85.5	70.51	56.29	77.99	54.76	70.29	55.39
Sin cache		78.96	65.95	55.63	72.4	54.13	67.17	54.69
Diferen.		6.54	4.56	0.66	5.59	0.63	3.12	0.7

Tabla 2.2: Resultados de compresión para una cache de 32 líneas de asignación directa aplicando predicción. Se varía el número de bits de precisión del rango y el valor de la probabilidad de fallo($-n \equiv 2^{-n}$). Se resume para cada imagen el mejor valor y se compara con el mejor valor posible sin cache.

paraciones. En las siguientes gráficas (figuras 2.24) se muestra la diferencia relativa de compresión para las distintas configuraciones respecto al mejor valor. Para cada imagen se ha calculado la mejor relación de compresión. Para cada configuración se ha calculado la diferencia relativa respecto a este valor $((ratio - ratio_{min})/ratio_{min})$ y finalmente se ha promediado entre las distintas imágenes para cada configuración.

A la vista de estas gráficas tenemos una visión más compacta del comportamiento de los resultados al variar los parámetros. Las diferencias de comportamiento aplicando predicción o no, no son muy acusadas.

Cuando no se utiliza predicción la mejor opción es utilizar la precisión más baja, al igual que en el caso sin cache. Los resultados son manifiestamente mejores lo que es una ventaja ya que a menor tamaño de palabra menor coste de implementación. Pero cuando se aplica predicción la situación ya no es tan clara. Hemos optado por utilizar un tamaño de palabra de 10 bits y un valor de P_{miss} de 2^{-10}. La cache de 8 líneas se separa ligeramente de la tendencia de las mayores debido a que para un cache tan pequeña reducir el coste de cada uno de los abundantes fallos es más importante que el coste de las correcciones. En la figura 2.25 se comparan los resultados promedio de compresión con y sin cache. Los mejores resultados se obtienen todavía cuando no se utiliza memoria cache. Esta tendencia es más acusada cuando se utiliza predicción. Haremos notar que sin predicción se logran resultados muy similares con una cache de 16 líneas, que es relativamente pequeña.

2.4.2 Influencia de la precisión en los productos

Parte de la complejidad de la codificación aritmética viene dada por las iteraciones de codificación y descodificación (ecuaciones 1.2 y 1.4). Si bien el coste del modelo es superior, su contribución no es desdeñable, y supone un problema durante la descodificación. Numerosos esquemas han sido propuestos para ajustar el coste de forma que los resultados no empeoren de manera sensible. En nuestro sencillo esquema truncamos el número de bits del multiplicador que se utilizan. Los resultados mostrados hasta el momento se han obtenido con una precisión alta, considerando 1 bit entero y 8 fraccionales. A continuación veremos hasta que punto podemos prescindir de la precisión sin empeorar los resultados. En la figura 2.26 se comparan las diferencias respecto al caso en que se utilizan 8 bits fraccionales.

La tendencia que se observa es que el salto de 8 a 4 bits fraccionales supone una merma insignificante en la eficiencia de la codificación. En promedio las imágenes ocupan entre un 0.01% y un 0.1% más, pero la precisión se ha reducido a la mitad. Siendo más ambiciosos se puede continuar disminuyendo la precisión, y así con tan solo 2 bits fraccionales se obtienen resultados con una merma sensiblemente inferior al 1%. En adelante utilizaremos 2 bits fraccionales para implementar los productos. De esta forma apostamos decididamente por una configuración que simplifique la

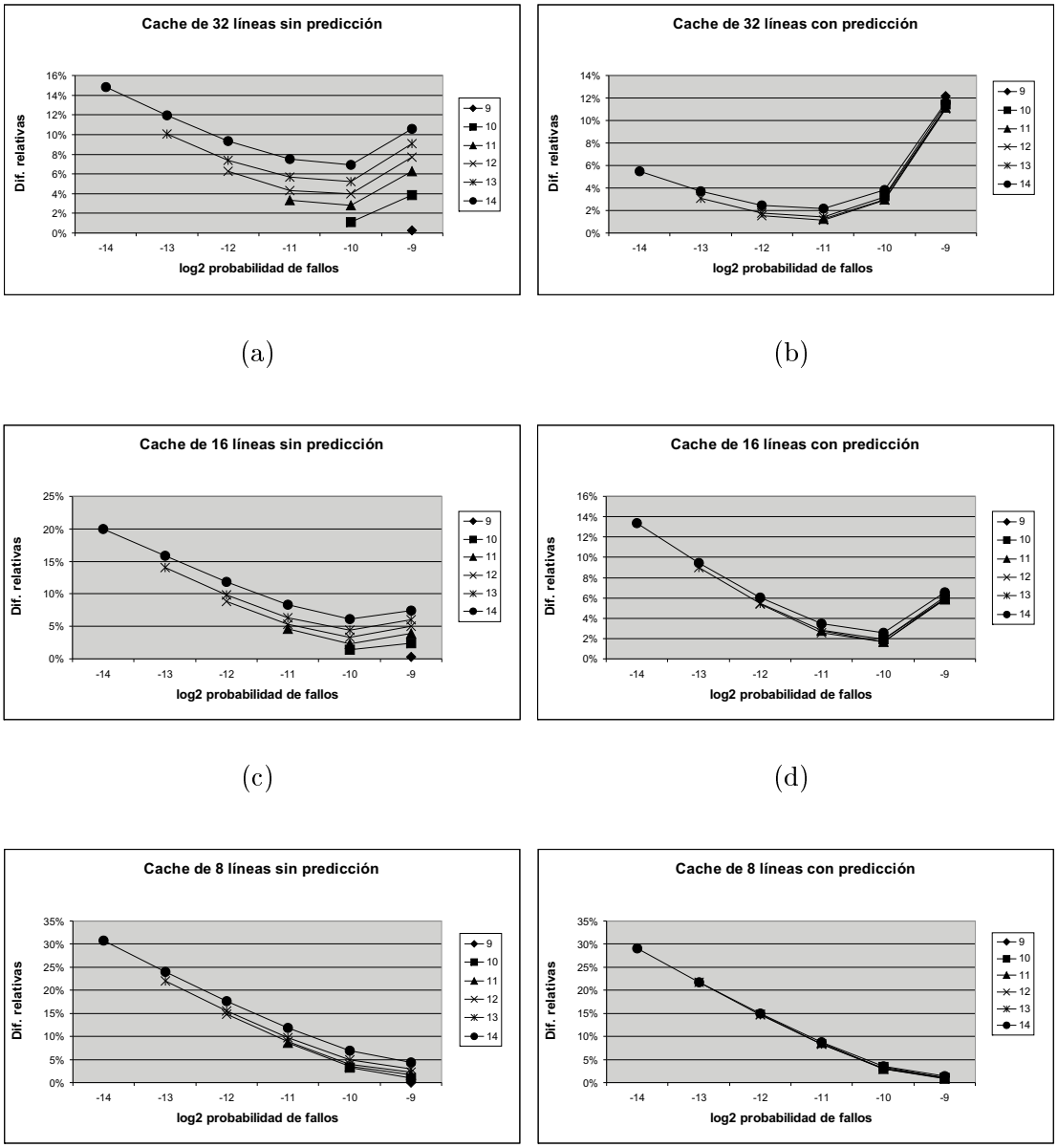


Figura 2.24: Diferencias de compresión para 32, 16 y 8 líneas respecto a la mejor configuración para cada caso. Menos es mejor.

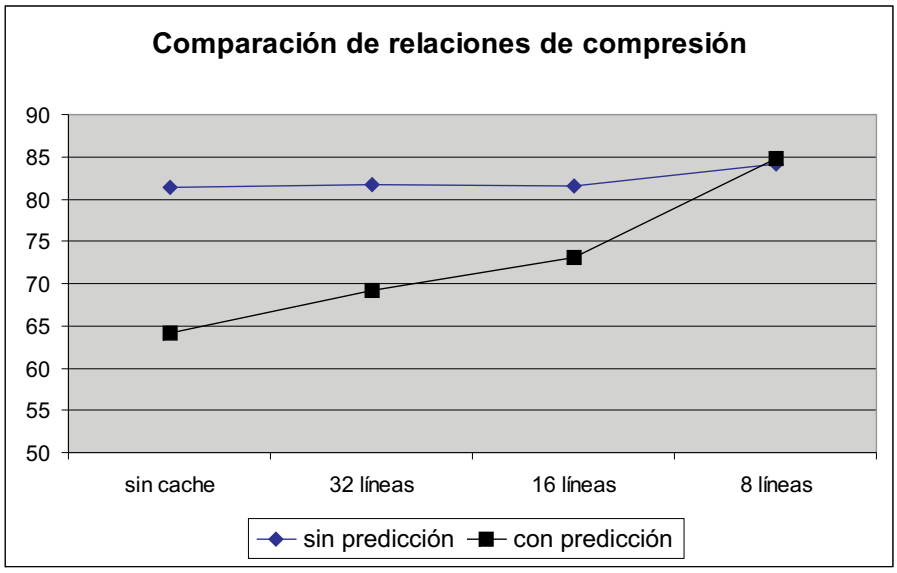


Figura 2.25: Relaciones de compresión promedio sin cache y con caches de 32, 16 y 8 líneas.

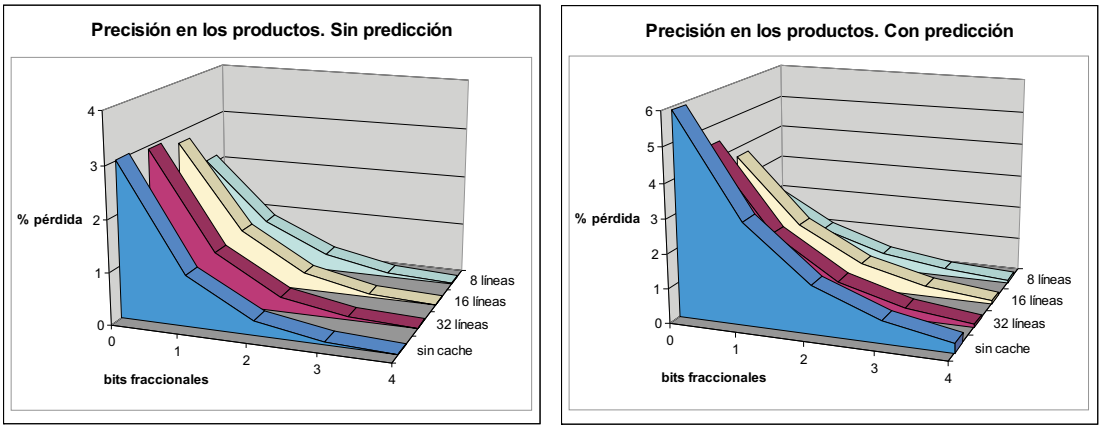


Figura 2.26: Pérdida de compresión al reducir el número de bits fraccionales utilizados al calcular los productos.

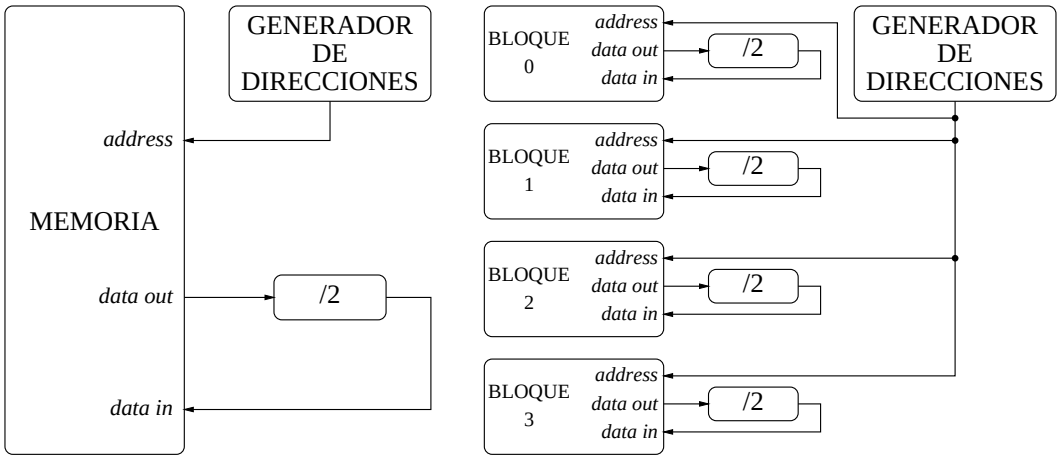


Figura 2.27: Estrategias de escalado de las probabilidades. (a) Serie. (b) Paralelo, cuatro símbolos por ciclo.

implementación hardware.

2.5 Correción de las probabilidades

La adaptabilidad del modelo que estamos utilizando se logra incrementando las probabilidades de los símbolos cada vez que estos son referenciados. A fin de reducir el coste de esta operación, las probabilidades reales son sustituidas por un contador del número de veces que un símbolo ha sido referenciado. No existe ninguna operación de escalamiento destinada a que la suma de las probabilidades tenga un valor normalizado, ya que su coste sería excesivo. Es por ello que tras procesar un determinado número de símbolos las probabilidades de todos ellos suman una cantidad que supera los límites de precisión del algoritmo. El método por el que se suele optar para corregir este problema consiste en dividir las probabilidades de todos los símbolos por 2 [WNC87]. Esta operación, si bien es sencilla, debe extenderse a todos los símbolos del alfabeto por lo que su implementación es problemática. Para realizar esta operación la codificación debe ser detenida, y a partir de ese momento se debe escalar el valor de todas las probabilidades. El tiempo necesario para completar la operación depende del grado de paralelismo que se desee introducir. En otras palabras, debe llegarse a una solución de compromiso entre el tiempo y el consumo de área tal y como se muestra en la figura 2.27.

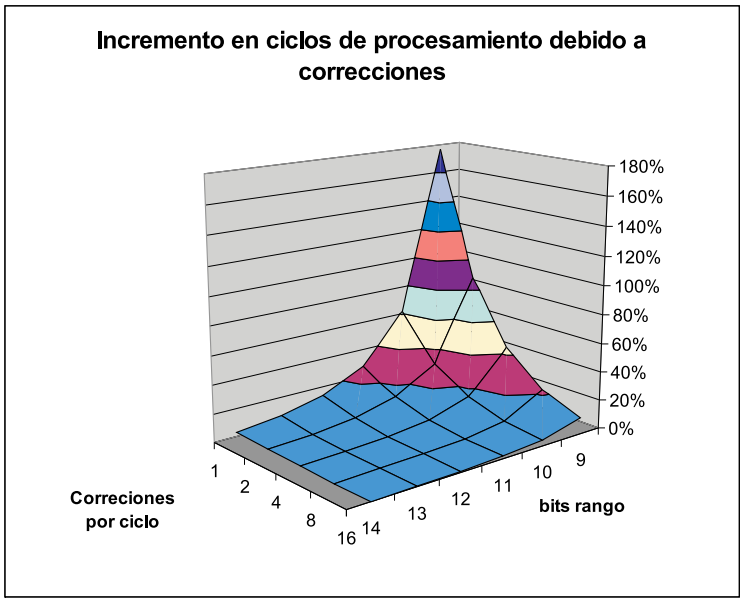


Figura 2.28: Coste temporal añadido debido a las correcciones dependiendo del entrelazado y la precisión.

En la figura 2.28 se muestra el porcentaje en que se incrementa el número de ciclos para las imágenes con las que trabajamos sin predicción y varios valores de tamaño de palabra y número de escalamientos por ciclo. La dispersión para las distintas imágenes es prácticamente nula y todas siguen la misma tendencia. Vemos que los valores más interesantes desde el punto de vista de la relación de compresión (poca precisión) suponen un aumento grande en el tiempo de computación. Así, para *lena*, una imagen de 262000 pixeles, con 8 bloques de memoria y 9 bits de precisión en el rango el número de ciclos de procesamiento crece hasta 320000, un 22% extra.

Las consideraciones anteriores se aplican a un modelo convencional. En nuestro caso tenemos un nivel de memoria adicional y las probabilidades de los símbolos almacenados en él han de ser escaladas también. En realidad la necesidad de escalar para corregir los valores surge de la cache misma. Dado que todas las operaciones salvo los reemplazos se realizan con los datos contenidos en la cache, es la suma de las probabilidades de los símbolos contenidos en la misma la que supera el límite permitido. En esta suma entra la tabla virtual, que aporta un valor fijo.

En principio sería necesario escalar la memoria principal y la cache. Por tanto la cache no aportaría ninguna ventaja en este aspecto. La posibilidad que surge de forma inmediata al plantear el problema es escalar únicamente el contenido de la cache. Esto hace que aparezca una incongruencia entre la situación de los datos

presentes en la cache y la de aquellos que se encuentran en la memoria principal. De hecho, sería posible que poco después de escalar, un cambio de contexto llenase la cache de datos altamente probables que provocaran un nuevo escalamiento. La convivencia en la cache entre datos escalados y no escalados no resulta problemática y puede suponer, por otra parte, una nueva forma de adaptabilidad del modelo a distintos contextos. Con el paso del tiempo todo el modelo se escala ya que los reemplazos llenan la cache de símbolos con su probabilidad escalda.

Nuevamente recurriremos a la simulación para obtener resultados concluyentes sobre esta posibilidad. Consideramos que un modelado del tipo de datos no sería una vía válida para obtener de forma analítica los resultados de las distintas modificaciones. En primer lugar porque la naturaleza de las imágenes es muy cambiante, y en segundo lugar porque al trabajar con la memoria cache, las propiedades estadísticas macroscópicas no son tan relevantes como los detalles de grano fino.

Consideraremos a continuación dos aspectos, las diferencias con respecto al esquema de cache que estamos considerando hasta el momento y las diferencias con un codificador sin cache. En la figura 2.29 se muestran ambos. Vemos que en general la introducción de esta modificación tiene efectos muy benignos sobre la compresión. En comparación con la última configuración de cache propuesta, se obtiene beneficios altos cuando no se utiliza predicción y moderados cuando se utiliza. Las diferencias con respecto a la codificación sin cache son muy diferentes dependiendo del número de líneas. La configuración con 8 líneas funciona muy bien sin predicción pero muy mal cuando se aplica, así que en lo sucesivo se descartará. En cambio la cache de 16 líneas tiene un comportamiento muy bueno en los dos casos. Sin predicción arroja los mejores resultados, pero con predicción todavía debe mejorar. En esta caso la cache de 8 líneas arroja muy malos resultados, casi un 18% peor que sin cache, pero un 1% mejor que la última configuración probada. La cache de 32 líneas, pese a su coste, arroja los mejores resultados cuando se utiliza predicción. El motivo de estos resultados debemos buscarlo en que la cache introduce un cierto grado de codificación de orden superior al considerar en cada momento un subconjunto de símbolos, los cuales tienen una probabilidad más alta de aparecer.

2.5.1 Resultados de compresión

Finalmente, en la figura 2.30, se muestran los resultados absolutos de compresión, comparándolos con los obtenidos por el método clásico de Witten y col. [WNC87] y con un codificador de precisión reducida sin cache. Los resultados sin predicción son los mejores. Se mejora el método de Witten, y no sólo para radiografías. Añádase a esto que la cache de 16 líneas obtiene mejores resultados que la de 32, por lo que el coste de la implementación puede llegar a ser realmente reducido. La cache de 8 líneas es también una opción a considerar porque en muchos casos es más

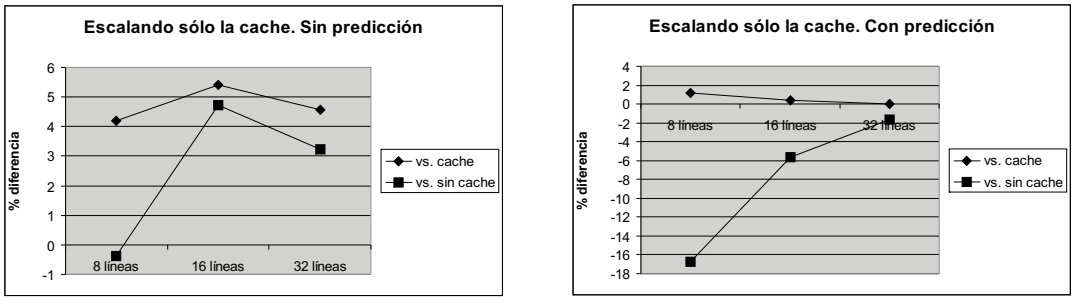


Figura 2.29: Escalando sólo la cache. Diferencias respecto a la cache con escalamiento completo y con respecto a la configuración sin cache (más es mejor). Los resultados negativos implican desventaja respecto a la configuración de referencia.

eficiente que los otros modelos sin cache. El hecho de que una cache pequeña dé mejores resultados que la más grande se explica porque su tamaño es suficiente para albergar los distintos contextos, mientras que la cache de 32 líneas acoge símbolos que no son utilizados. Se produce, por tanto, el mismo efecto, aunque a menor escala, que permite que un esquema con cache mejore a uno sin ella.

Cuando se utiliza predicción vemos que no se alcanza el objetivo marcado. La tendencia es la que cabría suponer: el método clásico obtiene los mejores resultados, seguidos del codificador sin cache con aritmética reducida y a continuación las tres caches. El número de líneas es importante ya que la compresión en este caso depende más del número de fallos que de la adaptabilidad ante cambios de contexto.

En resumen, podemos dar por finalizado el estudio del codificador con cache sin predicción. Los resultados obtenidos son los suficientemente buenos y no se abren nuevos caminos para mejorar la eficiencia o reducir la complejidad desde el punto de vista del algoritmo. Por otro lado cuando se utiliza predicción nos encontramos ante una importante diferencia entre nuestros resultados y los obtenidos por otros métodos. Dado que la mayor parte de los esquemas de compresión implementados utilizan predicción nos encontramos ante una importante deficiencia que debe ser subsanada, ya que las diferencias son lo suficientemente importantes como para que no se vean compensadas por la reducción de complejidad alcanzada. En lo sucesivo centraremos nuestros esfuerzos en estrategias dirigidas hacia este tipo de datos, con histogramas en forma de pico centrado.

Al margen de la importancia de las mejoras de compresión debemos considerar los beneficios que supone haber eliminado el escalamiento de los datos contenidos en la memoria principal. Dado que el coste del escalamiento en se reduce casi exclusivamente al coste de acceder a los datos, entonces hemos resuelto el problema del

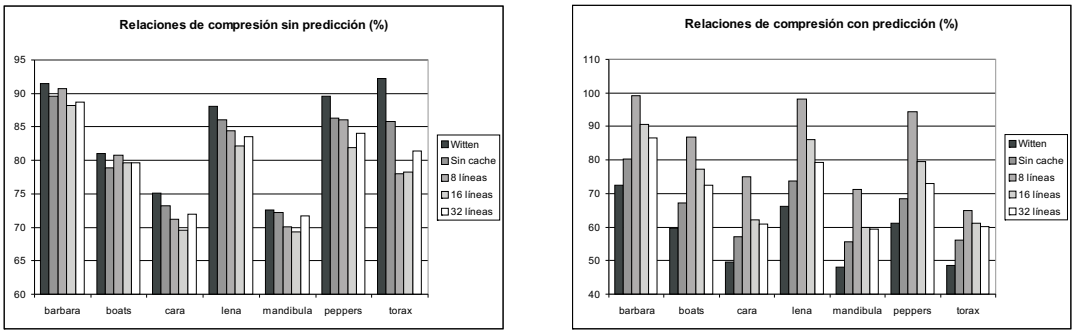


Figura 2.30: Comparativa de relaciones de compresión para varias imágenes. Se compara el método de Witten, un codificador con precisión reducida sin cache y tres caches de 8, 16 y 32 líneas. Los valores más bajos son los mejores.

escalamiento, ya que el acceso a los datos de la cache es totalmente paralelo. En este punto hemos conseguido una ventaja decisiva respecto al codificador convencional, y es que la codificación no ha de ser detenida para escalar las probabilidades, y esto puede suponer diferencias en tiempo de procesamiento muy altas, tal y como se ha visto en la figura 2.28.

2.6 Compresión con predicción

Al aplicar predicción el histograma resultante tiene forma de pico centrado, en el que los datos de máxima probabilidad se encuentran concentrados en el centro del histograma y cuya probabilidad decrece a medida que se alejan de este centro. La forma exacta de los mismos puede variar, si bien se suelen aproximar por ramas de exponenciales decrecientes o gaussianas. En estos últimos casos los perfiles son parametrizables, de forma que se puede llegar a un tratamiento del modelo dependiendo de estos parámetros. En este sentido resulta muy conveniente trabajar con un predictor que garantice histogramas del mismo tipo independientemente de las imágenes. De esta forma se puede optimizar el modelo para estos casos, obteniendo mayor eficiencia. El hecho de disponer de un modelo adaptativo nos libera de construir un codificador orientado hacia un tipo de datos determinado.

Ahora se impone un análisis de los motivos por los que el rendimiento de nuestra cache es inferior al de un codificador convencional. La desventaja tiene dos facetas, una es la diferencia con el algoritmo de precisión completa, que es comprensible, y otra es respecto al algoritmo con precisión reducida sin cache, que no es aceptable ya que ambos tienen la misma complejidad en las operaciones de codificación. La

diferencia entre ambos proviene únicamente del modelo, y es en él donde debemos buscar los motivos del poco favorable comportamiento del codificador.

La principal diferencia entre un modelo convencional y otro con cache reside en los fallos. Los fallos son una fuente de pérdida de compresión importante. Cada vez que un símbolo no se encuentra en la cache, su probabilidad es sustituida por P_{fallo}, que es una probabilidad de compromiso. Cuando el símbolo fallado tiene una probabilidad baja, la importancia de este hecho es pequeña, y la relación de compresión no se ve comprometida por ello. Sin embargo la situación es muy diferente cuando el símbolo fallado tiene una probabilidad alta. Esta probabilidad será varios órdenes de magnitud superior a P_{fallo} (en términos de aritmética binaria) y esta diferencia se traducirá en bits extra enviados a la salida. Por tanto, es importante que los símbolos con probabilidad alta se encuentren siempre en la cache.

Naturalmente, y a pesar del bajo porcentaje de fallos para datos a los que se ha aplicado predicción, esto es imposible de garantizar ya que todo símbolo tiene derecho a entrar en la cache cuando ha provocado un fallo, y esto puede suponer que símbolos poco probables expulsen sistemáticamente a los símbolos más probables. Imaginemos que los símbolos con mayor probabilidad son expulsados de la cache una vez por cada diez que son referenciados. Esto supone que una de cada diez veces serán codificados con P_{fallo} en lugar de su probabilidad real, con un gasto extra de $log_2(P) - log_2(P_{fallo})$ bits cada vez. A lo largo del procesamiento esto se traduce en una pérdida de eficiencia importante.

Además, por norma general los símbolos menos probables son expulsados de la cache en plazos muy cortos. De hecho no suelen ser referenciados en el período que transcurre entre su entrada y expulsión de la cache. Esto lo podemos ver en la figura 2.31 en la que se monitoriza el comportamiento de la cache. Por tanto, incluirlos en la cache no aporta ningún beneficio y supone la expulsión de un símbolo con alta probabilidad de ser referenciado en breve. Vetar el acceso a la cache a estos símbolos poco probables puede ser una medida para mejorar la compresión pero, ¿hasta qué punto hemos de llevar este razonamiento?.

Dada la forma extremadamente pronunciada de los histogramas que nos ocupan no cabe suponer que los símbolos poco probables vean aumentada su probabilidad localmente en ciertos periodos del procesamiento. Si esto ocurriese, por ejemplo en un borde, la dispersión de valores sería lo bastante alta como para que la cache fuese por completo vaciada de los símbolos más probables para ser sustituidos por otros que sólo la ocuparían durante unos pocos ciclos. Es decir, no se produciría la entrada de un conjunto reducido de símbolos asociados a esa transición, sino un continuo de valores con baja probabilidad. Así pues debemos considerar que los símbolos poco probables son mal recibidos siempre, y no tiene sentido aplicar un esquema flexible dependiente del contexto. El siguiente punto es definir la frontera entre los símbolos poco probables y los muy probables.

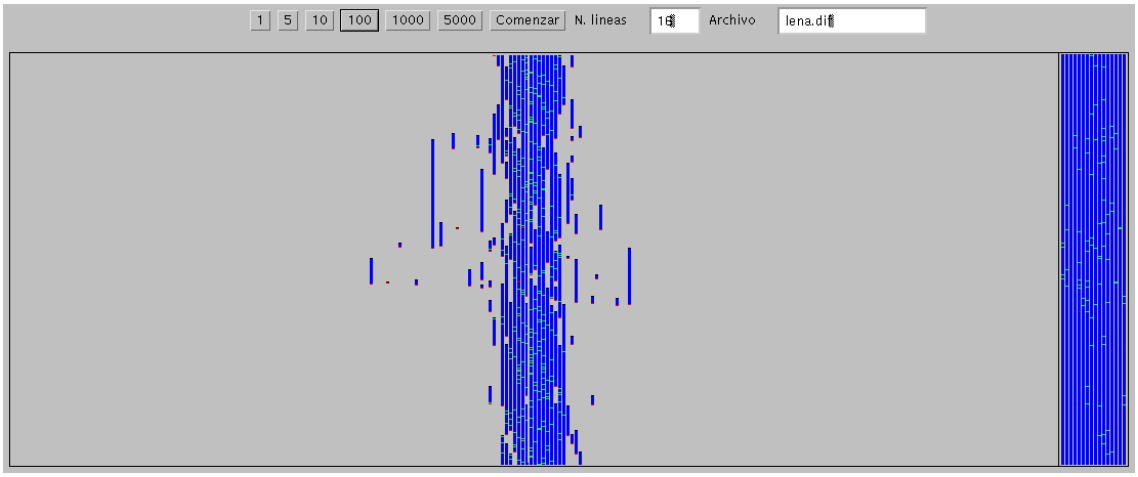


Figura 2.31: Monitor de la cache. El eje vertical representa la evolución temporal. El extremo derecho representa la cache, salpicada de puntos claros (fallos). El resto de la pantalla muestra los símbolos del alfabeto. Los más probables son los positivos y negativos cercanos a cero en el centro del histograma. Los puntos oscuros representan presencia en la cache. Vemos que los símbolos poco probables se mantienen en la cache durante muy poco tiempo.

Proponemos el siguiente método, dejar entrar en la cache a un conjunto de símbolos del entorno del centro del histograma hasta una cantidad múltiplo del número de líneas de la cache. Si el número de símbolos privilegiados ha de ser una, dos, tres, o más veces el número de líneas de la cache es algo que obtendremos por simulación.

En la figura 2.32 se compara la diferencia de compresión con respecto al último esquema de cache visto. Esto es, escalando únicamente las probabilidades de la cache e implementando los productos con sólo 4 bits fraccionales. Se varía el tamaño del conjunto de datos que tienen permitido el acceso a la cache. El tamaño sencillo equivale a que sólo pueden entrar los símbolos que caben en la cache. El tamaño doble equivale a que pueden entrar el doble de símbolos que líneas tiene la cache, y así sucesivamente. Los mejores resultados se consiguen para el tamaño más pequeño, lo que equivale a decir que la cache estará poblada siempre por los mismos símbolos. Iniciando la cache con estos símbolos no existirán reemplazos, y dado que los restantes símbolos serán codificados siempre utilizando P_{fallo}, el valor real de su probabilidad no es relevante y no es necesario almacenarlo. Hemos llegado así a una configuración en la que se prescinde totalmente de la memoria principal y la cache pasa a convertirse en el único elemento de la jerarquía de memoria.

Las implicaciones de eliminar la memoria principal son dobles. Por un lado los

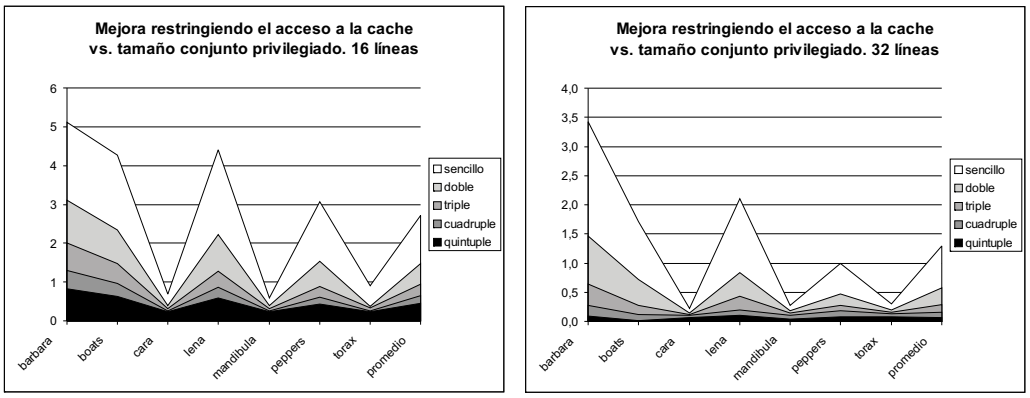


Figura 2.32: Mejoras de compresión (%) al restringir el acceso a la cache. Más es mejor.

resultados de mejora de compresión son evidentes (figura 2.32), sobre todo para las imágenes que peor comprimen. Para radiografías y para la cache de 32 líneas la mejora es menos evidente, pero aún así existe y entramos entonces en el segundo aspecto: haber eliminado la memoria RAM. Sin memoria principal y sin reemplazos el coste de la implementación se reduce considerablemente. Integrar una memoria RAM tiene un coste relativamente alto tanto en área como en tiempo y en consumo de potencia. Además, y tratándose de una memoria entrelazada, debemos considerar el coste de los buses de interconexión.

Al eliminar los reemplazos podemos prescindir de la lógica utilizada para ello, y además surgen múltiples ventajas en la organización interna de la cache al poder controlar mejor la evolución de las probabilidades acumulativas, evitar conflictos con los reemplazos durante los escalamientos, etc. Estos aspectos serán tratados en profundidad en el siguiente capítulo.

Estas pequeñas mejoras deben reflejarse en una nueva configuración capaz de competir con las implementaciones clásicas del algoritmo. En la figura 2.33 vemos que si bien la implementación con precisión completa de Witten [WNC87] sigue siendo superior, nuestros resultados se van aproximando a los de la configuración con precisión reducida y sin cache. La cache de 16 líneas ha mejorado en promedio un 4% y la de 32 un 1.8%, si bien se han producido mejoras por encima de la media en las imágenes que comprimían peor.

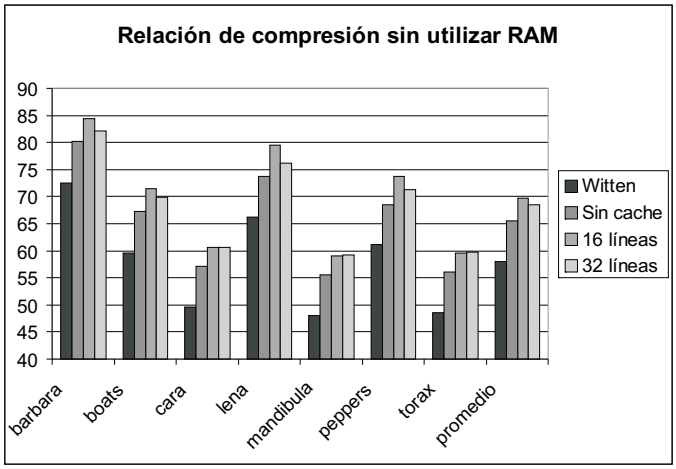


Figura 2.33: Comparación de compresión entre un codificador con precisión completa, uno de precisión reducida y dos caches de 16 y 32 líneas sin memoria principal. Menos es mejor.

2.7 Optimizaciones adicionales

A pesar de todas las mejoras introducidas, que han supuesto un incremento considerable en la eficiencia de la codificación y que podemos apreciar comparando las figuras 2.25 y 2.33, nos encontramos todavía por debajo de la implementación basada en el modelo clásico y que tomamos como punto de referencia. Nos aproximamos mucho a la versión con precisión reducida, pero todavía no es suficiente. En la sección anterior hemos eliminado una fuente de pérdida de compresión al eliminar los reemplazos no deseados (que han resultado ser todos), pero todavía no ha sido suficiente. Los fallos que todavía se producen provocan una pequeña pérdida de compresión que supone la diferencia entre nuestro modelo y el clásico. Sin embargo a este respecto es poco lo que podemos hacer, ya que hemos construido nuestro modelo sobre la premisa de reducir la complejidad de gestionarlo y desde un principio hemos asumido que existiría una pérdida de eficiencia debido a los fallos.

Pero si echamos la vista atrás veremos que hemos obtenido beneficios de compresión de aplicar pequeñas modificaciones, y que muchas de estas modificaciones eran resultado de una simplificación. De esta forma los resultados obtenidos cuando no se aplica predicción son mejores que los originales, a pesar de partir de un esquema que a priori debiera funcionar peor.

Podemos por tanto mejorar la compresión buscando cualquier fuente de ineficiencia y comprobar la mejora que se obtiene al subsanarla. Basándonos en que el modelo óptimo es aquel que se asemeja más al de precisión completa vemos que

nuestro modelo se diferencia de él en los fallos, en la precisión de la aritmética en la iteración y en la estimación de las probabilidades. Los dos primeros aspectos ya han sido comentados, reduciendo el número de fallos al mínimo y encontrando un equilibrio entre el coste de la aritmética y la merma en compresión. Nos queda por tanto centrarnos en la estimación de las probabilidades.

Realizar una normalización de las probabilidades de forma que el rango disponible esté siempre totalmente ocupado es algo que está fuera de nuestro alcance. En general nuestra cache aprovecha todo el rango disponible, a excepción de la porción que está reservada para la tabla virtual, salvo cuando se corrigen las probabilidades, ya que en este momento el valor de todas ellas cae a la mitad. La probabilidad acumulativa del tope de la cache crece de forma sostenida hasta que satura y entonces cae a la mitad de su valor. El tiempo que pasa hasta que el contenido de la cache se recupera supone que todos los símbolos se codificarán con una probabilidad menor durante un tiempo, y así será necesario emplear un bit más para cada símbolo.

Renunciar a que las probabilidades sufran esta caída supone renunciar a la adaptabilidad o bien modificar ligeramente el modelo para hacerla más suave. No consideraremos eliminar la adaptabilidad ya que esto supone construir modelos específicos para una determinada distribución de datos, y esta práctica ya está implementada en otros métodos de compresión [Ric79] [Gol66]. Para hacer que la evolución del modelo sea gradual existen dos opciones: seguir utilizando el mismo mecanismo de incremento de las probabilidades y modificar la corrección, o bien compensar el incremento de las probabilidades con el decremento de otras. Este último sistema está implementado en [HW98], utilizando una cola FIFO en la que los símbolos entrantes ven aumentada su probabilidad y la de los salientes es decrementada.

Comentaremos en primer lugar el sistema de decrementar una probabilidad por cada una que se incrementa. En nuestro caso no tiene sentido utilizar una cola FIFO porque sólo trabajamos con los símbolos situados en la cache. En la arquitectura descrita en [HW98] se utiliza un alfabeto de 256 símbolos y todos ellos pueden entrar en la cola, que introduce una codificación de orden superior de forma similar a como lo hace nuestra cache. Dado que nosotros ya implementamos un sistema similar, añadir una cola equivaldría a duplicar ciertas funciones de la cache. Otra alternativa es hacer una FIFO de tamaño superior a la cache de forma que ese efecto predominase sobre la cache, pero no resulta interesante ya que su coste sería superior al de la cache misma.

En su lugar hemos optado por establecer un turno para decrementar probabilidades. Cada vez que se produce un acierto la probabilidad de ese símbolo es incrementada y se decrementa la del símbolo al que corresponde el turno. En el siguiente ciclo el turno pasa a la siguiente línea y así sucesivamente como se muestra en la figura 2.34. En caso de que un símbolo tenga ya la probabilidad mínima, el turno pasa automáticamente a la línea siguiente o bien se cancela el incremento

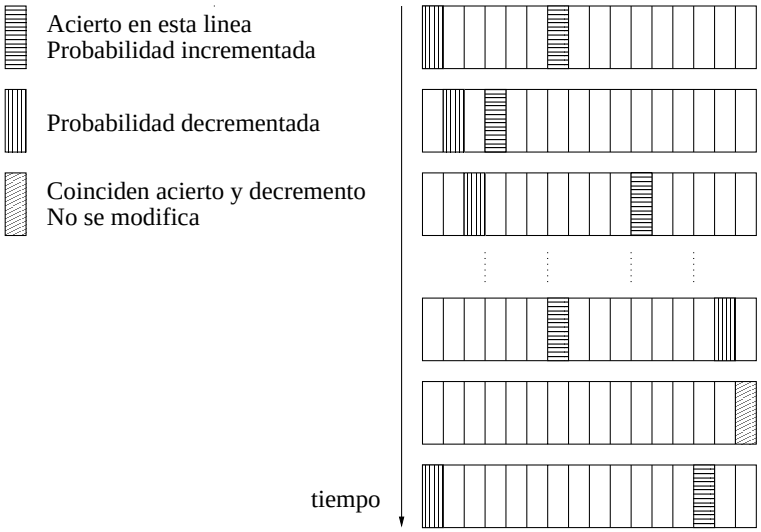
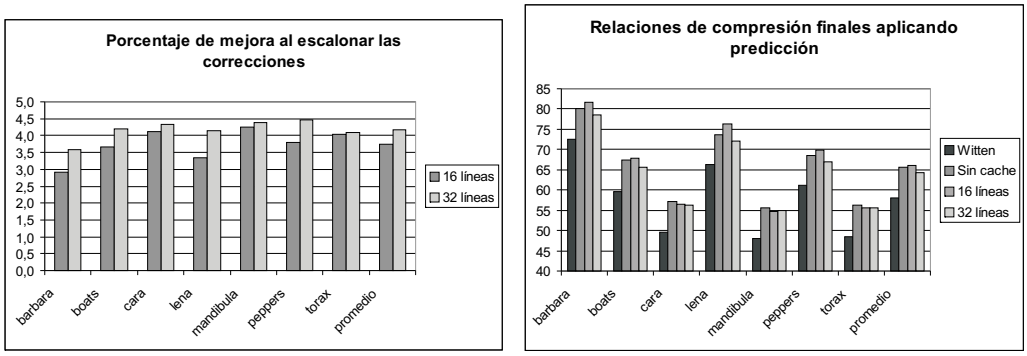


Figura 2.34: Escalamiento de probabilidades línea a línea.

realizado. En cualquier caso los resultados obtenidos son muy malos y no se tabulan. En general las imágenes no se comprimen sino todo lo contrario. El motivo es que en este sistema no se decrementa de forma proporcional a las probabilidades de los símbolos. Los símbolos menos frecuentes tienen las mismas posibilidades de ver decrementada su probabilidad que los más probables, y al cabo de un cierto tiempo sólo los símbolos más frecuentes tienen una probabilidad superior a la mínima.

Descartada esta posibilidad, proponemos conservar el mecanismo de actualización de las probabilidades pero hacer menos drástica la corrección de las mismas. Ya hemos visto en el caso anterior que es conveniente corregirlas de forma proporcional, de manera que seguiremos dividiendo su valor por 2. Procederemos dividiendo una única probabilidad cada vez que sea necesaria una corrección. Utilizaremos el mecanismo de turnos ya descrito. Con esta estrategia se pretende mantener el rango ocupado lo máximo posible, respetando la proporcionalidad que se debe mantener entre las probabilidades reales y las calculadas por el modelo. Naturalmente, existirá una discrepancia entre las probabilidades escaladas recientemente y aquellas que no lo han sido todavía o que ya se han recuperado. Sin embargo esto no debiera tener mayor efecto que el de aumentar el número de correcciones al disminuir los efectos de cada una de ellas. Sin embargo el aumento de las correcciones ha dejado de ser un problema para nosotros por cuanto que podemos realizarlas en un único ciclo sin detener la codificación.

Nuevamente evaluamos esta configuración por medio de simulaciones. Los resultados aparecen en la figura 2.35, donde hemos reflejado por un lado la mejora respecto a la configuración anterior y por otra la comparamos con los dos puntos de



(a) Mejoras relativas respecto a la configuración anterior. Más es mejor

(b) Comparación con modelos sin cache. Menos es mejor.

Figura 2.35: Resultados finales (relativos y absolutos) para la configuración sin memoria RAM escalando una probabilidad cada vez. A las imágenes se les ha aplicado un predictor.

referencia que tenemos: codificadores sin cache con precisión completa y reducida respectivamente.

Los resultados son concluyentes. Las mejoras obtenidas permiten salvar la diferencia existente con el modelo que no utiliza cache. Podemos optar por una cache de 32 líneas, la cual supera siempre los resultados del modelo original, o bien por una solución más económica y utilizar una cache de 16 líneas, que si bien no garantiza mejores resultados en todos los casos, tiene un coste de implementación menor. Dado que las diferencias promedio son inferiores al 1% consideraremos la cache de 16 líneas como la solución más atractiva para la implementación del modelo que proponemos.

El algoritmo con precisión completa sigue funcionando mejor que el que nosotros proponemos. Los motivos son los ya comentados: no resulta práctico implementar en hardware un algoritmo de tan alto coste. Sin embargo las diferencias no se pueden considerar excesivas teniendo en cuenta que estamos hablando del algoritmo de compresión que ofrece mejores resultados de cuantos se utilizan.

Además, la diferencia de coste entre la implementación de uno u otro método es considerable incluso en software. De hecho el algoritmo en software también es muy complicado y lento, motivo por el cual ha sido objeto de múltiples revisiones [Fen95] [Mof90] con el fin de mejorar, aún cuando sea sólo ligeramente los grandes tiempos de computación que necesita.

2.8 Colisiones en los reemplazos

Veremos ahora un detalle relativo a la implementación de la configuración con memoria RAM. Los reemplazos dan lugar a un pequeño problema que no hemos considerado hasta ahora: las colisiones en el acceso al mismo bloque de memoria. Es posible que el símbolo entrante y el reemplazado residan en el mismo bloque de memoria, y por tanto no será posible realizar las dos operaciones, de lectura y escritura, al mismo tiempo. Este pequeño detalle de índole eminentemente práctica puede suponer una importante degradación de las prestaciones en el codificador.

La solución por la que hemos optado es anular los reemplazos que provoquen conflictos. El símbolo que pretendiera entrar puede tener su oportunidad en otra ocasión sustituyendo a otro símbolo con el cual no haya conflicto. Las posibilidades que hemos considerado son dos, que tienen implicaciones en la velocidad de procesamiento que veremos en detalle en el siguiente capítulo.

La primera posibilidad consiste en realizar los dos accesos a memoria dentro del mismo ciclo. Se busca el símbolo entrante en la memoria, y una vez que se conoce el símbolo que se debe sustituir se comprueba si existe un conflicto. En caso afirmativo no sucede nada. Si no existe conflicto se direcciona la memoria para escribir en ella y el registro de la cache aceptará el dato que viene de la RAM cuando termine el ciclo.

Otra posibilidad un poco más elaborada posterga la escritura hasta el ciclo siguiente. La escritura no puede comenzar hasta que se sepa el dato que va a ser sustituido y se haya seleccionado. Si se espera al ciclo siguiente, la duración del ciclo puede ser más corta en caso de que dependa de esta operación. En caso de realizarse así los conflictos tendrían lugar entre la lectura del ciclo actual y la escritura postergada del ciclo anterior.

Los resultados para estas dos posibilidades se muestran en la figura 2.36. Se expresan como las pérdidas de compresión experimentadas con respecto a los resultados de la figura 2.30. Vemos que las diferencias son muy pequeñas, y con el segundo método prácticamente insignificantes. Así pues los reemplazos no son ninguna fuente de problemas ya que se pueden eliminar los conflictos con escasa influencia en la compresión. Una descripción más completa del funcionamiento de los reemplazos se puede encontrar en el siguiente capítulo.

2.9 La cache en el descodificador

A lo largo del presente capítulo hemos construido un nuevo modelo orientado a disminuir la complejidad de la codificación aritmética. Establecimos que se necesitaba un modelo más rápido en el codificador y reducir las búsquedas en el descodificador.

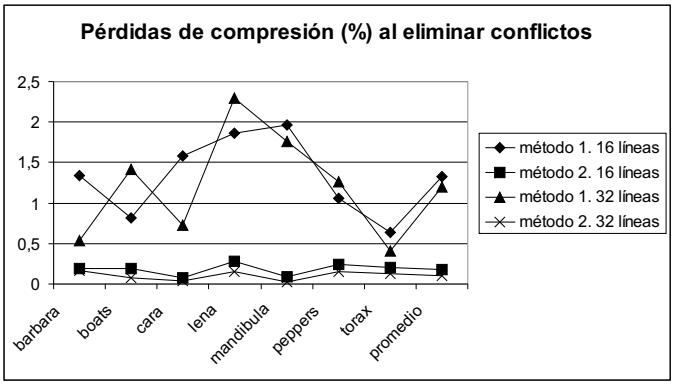


Figura 2.36: Influencia de eliminar los conflictos de memoria.

El primer punto podemos considerarlo cumplido al haber relegado la memoria RAM a un papel secundario e incluso al haberla eliminado en el caso de tratarse de datos con un histograma con las probabilidades concentradas alrededor de un punto. Además, hemos conseguido reducir el coste de la actualización del modelo de forma considerable, tanto en la ejecución normal del algoritmo como llegado el momento en que se han de reescalar las probabilidades. Todo esto nos ha llevado por otro lado a obtener relaciones de compresión competitivas.

Las ventajas de utilizar un modelo con cache en el descodificador aún se han de probar. Por el momento el reescalado de las probabilidades se ha eliminado al igual que sucede en el codificador, lo cual ya supone una ventaja, pero el verdadero problema con el descodificador se debe a la operación normal del mismo. En un modelo convencional se ha de comparar el valor del punto bajo del rango con todas las probabilidades acumulativas a fin de localizar el símbolo que generó ese código. La única forma de eliminar esta operación es introducir una búsqueda jerárquica en n niveles para reducir el número de comparaciones a costa de aumentar el número de niveles de comparación.

Ahora veremos como se integra la cache dentro de la descodificación. Nuestro modelo consta ahora de dos partes: la tabla virtual y la cache (figura 2.2). Por tanto tenemos un total de 256 símbolos en la tabla virtual más tantos como líneas tenga la cache. El problema se simplifica porque todos los símbolos de la tabla virtual tienen la misma probabilidad, P_{fallo}. De esta forma se puede conocer si se está descodificando un fallo o un acierto si se verifica la siguiente condición:

$$C_i < A_i \cdot (P_{fallo} \cdot N) \tag{2.4}$$

siendo A_i y C_i el rango y el punto bajo del intervalo respectivamente y N el número de símbolos. En caso de cumplirse, sabremos el símbolo que ha fallado con la

siguiente operación:

$$simbolo = \lfloor \frac{C_i}{A_i} \cdot \frac{1}{P_{fallo}} \rfloor \tag{2.5}$$

Y en caso de que se haya producido un acierto, la línea l que contiene el símbolo a decodificar es la más *alta* de la cache que cumpla la condición de que su probabilidad acumulativa multiplicada por A_i sea menor o igual que el punto bajo del rango.

$$A_i \cdot S_{linea\ l} \leq C_i \tag{2.6}$$

o equivalentemente, dado que la probabilidad acumulativa es la suma de las probabilidades y que la cache se encuentra sobre la tabla virtual:

$$A_i \cdot (\sum_{j=0}^{j<l} P_{linea\ j} + P_{fallo} \cdot N_{simbolos}) \leq C_i \tag{2.7}$$

Expresión que se puede simplificar hasta quedar:

$$\sum_{j=0}^{j<l} P_{linea\ j} \leq \frac{C_i}{A_i} - P_{fallo} \cdot N_{simbolos} \tag{2.8}$$

En las ecuaciones 2.5 y 2.8 hemos hecho trampa. Hemos obtenido una importante simplificación de las operaciones al sustituir una multiplicación por A_i en un término por una división en el otro. Aparentemente no supone ningún ahorro, pero si intentamos descodificar un fallo sin tomarnos esta licencia ya no podemos hacerlo en un sólo paso, sino que habría que realizar tantas comparaciones como símbolos tiene la tabla virtual. Este mismo artificio nos permite reducir la complejidad en la descodificación de los aciertos, ya que la comprobación original de la ecuación 2.7 implica un producto por A_i por cada comparación mientras que en la comprobación 2.8 el término de la derecha es constante para todas las líneas de la cache.

En pocas palabras, la descodificación utilizando la cache se puede resolver de forma fácil y eficiente. Podemos saber de forma sencilla si hay un fallo y descodificarlo (ecuaciones 2.4 y 2.5), y podemos descodificar los aciertos con un número de comparaciones igual al de líneas de la cache o bien menos si se descodifica en varios niveles. Por tanto se ha conseguido superar la limitación existente en el modelo original reduciendo la búsqueda sobre 256 símbolos a 16 o 32 dependiendo del tamaño de la cache. Todo esto es cierto siempre que seamos capaces de implementar en el descodificador un divisor a un coste factible. De lo contrario no habrá ninguna ventaja respecto al modelo sin cache.

2.9.1 División en el descodificador

En aplicaciones software implementar una division es un tarea trivial, pero en hardware resulta tremendamente costosa. A nuestro favor tenemos el hecho de que la precisión con que se implementaron los productos fue baja. Únicamente se utilizaron dos bits fraccionales, y el porcentaje de compresión que se perdió fue ínfimo.

Cuando se realiza la operación $C_{i+1} = C_i + A_i \cdot S_i$, el número entero C_i se convierte en un número con dos bits fraccionales debido a que A_i es un número fraccional con dos bits de precisión. Si queremos mantener a C_i como un número entero se deben truncar los dos bits fraccionales. Esta es la solución habitual siempre que se implementa codificación aritmética con aritmética de enteros. El hecho de truncar estos dos bits hace que la operación ya no sea reversible. Se puede comprobar como la eliminación de estos dos bits da lugar a errores en caso de intentar realizar una división. Estos errores son suficientes para dar lugar a una descodificación errónea, y de ahí a una cascada de errores que provocarían el fracaso de la descodificación.

La única solución posible para hacer que la multiplicación sea reversible es conservar los dos bits fraccionales, y realizar las operaciones aritméticas contando que ellos tanto en el codificador como en el descodificador. El precio no es demasiado grande ya que sólo utilizamos dos bits. En caso de que la compresión estuviese fuertemente comprometida con la precisión de los productos, nos encontrariamos ante la necesidad de ampliar todavía más en ancho de palabra.

Habiendo asumido que será necesario ampliar dos bits el tamaño de palabra, hemos de decidir un esquema para implementar la división en el descodificador, ahora que la multiplicación sí es reversible. Dado que con dos bits fraccionales sólo existen cuatro multiplicadores posibles la solución más fácil es almacenar los inversos de los multiplicadores con la precisión suficiente para asegurar que un producto por el inverso sea equivalente a calcular el cociente.

Dado un tamaño de palabra se pueden obtener los inversos de los multiplicadores con la precisión mínima que garantice que el resultado será el correcto. Así, dado un valor de A_i se procederá a seleccionar su inverso y multiplicarlo por C_i para obtener C_i/A_i. Esta sencilla operación sustituye de forma eficiente a muchas multiplicaciones. En el siguiente capítulo, dedicado a la implementación hardware de un codificador con modelo basado en cache, veremos una implementación eficiente de un divisor para un tamaño de palabra dado. Se describirá todo el proceso de cálculo de los inversos e implementación de la división en todos sus detalles.

2.10 Conclusiones

En este capítulo se ha introducido un nuevo modelo basado en una pequeña memoria cache [OB98, OB99]. Las operaciones más costosas de la codificación y descodifi-

cación se restringen ahora a la cache, habiéndose reducido de modo considerable el coste de las mismas. Se ha estudiado la mejor organización de la cache, incluyendo algoritmos de reemplazo para mantener la relación con la memoria principal. También se ha establecido la precisión idónea para obtener buenas relaciones de compresión al menor coste, y se ha eliminado el problema de escalar las probabilidades.

En el caso básico del codificador en que no se utiliza predicción, y por tanto no se conocen las propiedades estadísticas de los datos, nuestro modelo obtiene los mejores resultados. Conseguimos superar no sólo a un modelo convencional, sino al algoritmo clásico implementado en software, que es un punto de referencia importante aun cuando no puede ser trasladado al hardware.

Cuando sí se utiliza predicción tenemos un modelo mucho más sencillo en el que no existen reemplazos. Este caso es extensible a otro tipo de datos como codificación subbanda [GT88] y transformadas de distintos tipos. En este caso no se supera la eficiencia del codificador con precisión completa, pero si la del modelo sin cache con precisión reducida. Sin embargo se ha conseguido llegar a una implementación realmente económica en todos los aspectos.

La descodificación es ahora una tarea mucho más sencilla en la que sólo se ven involucrados los símbolos contenidos en la cache en lugar de todos los del modelo. Para ello se ha llegado a la conclusión de que era necesario implementar un divisor de bajo coste, a lo que ha ayudado el hecho de que durante toda el desarrollo se ha apostado por soluciones basadas en aritmética con baja precisión.

Capítulo 3

Arquitectura con jerarquía en dos niveles

En este capítulo presentamos la implementación hardware de un codificador y un descodificador basados en el modelo descrito en el capítulo anterior. Presentaremos una visión de la arquitectura partiendo de un punto de vista global hasta el nivel de detalle, incidiendo en los aspectos de valoración de velocidad y coste a fin de realizar una comparación con otras implementaciones.

3.1 Parámetros de configuración

En el capítulo anterior realizamos un estudio sobre las mejores configuraciones para dos casos: imágenes comprimidas directamente y tras aplicar un predictor. Éstas eran diferentes, de manera que optaremos por hacer nuestro estudio sobre el caso más completo. Resumiendo, la mejor configuración para cada caso era:

- Comunes
  - Cache de asignación directa
  - Productos implementados con 2 bits fraccionales
- Sin aplicar predicción
  - Reemplazos entre la cache y la memoria RAM
  - Aritmética con 9 bits de precisión
  - Se escala la cache, no la RAM
  - El tamaño más adecuado son 16 líneas

- Aplicando predicción
  - No se utiliza memoria RAM y por tanto no existen reemplazos
  - Aritmética con 10 bits de precisión
  - La cache se escala de forma escalonada
  - El tamaño más adecuado son 32 líneas

Si bien las características divergen ligeramente las diferencias son menores de lo que puede parecer. En primer lugar no existe gran diferencia entre utilizar una cache de 16 o 32 líneas, y la primera es mucho más sencilla. Con el tamaño de palabra sucede lo mismo, ya que las diferencias son muy pequeñas, de forma que nos decantaremos por un tamaño de 10 bits. Los reemplazos en cambio si son una diferencia mayor. Dado que la implementación sin reemplazos es más sencilla resulta más conveniente tratar en detalle una implementación con reemplazos y apoyarnos en ella para describir más adelante, en el capítulo 4, la versión sin reemplazos. Resumiendo, estas son las características que tendrá nuestra implementación de un modelo con jerarquía en dos niveles para imágenes sin aplicar predicción.

- Cache de asignación directa con 16 líneas
- Reemplazos al ocurrir un fallo
- Se escala la cache completa cuando se supera la precisión
- Aritmética con 10 bits de precisión
- Productos implementados con 2 bits fraccionales

3.2 Esquema general

Las estructuras del codificador y el decodificador pueden ser descritas de forma cualitativa a través el camino que siguen los datos durante su procesamiento.

Para describir el codificador hacemos uso de la figura 3.1 y de los puntos marcados en la misma. Las referencias numéricas que aparecen en el texto de esta sección se refieren los puntos indicados en la figura. El dato entrante, símbolo k, (1) debe ser localizado en la memoria cache (2). En paralelo, se direcciona la memoria RAM (3) en previsión de que se produzca un fallo. Esta operación puede ser abortada en caso de que se produzca algún conflicto de acceso.

Se sabe si el símbolo requerido se encuentra en la cache mediante una comparación en paralelo con todos los símbolos presentes en ella (4). En caso de acierto la

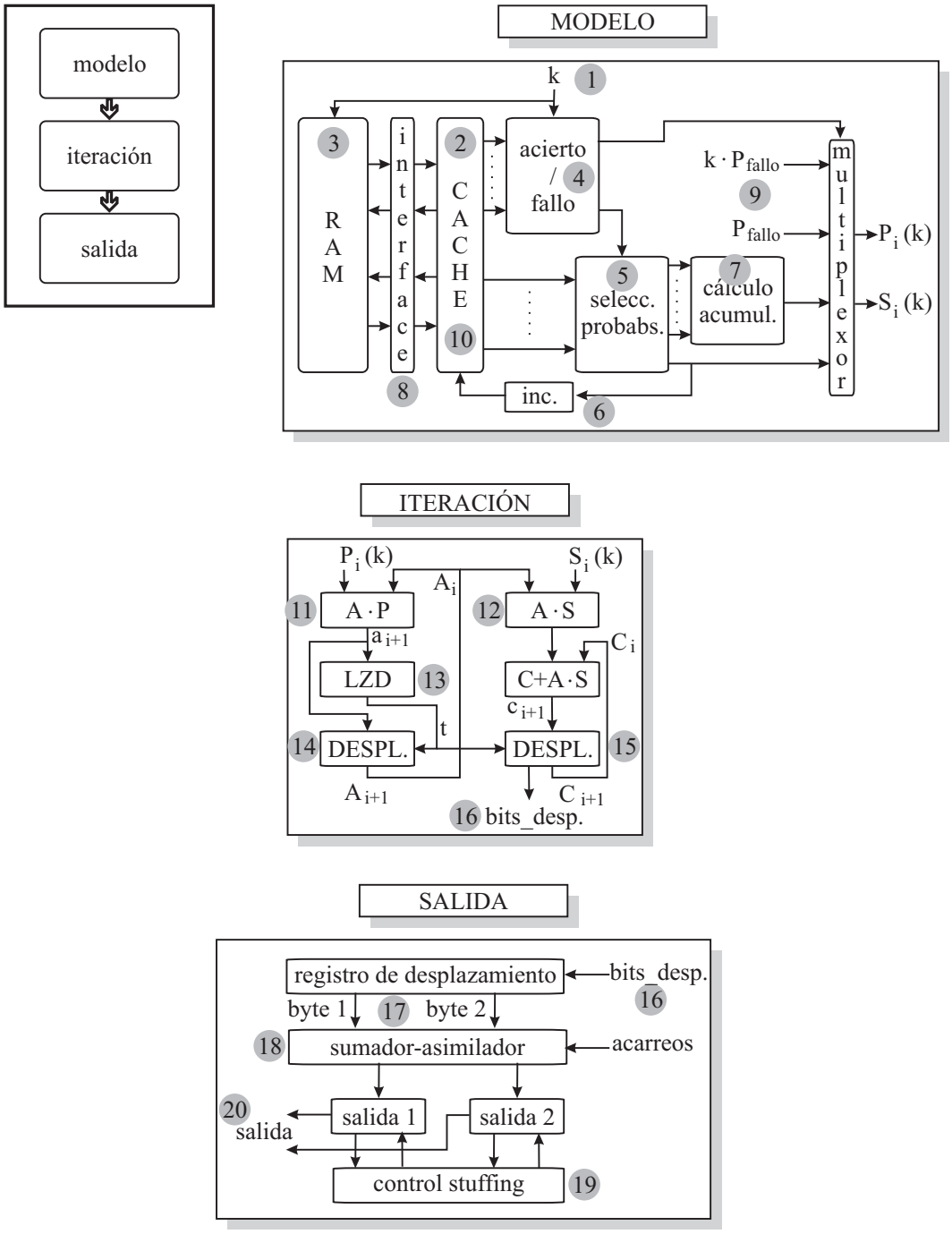


Figura 3.1: Esquema general de la arquitectura del codificador.

probabilidad es seleccionada (5) para ser utilizada por la iteración y también se incrementa (6) para actualizar el modelo. Al mismo tiempo se suman las probabilidades oportunas (7) para calcular la probabilidad acumulativa del símbolo referenciado (ver ecuación (2.2)).

En caso de fallo se detecta que línea será objeto de reemplazo y se emprenden las acciones para ello (8). Se comprobará en primer lugar si existe un conflicto de acceso a memoria. Esto dependerá de la estrategia que se adopte en los reemplazos. Si efectuamos el reemplazo en un sólo ciclo el conflicto aparecerá con el símbolo que se está leyendo, mientras que si se pospone la escritura hasta el ciclo siguiente el conflicto puede haber aparecido ya con una escritura del ciclo anterior. En cualquier caso, si el reemplazo prospera, un registro de la cache aceptará cargar datos desde la RAM al final del ciclo.

Al ocurrir un fallo, el modelo genera una probabilidad y una probabilidad acumulativa para codificarlo. Estas serán P_{fallo} y $k \cdot P_{fallo}$ (9) para un símbolo k. En este caso, el modelo no se actualiza.

Las funciones del modelo no terminan aquí, sino que resta todavía por considerar el caso en que las probabilidades acumulativas crezcan por encima del límite permitido, a lo que llamamos saturación del modelo. En todo momento se contabiliza el valor que toma la probabilidad acumulativa del tope de la cache (10), a la que llamaremos S_T. Si este valor supera el límite se escalan las probabilidades de todos los símbolos de la cache. El escalado en sí no es costoso, pero si puede serlo la detección, además los datos de la cache no estarán disponibles. Para subsanar este inconveniente sin detener la codificación es posible escalar la cache y procesar el símbolo entrante como un fallo (sin efectuar reemplazo) aún cuando el símbolo resida en la cache. La pérdida de compresión debida a este artificio es despreciable al ser muy bajo el número de saturaciones.

Los párrafos anteriores competen al modelo. Éste facilita una probabilidad y una probabilidad acumulativa que serán utilizadas en la iteración para actualizar los valores del rango. El proceso sigue las ecuaciones 1.5. Las operaciones sobre el rango A_i (11) y sobre el punto bajo C_i (12) comienzan al mismo tiempo, pero esta última se demora más. De forma que tras actualizar A_i, se calcula el desplazamiento t (13) y se normalizan A_i (14) y C_i (15). La velocidad con que se realizan estas operaciones es alta debido a la baja precisión utilizada. La normalización implica desplazar bits hacia la sección de salida (16).

Estos son empaquetados en bytes (17) y se previene la propagación de acarreos asimilándolos (18) o, en el caso en que la asimilación no sea posible, se introducen bits de *stuffing* (19) intercalados en la secuencia de salida (20).

Como vemos todo el codificador es muy similar a lo descrito en el primer capítulo excepto el modelo, en el que nos hemos detenido algo más. La salida del codificador se almacena o se transmite hasta que llega el momento de la descompresión.

En el decodificador (figura 3.2), dados los valores de A_i (21) y C_i (22) el primer paso hacia la descodificación pasa por calcular el cociente C/A (23) como ya hemos visto en la sección 2.9. El valor del cociente (24) nos dice si se está descodificando un fallo o un acierto (25). En el primer caso la descodificación es trivial (26) dado que todos los fallos tienen la misma probabilidad.

En caso de acierto debe compararse con las probabilidades acumulativas de todos los símbolos de la cache (27). Éstas deben haber sido calculadas (28) en paralelo con la división, solapando ambas operaciones en el mayor grado posible, ya que supone una cantidad de operaciones aritméticas considerable. Una vez obtenidos el cociente y las probabilidades acumulativas llega el momento de compararlos (27) y obtener así la línea (29) que contiene el símbolo requerido. Éste es seleccionado, al tiempo que su probabilidad y su probabilidad acumulativa (30). La probabilidad es incrementada (31) y vuelve a la cache.

En caso de haberse producido un fallo, se utiliza P_{fallo} y $k_{fallo} \cdot P_{fallo}$ como probabilidad y probabilidad acumulativas (32), siendo k_{fallo} el símbolo que ha fallado (26). Las operaciones de reemplazo (33) se realizan igual que en el codificador y lo mismo sucede con la corrección de las probabilidades.

La iteración sobre el rango A (34) es igual que el codificador. Sin embargo la iteración sobre C (35) se diferencia en un signo (36) y en la entrada de nuevos bits (37). Estos residen en un buffer del que se extraen a cada ciclo tantos bits como demande el descodificador. Esta cantidad depende del desplazamiento necesario para la normalización (38) y es equivalente al que se realiza en el codificador.

Como vemos, a causa del divisor y de las comparaciones el descodificador abarca muchas más operaciones que el codificador. La iteración es prácticamente idéntica a la del codificador, salvo que se realiza una resta en lugar de una adición. Además la normalización da lugar a la entrada de nuevos bits, en lugar de desplazarlos fuera.

3.3 El modelo en el codificador

El procesamiento en el codificador comienza por el modelo de probabilidades. A cada símbolo se asigna una probabilidad y una probabilidad acumulativa cuya obtención dependerá de la presencia o no del símbolo en la cache. Los elementos básicos a nivel físico son los siguientes.

- Memoria RAM para almacenar las probabilidades de 256 símbolos
- Memoria cache con capacidad para 16 símbolos con sus probabilidades
- Interface entre la cache y la memoria principal
- Lógica de detección de aciertos y fallos

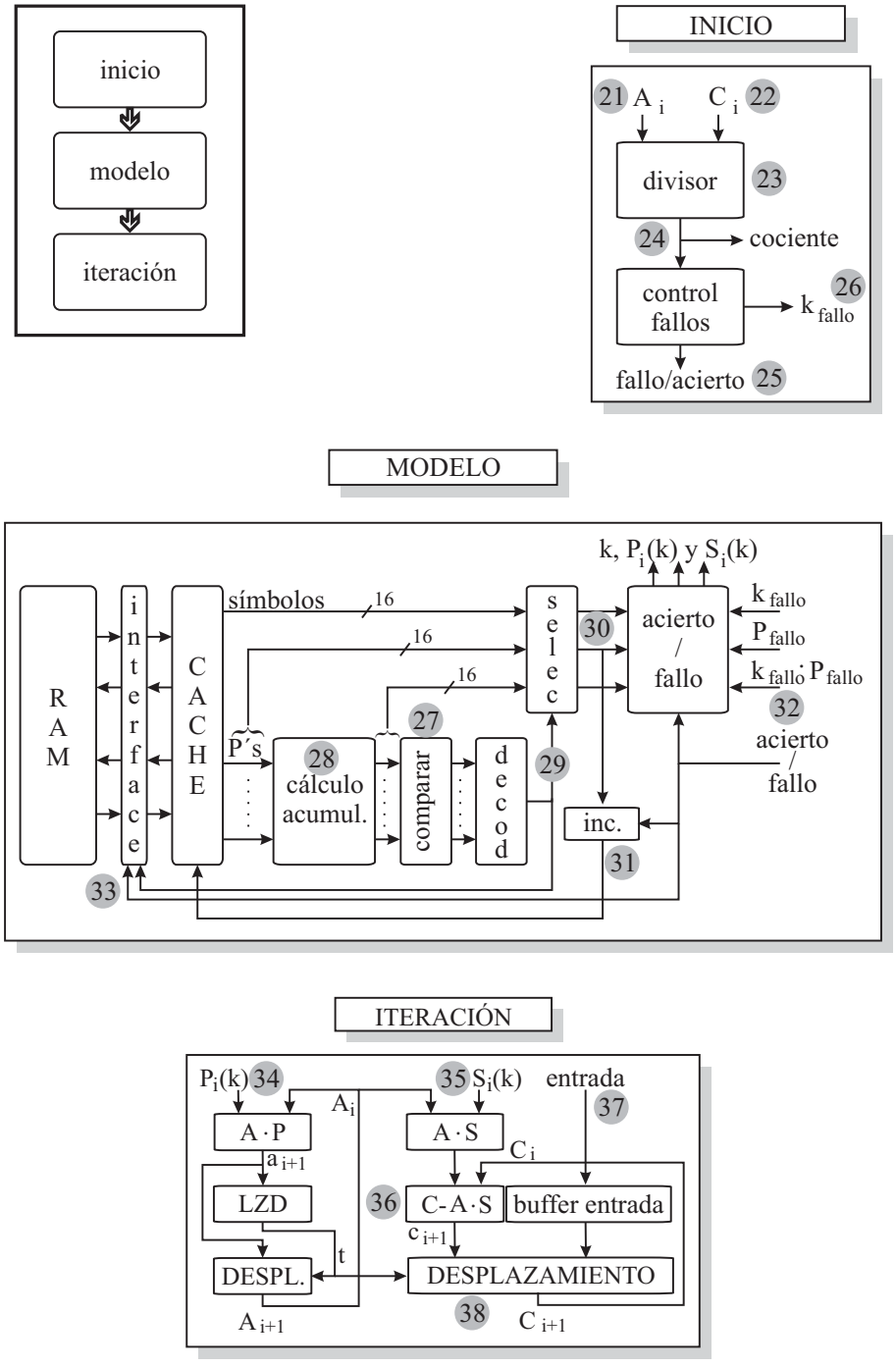


Figura 3.2: Esquema general de la arquitectura del descodificador.

- Lógica de selección de probabilidades
- Sumadores para el cálculo de las probabilidades acumulativas
- Lógica de actualización del modelo
- Control de saturación del modelo y corrección de probabilidades

3.3.1 Aritmética utilizada

Entre las características del codificador hemos establecido que se utilizarán 10 bits de precisión. Dejaremos entonces que las probabilidades acumulativas crezcan hasta agotar la precisión. En aritmética entera esto supone valores menores de 1024, y si el valor asignado a P_{fallo} es de 2^{-10} esto se traducirá en $P_{fallo} = 1$. Sólo queda entonces establecer el valor máximo que pueden tomar las probabilidades para definir adecuadamente los tamaños de palabra.

Dado un alfabeto de 256 símbolos y los valores anteriores, la tabla virtual ocupará un espacio de 256 unidades de un rango de 1023. Por tanto el espacio disponible para la cache es de sólo 767 unidades, 3/4 del total, y la precisión con que deben expresarse las probabilidades será reducida también. Limitaremos entonces el crecimiento de las probabilidades de forma que tomen siempre valores menores de 256 (8 bits). Una vez llegadas al valor máximo no se incrementará la probabilidad aunque el símbolo sea referenciado. Esto se denomina aritmética de saturación.

Se utilizará aritmética con acarreo almacenado para el cálculo de las probabilidades acumulativas y aritmética convencional con saturación para actualizar las probabilidades.

3.3.2 La memoria RAM

Desde años atrás es habitual integrar memoria RAM dentro de una arquitectura de propósito específico y está posibilidad está contemplada en casi todos los programas de diseño. La memoria RAM proporciona una densidad de integración muy superior a la del almacenamiento en registros, en parte por su regularidad y en parte por el hecho de que las conexiones internas han sido optimizadas por los creadores del software de diseño CAD. Sin embargo su velocidad de acceso y la flexibilidad en los mismos son sus puntos débiles. En su implementación básica sólo es posible acceder a un dato por cada ciclo y éste es bastante dilatado. En caso de necesitar un acceso de lectura-escritura o accesos múltiples la complejidad crece de forma rápida, perdiéndose las ventajas de utilizar este tipo de memoria.

Necesitamos un total de 256 bytes, divididos en cuatro bloques para permitir una concurrencia aceptable en los accesos. Este es el esquema que mostramos en la

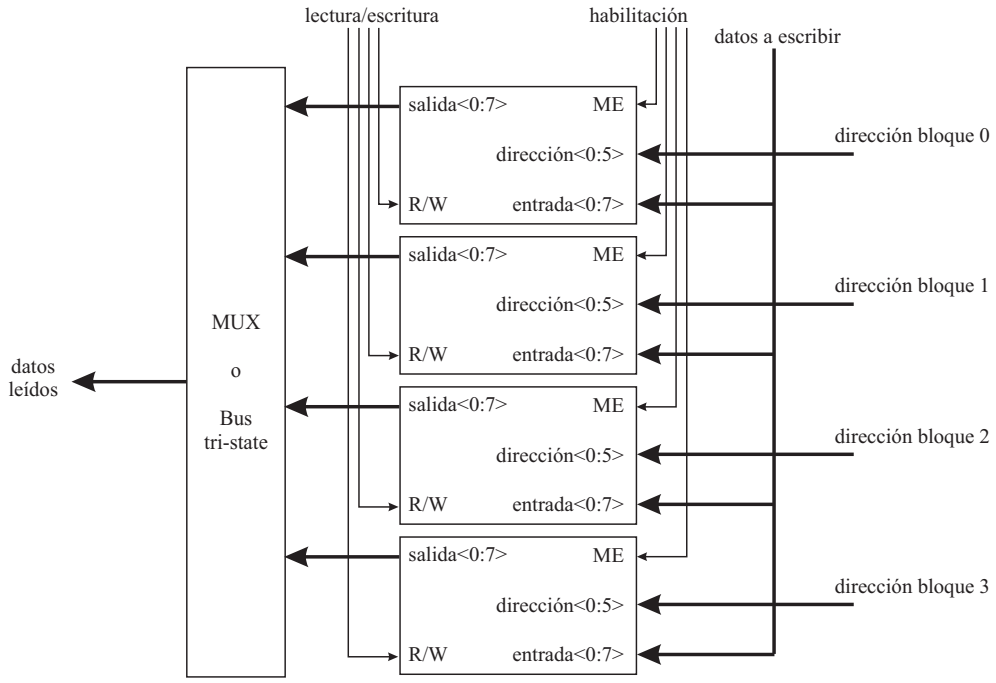


Figura 3.3: Entrelazamiento de la RAM. Señales de control y ruta de datos.

figura 3.3. Los cuatro bloques tienen señales individuales de habilitación y selección de lectura/escritura, así como el direccionamiento. Cada bloque necesita 6 bits para direccionar su contenido, ya que contienen 64 bytes cada uno. De los 8 bits que forman cada símbolo se utilizan los bits de pesos 2^7, 2^6 y del 2^3 al 2^0 para direccionar el contenido de los bloques de memoria. Los bits de peso 2^5 y 2^4 direccionan el un bloque de entre cuatro. Ya que se pueden realizar hasta dos accesos a memoria por ciclo (pero a distintos bloques) estos dos bits no direccionan directamente los bloques de memoria, sino que en el interface entre la memoria y la RAM se habilitan las señales oportunas en virtud de las operaciones que haya que realizar y la existencia o no de reemplazos. Ya que sólo puede existir una operación de escritura y/o una de lectura cada ciclo, la entrada de datos es la misma para todos los bloques y la salida se vuelca bien a un multiplexor, o bien a un bus tri-state. Esta última solución puede ser más rápida y económica.

Para estimar el tiempo de acceso utilizaremos los datos facilitados en las librerías de ES2 para 0.7 micras [Str92]. Y a fin de hacer la estimación lo más independiente posible de la tecnología traduciremos el retardo obtenido a un tiempo equivalente expresado en forma de niveles de puerta *nand* con una carga de tres puertas del mismo tipo a la salida. Esta forma de evaluar los tiempos será la utilizada a lo largo de esta memoria. El tiempo de un ciclo de lectura o escritura es de aproximadamente

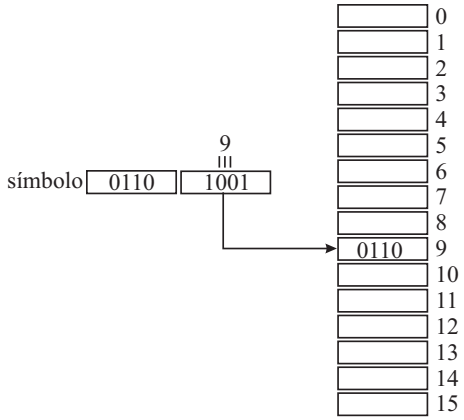


Figura 3.4: Direccionamiento de la cache y almacenamiento de los símbolos.

19.4 retardos de puerta *nand* es un tiempo sensiblemente superior al de un registro, que necesita 7.4.

3.3.3 Estructura de la memoria cache

Las características principales ya han sido comentadas, contiene un total de 16 líneas y para cada una de ellas se almacena un símbolo y una probabilidad. Veremos ahora algunos detalles de su implementación.

Las probabilidades ocupan 8 bits, pero los símbolos se pueden almacenar utilizando sólo 4. Esto se debe a que tratándose de una cache de asignación directa, la posición de un símbolo dentro de la cache viene dada por una porción de sus bits. Entonces dada la posición de un símbolo en la cache, parte de sus bits se sobreentienden. Un ejemplo se ve en la figura 3.4. En nuestro caso, direccionar una línea de entre 16 precisa 4 bits, que se pueden obviar al almacenar el símbolo en la cache.

Cada línea contiene no sólo los registros, sino la lógica para habilitar la carga de nuevos datos y escalar la probabilidad en caso de saturación. La estructura es muy sencilla, tal y como se ve en la figura 3.5, se trata de registros, multiplexores y las señales de control para cargar cada dato. Las posibilidades para actualizar el símbolo son mantener el mismo, cargar un nuevo símbolo fruto de un reemplazo, o cargar el símbolo por defecto al iniciar la cache.

Para la probabilidad existen dos posibilidades más, escalamiento de la probabilidad, e incremento de la misma tras un acierto. Se utilizan dos multiplexores. El primero de ellos, a la salida del registro selecciona entre la probabilidad y un medio de la misma. La operación que se realiza para dividir entre 2 es trivial, se introducen en el multiplexor los bits desplazados una posición a la derecha, por lo que se pierde

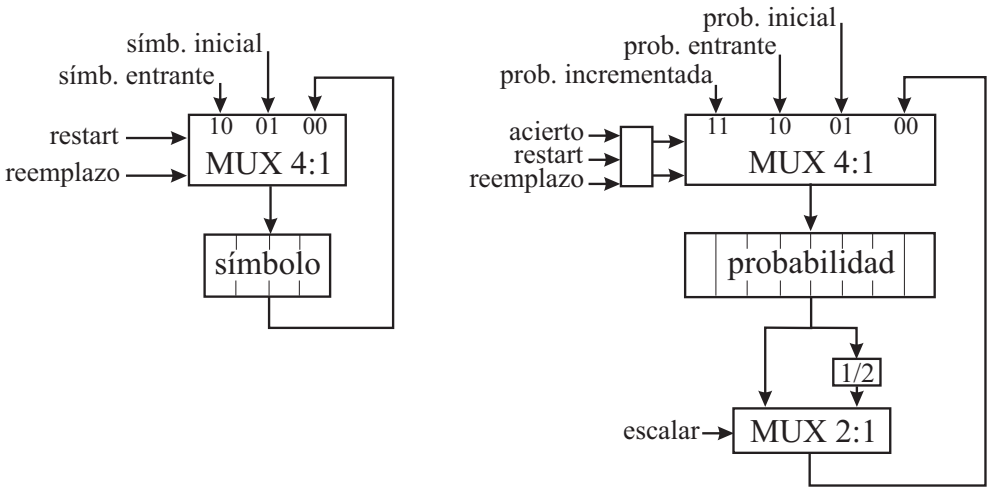


Figura 3.5: Estructura de una línea de la cache, incluyendo control de carga de datos y escalamiento de las probabilidades.

el bit menos significativo. Para impedir que la operación 1/2 de por resultado cero, el bit menos significativo de la probabilidad escalada es siempre 1. Esta pequeña modificación no tiene efectos sobre la compresión. Al poner este multiplexor en esta posición disponemos del valor escalado de la probabilidad en caso de que haya escalamiento. Éste se utiliza para recalcular S_T. Dado que la operación de escalamiento no es lineal (tampoco lo es en el algoritmo original en el que $P \leftarrow (P+1)/2$), no se puede obtener el nuevo valor de S_T escalando el valor previo sin más , sino que tendremos que introducir una corrección. Un pequeño inconveniente de utilizar aritmética entera.

El segundo multiplexor que controla las probabilidades permite cargar el valor realimentado (escalado o no), la probabilidad incrementada tras un acierto, una nueva probabilidad tras un reemplazo, o la probabilidad al reiniciar la cache. Aunque hay tres señales de control, se pueden resumir en dos de forma trivial. Posteriormente, al estudiar las distintas posibilidades que se pueden dar durante la operación del modelo veremos como no se puede dar el caso de más de una señal de control esté activada al mismo tiempo.

3.3.4 Detección de aciertos y fallos y selección de las probabilidades

Tras la llegada de un nuevo símbolo, se compara éste con el contenido de la cache. En la figura 3.6 se muestra como se realiza la comparación teniendo en cuenta que en la cache sólo se guardan 4 bits de cada símbolo. Como resultado de las comparaciones

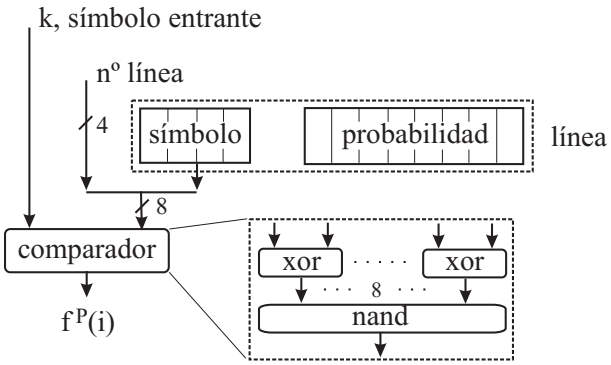


Figura 3.6: Comparaciones en cada línea de la cache del codificador.

se generan 16 señales que indican fallo o acierto en cada línea y que llamaremos $f^P(i)$ para cada línea i.

Estas señales nos sirven para tres propósitos: obtener una señal unificada f/a que indica fallo o acierto, seleccionar la probabilidad del símbolo que ha generado el acierto, y seleccionar las probabilidades que se utilizarán para calcular la probabilidad acumulativa. En la figura 3.7 vemos como se obtienen éstas y como se seleccionan las probabilidades.

Si alguna de las $f^P(i)$ tiene un valor no nulo entonces se ha encontrado el símbolo en la cache. La señal $fallo/acierto$ (f/a) se obtiene entonces con una simple operación *or* con todas las $f^P(i)$. La probabilidad del símbolo requerido se selecciona con un multiplexor 16:1 en el que las señales de control, $f^P(i)$, están descodificadas. Una vez obtenida se incrementa y se devuelve a la cache. En caso de haber ocurrido un fallo se selecciona en su lugar P_{fallo}.

Las probabilidades almacenadas en líneas situadas por debajo de la línea que provocó el acierto se utilizan para calcular la probabilidad acumulativa. Para seleccionarlas utilizamos las señales $f^S(i)$ que se obtienen utilizando sólo las líneas j mayores que i, es decir:

$$f^S(i) = OR_{j=i+1}^{j<16} f^P(j) \tag{3.1}$$

Con ellas se deja pasar cada probabilidad a los sumadores para calcular la probabilidad acumulativa o bien se anula su valor de forma que no contribuya a la suma. A estos valores les llamamos $B(i)$.

3.3.5 Cálculo de las probabilidades acumulativas

Las probabilidades que hayan sido seleccionadas para calcular la probabilidad acumulativa se introducen en un árbol de sumadores de acarreo almacenado (figura 3.8)

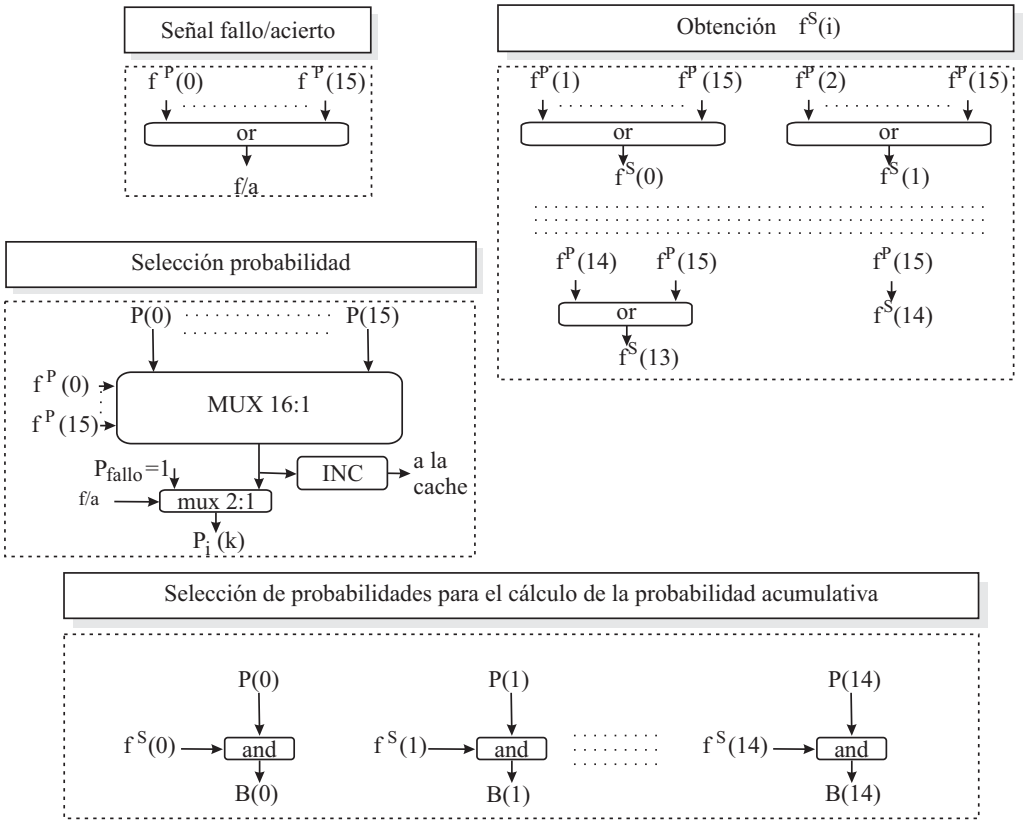


Figura 3.7: Selección de las probabilidades en la cache del codificador. Obtención de la señal fallo/acierto (f/a).

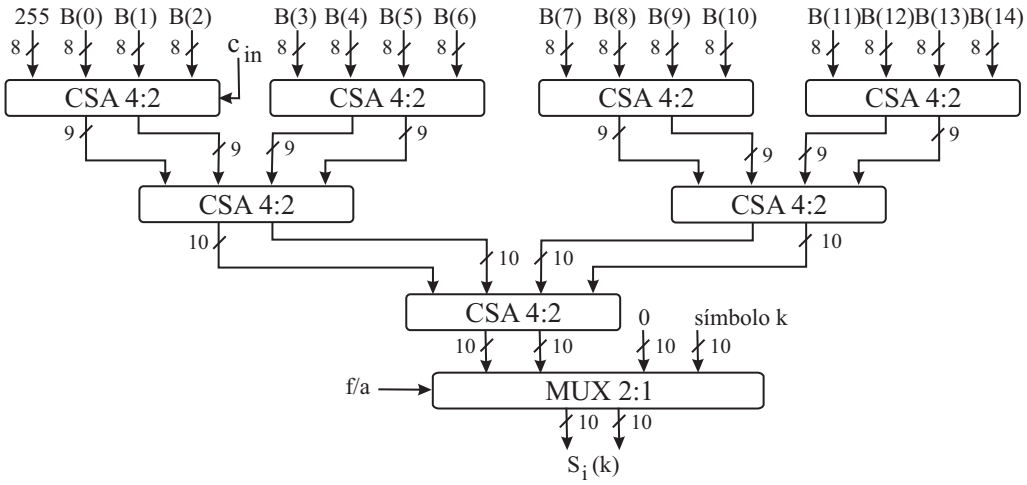


Figura 3.8: Cálculo de las probabilidades acumulativas en la cache del codificador.

que nos dará su valor en formato redundante, una palabra de semi-suma y otra de acarreo. Se utilizan sumadores del tipo CSA 4:2, que reducen 4 operandos a sólo 2 en un tiempo inferior al que sería necesario con los clásicos sumadores 3:2. La cache consta de 16 líneas y tendremos que sumar igual número de probabilidades. Algunas habrán sido sustituidas por ceros, pero se suman todos los valores. Reducir 16 sumandos a sólo 2 utilizando sumadores 4:2 supone utilizar tres niveles, con un total de 7 sumadores.

Realmente el número máximo de probabilidades que es necesario sumar para obtener la probabilidad acumulativa de un símbolo de la cache es de 15, ya que se suman las probabilidades de los símbolos situados por debajo en la cache. Sin embargo es necesario sumar la longitud de la tabla virtual a mayores.

A un primer nivel de sumadores llegan 15 probabilidades, números de 8 bits. Éstas se han seleccionado en la etapa anterior (sección 3.3.4). Al sumar 4 de ellas se obtienen dos operandos de longitud 9 bits. En el primero sumador se introduce también la longitud de la tabla virtual, 256, como 255 más un acarreo. Esto se hace así para no agrandar el tamaño del sumador. Tras el segundo nivel de sumadores se obtienen dos parejas de palabras de 10 bits. Dado que el valor máximo de las probabilidades acumulativas es siempre menor que 2^{10}, el tamaño del resultado no crecerá más.

Únicamente nos queda por ver que ocurre cuando se produce un fallo. En este caso, la señal f/a selecciona en el multiplexor que se utilice $k \cdot P_{miss}$ como probabilidad acumulativa del símbolo k que ha provocada el fallo. Como $P_{miss=1}$ se utiliza k como palabra de suma y 0 como palabra de acarreo.

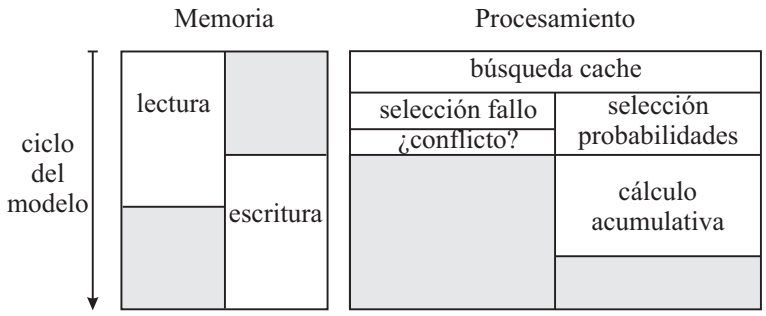


Figura 3.9: Diagrama de tiempos para los reemplazos en un solo ciclo.

3.3.6 Los reemplazos

En el capítulo anterior (sección 2.8) vimos como evitar las colisiones en los reemplazos anulando aquellos que fuesen conflictivos. Consideramos dos posibilidades: realizar el reemplazo en uno o dos ciclos. Los mejores resultados se obtenían con la segunda opción, pero las diferencias no eran grandes. Por ello nos decantaremos por una u otra opción de acuerdo con criterios arquitecturales.

La opción de utilizar un sólo ciclo es muy sencilla. En la figura 3.9 se muestra un diagrama de tiempos en el que se resumen las distintas etapas de operación. Se ha procurado mantener lo mejor posible la relación entre el tiempo de computación de cada etapa y el espacio que ocupa en el esquema. Al comenzar el ciclo, y en previsión de un fallo, se lee la RAM. Mientras, se busca el símbolo en la cache, y en caso de no encontrarse se producirá un fallo. Se selecciona el símbolo que va a ser sustituido y su probabilidad. Finalmente, si no existe conflicto, se escribe en memoria la probabilidad sustituida. Nótese que es imprescindible que no exista conflicto de acceso ya que los accesos para lectura y escritura se solapan. Los espacios sombreados son tiempos muertos para esa etapa de procesamiento y la parte de procesamiento relativa al cálculo de probabilidades acumulativas se añade para tener una referencia temporal. Este esquema implica que la longitud del ciclo está fijada por la detección y selección del símbolo a sustituir y la escritura en memoria, con lo que el tiempo es superior al del otro esquema como veremos a continuación.

El diagrama de tiempos para el esquema que utiliza dos ciclos aparece en la figura 3.10. Se muestran dos ciclos consecutivos, pero sólo es necesario un ciclo para procesar cada símbolo. Suponemos que en ciclo anterior no hay ningún fallo para facilitar la explicación. Al comenzar, se busca el símbolo en la cache. Suponemos que no se encuentra. La lectura del símbolo entrante de la RAM no comienza al empezar el ciclo, porque existe un tiempo previo para comprobar que no hay un fallo pendiente del ciclo anterior con el que exista conflicto.

Tras decidirse el reemplazo, el símbolo saliente y su probabilidad se introducen

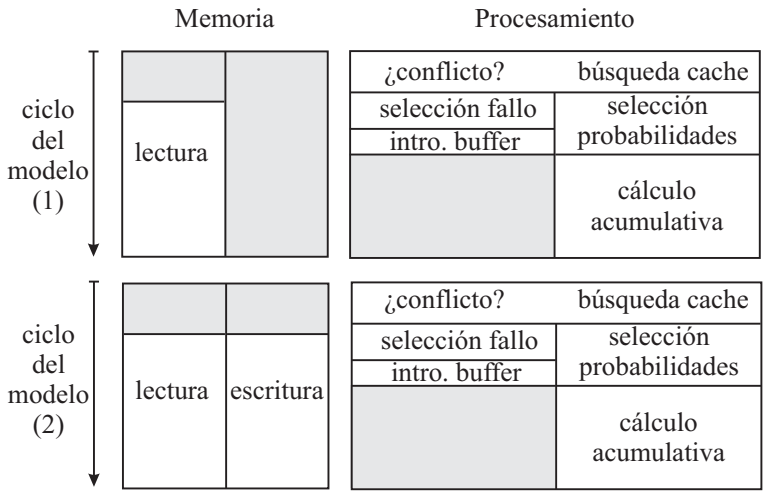


Figura 3.10: Diagrama de tiempos para los reemplazos en dos ciclos.

en un buffer para ser escritos en memoria el ciclo siguiente. El tiempo de lectura de la memoria, más el tiempo de espera previo, es similar al de procesamiento de un acierto, por lo que la duración del ciclo está muy equilibrada.

En el ciclo siguiente supondremos que se vuelve a producir un fallo. Si la escritura de los datos que quedaron pendientes del ciclo anterior entra en conflicto con la lectura del dato entrante, éste no se lee. De esta forma se anula el reemplazo. En caso contrario, se procede como en el ciclo descrito en el párrafo anterior. Mientras, y en otro bloque de memoria, se escribe la probabilidad del símbolo que fue expulsado de la cache en el ciclo anterior. Existe una posibilidad adicional, y es que el símbolo que provoca el último fallo sea el mismo que fue expulsado de la cache en el ciclo anterior. En este caso, se lee del buffer y se expulsa el símbolo que está en la cache. El tiempo necesario para esta operación no es superior a comprobar si existe conflicto.

Dado que la duración del ciclo se puede reducir utilizando el segundo esquema, será este el que utilicemos. Con ello se consigue además reducir la degradación en la relación de compresión debida a los conflictos a niveles insignificantes.

3.4 Control de saturación del modelo

Conocer en todo momento cual es el valor de la probabilidad acumulativa del tope de la cache, S_T, es una tarea complicada debido a los reemplazos. De no existir estos, sólo habría tres opciones: fallo, acierto y acierto en el que la probabilidad ya está a su valor máximo.

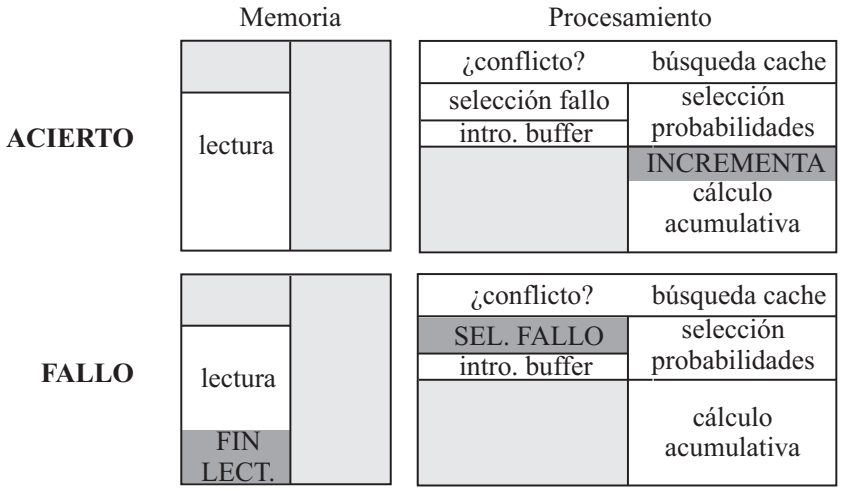


Figura 3.11: Se muestran resaltados los momentos en que se obtienen parámetros relevantes para actualizar S_T.

Con reemplazos las posibilidades son más: acierto, acierto sin incremento (por alcanzar el límite de crecimiento de la probabilidad), fallo con reemplazo y fallo con reemplazo anulado por conflicto. Sumar el valor de todas las probabilidades supone utilizar un árbol de sumadores como el de la figura 3.8, que es bastante costoso. En lugar de ello, se mantiene en un registro su valor y se actualiza dependiendo del evento que se produzca. Todos ellos se resuelven de forma trivial excepto los reemplazos.

De todas formas los elementos necesarios para saber en que momento se debe producir un escalamiento de las probabilidades no están disponibles en el momento en que sería deseable. En la figura 3.11 vemos que en caso de acierto se conoce si hay incremento o no muy cerca del final del ciclo. En caso de fallo la situación es peor, porque la probabilidad del símbolo saliente se conoce relativamente pronto, pero la del símbolo entrante sólo se conoce al final del ciclo. Por ello, optamos por guardar en un registro estos datos y tomar la decisión sobre si escalar o no en el ciclo siguiente.

La actualización supone sumar una unidad en caso de acierto o, en caso de fallo, sumar la probabilidad entrante y restar la saliente. La probabilidad saliente se habrá almacenado en complemento a 2 en el ciclo anterior, y todo se reduce a sumar tres sumandos. En caso de acierto sin incremento o de fallo que provoque un conflicto, no se emprende ninguna acción.

En la figura 3.12 mostramos los casos posibles, obviando aquellos en que no se actualiza S_T. Las entradas al sumador 3:2 están numeradas, y en la tabla de la derecha se muestran los valores para cada entrada en cada caso. Si se produce un

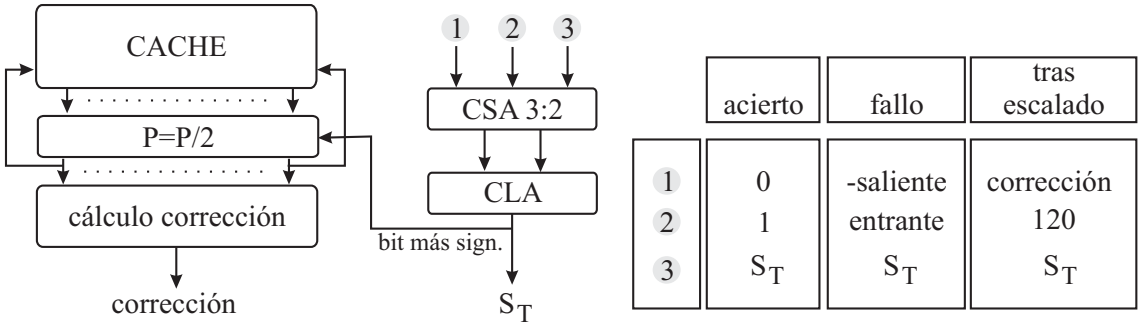


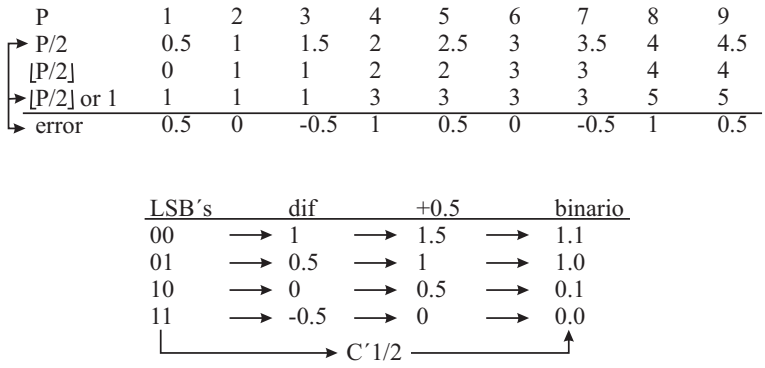
Figura 3.12: Escalamiento de las probabilidades y actualización de S_T.

acierto, en el que se incremente la probabilidad de un símbolo, debe incrementarse también S_T. Así, se suma una unidad al valor previo. Si se produce un reemplazo debe restarse la probabilidad saliente y sumarse la entrante. El sumador 3:2 y el sumador de acarreo adelantado (CLA) obtienen pronto el resultado.

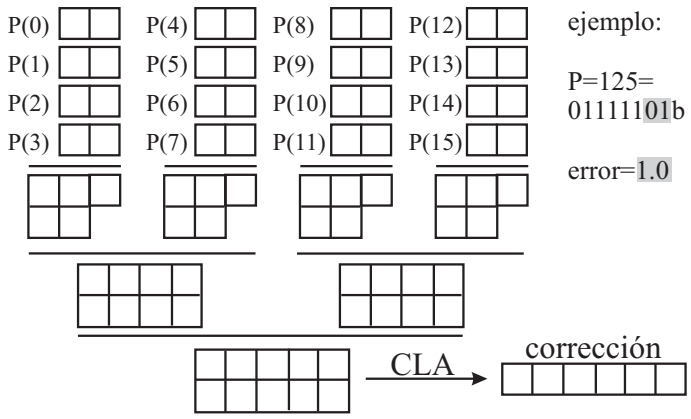
En estos dos casos es posible que se produzca una saturación del modelo. Esto se detecta comprobando el bit de peso 2^{10} a la salida del sumador. En caso de ser no nulo se escalan las probabilidades y se recalcula S_T. Se podría utilizar el árbol de sumadores que calcula las probabilidades acumulativas. Éste está desocupado ya que al detectarse la saturación se fuerza un fallo sin reemplazo para liberar la cache de toda actividad. Sin embargo, el tiempo de detección sumado al retardo del árbol es excesivo. Esto se debe a que el coste de los sumadores totaliza casi toda la duración del ciclo en el modelo.

En lugar de ellos corregiremos el valor conocido de S_T de la siguiente manera. Si miramos la figura 3.13.(a) veremos que la operación $\lfloor P/2 \rfloor or 1$ genera un error respecto a $P/2$ que se repite con periodicidad 4. Basta comprobar los 2 bits menos significativos de la probabilidad para conocer el error (figura 3.13.(a)). Tras un escalado el nuevo valor de S_T será la mitad más la suma de todos los errores. Los errores se pueden calcular desde el inicio del ciclo, antes de saber si se ha producido una saturación, y por tanto se pueden obtener con tiempo suficiente antes de que termine el ciclo. Ya que son posibles errores negativos se introduce un offset de $+0.5$ para limitar las operaciones con números negativos. De esta forma el error asociado a los 2 bits menos significativos se obtiene haciendo el complemento a uno de los mismos.

Sumar los 16 errores tiene el mismo coste en tiempo que sumar las 16 probabilidades acumulativas, pero se comienza al inicio del ciclo y el coste en área es mucho menor al tratarse de sólo 2 bits. En la figura 3.13.(b) se muestra el procedimiento, utilizando sumadores 4:2 y un CLA final para obtener la suma de los errores. Al siguiente ciclo se propagan $S_T/2$, la suma de los 16 errores, y el término 120, que



(a) Estimación de los errores



(b) Mecanismo de cálculo

Figura 3.13: Corrección durante el escalamiento de las probabilidades.

procede de $128 - 16 \cdot 0.5$. El primer término compensa que al dividir S_T por dos también se ha dividido la longitud de la tabla virtual ($256/2 = 128$), y el término negativo se debe al offset de 0.5 en los 16 errores. En la figura 3.12 estos tres términos se introducen en los sumadores tras el escalado para obtener un valor no redundante de S_T.

El hecho de codificar como un fallo permite corregir las probabilidades sin detener la codificación y además su influencia sobre la eficiencia del codificador es mínima. Tal y como se han ajustado las operaciones de detección de saturación y escalado, éstas no aumentarán la duración del ciclo.

3.4.1 Resumen del funcionamiento del modelo en el codificador

En este punto deberíamos, una vez mostrados los componentes por separado, resumir el funcionamiento del modelo en su conjunto. Hemos optado por proponer un ejemplo simulando una secuencia de entradas que provocan todo tipo de situaciones en distintos contextos. Éste se muestra en la figura 3.14, en la que el tiempo evoluciona ciclo a ciclo en sentido vertical y las distintas acciones están clasificadas por categorías. El contenido de la cache se muestra en la figura 3.15. Las filas representan los distintos ciclos y las columnas líneas de la cache. La casilla sombreada contiene el símbolo y en la otra tenemos la probabilidad. Las líneas que no aparecen en el ejemplo no se muestran para dar mayor claridad. Podemos apreciar en la figura el escalamiento de las probabilidades y los distintos reemplazos.

El ejemplo comienza por la llegada del símbolo 6. Se produce un acierto porque se encuentra en la cache. Su probabilidad es 29 y será utilizada en la iteración del algoritmo. Además, es incrementada y se vuelve a almacenar en la cache. Su probabilidad acumulativa se calcula y es 571. El valor de S_T en este ciclo es de 1022.

El siguiente símbolo es el 74, que no se encuentra en la cache. El valor de su probabilidad se lee de la cache y mientras se resuelve expulsar de la cache el símbolo 10, que ocupa la línea que corresponde al símbolo 74. La probabilidad es la de fallo, 1, y la probabilidad acumulativa 74. La probabilidad del símbolo saliente es 43. Mientras, se actualiza S_T sumando la unidad correspondiente al incremento del ciclo anterior. Al final del ciclo se ha cargado la probabilidad del símbolo 74, que es 40.

Nuevamente se produce un fallo, esta vez con el símbolo 172. Se codifica el fallo con probabilidad 1 y probabilidad acumulativa 172. El símbolo que será expulsado de la cache es el 44. La lectura del símbolo 172 se realiza al mismo tiempo que la escritura del 10, que fue expulsado de la cache el ciclo anterior. S_T se actualiza sumando 40, la probabilidad del símbolo 74, y restando 43, la probabilidad del 10. Al final del ciclo ya se ha cargado la probabilidad del símbolo 172, que es de 90.

Al procesarse el símbolo 110 se produce un nuevo fallo pero no se puede emprender el reemplazo porque el símbolo 110 reside en el mismo bloque de memoria que el 44, que está siendo escrito en este ciclo. El fallo se codifica de la forma habitual y se actualiza S_T.

A continuación hay una serie de aciertos al procesarse el símbolo 6. La primera vez su probabilidad es seleccionada, 30, e incrementada. La probabilidad acumulativa es 571, y S_T no se actualiza porque el ciclo anterior no hubo acierto ni reemplazo.

En el siguiente ciclo se incrementa S_T debido al acierto. Se detecta que se ha alcanzado el límite de la precisión y se cancela la codificación del acierto. Es necesario escalar las probabilidades de la cache y estas no pueden ser utilizadas para

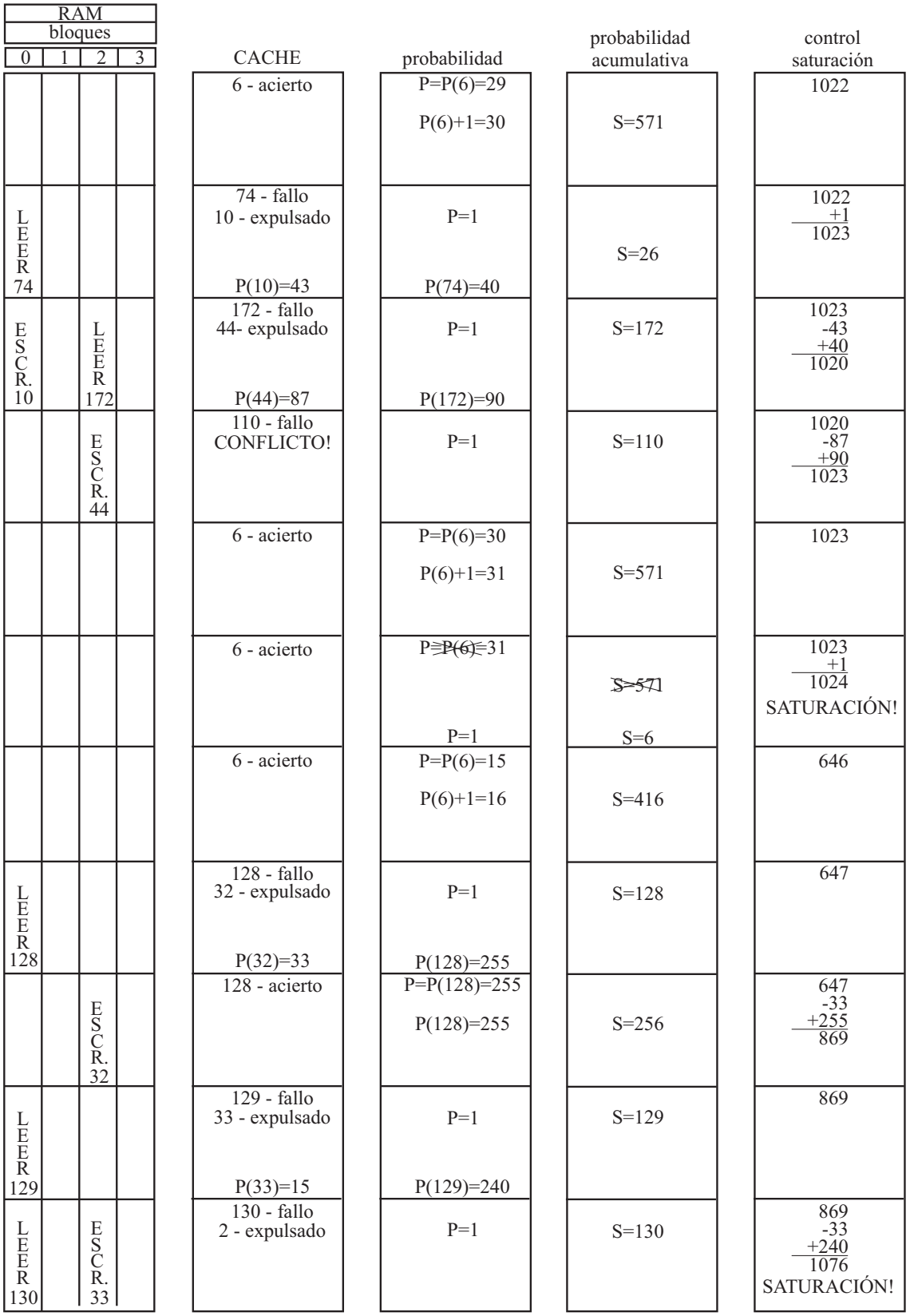


Figura 3.14: Ejemplo de procesamiento en el modelo del codificador.

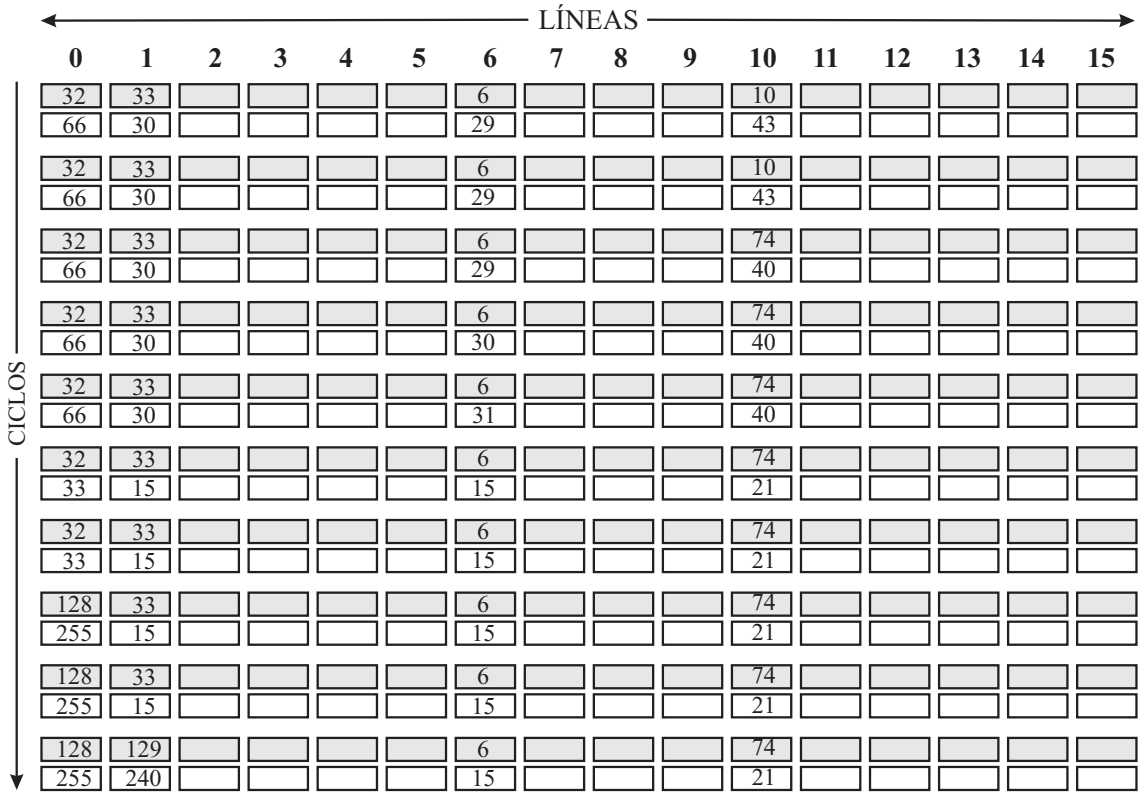


Figura 3.15: Contenido de la cache según el ejemplo de la figura anterior.

ningún otro motivo. Por ello, el símbolo 6 se codifica como un fallo y no se produce ningún reemplazo. El hardware de cálculo de las probabilidades acumulativas se utiliza para disponer al ciclo siguiente del valor actualizado de S_T.

Nuevamente se procesa el símbolo 6, y vuelve a haber un acierto. Su probabilidad ha sido escalada, pasando de 31 a 15. S_T también ha sido escalado y su valor es de 646.

A continuación se produce un fallo al no encontrarse el símbolo 128 en la cache. Se expulsa el símbolo 32 y se prepara la lectura del 128. El fallo se codifica y se actualiza S_T incrementándola en una unidad a causa del acierto del ciclo anterior.

El símbolo 128 vuelve a ser referenciado y se produce un acierto. Como su probabilidad era ya la máxima (255), no se incrementa. En memoria, se escribe el símbolo 32 que fue expulsado el ciclo anterior. S_T se actualiza acorde con el fallo del ciclo anterior.

En el siguiente ciclo hay un nuevo fallo. El símbolo 129 expulsa al 33 y se lee su probabilidad de la RAM. S_T se actualiza porque en el acierto anterior no se incrementó la probabilidad.

A raíz del reemplazo anterior ha entrado en la cache un símbolo con probabilidad muy alta. En la actualización de S_T se supera el límite y es necesario reescalar las probabilidades. La escritura en memoria continúa, pero la lectura se cancela. El símbolo 130 hubiera provocado un fallo de cualquiera manera, pero ahora además se anula el reemplazo.

3.5 Actualización del intervalo en el codificador

Visto el modelo en la sección anterior pasaremos a considerar la actualización del intervalo. La mayor parte de las contingencias posibles han sido consideradas en el modelo, fallos y saturación de las probabilidades se resuelven sin detener el procesamiento de forma que la iteración del algoritmo se calcula ciclo a ciclo hasta terminar el procesamiento.

Por ello, la iteración se reduce a una función de cuatro variables: valores del rango A_i y del punto bajo del intervalo C_i, probabilidad $P_i(k)$ y probabilidad acumulativa $S_i(k)$. Los resultados son los nuevos valores A_{i+1} y C_{i+1} y los bits que se envían hacia la salida.

Los cálculos se puede separar en dos partes, la actualización de A_i y la de C_i, pero existe una ligadura entre ambos ya que el factor de normalización se obtiene durante el procesamiento de A_i. Las operaciones involucradas son la de las ecuaciones 1.5 y un esquema general de la implementación de las mismas se puede ver en la figura 3.1 en el bloque dedicado a la iteración. Un desglose en mayor detalle se muestra en la figura 3.16 que ahora procedemos a describir.

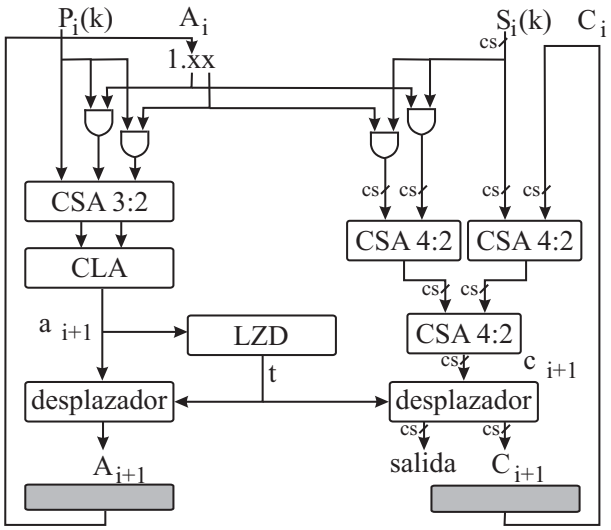


Figura 3.16: Esquema de la actualización del intervalo en el codificador.

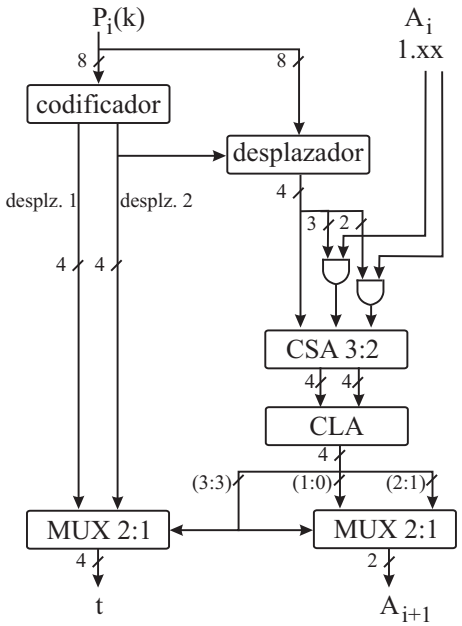


Figura 3.17: Actualización del rango del intervalo y obtención del desplazamiento de normalización.

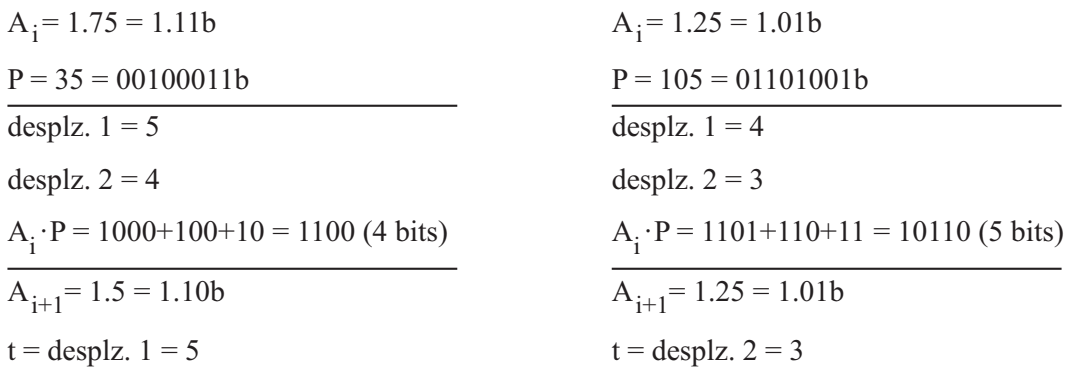


Figura 3.18: Ejemplos de actualización del rango.

Se comienza por implementar las operaciones $A_i \cdot P_i(k)$ y $A_i \cdot S_i(k)$. De A_i utilizamos sólo dos bits fraccionales, de forma que el producto se implementa como la suma de tres términos, uno gobernado por el bit entero, que siempre es 1, y los otros dos gobernados por los bits fraccionales. Como $S_i(k)$ está expresado en formato de acarreo almacenado (todos los operandos involucrados en la actualización de C lo están y aparecen marcados como cs) el número de sumandos se multiplica por 2. A partir de este punto la secuencia de operaciones diverge para la A y para C.

Para A es necesario obtener el número de bits t que es necesario desplazar para normalizar el rango. En principio esto se realiza con un detector del bit no nulo más significativo (LOD o LZD) [BL, HM90], pero dado el caso particular con aritmética de tan baja precisión hemos optado por optimizar esta operación. En primer lugar, al ser A_i un número mayor o igual que 1 y menor que 2, el valor de $A_i \cdot P_i(k)$ será menor que $2 \cdot P_i(k)$, y por tanto el bit más significativo no nulo de $A_i \cdot P_i(k)$ estará en la misma posición que el de $P_i(k)$ o a lo sumo en la siguiente. Por ello, es más conveniente operar con $P_i(k)$ y no con $A_i \cdot P_i(k)$.

En la figura 3.18 vemos dos ejemplos. En el primero de ellos suponemos dos desplazamientos posibles, y seleccionamos la cantidad de bits de $P_i(k)$ que necesitamos para realizar la operación. Tras calcular $A_i \cdot P_i(k)$ comprobamos que el segundo desplazamiento era el correcto porque el resultado tiene su bit más significativo en una posición de más peso que $P_i(k)$. En el segundo caso, el primer desplazamiento es el correcto.

En la figura 3.17 vemos como se implementa. Dependiendo del bit más significativo no nulo de $P_i(k)$ se seleccionan cuatro bits de $P_i(k)$ para calcular $A_i \cdot P_i(k)$. También se conocen dos posibles valores para el desplazamiento t, dependiendo de si $A_i \cdot P_i(k)$ ocupa los mismos bits que $P_i(k)$ o uno más. La selección de los bits de $P_i(k)$ que se van a utilizar se obtienen mediante un desplazador. La operación $A_i \cdot P_i(k)$ se calcula con rapidez: se seleccionan los tres sumandos (nótese como

entran desplazados a la derecha y por ello se utilizan cada vez menos bits) y se asimilan en un sumador 3:2. La propagación del acarreo en el último sumador (CLA) está limitada a 4 bits. Una vez realizada se sabrá también cual de los dos valores del desplazamiento es el correcto y se termina de normalizar $A_i \cdot P_i(k)$. Sólo se seleccionan 2 bits ya que el bit entero es siempre 1.

Esta forma de implementar la actualización de A es más rápida y compacta que la habitual ya que sacando ventaja de la baja precisión utilizada se consigue integrar el producto, la detección y la normalización en pocas operaciones.

La iteración sobre C_i se calcula de una forma más convencional. Mientras que de A_i sólo necesitábamos conocer 3 bits, C_i ha de ser calculado con total predicción. Vemos la implementación en la figura 3.19. Se utiliza aritmética de acarreo almacenado, así pues tanto C_i como $S_i(k)$ (probabilidad acumulativa) están compuestos de 2 palabras, una de semisuma (C_s y S_s) y otra de acarreo (C_c y S_c). $S_i(k)$ es un número de 10+10 bits (10 de semisuma y 20 de acarreo) ya que tal es la precisión utilizada. Sin embargo C_i es un número de 12+12 bits. Los bits adicionales se deben a que se deben mantener los bits fraccionales en los productos para garantizar la descodificación mediante un cociente. Debido a esta diferencia en el tamaño del ancho de palabra el alineamiento es el siguiente: $S_i(k)$ crece hasta los 12+12 bits añadiendo 2+2 bits a la derecha (fraccionales). El valor de $S_i(k)$ desplazado una posición a la derecha para efectuar el producto se alinea dejando 1+1 bits a la derecha, y al que se desplaza dos posiciones no es necesario añadir bits. Se reduce ligeramente el tamaño de los sumadores adaptando cada uno de ellos a la longitud de palabra mínima para cada caso. Al final las longitudes se unifican totalizando 14+14 bits. Realmente, con un análisis cuidadoso de los acarreos se llega a que realmente son 13+14 bits. Dado que la longitud de C_i son sólo 12 bits, los bits extra corresponden a acarreos hacia la parte del código que ya ha sido procesada. Este es un detalle a tener en cuenta en el desplazador.

El desplazador recibe el número de posiciones t desde la iteración sobre A_i y genera un nuevo valor de C_i introduciendo $t + t$ bits por la derecha. Hacia la salida son enviados $t + t$ bits, y a mayores los acarreos. Estos han de ser asimilados por la etapa de salida, por lo que se generarán nuevos acarreos que es necesario compensar. Nuevamente vemos un ejemplo en la figura 3.20 en la que se muestran la distribución de los operandos en los dos niveles de suma y la obtención del resultado final, que es desplazado 5 posiciones a la izquierda, dando lugar a una salida de 5+5 bits además de los acarreos que se propagan sobre la parte ya asimilada.

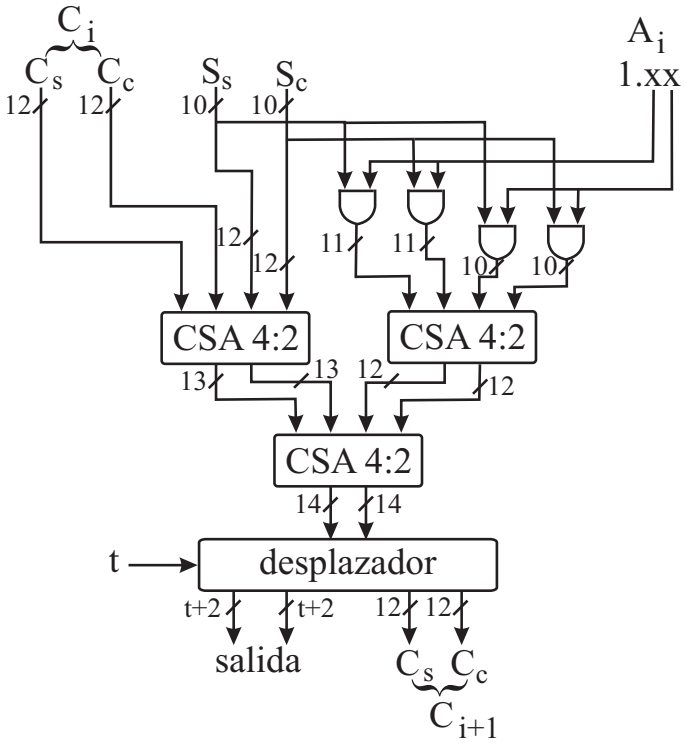


Figura 3.19: Actualización del punto bajo del intervalo.

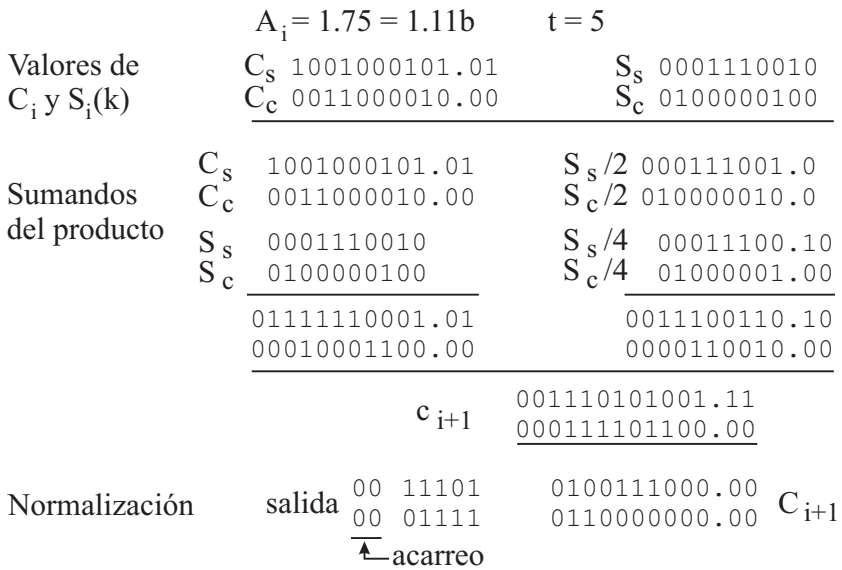


Figura 3.20: Ejemplo de la iteración sobre el punto bajo del rango.

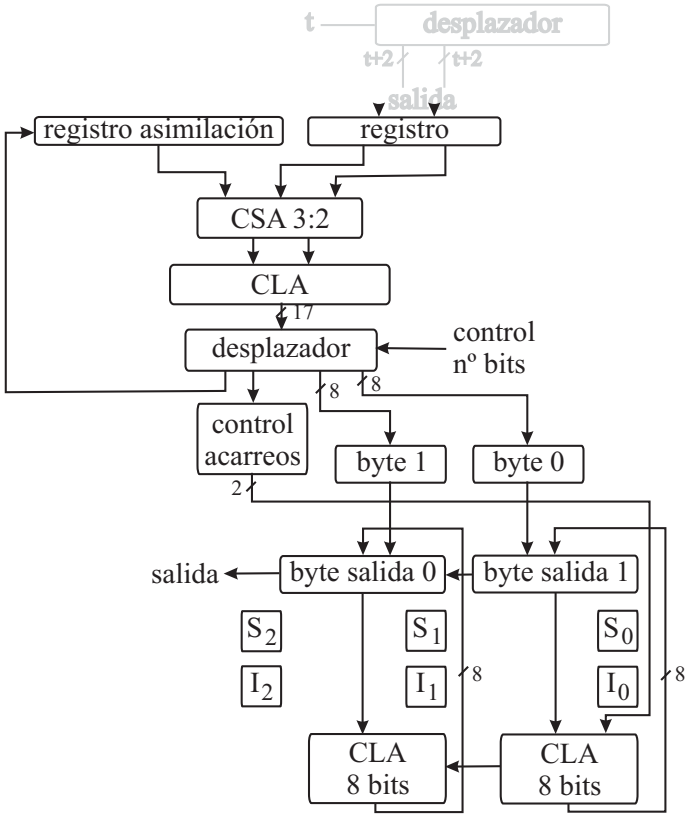


Figura 3.21: Etapa de salida.

3.6 La sección de salida

Es la encargada de convertir la secuencia irregular de bits en una cadena de bytes, eliminando los acarreos que se propagan y, en el caso de esta implementación, convertir los bits a formato no redundante [BBL97]. Es necesario señalar que en este codificador se producen bits todos los ciclos, al contrario que en un codificador binario que puede haber ciclos en los que no se produce ninguna salida. A nuestro favor tenemos el hecho de que no existe ningún tipo de recurrencia por lo que esta sección puede ser segmentada tanto como sea necesario para ajustar la duración del ciclo con el resto del hardware.

El esquema de esta etapa se muestra en la figura 3.21 donde apreciamos dos partes, la superior de asimilación y la inferior en al que se eliminan los acarreos que se propagan mediante la técnica de bit stuffing [LR81].

La asimilación consiste en construir una secuencia de bits a partir de los datos que llegan desde la iteración del codificador y particionarla en bytes. Recordemos

que la salida de la iteración es una palabra de suma y otra de acarreo, y además se han generado acarreos extra sobre la parte ya asimilada. Por un lado debemos convertir las dos palabras en una sola y por otra se han de considerar los acarreos.

El tiempo durante la asimilación no es un problema, por tanto daremos prioridad a la sencillez. La primera medida será reducir los acarreos extra mediante un sumador 3:2, y a continuación convertiremos el resultado a formato no redundante mediante un sumador rápido, ya que un sumador convencional de acarreo propagado es excesivamente lento. El resultado es una única palabra que sufrirá un desplazamiento a la izquierda como resultado de la normalización en la próxima iteración del algoritmo.

Los desplazamientos posibles son entre 2 y 10 posiciones. Al ser el desplazamiento máximo mayor que 8 bits es posible que en un ciclo sea necesario dar salida a dos bytes. En otros, en cambio, sólo se dará salida a uno o a ninguno. El control del número de bytes que se expulsan se lleva mediante un acumulador que cuenta el número de bits que se añaden a cada ciclo. Cada vez que se completan uno o dos bytes se envían a los registros del segundo nivel. Su función es la de un mero separador entre etapas. Los acarreos generados durante la asimilación se envían a finalmente a sumarse con el contenido de los registros de salida (byte de salida 0).

En esta última parte es donde se introducen los bits de stuffing para compensar los acarreos que no pueden ser eliminados de otra forma. El funcionamiento de la técnica es el siguiente. Utilizamos tres pares de bits. Cada par corresponde a un byte de salida. Los pares 0 y 1 corresponden a los bytes de salida 0 y 1, y el par 2 corresponde al último byte expulsado. Cada par está formado por un bit de stuffing S_i y uno de información I_i. El bit I informa si se está utilizando un bit de stuffing, que puede ser '1' o '0' y se almacena en el correspondiente bit S. Si un byte va ser expulsado y su bit I está activado, se emitirá también el bit S correspondiente, y el descodificador sabrá en que momento debe esperar la llegada de un bit de stuffing y el proceso se invierte sin problemas.

Los bits de stuffing se activan cada vez que se genera un acarreo que no puede ser anulado o bien existe riesgo de que ocurra. En caso de que el byte más significativo (byte salida 1) contenga el valor máximo (255) un acarreo futuro puede producirá un desbordamiento. Por ello, en tal caso se introduce un bit de stuffing. En él se introducirá el acarreo que se genere en el byte menos significativo. Existe una excepción a esta norma, si ya se introdujo un bit de stuffing para el byte ya expulsado (S_2, I_2), no se cree uno nuevo, sino que el acarreo se cancela haciendo uso de S_2.

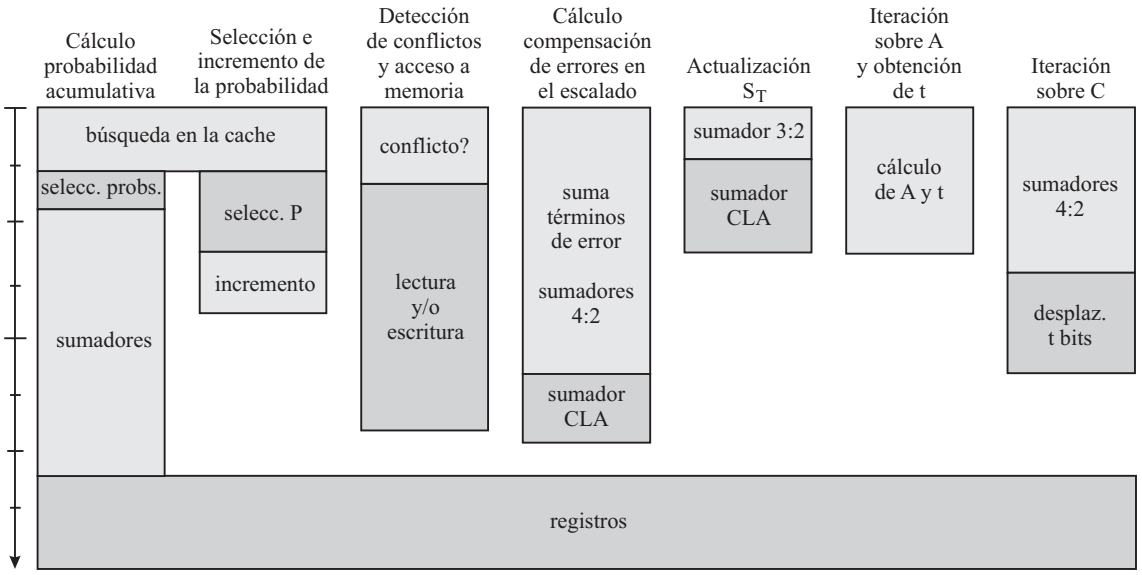


Figura 3.22: Tiempos de computación en el codificador. La duración total del ciclo es de aproximadamente 37 retardos de puertas nand.

3.7 Tiempo de procesamiento en el codificador

Habiendo estudiado los distintos elementos del codificador en las secciones anteriores podemos resumir el tiempo necesario para la codificación. Las estimaciones se han realizado por medio del programa de síntesis Synopsys. Se utilizan librerías de ES2 [Str92] de 0.7 micras y los tiempos obtenidos se traducen a retardos de puertas nand (t_{nand}) con una carga de tres puertas de igual tipo a la salida. Este tipo de medida es relativamente independiente de la tecnología utilizada y supone unas situaciones de carga en los circuitos realistas. Sin embargo los datos que presentamos no son más que estimaciones ya que los retardos reales dependen fuertemente de las cargas reales, longitud de las líneas y otras características físicas de los circuitos reales.

En la figura 3.22 se muestran los tiempos para las etapas que implican cálculos tanto en el modelo como en la iteración. No hemos incluido aspectos tales como la selección de símbolos durante los fallos porque carecen de complejidad.

Cada columna corresponde a una acción del codificador y las partes que la componen tienen una altura proporcional al tiempo que necesitan para completarse. La duración total del ciclo es de aproximadamente 37 retardos de puerta *nand*, y viene fijada por el tiempo necesario para detectar un acierto, calcular su probabilidad acumulativa y los tiempos de carga y acceso de los registros. El tiempo de los registros es común a todas las tareas, y la búsqueda en la cache es común a la obtención de la probabilidad y de la probabilidad acumulativa. Los restantes elementos no aparecen

en más de una tarea.

El cálculo de las probabilidades acumulativas es la tarea más lenta del codificador. Esto se debe principalmente al coste de los sumadores, que son tres niveles 4:2. La selección de las probabilidades para las sumas necesita poco tiempo y también la búsqueda del símbolo de la cache.

La selección de la probabilidad del símbolo en caso de acierto es ligeramente más costosa porque no consiste en activar algunas u otras, sino en seleccionar un valor entre 16, por ello necesita más tiempo que su equivalente en el cálculo de las probabilidades acumulativas. El incremento de la probabilidad tras la selección es una operación rápida, y como vemos el total de esta tarea ocupa poco más de la mitad del ciclo.

Los accesos a memoria son una tarea lenta que sin embargo no define la vía crítica. Como vemos en la figura, aun sumando el tiempo necesario para verificar la no-existencia de conflictos, hay suficiente tiempo para completar la lectura y/o escritura. Esto es posible al haber eliminado ciclos de lectura-modificación-escritura y haber retrasado un ciclo la escritura en los reemplazos.

Las operaciones relativas a la corrección de las probabilidades ocupan las dos siguientes columnas. La primera corresponde al cálculo de los factores de corrección. La suma de todos los términos de error tiene el mismo coste en tiempo que sumar todas las probabilidades. Al final, un pequeño sumador rápido los reduce a una sola palabra. El tiempo es muy ajustado pero inferior al disponible.

En el ciclo siguiente a haberse detectado la saturación se suman todos los términos involucrados en el cálculo del nuevo valor de S_T. Es una operación con un coste muy bajo, un sumador 3:2 y un sumador no redundante.

Las dos siguientes columnas las comentaremos juntas. Si las anteriores concernían a la gestión del modelo, estas corresponden a la iteración. La actualización del rango A_i y la obtención del desplazamiento t para la normalización son operaciones muy optimizadas que se resuelven en muy poco tiempo. La actualización de C_i es más lenta, no mucho más, de forma que cuando finaliza ya está disponible el valor de t para proceder a la normalización.

En total, la iteración tiene un coste temporal inferior al del modelo. Esto se debe a dos motivos. Por un lado el coste de manejar el modelo está condicionado por la cantidad de sumas que se realizan para el cálculo de las probabilidades acumulativas y por los accesos a memoria, siendo muy difícil reducir el ciclo. Por otro lado el coste de la iteración ha disminuído espectacularmente respecto a otras implementaciones con multiplicaciones debido a la baja precisión con que se implementan todas las operaciones, a pesar de que ello tiene escasa influencia en la compresión.

3.8 La descodificación

Visto el codificador mostraremos la arquitectura del descodificador (figura 3.2) comenzando por el primer elemento importante: el divisor invierte el producto calculado en el codificador, facilitando de esta forma la descodificación. A partir de la división se puede conocer si ha habido un fallo o no, y además muchos de los detalles relacionados con la descodificación de los aciertos encuentran su explicación en la implementación del divisor.

3.8.1 El divisor

Su propósito es dividir C_i entre A_i evitando que aparezca el término $A_i \cdot S_i$ en todas las comparaciones que se realizan durante la descodificación. Tal y como se explicó en la sección 2.9.1 la división hace posible descodificar los fallos en una sola operación y descodificar los aciertos sin implementar los productos.

La expresión a la que habíamos llegado para los aciertos, con 256 símbolos y P_{fallo} igual a la unidad, es la siguiente:

$$\sum_{j=0}^{j<l} P_{linea\ j} \leq \frac{C_i}{A_i} - 256 \tag{3.2}$$

Se descodificará el símbolo contenido en la línea l más alta que cumpla esta condición. A fin de ser implementada en hardware conviene llegar a una expresión en la que la condición se pueda evaluar con mayor facilidad.

$$(\sum_{j=0}^{j<l} P_{linea\ j}) - (\frac{C_i}{A_i} - 256) \leq 0 \tag{3.3}$$

Sin embargo, comprobar la condición menor o igual que cero acarrea una pequeña dificultad, ya que hay que comprobar por un lado si el número es cero y por otro si es negativo. Sería más conveniente que la condición fuese estricta y de esta forma la condición menor que cero se comprueba utilizando el bit de signo del resultado ya que trabajaremos en complemento a 2. Para ello restamos una unidad y la condición (3.3) se transforma en

$$(\sum_{j=0}^{j<l} P_{linea\ j}) - (\frac{C_i}{A_i} - 256 + 1) \leq 0 \tag{3.4}$$

De esta manera basta calcular el valor de $-(C_i/A_i - 255)$ y al sumarlo a las probabilidades acumulativas de los símbolos de la cache conoceremos el resultado de la comparación. Por tanto, el objetivo del divisor es obtener este valor. Veremos como la suma del factor adicional 255 se puede integrar sin problemas en la división.

A	$1/A$ (binario)	$1/A$ (radix-4)
1.00	1.0000000000000	$2000000 \cdot 2^{-13}$
1.01	0.1100110011001	$1212121 \cdot 2^{-13}$
1.10	0.1010101010101	$1111111 \cdot 2^{-13}$
1.11	0.1001001001001	$1021021 \cdot 2^{-13}$

Tabla 3.1: Inversos de los valores de A_i

Para calcular el cociente C_i/A_i recurriremos a un método sencillo: calcular los inversos de los posibles valores de A_i y sustituir la división por una multiplicación por estos inversos. Otros métodos de división más generales pueden ser más eficientes en otras aplicaciones, pero en nuestro caso esta solución directa parece la más rápida y económica. En la tabla 3.1 se muestran los inversos de los posibles valores de A_i. Se representan en binario y en radix-4. Vemos que en este último formato los coeficientes toman valores 0, 1 y 2, que permiten una implementación sumamente sencilla de los productos.

El número de bits con que se ha expresado $1/A$ es el mínimo necesario para garantizar la descodificación. Este valor se ha obtenido probando todas las posibles combinaciones de valores de C_i y A_i. Son necesarios 6 bits fraccionales en los sumandos para evitar pérdidas de precisión con los multiplicandos de menor peso y además se ha de sumar un factor de redondeo cuyo valor se ha estimado en 7/64. De esta manera es posible obtener los cocientes con un error inferior a la unidad, lo que garantiza la descodificación correcta de todos los símbolos, ya que sus probabilidades acumulativas difieren en al menos una unidad.

Los cálculos se realizan utilizando aritmética de acarreo almacenado. en dos niveles de sumadores tal y como se observa en la figura 3.23. Naturalmente, nos interesa integrar la mayor cantidad posible de operaciones de forma que el tiempo de computación sea lo más bajo posible. Por ello se incluye dentro de los cálculos de la división la suma del factor constante -256 junto con el factor de redondeo.

Los valores de $1/A_i$ se encuentran almacenados en una PLA. Los dígitos de $1/A_i$ gobiernan una serie de multiplexores, uno para cada valor desplazado de C_i. Dependiendo de los dígitos se seleccionará el valor desplazado de C_i, el doble o cero. Hay siete sumandos, que se completan con -256 más el factor de redondeo, ocupando las ocho entradas del árbol de sumadores. Finalmente se selecciona la parte entera del cociente resultante y se le cambia el signo. Estas operaciones las describiremos en detalle a continuación. Hacemos notar que C_i se encuentra en formato no redundante. Esto es necesario porque de lo contrario nos encontraríamos con un número doble de operandos. Para ello la iteración del descodificador termina convirtiendo C_i a formato no redundante.

Veremos a continuación el proceso que nos lleva a obtener $-(C_i/A_i - 255)$ de

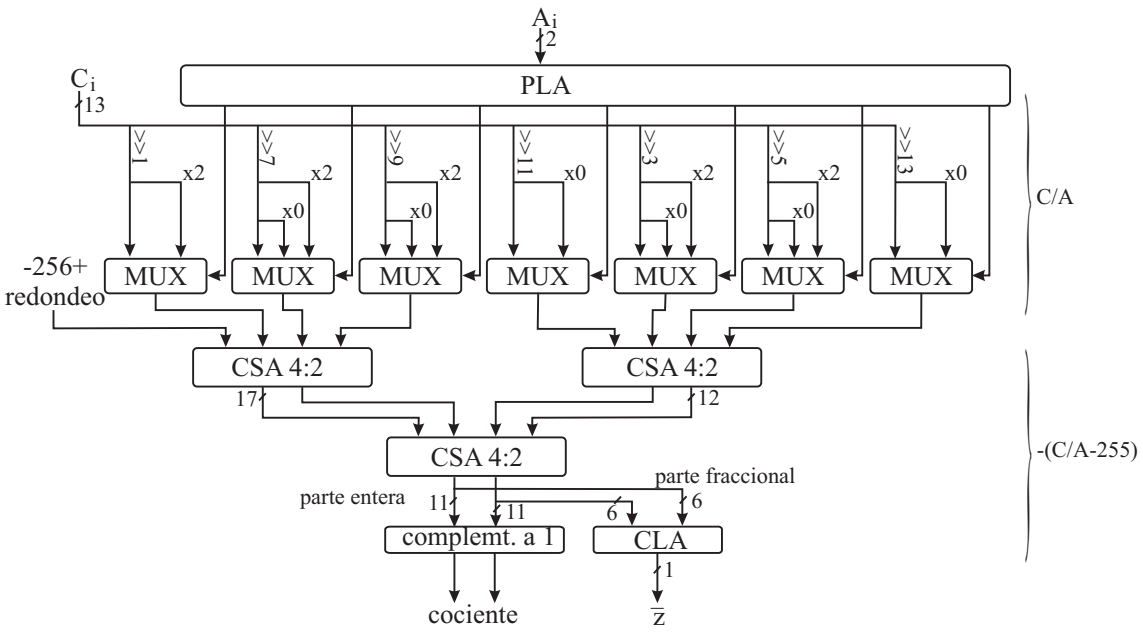


Figura 3.23: Estructura del divisor.

la forma más conveniente. Llamamos Q al resultado arrojado por nuestro árbol de sumadores, que está compuesto en realidad por una palabra de semisuma y una de acarreo. Ambas tienen 6 bits fraccionales, pero sólo estamos interesados en su valor entero. Por razones de velocidad en los cálculos trabajaremos por separado con la parte entera de Q y con el acarreo propagado desde la parte fraccional, al que llamaremos Z.

$$\begin{aligned} Q &= \frac{C_i}{A_i} - 256 \\ Q &= Q_s + Q_c \\ Z &= acarreo(frac(Q_s) + frac(Q_c)) = \lfloor (frac(Q_s) + frac(Q_c)) \rfloor \\ \lfloor Q \rfloor &= \lfloor Q_s \rfloor + \lfloor Q_c \rfloor + Z \end{aligned} \tag{3.5}$$

El valor en que realmente estamos interesados es $-\lfloor Q \rfloor - 1$ y lo obtenemos teniendo en cuenta que en complemento a 2, el signo se invierte complementando a 1 los bits del operando y sumando una unidad. También es necesario invertir el signo de Z de la misma manera, si bien aquí cabe una pequeña optimización dado que Z es un número de un sólo bit al que llamaremos z, entonces $-Z = \bar{z} - 1$.

$$-(\lfloor Q \rfloor) - 1 = \lfloor \bar{Q_s} \rfloor + \lfloor \bar{Q_c} \rfloor + 2 - Z - 1 = \lfloor \bar{Q_s} \rfloor + \lfloor \bar{Q_c} \rfloor + 2 - 1 + \bar{z} - 1$$

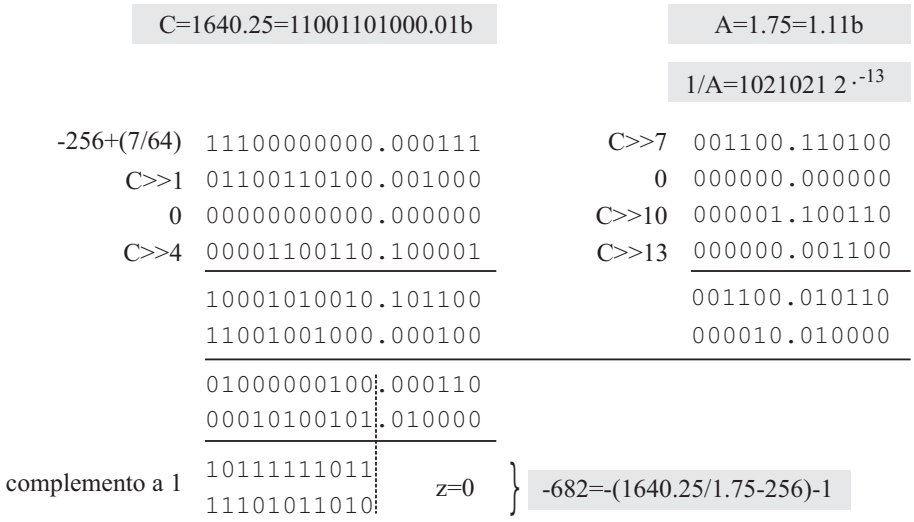


Figura 3.24: Ejemplo de división.

$$= \lfloor \bar{Q}_s \rfloor + \lfloor \bar{Q}_c \rfloor + \bar{z} \tag{3.6}$$

De manera que toda la operación se reduce a complementar a uno las partes enteras de las palabras de semisuma y acarreo de Q y sumar el complemento a 1 de z. En la siguiente sección veremos como a pesar de que la obtención de z es lenta debido a que supone una suma con propagación de acarreo, podemos introducirlo en un momento posterior de forma que este retraso se vea compensado.

Para terminar con la descripción del divisor mostramos en la figura 3.24 un ejemplo numérico de toda la secuencia de operaciones. La división se realiza con una suma de 8 términos. Uno de ellos es una constante que engloba restar la longitud de la tabla virtual (256) y sumar un factor de redondeo (7/64). Los 7 sumandos restante provienen de cada uno de los dígitos en que se codifica el valor de $1/A_i$. La suma final consta de una parte entera, de la cual se calcula el complemento a 1, y de una parte fraccional que se asimila por separado para obtener z. El resultado final es el cociente deseado con el símbolo invertido.

3.9 Descodificación del símbolo

La descodificación puede venir por dos caminos: como un fallo o como un acierto. El primer caso es el más sencillo y el que veremos primero. En el segundo nos detendremos más ya que el hardware es más complejo y define la vía crítica del descodificador.

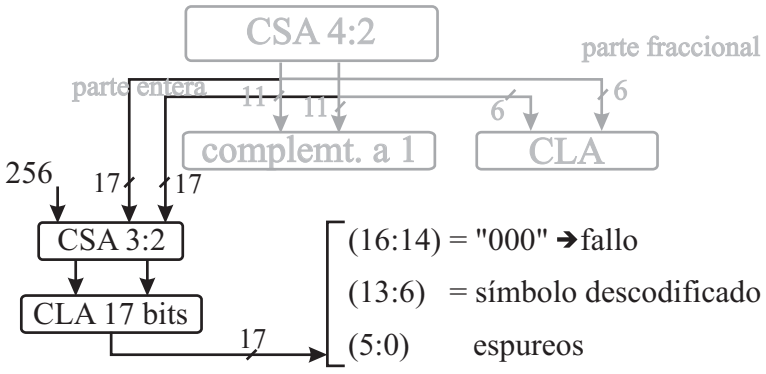


Figura 3.25: Descodificación de un fallo.

3.9.1 Descodificación de un fallo

En la figura 3.2 vemos la descodificación de los fallos en el módulo de inicio a continuación del divisor. Utilizaremos el cociente para ello, pero dado que la división se ha optimizado para descodificar aciertos (que es la parte más costosa), no disponemos de C_i/A_i, sino de $C_i/A_i - 256$. Para descodificar un fallo es necesario sumar primeramente 256 y a continuación convertir el resultado a formato no redundante. La suma de 256 se realiza con un sumador 3:2 y la suma final con un sumador rápido de forma que la duración total de la operación no sea superior a la descodificación de los aciertos. El esquema de esta circuitería es el que se muestra en la figura 3.25.

La longitud real del sumador 3:2 es de tres bits, ya que los bits menos significativos no se ven afectados. El desglose por bits del resultado nos muestra como saber si ha habido un fallo o un acierto y en el primer caso cual es el símbolo que se descodifica.

3.9.2 Descodificación de un acierto

La descodificación pasa por una comparación entre el cociente obtenido y las probabilidades acumulativas de todos los símbolos, por lo que en paralelo con la división se calculan estas últimas. Es una operación costosa ya que al contrario que en el codificador no basta con obtener una dada, sino que es necesario obtener todas ellas. El esquema utilizado para ello se muestra en la figura 3.26, junto con la implementación de las comparaciones. Sólo se muestra a modo de ejemplo la obtención de una probabilidad acumulativa. En primer lugar hay tres niveles de sumadores de distinto tipo, correspondientes al cálculo de las probabilidades acumulativas. A continuación una fila de sumadores 4:2, principalmente, para sumar el cociente y las probabilidades acumulativas. Por último un nivel de sumadores CLA para obtener

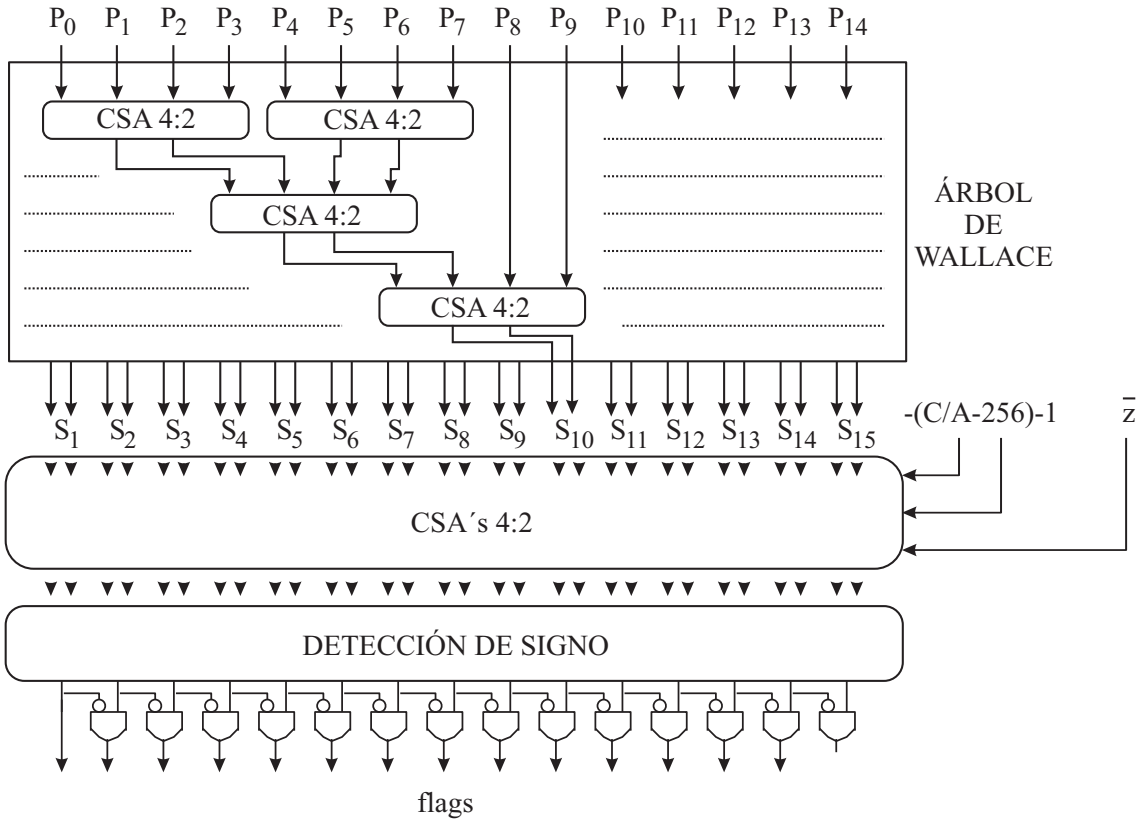


Figura 3.26: Descodificación de un acierto.

el signo de los resultados y conocer que línea contiene el símbolo buscado.

La obtención de cada una de las probabilidades acumulativas tiene un coste creciente. Algunas se obtienen directamente o bien a través de un sumador 3:2 o 4:2. Las restantes se obtienen a través de más de un nivel de sumadores y es necesario escoger adecuadamente que resultados intermedios son los más adecuados para minimizar el número de sumadores sin incrementar el retardo. en el peor de los casos son necesarios tres niveles de sumadores 4:2.

La suma de las probabilidades acumulativas con la salida del divisor se realiza a través de un nuevo nivel de sumadores 4:2, con lo finalmente tendremos dos operandos que han de ser reducidos a uno sólo. En realidad no nos interesa conocer el valor de la suma, sino tan solo su signo. Para ello utilizamos el esquema de un sumador de acarreo adelantado para obtener el valor del bit más significativo del resultado, que es también el bit de signo [LB99]. Como entrada adicional en cada CLA se ha incluido el bit $\bar{z}$. Si revisamos el divisor veremos que su obtención es lenta, y el hecho de haber expresado el resultado como se hizo nos permite comenzar

la comparación sin esperar a que esté disponible.

Tras la suma, en la que realmente se implementa la comparación, tenemos un conjunto de bits de signo que debemos interpretar. La línea correcta será aquella en la cual se detecte un cambio de signo con respecto a la anterior, y esta es la operación que se realiza en las puertas *and* con una entrada negada en la parte inferior de la figura. Este indicador nos sirve para seleccionar el símbolo, probabilidad y probabilidad acumulativa. El primero es el resultado de la descodificación, y las dos restantes son necesarias para completar la iteración del descodificador. La probabilidad, además, ha de ser incrementada antes de volver a la cache.

3.10 La iteración en el descodificador

Es la etapa final de la descodificación y difiere ligeramente de lo visto en el codificador. La iteración sobre el rango, A, es exactamente igual por lo que no repetiremos lo ya dicho, pero la iteración sobre el punto bajo, C, es distinta por dos motivos: hay un cambio de signo y el resultado debe ser convertido a formato no redundante.

El cambio de signo se refiere a la resta que aparece en las ecuaciones 1.4, que será implementada como una suma invirtiendo el signo. El cambio de signo no es una gran complicación, salvo que han de tenerse en cuenta sus efectos sobre los desplazamientos ya que el bit de signo ha de ser replicado si se desplaza un operando a la derecha.

El resultado ha de estar en formato no redundante por el motivo que se adujo al comentar el divisor, ya que de utilizarse acarreo almacenado el coste del divisor se multiplicaría por 2, haciéndolo poco práctico. En cambio el coste de realizar una conversión a formato no redundante tiene un coste aceptable como veremos.

En la figura 3.27 se esquematiza la iteración sobre C. La operación $C_i + A_i \cdot S_i$ se realiza por medio de dos niveles de sumadores 4:2. Una de las entradas se utiliza para el valor realimentado de C_i y las restantes para los sumandos en que se descompone el producto. Por una de ellas se introduce el valor $1A_{-1}.A_{-2}0$, equivalente a $2 \cdot A$. El motivo es el siguiente: el cambio de signo, complemento a 2, se realiza mediante un complemento a 1 seguido de la suma de una unidad. Para cada uno de los sumandos, tres de suma y tres de acarreo, habrá que sumar una unidad, a no ser que ese sumando esté multiplicado por cero.

Tras la suma un sumador rápido convierte C_{i+1} a formato no redundante. Se procede entonces a desplazar el resultado a la izquierda t bits, los mismos que se ha desplazado el rango para normalizarlo. Este desplazamiento da lugar a la entrada de t nuevos bits desde la corriente de entrada.

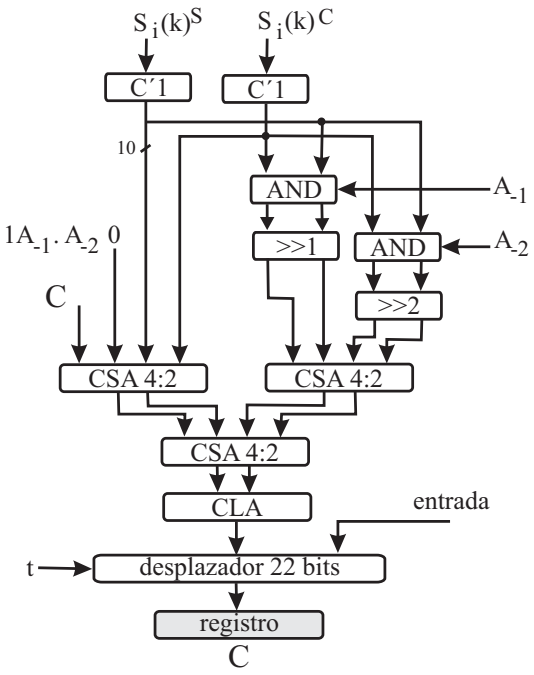


Figura 3.27: Iteración sobre C en el descodificador.

3.11 Los reemplazos y la saturación en el descodificador

El ciclo del descodificador es considerablemente más largo que el del codificador, pero ambos han de realizar las operaciones de la misma forma. Así aunque el ciclo del descodificador sea lo suficientemente holgado como para completar los reemplazos en un único ciclo, debemos realizarlos en dos para mantener la consistencia con el modelo del descodificador. El esquema, por tanto, es el mismo que se describe en la sección 3.3.6 con la única diferencia de que el tiempo no es ningún problema. Tras descodificarse un fallo se direcciona la RAM para leer el dato requerido excepto si hay un conflicto con una escritura de un fallo que se haya producido el ciclo anterior. Habría tiempo suficiente para realizar la escritura en el mismo ciclo pero se demora hasta el siguiente y de esta forma codificador y descodificador son compatibles.

Respecto al control de saturación ocurre algo muy similar. La saturación se podría detectar y corregir en el mismo momento en que se produce, pero se demora hasta el ciclo siguiente, y en caso de producirse se simula un fallo para escalar las probabilidades. En la siguiente sección veremos el reparto de tiempos en el descodificador y comprobaremos como estas operaciones tienen tiempo suficiente para desarrollarse.

3.12 Tiempo de procesamiento en el descodificador

Al igual que en el codificador mostramos en la figura 3.28 un diagrama donde situaremos las distintas tareas y podremos estimar la duración de cada una de ellas.

Tiene algunos elementos en común con el ya visto para el codificador. En particular, y dado que su comportamiento no difiere en absoluto en lo ya explicado y ya que no intervienen en la vía crítica, hemos englobado en el bloque *"Otras tareas"* las operaciones de escritura de datos después de un reemplazo, cálculo de factores de corrección ante una saturación y recálculo de S_T.

El primer elemento que comentaremos es el divisor. Como ya se ha visto la división completa incluye el cálculo de un factor adicional que llamamos z. Sin embargo consideramos concluida la división antes de obtenerla porque el resultado sin conocer z se puede utilizar para comprobar si ha habido un fallo o no. Por ello vemos que al terminar la división comienza la descodificación del fallo (si lo hubiese).

Los fallos se detectan asimilando el valor de la división y comprobando si es inferior a la longitud de la tabla virtual. Esto supone una suma no redundante con operandos de 17 bits, por ello necesita tanto tiempo. En caso de detectarse fallo se comprueba si existe algún conflicto de acceso a memoria y si no es así se escribe en la RAM.

La tercera columna se refiere al coste de calcular las probabilidades acumulativas de todos los símbolos contenidos en la cache para compararlos con el resultado de la división y descodificar un acierto si se da el caso. Son necesarios hasta tres niveles de sumadores 4:2 y por ello el tiempo que se necesita es alto. El resultado obtenido se utiliza para las comparaciones tal y como se describe en la siguiente columna. Dado que tanto las probabilidades acumulativas como el cociente obtenido se expresan en formato redundante, es necesario un sumador 4:2 para reducirlos a 2 operandos. Para esta primera operación no es necesario conocer el valor de z. A continuación se utiliza una suma no redundante para obtener el signo de los resultados y deducir que línea contiene el símbolo buscado. Éste es el resultado de la descodificación, pero también es necesario seleccionar la probabilidad y la probabilidad acumulativa que será utilizada en la iteración. La primera ha de ser incrementada antes de volver a la cache.

Sin esperar a que se incremente la probabilidad, comienzan en las dos siguientes columnas las actualizaciones de A y C respectivamente. La iteración sobre A no se diferencia en absoluto de la vista para el codificador. No así la iteración sobre C que incluye una suma no redundante. Por ello, necesita tiempo para completarse y define la vía crítica del descodificador.

En total, incluyendo el tiempo de los registros, nuestro descodificador necesita

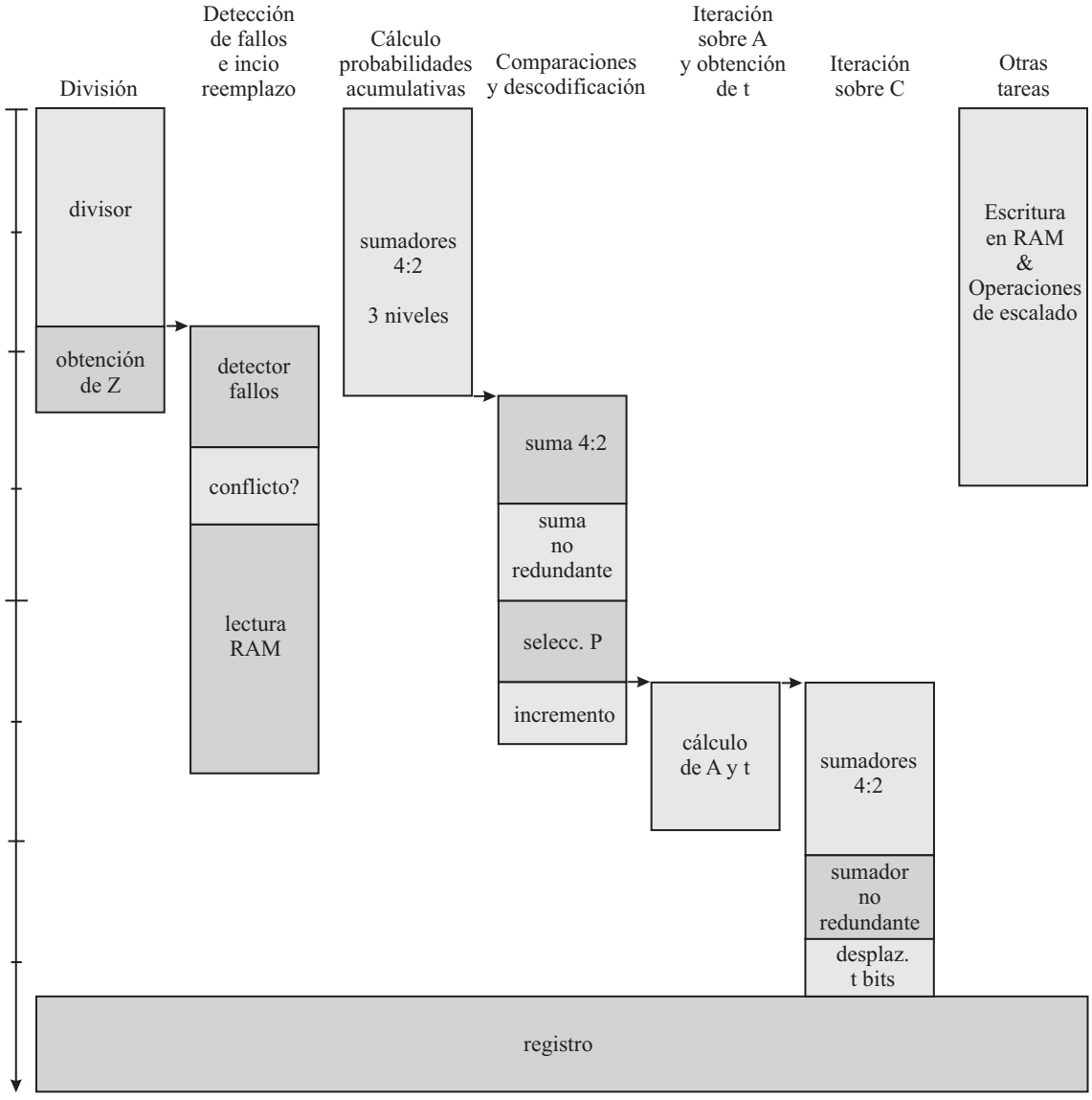


Figura 3.28: Tiempo de computación de las distintas tareas en el descodificador.El tiempo total de computación es de aproximadamente 77 retardos de puerta nand.

alrededor de 77 veces el retardo de una puerta *nand* para completar el ciclo. Como era de suponer es superior al del codificador (aproximadamente el doble), debido principalmente a que la recursividad del algoritmo no permite en este caso separar el modelo de la iteración.

3.13 Segmentación

Introducir segmentación en una arquitectura es la forma más fácil de mejorar la velocidad. Sin embargo, no siempre es posible hacerlo, ya que pueden existir dependencias entre los datos. Este es el caso de los algoritmos recursivos, como el de codificación aritmética, puesto que el resultado de una iteración es imprescindible para comenzar la siguiente.

Por este motivo no es sencillo segmentar los algoritmos de codificación ni de descodificación. En el primer caso no es demasiado problemático, ya que la duración del ciclo no es excesivamente grande debido a que el modelo se puede separar de la iteración. De hecho, segmentar sólo la iteración no tendría interés ya que la vía crítica la define el modelo. En el descodificador la posibilidad de segmentar es mucho más atractiva debido a la elevada duración de su ciclo de procesamiento.

De forma inmediata es posible segmentar todas las partes no involucradas dentro de la iteración, como es el modelo y la sección de salida del codificador. Siempre podremos disminuir el tiempo de procesamiento de estas etapas sin más que añadir registros para separar tareas dentro de la misma etapa. Sin embargo, no es trivial segmentar la iteración.

En [OB97] se presentó un método para segmentar a pesar de la recursividad. Se trata sencillamente de ejecutar dos iteraciones del algoritmo que se alternan en el uso del hardware segmentado (figura 3.29). Sobre este aspecto ya hemos hablado en la sección 1.4.6 pero al introducir la cache las restricciones varían ligeramente, ya que se debe garantizar la actualización del contenido de la cache dentro de un mismo ciclo y la secuencia de operaciones en los reemplazos.

La segmentación en el codificador es una mera formalidad para mantener la coherencia con el descodificador, y ya que la iteración está fuera de la vía crítica, podemos dividir la iteración de forma arbitraria. En la figura 3.30 mostramos como se efectuaría la división. El modelo está separado de la iteración por medio de un registro que no forma parte de la segmentación. Ésta se realiza a nivel de la iteración, por ejemplo separando las sumas y productos de la normalización. Cada una de las etapas, separadas mediante un registro, trabajan con valores distintos de A_i y C_i. El tamaño de los distintos bloques da una idea del tiempo de computación que necesitan y podemos apreciar que no importa como se haga la segmentación, la vía crítica está siempre definida por el modelo. La etapa de salida se añade

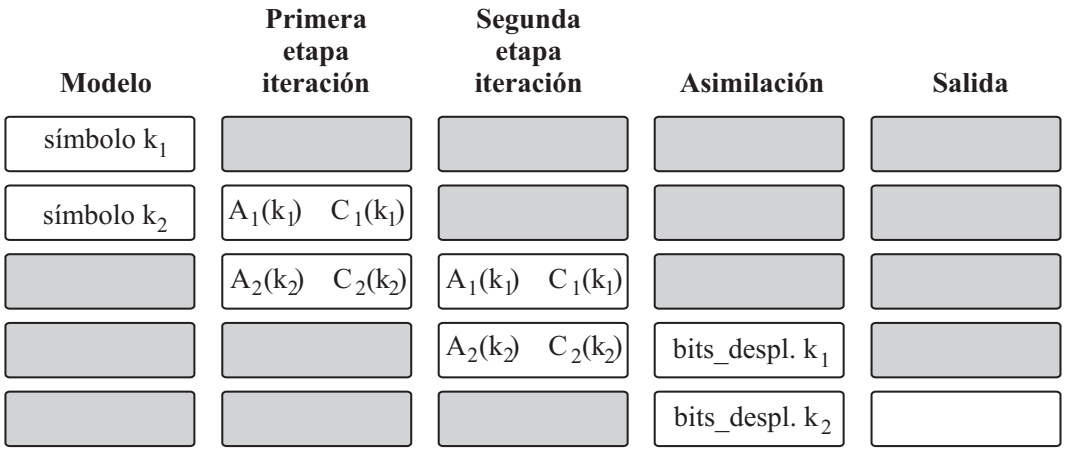


Figura 3.29: Ejemplo de segmentación. Se observa el procesamiento de dos símbolos, k_1 y k_2. En la etapa de salida ya no hay distinción entre los bits que provienen de codificar un símbolo u otro (ya que se empaquetan en bytes).

simplemente por completar el codificador ya que no influye en la vía crítica al poder ser segmentado cuanto se desee.

En el descodificador se consigue una reducción de la duración del ciclo, pero no es demasiado significativa, sólo un 10%, y puede llevar a plantearnos si vale la pena. El motivo de ello es que para mantener la coherencia en el modelo la segmentación debe introducirse entre éste y la actualización del intervalo, y esta última ocupa sólo una pequeña parte del ciclo. En la figura 3.31 vemos los bloques en que queda dividido el descodificador. Esta vez la segmentación no se hace a nivel de iteración, que ocupa una etapa, sino que se separa la operación en el modelo y la iteración. Esto es congruente con el codificador en tanto que en cada ciclo se alternan las dos ejecuciones del algoritmo, cada una con su valor de A_i y C_i. El modelo se actualiza a cada ciclo y también mantiene la coherencia con el codificador.

El diagrama temporal de la arquitectura tras segmentar se muestra en la figura 3.32, y apreciamos que no se llega a un equilibrio entre las dos etapas, las cuales se muestran separadas por una línea vertical.

3.14 Conclusiones y resumen de resultados

A continuación compararemos nuestra arquitectura con la arquitectura sin cache descrita en la sección 1.4 [POB97] [BBL97]. Aunque también hemos propuesto un modelo sin memoria RAM ni reemplazos que a priori promete ser más sencillo, y que funciona en casos con gran interés como es la codificación subbanda, DCT, etc,

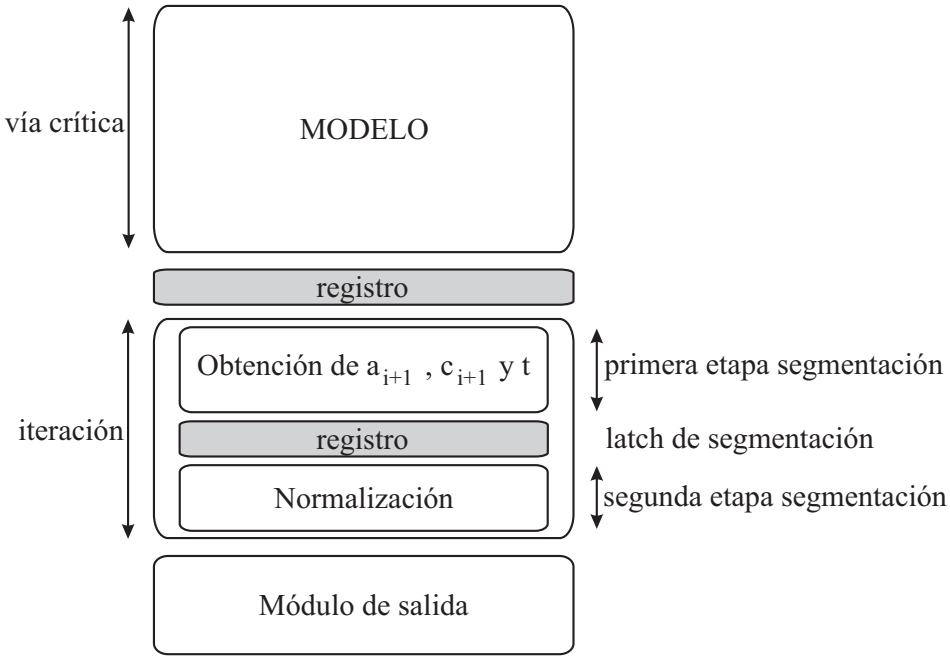


Figura 3.30: Estructura del codificador segmentado.

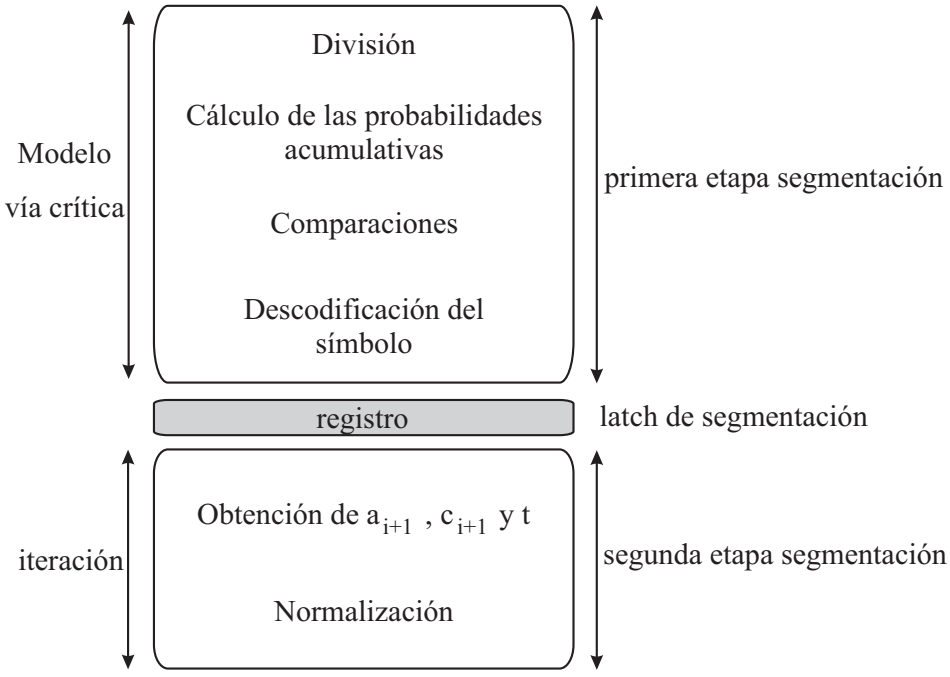


Figura 3.31: Estructura del descodificador segmentado.

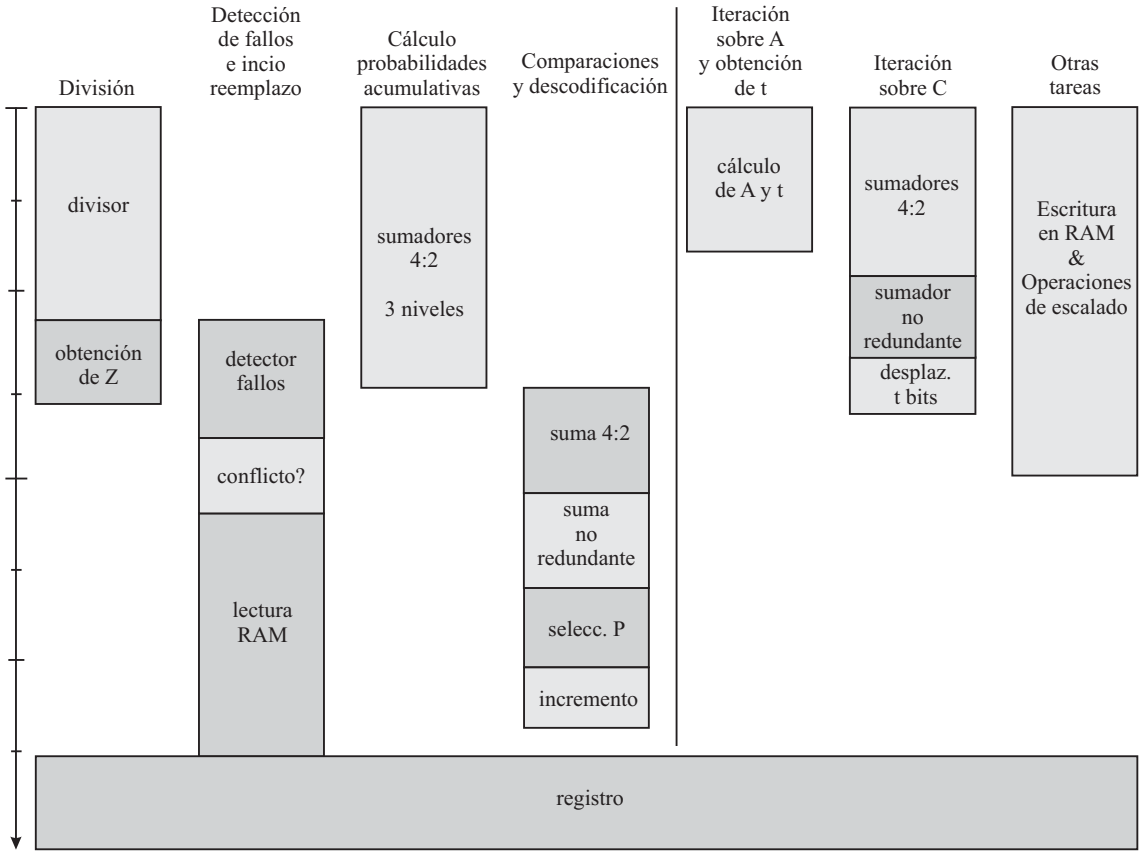


Figura 3.32: Tiempo de computación en el descodificador segmentado. A la izquierda las tareas de la primera etapa y a la derecha las de la segunda. La duración del ciclo es ahora de aproximadamente 70 retardos de puerta nand.

Concepto	Medida	Sin cache	Con cache
ciclo	t_{nand}	41	37
Segmentado	t_{nand}	39	37
RAM	Kbits	3	2
registros	bits	192	128
modelo	sumadores completos	192	99
iteración	cualitativa	medio	bajo

(a) Codificador

Concepto	Medida	Sin cache	Con cache
ciclo	t_{nand}	141	77
Segmentado	t_{nand}	93	70
RAM	Kbits	3	2
registros	bits	192	128
divisor vs multiplicadores	sumadores completos	960	110
sumadores y comparadores	relativo	24	8
iteración	cualitativa	medio	medio

(b) Descodificador

Tabla 3.2: Comparación de arquitecturas.

hemos preferido presentar la arquitectura más completa, y la compararemos con la mostrada en la sección 1.4 ya que la hemos tomado como punto de partida y tienen muchas características en común. Haremos en el capítulo 5 evaluaciones con otras arquitecturas centrándonos en aplicaciones concretas.

En el capítulo anterior nuestro codificador ha demostrado ser más eficiente desde el punto de vista de la compresión. Ahora compararemos los otros dos puntos de interés: tiempo de computación y coste en área. Este último es muy difícil de evaluar por lo que sólo compararemos el coste de los elementos más significativos. Los resultados obtenidos y que comentaremos a continuación son los que se muestran en la tabla 3.2.

3.14.1 El tiempo de computación

En primer lugar consideraremos la velocidad de procesamiento. En el codificador no hay grandes diferencias, 41 retardos de puerta *nand* frente a 37 que hemos obtenido nosotros, aunque los márgenes de error pueden eliminar esta diferencia. Sabíamos que la diferencia no iba a ser importante, ya que el modelo del codificador tenía un ciclo no muy superior al de un acceso a memoria y este se mantiene en nuestra arquitectura. Por otro lado, y también dentro del codificador, el coste de la iteración es bastante inferior en nuestro caso debido a la aritmética de baja precisión, mientras que en [BBL97] definía la vía crítica.

En cambio, el descodificador es el gran beneficiado de la introducción de la cache. Con respecto a la implementación mostrada en [OB97] se consigue una reducción desde 141 t_{nand} hasta sólo 77. En este caso la diferencia es notoria y se consigue el objetivo de casi duplicar la velocidad del punto débil de la codificación aritmética. El motivo de esta diferencia es que hemos reducido las comparaciones a un único nivel, y que estas además son muy rápidas ya que el divisor sustituye a los multiplicadores en las comparaciones. Además, utilizamos aritmética con precisión reducida que acelera la iteración. A mayores hemos de recordar que el escalado de probabilidades se resuelve sin ningún coste en tiempo, por lo que la diferencia real con respecto a todas las implementaciones que no lo consideran es todavía mayor.

En cuanto a la versión segmentada el codificador no experimenta diferencias, pero el descodificador consigue una pequeña mejora. El ciclo se reduce de 77 t_{nand} a 70, mientras que en el descodificador segmentado de [OB97] era de 93. En realidad, nuestra configuración básica ya es mejor que la configuración de partida segmentada.

3.14.2 Consumo de área

Tal y como ya dijimos haremos una comparación del coste elemento a elemento de forma aproximada en lugar de hacerlo en términos de área de chip, ya que la obtención de este dato implicaría el desarrollo completo de la arquitectura que tomamos como referencia.

El primer elemento a considerar es la memoria RAM. Nosotros utilizamos 8 bits para expresar las probabilidades, mientras que el tamaño de palabra habitual es de 12 bits. Por tanto, la relación es de 12 a 8, un 50% más que nuestra propuesta. Por otro lado, hemos entrelazado la memoria para disminuir el número de conflictos de acceso. Sólo hemos necesitado utilizar 4 bloques, mientras que la arquitectura que tomamos como referencia necesita 8, lo cual tiene implicaciones en el área consumida para el routing. A continuación veremos los registros utilizados para almacenamiento del modelo. En la sección 1.4 vimos que se utilizaban 16 registros de 12 bits para las probabilidades acumulativas de referencia. En equivalente en nuestra arquitectura

son los registros de la caches. Estos también son 16, pero la longitud vuelve a ser menor, sólo 8 bits.

El codificador

Los restantes elementos son particulares para el codificador o el descodificador. Comenzaremos por el modelo del primero. La arquitectura de 1.4 utiliza 16 incrementadores de 12 bits para actualizar las probabilidades acumulativas de referencia y un conjunto de sumadores CSA 4:2 y 3:2 para calcular las restantes probabilidades. Nuestra implementación calcula las probabilidades acumulativas con un árbol de sumadores 4:2. Dado que el coste de un sumador 3:2 de un bit es el de un sumador completo, y que el de un sumador 4:2 es aproximadamente un 50% mayor, podemos reducir el coste de todos los sumadores a términos de sumadores completos de un bit. El coste de los incrementadores se puede equiparar al de un sumador normal, aunque realmente sea mayor al estar optimizados para ganar velocidad. En total la relación de costes es de aproximadamente 192 frente a 99, con lo que nuestra implementación tiene la mitad de coste.

Consideraremos ahora el coste de la iteración. La actualización de A tiene un coste muy bajo tal y como se vio, si bien actualizar C es más costoso. En cambio en la implementación vista en 1.4 incluye operaciones de coste mayor y además utilizan mayor número de bits. Si bien ambas utilizan 3 sumadores 4:2, la diferencia de tamaños de palabra desequilibra la balanza a nuestro favor. Sucede lo mismo para con el desplazador que normaliza el resultado, y el coste de multiplicar la probabilidad acumulativa por dígitos radix-4 también es superior al esquema que proponemos. Añádase el hecho de que la iteración sobre A incluye un recodificador a radix-4, un LZD que trabaja en formato carry-save y un desplazador y tendremos que el coste es muy superior, ya que el área ocupada por nuestra iteración sobre A es sólo un 15% de la iteración sobre C según datos de Synopsys. Así pues también en este aspecto nuestra implementación se muestra superior si bien debido a la gran variedad de elementos involucrados no hemos hecho un análisis muy detallado.

La circuitería dedicada a gestionar los fallos es nueva y supone un coste que antes no existía. Sin embargo consideramos que en ningún momento es excesiva y podemos equipararla con la que sería necesaria, por ejemplo, para escalar las probabilidades.

La sección de salida no presenta ninguna diferencia entre las dos arquitecturas de forma que no la comentaremos sino que pasaremos directamente al descodificador.

El descodificador

El descodificador ha cambiado de forma tan radical que nos será más complicado comparar los distintos elementos entre sí. Nuestra implementación incluye un divisor

que antes no existía pero que viene a sustituir a un número elevado de multiplicadores. La comparación se realiza en sólo un nivel en lugar de en 2, y la iteración sobre C incluye una operación no redundante.

En primer lugar calcularemos el coste del divisor en términos de sumadores completos para compararlo con el de los dos niveles de 16 multiplicadores utilizados en la arquitectura que tomamos como referencia. Mirando el esquema de la figura 3.23 podemos apreciar que el coste es de 110 sumadores completos. Hemos incluido los sumadores rápidos utilizados para calcular z y para descodificar los fallos (que no aparece en la figura), considerando que tienen un coste doble que uno normal. En el otro plato de la balanza tenemos un total de 2 niveles de 16 multiplicadores de 12 bits, cada uno de los cuales supone dos sumadores 3:2 y uno 4:2. En total, la cifra se puede estimar en 960 sumadores completos de un bit. El coste del divisor está entonces compensado con creces.

Los comparadores y los sumadores para construir las probabilidades acumulativas también son más sencillos en nuestro caso. El cómputo se puede simplificar si consideramos que son dos niveles frente a uno solo y que el tamaño de palabra es de 12 frente a 8. En total la relación será de 24 a 8 aproximadamente.

Por último con la iteración sucede lo mismo que en el codificador salvo que añadimos su sumador no redundante. Respecto a esto consideramos que si bien representa un coste añadido, se ve compensado en buena medida por el hecho de que el desplazador es más pequeño al no tener que desplazar una palabra de suma y otra de acarreo. Sumando a esto la diferencia de tamaños de palabra y la simplicidad de la iteración sobre A podemos decir que la nueva implementación sigue siendo más ventajosa.

Estas diferencias son también extensibles a muchas otras arquitecturas para codificación y descodificación aritmética multinivel, si bien se trata de meras proposiciones en las que no se concretan aspectos importantes y en las que el coste no está debidamente evaluado.

Capítulo 4

Arquitecturas con jerarquía de un único nivel

En el capítulo anterior hemos visto en detalle una arquitectura con un modelo con dos niveles en la jerarquía. Ésta es la arquitectura más completa y por ello le hemos dedicado un capítulo. Ahora veremos otras arquitecturas que tienen en común el haber prescindido de la memoria principal del modelo para pasar a gestionarlo con la información contenida en la cache únicamente. Estas arquitecturas son adecuadas para histogramas centrados, obtenidos tras aplicar predicción u otras técnicas con efectos similares, y que es el caso con mayor interés en compresión de datos, especialmente en multimedia. Éstas tienen muchas características comunes con la ya vista en el capítulo 3, de manera que obviaremos los detalles comunes.

Las nuevas arquitecturas presentan importantes mejoras que las diferencian de la ya vista. Éstas atañen principalmente a la gestión del modelo, pero también en algún caso a la iteración. Tendremos en cuenta distintas posibilidades para aumentar la capacidad de la cache sin aumentar su tamaño y describiremos también una arquitectura para un algoritmo no adaptativo.

Si bien la desaparición de la memoria principal hace que la cache sea el único elemento de la jerarquía de memoria, seguiremos utilizando el término cache para referirnos al banco de registros. Estrictamente hablando ya no es una cache, pero mantiene algunas de sus características.

4.1 Codificador/descodificador sin memoria RAM

Estas arquitecturas están diseñadas para comprimir datos con los símbolos más probables concentrados en el centro del histograma. Esto supone que la parte más representativa del histograma se puede introducir en la cache (figura 4.1) y pres-

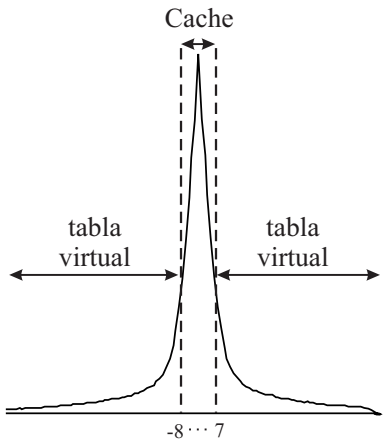


Figura 4.1: Porciones del histograma que se codifican utilizando la cache y la tabla virtual.

cindir del resto, sin almacenarlo en memoria RAM. Al eliminar la memoria RAM tenemos una implementación del codificador que ocupa menos área, y además obtenemos muchos beneficios al desaparecer uno de los aspectos más complejos del funcionamiento de nuestro modelo: los reemplazos. Las consecuencias de esto son principalmente ventajas en términos de área y tiempo.

El funcionamiento básico del modelo sin memoria lo conocemos del capítulo 2, y podemos resumirlo, al igual que se hizo con el modelo básico en capítulo anterior, en un diagrama como el que presentamos en la figura 4.2. Nótese la ausencia de RAM y de su interfaz con la cache. Así mismo, y como ya veremos, la detección de fallos no hace uso de la cache. Las modificaciones sólo afectan al modelo, la iteración y la sección de salida no sufren cambios.

Los elementos que no varían con respecto a la arquitectura básica no se muestran en detalle. Las partes de la arquitectura que deben ser rediseñadas son las siguientes:

- Estructura de la línea: se simplifica al no existir reemplazos.
- Detección de fallos.
- Cálculo de las probabilidades acumulativas.
- Control de saturación: simplificado al desaparecer los reemplazos.
- Escalado de las probabilidades: sólo se escala una probabilidad.

El resto del hardware no se modifica. En particular la iteración no se ve afectada ya que estos cambios sólo implican al modelo. Todo lo dicho es extensible al descodificador, salvo la detección de fallos que se sigue realizando de la misma manera.

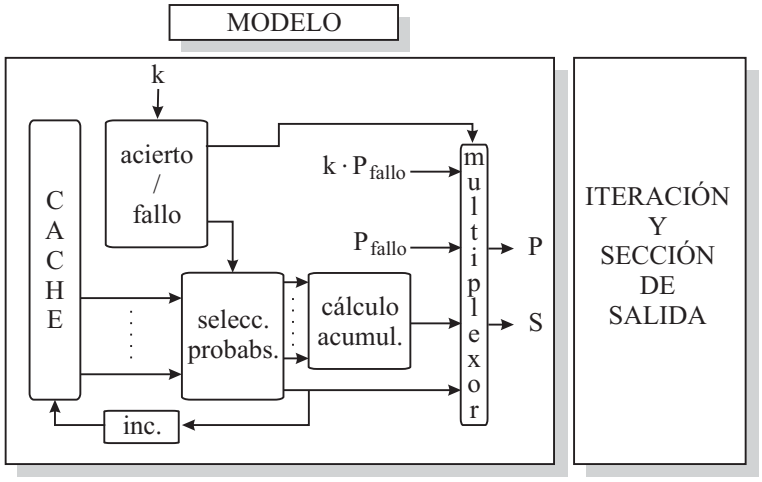


Figura 4.2: Esquema general de un codificador sin RAM.

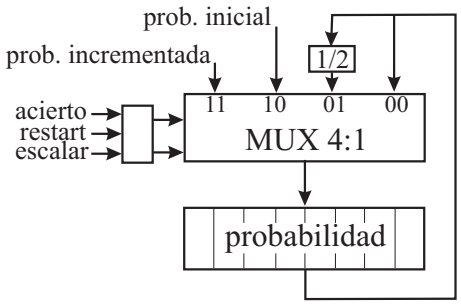


Figura 4.3: Nueva estructura de una línea de la cache.

A continuación revisaremos todas las partes del modelo afectadas por este cambio y finalmente haremos una reevaluación del coste de la implementación.

4.1.1 Estructura de la línea

En la figura 4.3 mostramos la estructura de una línea, que difiere de la vista en la figura 3.5. En esta nueva arquitectura ya no es necesario almacenar el símbolo que hay en cada línea, sino que se sobreentiende. Si en la arquitectura anterior se sobreentendían 4 bits, ahora, al no haber reemplazos, el contenido de la cache es siempre el mismo. El almacenamiento de las probabilidades no ha cambiado pero ahora sólo hay cuatro posibilidades: inicio, sin cambios, incremento en caso de acierto, y escalamiento.

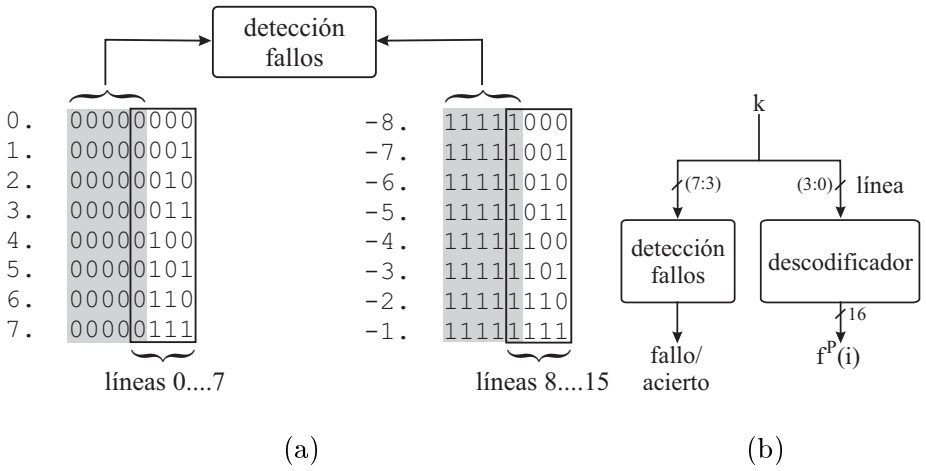


Figura 4.4: (a) Detección de fallos y asignación de los símbolos a las líneas de la cache. (b) Obtención de las señales de control.

4.1.2 Detección de fallos

En lugar de comparar el símbolo entrante con los 16 símbolos de la cache (figura 3.7), se generan directamente las señales de control mediante un descodificador. La forma de implementar la descodificación se muestra en la figura 4.4. Hacemos esto gracias a que los símbolos almacenados en la cache, tienen un patrón de bits fácilmente identificable. Cómo sólo se considera la parte central del histograma (figura 4.1), los símbolos que pueden entrar en la cache van del -8 al -1 y del 0 al 7 sus representaciones binarias hacen que sea muy sencillo determinar si ha ocurrido o no un fallo.

Comprobando los 5 bits más significativos discerniremos entre fallos y aciertos ya que se producirá un acierto si y sólo si los 5 bits son todos '0' o todos '1', tal y como se ve en la figura 4.4.(a). En caso de acierto, los cuatro bits menos significativos indican la línea a la que se dirige cada símbolo (figura 4.4.(b)). Introduciendo estos cuatro bits en un descodificador se obtienen las 16 señales f^P, que necesitamos para seleccionar las probabilidades, de forma más sencilla a como hemos hecho en el capítulo anterior.

4.1.3 Cálculo de las probabilidades acumulativas

El sistema implementado en el capítulo anterior (figura 3.8) era razonablemente rápido y poco costoso. Si bien el cálculo de las probabilidades acumulativas definía la vía crítica, el hecho de que la diferencia con respecto al acceso a memoria fuese

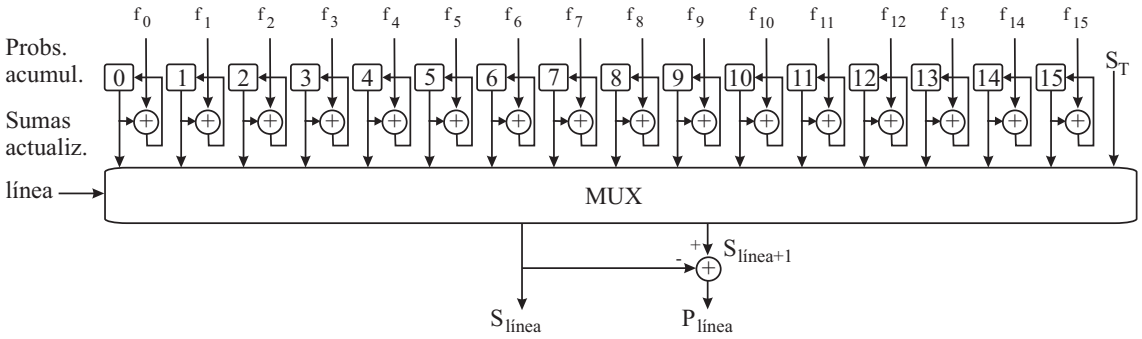


Figura 4.5: Gestión del modelo basado en almacenar las probabilidades acumulativas.

pequeña limitaba la eficacia de cualquier implementación alternativa. Esto es, la memoria definía un mínimo en la duración del ciclo que hacía inútiles los esfuerzos por acelerar otros aspectos del hardware. Ahora el caso es distinto ya que no existe acceso a memoria y una reducción significativa en el coste de calcular las probabilidades acumulativas sería bienvenido. Esta reducción puede venir del comportamiento más predecible de la cache. Al no existir reemplazos los únicos cambios que se pueden producir son incrementos y escalamientos.

Con respecto a los escalamientos que se producen cuando S_T alcanza su valor máximo, recordamos que la solución óptima es escalar una única línea (sección 2.7). Mediante un sistema de turnos todas las líneas terminan por ver escalada la probabilidad que contienen, pero de una en una. De esta manera los cambios en las probabilidades acumulativas son fácilmente controlables. En cada ciclo las probabilidades acumulativas de los símbolos pueden sufrir los siguientes cambios:

- incrementadas debido a un acierto en una línea inferior.
- disminuir en una cantidad $P_{esc}/2$ al escalarse la probabilidad P_{esc} de un símbolo situado también en una línea inferior

La primera posibilidad que se abre es almacenar las probabilidades acumulativas y actualizarlas ciclo a ciclo. Las probabilidades no se almacenan. Se calculan restando las probabilidades acumulativas de símbolos consecutivos y el coste de las actualizaciones es inferior, en tiempo, al del cálculo on-line. Este esquema se muestra en la figura 4.5, consta de un conjunto de registros para almacenar las probabilidades acumulativas y un sumador para actualizar cada una de ellas. La actualización se realiza sumando los factores f_i que pueden tomar los siguientes valores:

$$f_i = \begin{cases} 1 & \text{si ha habido un acierto en una línea situada por debajo} \\ -P_{esc}/2 & \text{si se ha escalado una línea situada por debajo} \\ 0 & \text{en los restantes casos} \end{cases} \quad (4.1)$$

Se seleccionan la probabilidad acumulativa del símbolo requerido y la del siguiente para obtener la probabilidad restando ambas, ya que éstas difieren precisamente en esa cantidad

$$P_i(k) = S_{i+1}(k) - S_i(k) \quad (4.2)$$

Nótese que se necesita el concurso de S_T, la probabilidad acumulativa del tope de la cache, ya que la probabilidad del símbolo de la línea 15 se calcula restando la probabilidad acumulativa de esta línea de la probabilidad acumulativa de la línea 16, y ésta última no existe como tal sino como S_T, la probabilidad acumulativa del tope de la cache.

Este esquema es rápido pero tiene una serie de inconvenientes. El primero es que las necesidades de almacenamiento aumentan mucho. En lugar de 16 registros de 8 bits tenemos 32 registros de 10 bits si se quiere utilizar aritmética redundante o 16 de 10 bits en caso contrario. La ventaja de utilizar aritmética redundante estriba en que los sumadores redundantes para actualizar las probabilidades acumulativas serían muy rápidos. Los sumadores no redundantes en cambio deben aumentar su área para ganar velocidad. En cualquier caso sería necesario al menos un sumador para cada símbolo y el coste en área de una celda de sumador es muy similar al de una celda de registro.

Una posibilidad más sencilla es reproducir el esquema visto en la sección 1.4. Introduciendo probabilidades acumulativas de referencia se reduce el nivel de sumadores para calcular las probabilidades acumulativas. Si en el capítulo anterior eran necesarios tres niveles de sumadores 4:2, con una probabilidad acumulativa de referencia los niveles son sólo 2, e introduciendo más se pueden reducir a un único nivel. Ambas posibilidades se muestran en la figura 4.6.

Las probabilidades (nótese que sólo necesitamos 15) se almacenan en registros de 8 bits (cuadros blancos), y las probabilidades acumulativas de referencia en registros de 10 bits (cuadros oscuros) que pueden ser dobles si se utiliza aritmética redundante. En caso de utilizar dos niveles de sumadores (figura 4.6.(a)) sólo se necesita una probabilidad acumulativa de referencia. Si se quiere reducir a un único nivel (figura 4.6.(b)) será necesario almacenar 4.

Existen sumadores para actualizar las probabilidades acumulativas de referencia (con los factores f_i) y sumadores para calcular las probabilidades acumulativas restantes. Los datos que se introducen en estos sumadores dependen del símbolo cuya probabilidad acumulativa se desee calcular. Así, en el esquema de la figura

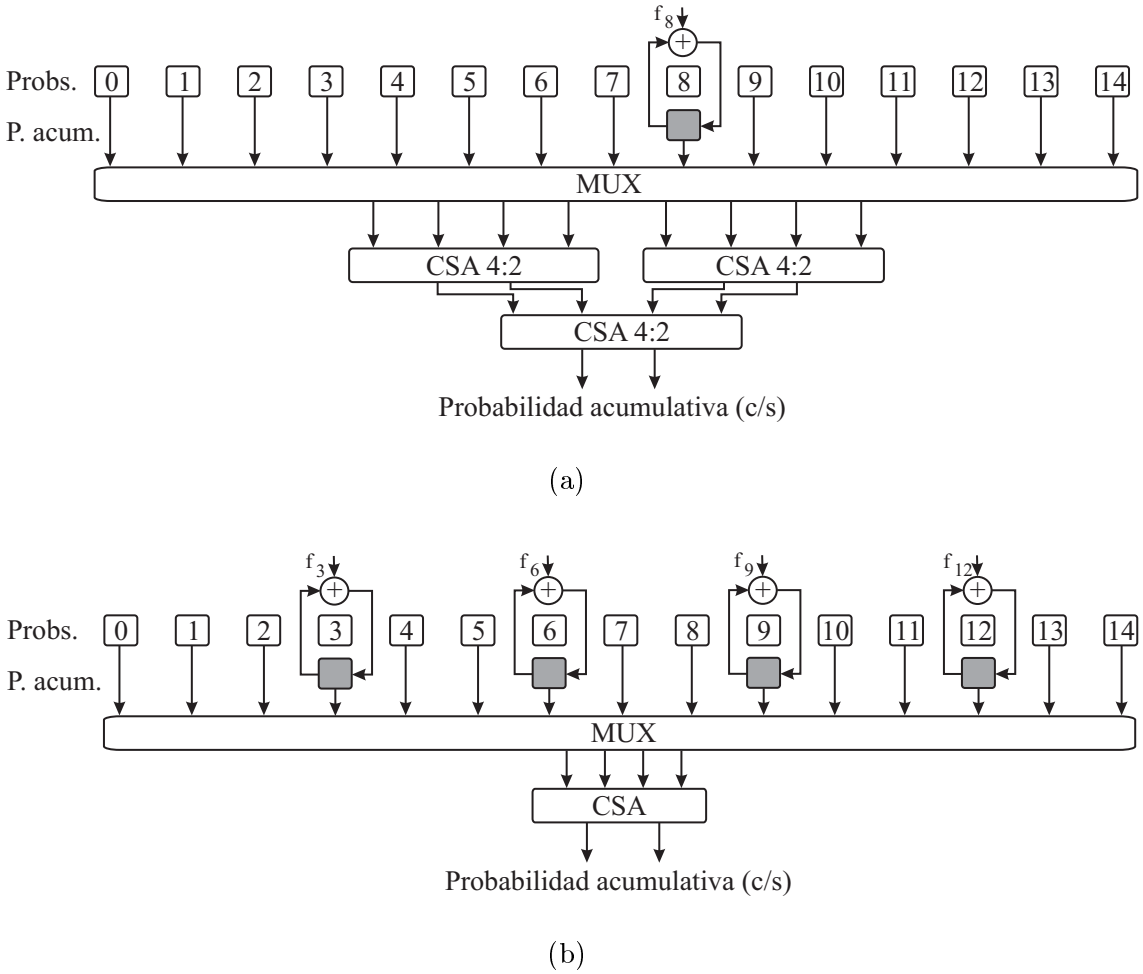


Figura 4.6: (a) Cálculo de las probabilidades acumulativas en dos niveles. (b) Cálculo en un único nivel de sumadores.

	2 niveles no redund	2 niveles redund	1 nivel no redund	1 nivel redund
área registros	138	148	168	208
área CSA's	45	45	10	15
área sumadores actualizaciones	15	10	60	40
total	198	203	238	263
tiempo CSA's + MUX	15	15	8	8
tiempo actualizaciones	8	5	8	5

Tabla 4.1: Comparación entre distintos esquemas para el cálculo de las probabilidades acumulativas. El área se expresa en términos de sumadores completos de un bit, y el tiempo como retardos de puerta *nand* (de forma aproximada).

4.6.(a) seleccionaremos una de las dos mitades de la cache, y en la figura 4.6.(b) las posibilidades son 5. Un multiplexo es el encargado de seleccionar estos datos. Para calcular la probabilidad acumulativa del símbolo de la línea 5 en el segundo esquema, se seleccionan para sumar la probabilidad acumulativa de referencia 3 y la probabilidad del símbolo 4. De esta forma, con un número mínimo de sumadores conseguimos calcular todas las probabilidades acumulativas. El sumador CSA será del tipo 3:2 si las probabilidades acumulativas de referencia se almacenan en formato no redundante y del tipo 4:2 en caso contrario.

Si se utiliza aritmética redundante los registros para almacenar las probabilidades acumulativas de referencia serán el doble de grandes, pero los sumadores para actualizar su valor serán redundantes y su velocidad de procesamiento no estará comprometida con el área. En cambio, si se utiliza aritmética no redundante los registros son más pequeños pero actualizar con rapidez las probabilidades acumulativas de referencia implica utilizar un sumador costoso.

La alternativa por la que hemos optado es el esquema con cuatro probabilidades acumulativas de referencia (un sólo nivel de sumadores) con aritmética no redundante. El motivo es la sencillez y la velocidad. Igualando el coste en área de un registro de un bit con un sumador completo, y el de un sumador rápido con 1.5 sumadores completos, hemos construido la tabla 4.1. Hemos calculado el área de los distintos elementos y el tiempo de las tareas involucradas. El cómputo de registros incluye los que conforman la cache. El tiempo de los sumadores y actualizaciones no se suman porque ambas tareas se realizan en paralelo.

El coste en área de nuestro esquema es relativamente alto, pero no mucho mayor que el más económico, y lo que es más importante es que el tiempo de computación es muy inferior. El coste de los CLA's que se utilizan para actualizar las probabilidades acumulativas de referencia se puede reducir si los requerimientos de velocidad no son muy estrictos. Las estimaciones de tiempo son muy aproximadas pero vemos que

los tiempos de cálculo y de actualización son bastante similares.

Nos resta sólo detallar en que momento se realiza la actualización de las probabilidades acumulativas de referencia. Ya se ha dicho que se hace en paralelo con el cálculo de las probabilidades acumulativas, pero no las dependencias con otras operaciones. En primer lugar, cuando se obtienen las señales f^P se obtienen también las señales que indican que probabilidades y que probabilidades acumulativas se utilizarán en los cálculos de forma similar a lo visto en la figura 3.7, y además se obtienen señales adicionales para indicar el incremento de las probabilidades acumulativas de referencia. El tiempo de obtención de estas señales es inferior al de seleccionar la probabilidad que ha de ser incrementada.

El tiempo de incremento es similar al de incrementar la probabilidad (ligeramente superior) por lo que ambas operaciones se culminan prácticamente a la par. En caso de que la probabilidad no se haya incrementado por ser ya el valor máximo, los registros que contienen las probabilidades acumulativas de referencia no cargan los valores incrementados. De esta forma evitamos tener que esperar a conocer el valor de la probabilidad antes de incrementar. La actualización cuando se produce un escalamiento la describiremos en la siguiente sección.

4.1.4 Control de saturación y escalamiento de probabilidades

La detección del momento en que se satura la tabla de probabilidades se hace de forma distinta a como se vio en el capítulo anterior. Los únicos incrementos que pueden sufrir las probabilidades acumulativas, y en particular S_T, son de una unidad tras un acierto. Los reemplazos podían provocar cambios bruscos, pero éstos ahora no existen. Así, controlar el valor de S_T es más sencillo y la saturación se detecta en muy poco tiempo. Cuando se produce una saturación se corrige una probabilidad escalándola, y al igual que antes se fuerza un fallo para tener libre la cache durante el ciclo que se dedica al escalado. El símbolo que sufrirá el escalamiento se decide por turnos, por lo que surgen diferencias con el esquema ya comentado.

Éstas se deben a que es posible que la probabilidad que se escala sea el valor mínimo, por lo que ya no disminuiría más su valor y no se produciría ninguna reducción en el valor de S_T. Hemos decidido utilizar un sistema de escalado diferente, que incumbe también a la actualización de las probabilidades acumulativas de referencia. Cuando se escala una probabilidad, se comprueba en primer lugar si es la mínima posible. En este caso no se hace nada y se espera al siguiente ciclo para que avance el turno y escalar la probabilidad de la línea siguiente. El proceso puede continuar hasta llegar a un símbolo con probabilidad mayor que la mínima. En la práctica esta contingencia no debiera darse mas que una o dos veces. Al descartarse que un escalamiento provoque que una probabilidad sea nula, podemos escalar dividiendo

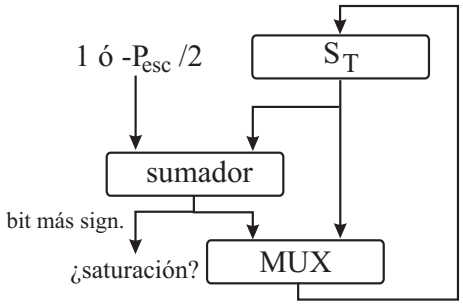


Figura 4.7: Control de saturación.

por dos sin más, sin forzar a que el bit menos significativo sea un '1'.

La actualización de las probabilidades acumulativas es sencilla ya que se le resta la cantidad que se pierde al escalar la probabilidad, esto es, $P/2$ ó $P/2+1$. Como ya hemos dicho una probabilidad de referencia sólo se ve afectada por un escalamiento si se produce en una línea situada por debajo de ella. Por tanto, corregiremos las probabilidades acumulativas de referencia que se vean afectadas. El valor de S_T se ve afectado por todos los escalamientos ya que ocupa el punto alto de la cache.

En la figura 4.7 se muestra como se realiza la detección y corrección. El mismo hardware se utiliza con las probabilidades acumulativas de referencia. La detección consiste en ver si el valor es igual que 1024, valor máximo de S_T, por lo que no requiere hardware especial. La operación normal es el incremento, salvo si no se incrementa la probabilidad, en cuyo caso se utiliza el antiguo valor de S_T. Sólo tras detectarse una saturación se resta la probabilidad escalada para corregir su valor.

4.1.5 Modelo del descodificador

Al igual que el codificador, la desaparición de la RAM implica únicamente al modelo. Pero en este caso no se obtiene un mejora importante de velocidad por los motivos que ahora comentamos. Si observamos la figura 3.28 veremos que el cálculo de las probabilidades acumulativas termina poco después de haber terminado la división.

Para nuestros propósitos sería conveniente que la descodificación dependiese únicamente de las probabilidades acumulativas, de forma que al simplificar el cálculo de estas se llegase a un ciclo más corto. Sin embargo, sólo podemos obtener una pequeña mejora ya que se mantiene la dependencia con el divisor. Hemos conseguido, eso sí, eliminar la RAM y los reemplazos. El problema de la búsqueda del símbolo, el más costoso en términos de área y tiempo en un descodificador aritmético, ha sido resuelto en el esquema con jerarquía de memoria en dos niveles y se mantiene en este esquema con un solo nivel.

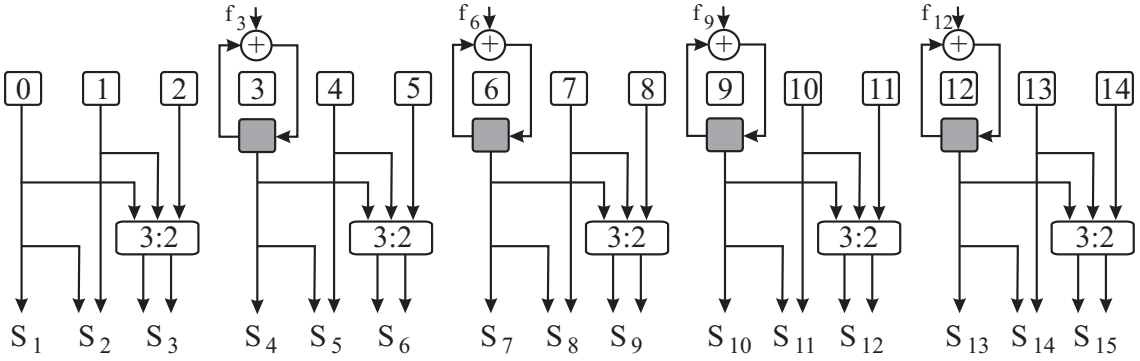


Figura 4.8: Modelo del descodificador. Obtención de las probabilidades acumulativas. No se muestran las comparaciones.

El cálculo de las probabilidades acumulativas cambia ligeramente, ya que ahora algunas de ellas están almacenadas (probabilidades acumulativas de referencia). Si el esquema visto en el capítulo anterior (figura 3.26) existía un importante consumo de área debido a los sumadores, ahora hemos optado por un esquema en el que los registros son la parte más importante.

El esquema se muestra en la figura 4.8, donde vemos que muchas de las probabilidades acumulativas se obtienen directamente de las de referencia. Como tanto éstas como las probabilidades de los símbolos se almacenan en formato no redundante, el valor de las probabilidades acumulativas de algunos líneas (2,5,8...) se obtiene combinando la probabilidad acumulativa de referencia inmediatamente anterior y una probabilidad, ya que la probabilidad acumulativa se obtiene en forma de una palabra de semisuma y otra de acarreo. Otras, en cambio, (3,6,9...) necesitan el concurso de un sumador 3:2 para calcularse. Nótese el importante ahorro en sumadores que se ha conseguido. En la siguiente sección evaluaremos la arquitectura y veremos las mejoras obtenidas en el descodificador.

4.1.6 Evaluación de la arquitectura sin RAM

Procederemos ahora a evaluar la duración del ciclo de procesamiento y el coste de implementación de la nueva arquitectura. Para el caso del codificador, en términos de área será necesariamente ventajosa salvo por el pequeño incremento que se ha producido en el modelo pero que será compensado por la eliminación de la RAM. El tiempo de computación también se reduce de forma significativa, y para estimarlo hemos construido el diagrama de la figura 4.9. Para el descodificador la reducción es menor como podemos ver en la figura 4.10.

Las principales diferencias con respecto a las figuras 3.22 y 3.28 son la desapari-

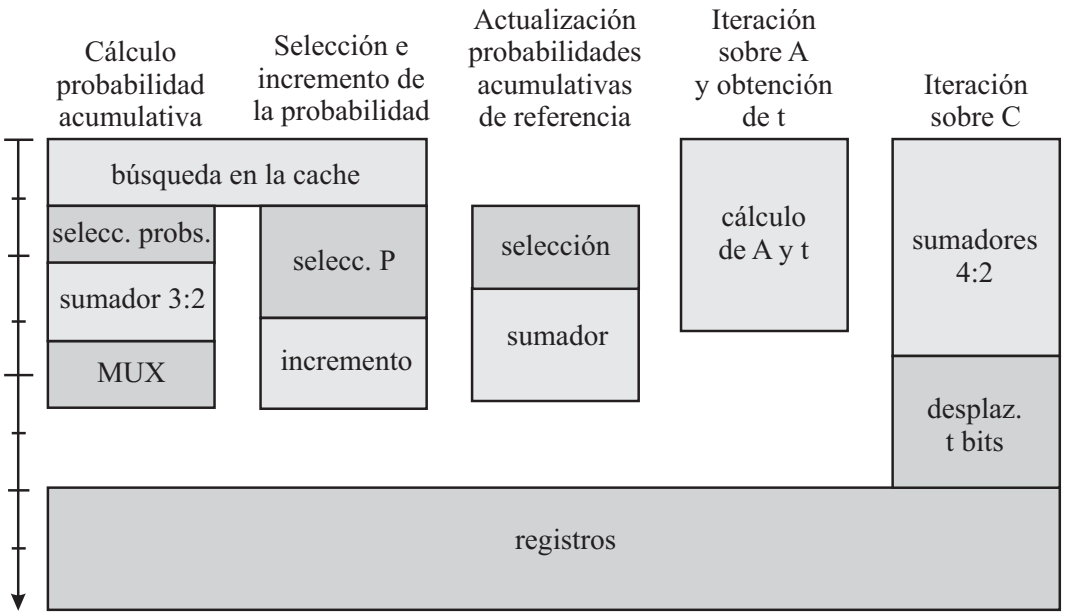


Figura 4.9: Diagrama de tiempos para el codificador sin RAM. La duración del ciclo es de aproximadamente 28 t_{nand}.

ción de la memoria RAM y todas las tareas asociadas a los reemplazos. Además, ya no consideramos el control de saturación de las probabilidades como una tarea especialmente crítica sino que es un caso particular de la gestión de las probabilidades acumulativas de referencia.

La disminución del tiempo necesario para calcular las probabilidades acumulativas es otro punto importante. En el codificador nos permite reducir la duración del ciclo, haciendo que esta dependa ahora de la iteración, y no del modelo. En el descodificador no se da esta situación, el divisor es quien fija el momento en que pueden comenzar las codificaciones sin importar que ahora las probabilidades acumulativas estén disponibles desde mucho antes.

Las características principales de codificador y descodificador se resumen en las tablas 4.2 y 4.3, comparándolas con la arquitectura del capítulo anterior. También mostramos los tiempos para la versión segmentada de esta arquitectura.

En el codificador el nuevo método para calcular las probabilidades acumulativas supone un gasto adicional en registros que se compensa en parte por un ahorro en sumadores. Por el contrario, el descodificador consigue una mejora neta, pues aunque el número de registros aumenta, el número de sumadores desciende de forma importante. El descenso se produce tanto porque se simplifica el cálculo de las probabilidades acumulativas como porque se simplifican las comparaciones.

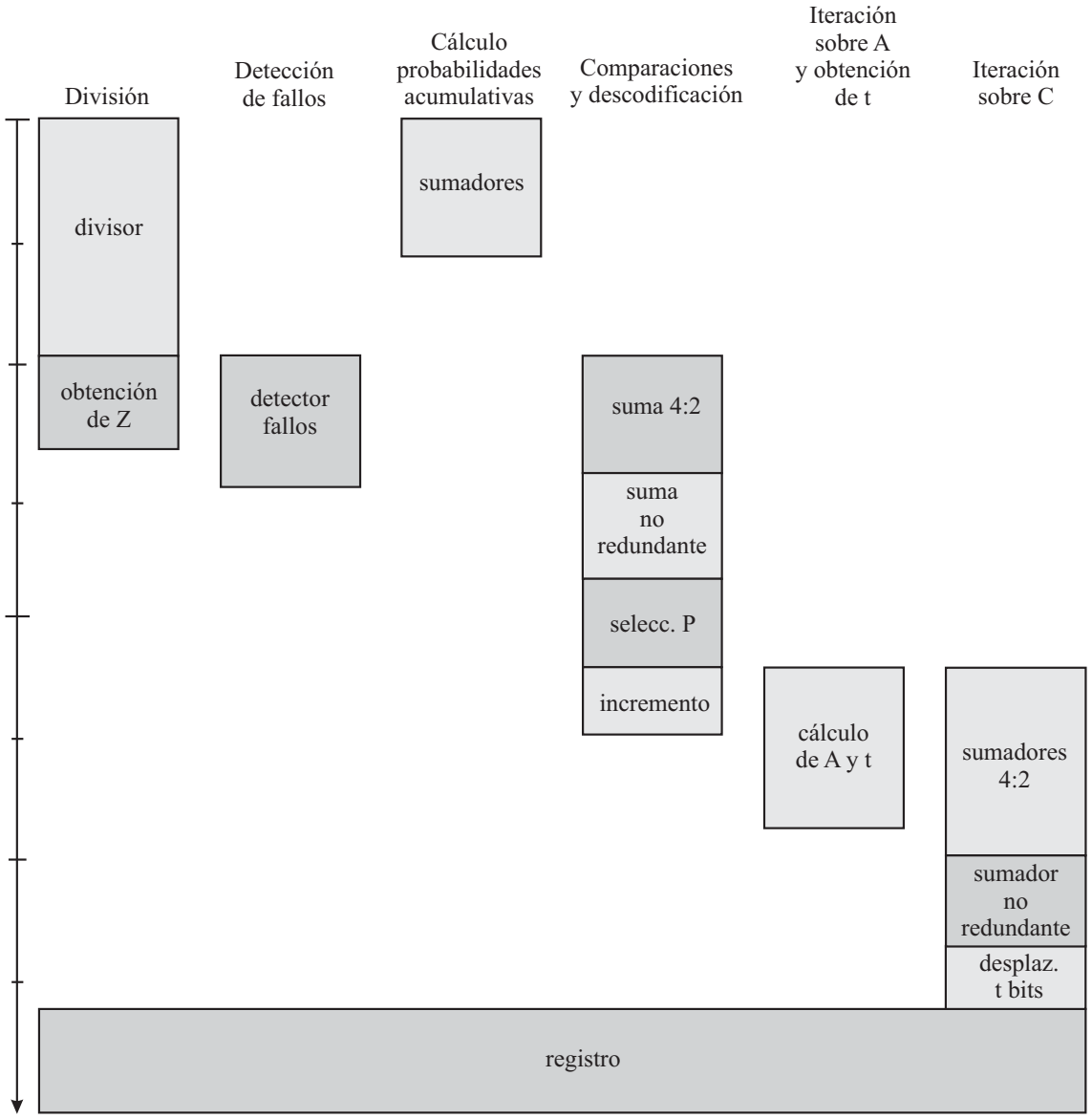


Figura 4.10: Diagrama de tiempos para el descodificador sin RAM. La duración del ciclo es de aproximadamente 72 t_{nand}.

Concepto	Medida	Sin RAM	Con RAM
ciclo	t_{nand}	28	37
ciclo segm.	t_{nand}	24	37
RAM	Kbits	0	2
registros	bits	168	128
modelo	sumadores completos	70	99

Tabla 4.2: Comparación de arquitecturas para el codificador

Concepto	Medida	Sin RAM	Con RAM
ciclo	t_{nand}	72	77
ciclo segm.	t_{nand}	52	70
RAM	Kbits	0	2
registros	bits	168	128
modelo	sumadores completos	110	244
comparaciones	sumadores completos	350	370

Tabla 4.3: Comparación de arquitecturas para el descodificador

Las comparaciones (ver parte inferior de la figura 3.26), necesitan en primer lugar reducir el número de sumandos a dos, para aplicar a continuación una suma no redundante y detectar el signo. La reducción a sólo dos sumandos se hacía en el capítulo anterior con sumadores 4:2. En nuestro caso, se pueden utilizar sumadores 3:2 en muchos casos porque las probabilidades acumulativas de referencia están en formato no redundante (figura 4.8).

El coste de la suma no redundante no es alto ya que en realidad sólo necesitamos obtener el signo del resultado, que es el bit más significativo. Por ello el esquema es el de un generador de acarreo adelantado y un único sumador completo de un bit. Las estimaciones de área que hemos hecho nos dicen que su coste es equivalente al de un CSA 3:2 de 10 bits.

En resumen, esta arquitectura consigue importantes mejoras en velocidad (sobre todo en el codificador) y en área, especialmente en el descodificador. La eliminación de la RAM es un logro por sí, al que hemos de añadir la desaparición de la circuitería para realizar los reemplazos, si bien su tamaño no es relevante y no la hemos considerado en las comparaciones. En términos de área la memoria RAM es una forma muy eficiente de almacenar información, pero el gran tamaño del alfabeto que manejamos (2 kbits) y la necesidad del entrelazado, suponían una importante contribución al área, que hemos logrado eliminar.

4.2 Mejora de la relación de compresión

En esta sección presentaremos modificaciones del modelo sin memoria RAM que nos permiten mejorar la relación de compresión sin aumentar el coste. Las mejoras vendrán por distintas vías:

- Aumentando la capacidad de la cache para disminuir el número de fallos.
- Modificando la forma en que se actualiza el modelo, llegando a una gestión más eficiente de las probabilidades.
- Codificando los fallos de forma diferente a como hemos hecho hasta ahora, de forma que disminuye el impacto que la tabla virtual tiene en la compresión.

Comprobaremos como podemos llegar a implementaciones más sencillas del codificador y del descodificador mejorando también la relación de compresión.

4.2.1 Cache con capacidad aumentada

Si bien la eficiencia de nuestro modelo es bastante competitiva, sería deseable mejorar un poco más las relaciones de compresión que se pueden obtener con él. En la sección anterior hemos implementado una cache con 16 líneas debido a su buena relación prestaciones/coste, pero la cache de 32 líneas es la que obtiene mejores resultados, tal y como hemos visto en la figura 2.35.

Buscando una forma de acercarnos a los resultados de una cache de 32 líneas sin renunciar a la sencillez de la de 16, hemos probado con un modelo ligeramente distinto. Estudiando los histogramas de muchas imágenes tras aplicar un predictor hemos comprobado que en todas ellas las probabilidades de los símbolos cambian con una cierta suavidad. En la figura 4.11 vemos el histograma para *Barbara*. La región central es la que experimenta las mayores variaciones de símbolo a símbolo, pero en valor relativo no son grandes. La idea que surge inmediatamente es considerar iguales las probabilidades de símbolos próximos y reducir así la cantidad de información que ha de almacenar el modelo.

La idea es muy atractiva y acepta distintas implementaciones, desde conjuntos de longitud fija hasta un sistema adaptativo. Una de nuestras prioridades es mantener la complejidad baja, de forma que sólo optaremos por las implementaciones más sencillas.

Tomando como referencia los resultados de la figura 2.35 debemos llegar a mejoras significativas en la compresión sin aumentar en exceso la complejidad. La primera posibilidad que contemplamos es la de incluir dos símbolos en cada línea de la cache, que compartirán la misma probabilidad. Los resultados obtenidos se

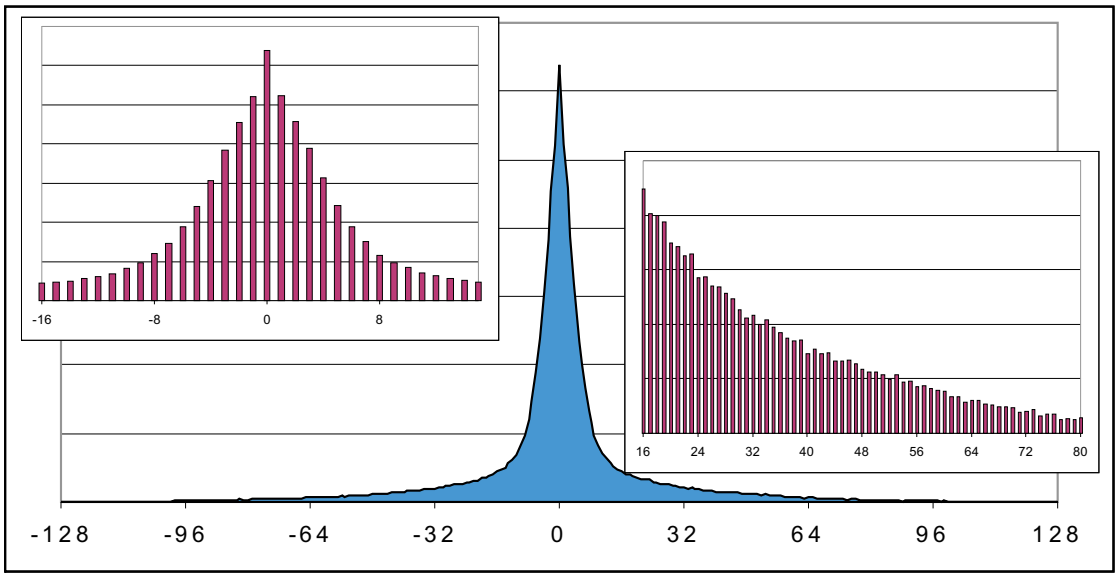


Figura 4.11: Histograma de la imagen *Barbara* tras aplicar predicción. En detalle se muestran la parte central y una región alejada de la misma.

muestran en la figura 4.12, donde también podemos ver los resultados de la figura 2.35 para 16 y 32 líneas. Vemos que mejoran de forma clara a los presentados en la figura 2.35. En esencia casi se logra emular el comportamiento de una cache normal utilizando otra con la mitad de líneas. Recordamos que son resultados de compresión promediados sobre las imágenes estudiadas en el capítulo 2.

Se almacena una única probabilidad para ambos, que se incrementa por acierto en cualquiera de los dos símbolos. La probabilidad acumulativa de ambos ha de ser distinta, y diferirá en un valor igual a su probabilidad. Para una línea l que almacena los símbolos a y b tendríamos:

$$S_l^a = \sum_{j=0}^{j<l} (2 \cdot P_j) \tag{4.3}$$

$$S_l^b = \sum_{j=0}^{j<l} (2 \cdot P_j) + P_l \tag{4.4}$$

Podríamos haber optado por asignar a cada símbolo una probabilidad igual a 1/2 de la almacenada en la cache, ya que esta se incrementa por aciertos en cualesquiera de los símbolos. Sin embargo hemos preferido ser más simples y asignar el valor total a ambas. El modelo saturará con mayor frecuencia, pero las probabilidades evolucionan con mayor velocidad.

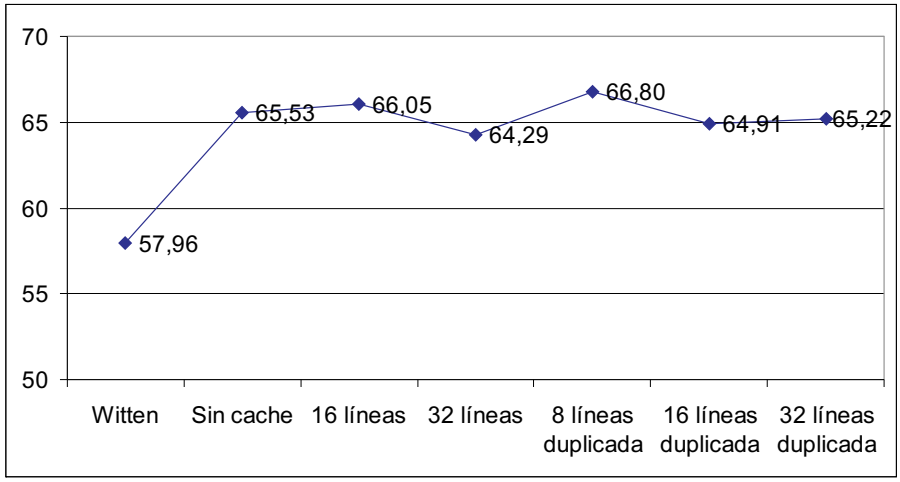


Figura 4.12: Relación de compresión duplicando la capacidad de la cache.

El esquema que sale más favorecido es el de 16 líneas. Duplicar la cache de 32 líneas conlleva en realidad una cierta penalización. El motivo es que aumentan demasiado los escalamientos de las probabilidades. En general, todos los parámetros de la cache guardan un delicado equilibrio entre ellos, y la cache de 16 líneas que hemos utilizado para desarrollar las arquitecturas es la más atractiva para la implementación.

La siguiente posibilidad es explotar más la idea y almacenar 4 símbolos dentro de cada línea. Los resultados los mostramos en la figura 4.13 comparándolos con los de la figura 4.12.

Los resultados, que no son malos, especialmente para las caches más pequeñas, están lejos de los obtenidos anteriormente. El motivo lo podemos deducir de lo ya visto en el caso anterior para la cache de 32 líneas: el aumento en el número de escalamientos. Por otro lado las caches más pequeñas tienen sus limitaciones a la hora de reflejar con fidelidad una parte significativa del modelo.

En resumen, nos decantamos por utilizar una cache de 16 líneas cada una de las cuales almacena dos símbolos consecutivos. De esta manera los resultados son casi equivalentes a utilizar una cache de 32 líneas.

4.2.2 Algoritmo sin multiplicaciones

En la sección anterior hemos llegado a una nueva configuración del modelo que nos permite mejorar la compresión ligeramente. Estudiando la operación del codificador hemos llegado a la conclusión de que se puede mejorar todavía más permitiendo a

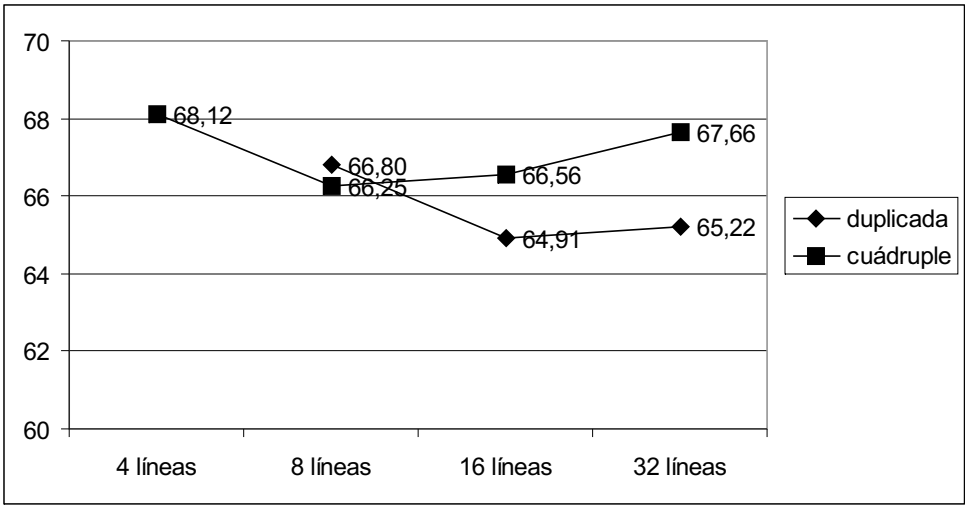


Figura 4.13: Relación de compresión cuadruplicando la capacidad de la cache.

su vez importantes simplificaciones. Estas modificaciones sólo son aplicables a casos en los que la adaptabilidad del modelo no es un factor crítico en la compresión, por lo que se podría dudar de su aplicación a todos los casos.

Las conclusiones a las que hemos llegado provienen de un estudio para simplificar la implementación del modelo con dos símbolos por línea. En particular estábamos interesados en agilizar la actualización de las probabilidades y en reducir el número de fallos. Como resultado hemos llegado al esquema que describimos a continuación.

Cada línea de la cache contiene dos símbolos consecutivos. La probabilidad se incrementa por aciertos en cualquiera de los dos símbolos. Sin embargo, y aquí reside la diferencia con el método anterior, en los cálculos se utiliza una aproximación de la probabilidad que sea potencia entera de 2. Elegimos la potencia entera de 2 más alta tal que sea menor o igual que la probabilidad real. En esencia se trata de utilizar únicamente el bit más significativo de la probabilidad. Al aproximar las probabilidades por potencias de 2 nuestro codificador emparenta con el codificador de Huffman, si bien sigue siendo adaptativo con un coste muy inferior a cualquiera de las implementaciones adaptativas del método de Huffman que vimos en la sección 1.2.1.

Aproximar las probabilidades por potencias de 2 tiene dos efectos. El primero es que al ser la iteración sobre el rango de la forma

$$a_{i+1} = A_i \cdot P_i(k) \tag{4.5}$$

el valor de a_{i+1} tras la normalización (A_{i+1}) será siempre la unidad ya que $P_i(k)$ sólo

tiene un bit no nulo.

$$P_i(k) = 2^x \tag{4.6}$$
$$a_{i+1} = A_i \cdot 2^x \tag{4.7}$$
$$A_{i+1} = a_{i+1} \cdot 2^t \tag{4.8}$$
$$t = 10 - x \tag{4.9}$$

De esto se deduce que siendo el rango la unidad, no es necesario realizar multiplicaciones en la iteración, ni tampoco utilizar un divisor en la descodificación. Los algoritmos de codificación sin multiplicaciones están bastante extendidos, pero implican siempre una importante pérdida en compresión tal y como se vio en la figura 2.26.

Sin embargo, nosotros no experimentamos esa merma en compresión sino el efecto contrario porque existe otro efecto que se alía a nuestro favor, el menor número de saturaciones del modelo sin que se pierda adaptabilidad. Al utilizarse siempre probabilidades iguales o menores que las reales, su suma es menor y al valor de S_T se mantiene por debajo del real. Sólo cuando la probabilidad de una línea (dos símbolos) crece hasta la próxima potencia de 2, se igualan el valor real y el utilizado.

En términos de compresión la posición del bit más significativo es el factor dominante para definir la longitud del código generado, mientras que los restantes bits influyen de forma secundaria. Sin embargo su contribución es importante ya que el hecho de tenerlos en cuenta marca la diferencia entre la codificación aritmética y otros métodos de compresión. El hecho de que en nuestro caso eliminar esta contribución sea beneficioso se debe a efectos producidos por la cache. Nuestro modelo sólo almacena algunas probabilidades y su comportamiento no se puede predecir siguiendo los mismos razonamientos que otros modelos.

Los resultados obtenidos se muestran en la figura 4.14 comparándolos con las restantes configuraciones vistas hasta el momento incluyendo el algoritmo clásico de Witten. Las diferencias respecto a este último se mantienen pero se aprecia una tendencia a mejorar.

Sobre los efectos sobre la complejidad hardware de estas modificaciones volveremos más adelante en esta misma sección.

4.2.3 Gestión de los fallos

En el capítulo 2 desarrollamos un método para codificar los fallos producidos en la cache con un coste aceptable y sin detener la codificación ni demorarla. En este momento, habiendo conseguido ya una arquitectura rápida y poco costosa, centramos nuestros esfuerzos en mejorar la relación de compresión. El aumento de la capacidad de la cache y la aproximación de las probabilidades por potencias de 2

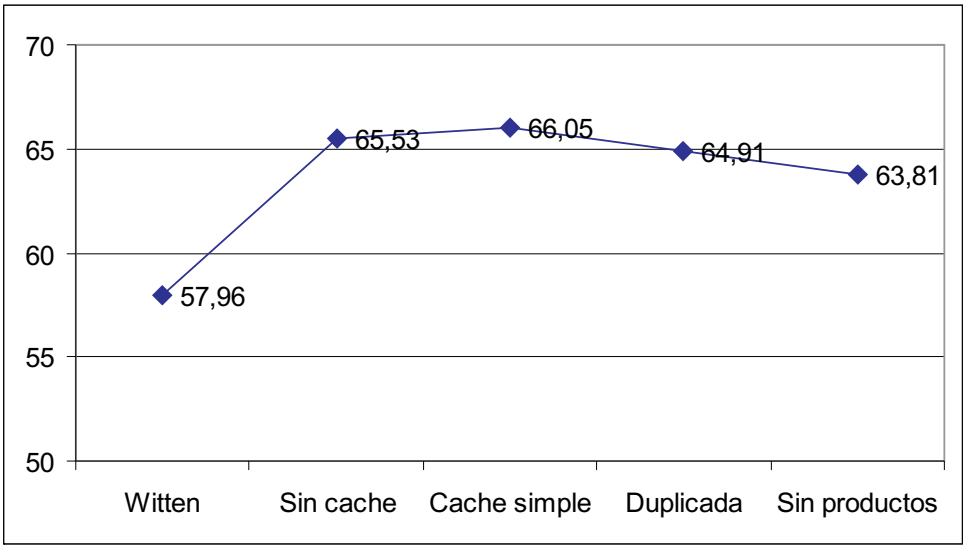
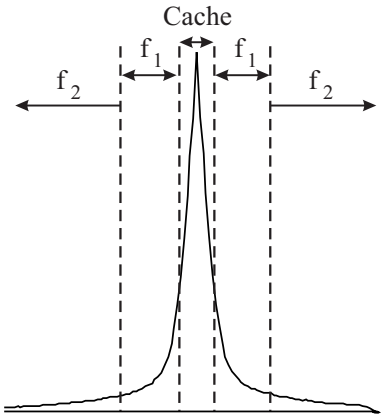


Figura 4.14: Resultados obtenidos sin utilizar productos para caches de 16 líneas.

ha sido dos pasos guiados por la búsqueda sistemática de pequeñas imperfecciones en el modelo.

En este punto le llega el turno a la gestión de los fallos, que se codifican de una forma muy eficiente en términos de tiempo de computación pero no tanto en términos de eficiencia en la compresión. Al codificarse todos los fallos con una frecuencia fija no existe un coste proporcional a la probabilidad de cada uno. Por otro lado la tabla virtual es muy útil pero limita el crecimiento de las probabilidades por suponer una aportación importante al valor de S_T. Si la eliminamos, conseguiremos reducir el número de actualizaciones y esto tendría un efecto beneficioso sobre la compresión.

Por tanto, codificaremos los fallos en dos ciclos. Esto supondrá una pérdida de velocidad pero, tratándose de una arquitectura muy rápida, creemos que es justificable en aras de una mejor compresión. En promedio, para las imágenes que estamos considerando, supone un 10% adicional de ciclos para procesar los fallos. En el primer ciclo se utiliza un símbolo especial para indicar que se ha producido un fallo, y en el segundo se codifica el símbolo utilizando el mismo esquema de la tabla virtual. Este procedimiento admite un refinamiento adicional que es el de utilizar más de un símbolo para codificar los fallos. Cada uno de ellos representa un fallo en una determinada región del histograma, y tendrá una probabilidad acorde con la probabilidad que existe de que se produzca un fallo en esa región. La figura 4.15 muestra un ejemplo cuando se utilizan dos símbolos, f_1 y f_2 para codificar los fallos. Cada símbolo tiene asociada una región a cada lado del histograma, y cada uno de ellos utiliza una línea de la cache.



(a)

0 , 1	0
2, 3	1
4, 5	2
6, 7	3
8, 9	4
10 , 11	5
12 , 13	6
f 1	7
f 2	8
-13 , -14	9
-11, -12	10
-9 , 10	11
-7 , -8	12
-5 , -6	13
-3 , -4	14
-1 , -2	15

(b)

Figura 4.15: Codificación de los fallos con dos símbolos especiales (f_1 y f_2). (a) codificación de las distintas regiones del histograma. (b) símbolos contenidos en la cache.

En la figura 4.16 se comparan los nuevos resultados con los obtenidos anteriormente. Estos resultados se han obtenido utilizando dos símbolos para codificar los fallo. Un primer símbolo codifica los 32 símbolos más cercanos a la cache que no están incluídos en ella y el segundo codifica los restantes. Podemos apreciar que continúa la tendencia a mejorar, si bien en este caso hemos pagado el precio de que la codificación sea ligeramente más lenta. Otras configuraciones con uno y dos símbolos para codificar los fallos han obtenido peores resultados y no se muestran.

Observando con detenimiento la figura vemos que se ha dado un paso muy importante. La diferencia con respecto al algoritmo que se acostumbra a considerar óptimo es ahora inferior al 3%, y mejora en casi un 5% a la configuración sin cache.

Por otro lado, el tiempo total de computación se verá alargado debido a los fallos que se produzcan. La duración de cada ciclo no aumentará, pero si el número de estos. Con el conjunto de imágenes con el que trabajamos el número extra de ciclo necesarios varía entre el 1% y el 25%, siendo el valor medio del 8%. Consideramos que la mejora en compresión, así como las simplificaciones en el modelo y el codificador en general introducidas hasta el momento, justifican sobradamente este pequeño incremento.

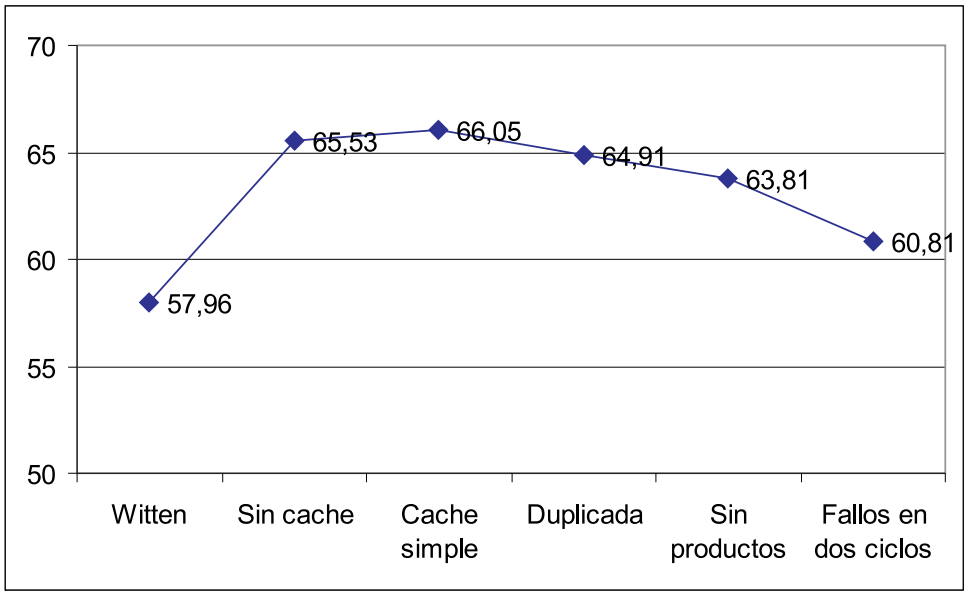


Figura 4.16: Codificación de los fallos en dos ciclos para caches de 16 líneas. Se utilizan dos símbolos para codificar fallos en distintas regiones del histograma.

4.2.4 Implementación de la nueva arquitectura

Veremos ahora los detalles más relevantes de la implementación y haremos una evaluación de la misma. Debido a que la mayor parte de los elementos ya han sido descritos no entraremos en detalles concretos salvo en casos muy particulares.

El modelo en el codificador

Se mantiene, con respecto a las otras arquitecturas, el almacenamiento de las probabilidades y de las probabilidades acumulativas de referencia en registros. Llamaremos P_i' a las aproximaciones de P_i por potencias enteras de 2, y se obtendrán a partir de las P_i cada vez que sean necesarias. Es decir, se almacena el valor P_i pero no su aproximación. El coste de almacenarlas y actualizarlas sería demasiado grande, así que es preferible utilizar lógica combinacional sencilla y rápida para calcularlas on-line. La función lógica necesaria es relativamente sencilla y se pueden balancear la velocidad y el área. La implementación básica la mostramos en la figura 4.17.

Se mantiene el esquema de la figura 4.6.(b) con probabilidades acumulativas de referencia. Como cada línea engloba dos símbolos, la obtención de las probabilidades acumulativas se realiza sumando las probabilidades multiplicadas por 2, salvo cuando se sume la probabilidad de un sólo símbolo de la línea. Por ejemplo: las

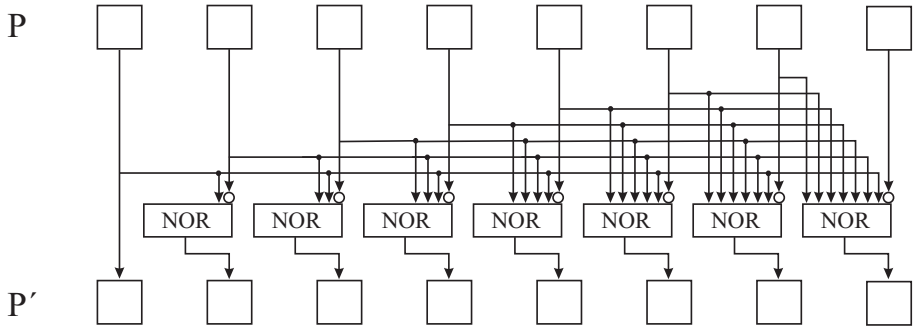


Figura 4.17: Conversión de la probabilidad de un símbolo a una aproximación por una potencia entera de 2.

probabilidades acumulativas de los símbolos 3 y 4 serían:

$$S_3 = P'_0 + P'_1 + P'_2 = 2 \cdot P'_{0,1} + P'_{2,3} \quad (4.10)$$

$$S_4 = P'_0 + P'_1 + P'_2 + P'_3 = 2 \cdot P'_{0,1} + 2 \cdot P'_{2,3} \quad (4.11)$$

Es decir, existe una salvedad con los símbolos que no *encabezan* su línea. Los símbolos utilizados para codificar los fallos también tienen un tratamiento especial ya que ocupan cada uno de ellos una línea. Esto no representa ninguna dificultad ya que las probabilidades que se han de sumar se gestionan principalmente a nivel de línea y no a través de un control central.

El esquema propuesto aparece en la figura 4.18. Para calcular una probabilidad acumulativa se selecciona la de referencia más próxima y las probabilidades que sea necesario sumar. Si el símbolo que se procesa encabeza su línea serán necesarios a lo sumo tres sumandos, de lo contrario se suman cuatro (ver figura 4.15 para comprender que probabilidades se suman)

$$S_8 = \sum_{j=0}^{j=7} P_j = S_3^{refer} \quad (4.12)$$

$$S_9 = \sum_{j=0}^{j=8} P_j = S_3^{refer} + P_8 = S_3^{refer} + P_{8,9} \quad (4.13)$$

$$S_{10} = \sum_{j=0}^{j=9} P_j = S_3^{refer} + P_8 + P_9 = S_3^{refer} + 2 \cdot P_{8,9} \quad (4.14)$$

$$S_{11} = \sum_{j=0}^{j=10} P_j = S_3^{refer} + P_8 + P_9 + P_{10} = S_3^{refer} + 2 \cdot P_{8,9} + P_{10,11} \quad (4.15)$$

$$S_{12} = \sum_{j=0}^{j=11} P_j = S_3^{refer} + P_8 + P_9 + P_{10} + P_{11} \quad (4.16)$$

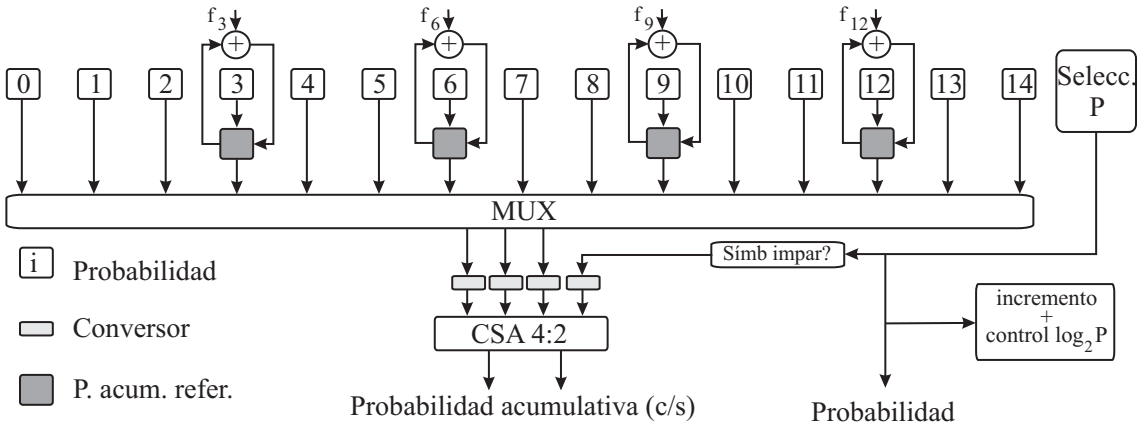


Figura 4.18: Modelo del codificador con dos símbolos por línea.

$$= S_3^{refer} + 2 \cdot P_{8,9} + 2 \cdot P_{10,11} \tag{4.17}$$

$$S_{13} = \sum_{j=0}^{j=12} P_j = S_3^{refer} + P_8 + P_9 + P_{10} + P_{11} + P_{12} \tag{4.18}$$

$$= S_3^{refer} + 2 \cdot P_{8,9} + 2 \cdot P_{10,11} + P_{12,13} \tag{4.19}$$

El control de la saturación, que es el control de la actualización de S_T, se realiza en el incrementador de la probabilidad. Se incrementa P_i y se convierte a su aproximación como potencia de 2, que llamaremos P_i''. Este valor se compara con P_i' y en caso de ser distintos se actualiza S_T, tal y como vemos en la figura 4.19. El mismo esquema se aplica a la actualización de las probabilidades acumulativas de referencia.

El modelo en el descodificador

En esta nueva arquitectura hay importantes simplificaciones como podemos apreciar en la figura 4.20. Ha desaparecido el divisor y la iteración en más sencilla (la iteración la veremos en la siguiente sección).

En el descodificador todas las probabilidades acumulativas deben ser calculadas para poder proceder con la comparación. Realmente se podría implementar un esquema en más de un nivel, ahorrando hardware, pero aumenta el tiempo necesario, y el hecho de que las comparaciones no puedan comenzar hasta que termine la operación del divisor supone que se acumula un retraso importante. Sin embargo, esta nueva arquitectura que proponemos no realiza multiplicaciones y por tanto no es necesario un divisor. Resulta entonces factible reducir el coste de las comparaciones aumentando el número de niveles de comparación.

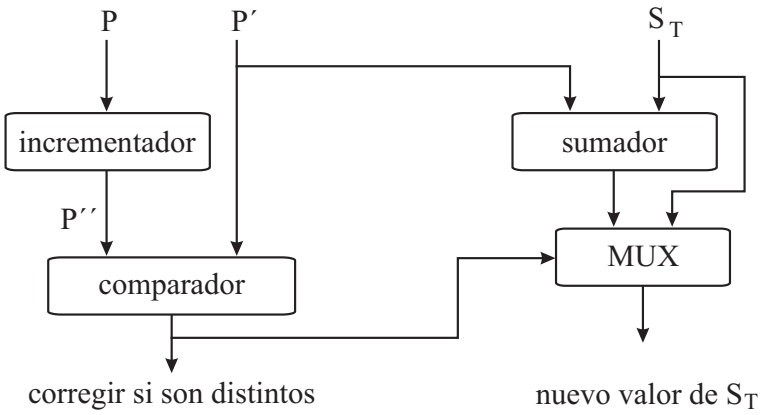


Figura 4.19: Control del crecimiento de las probabilidades cuando se aproximan por potencias enteras de 2.

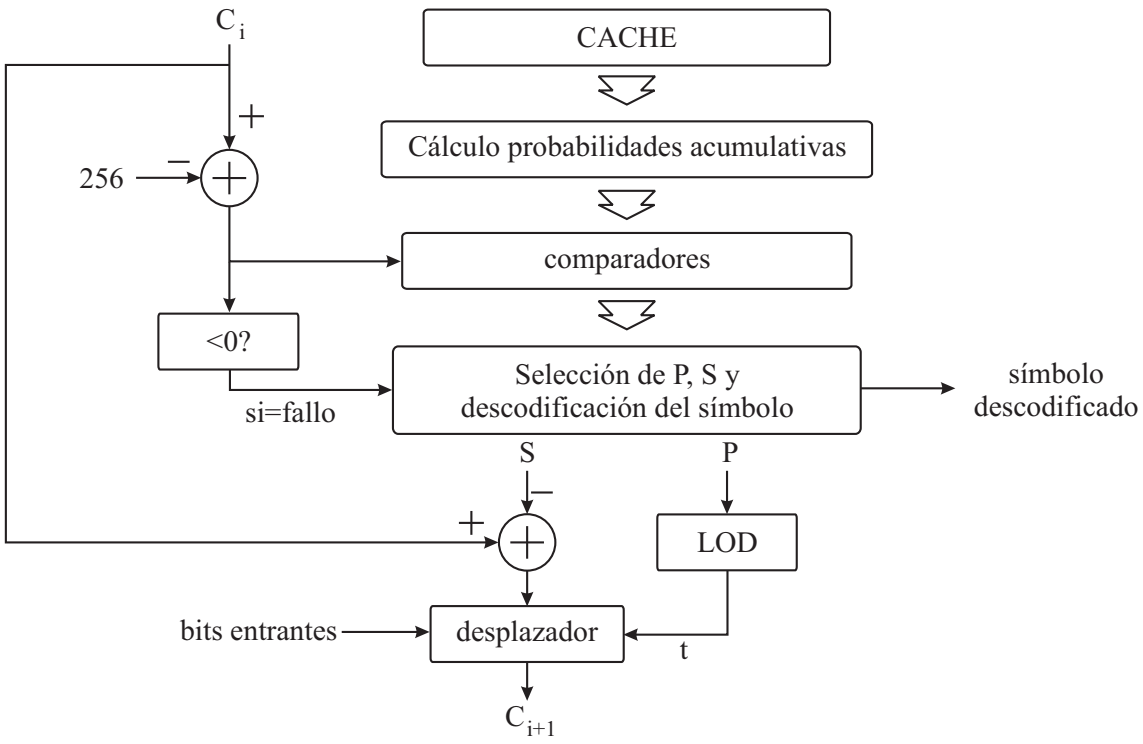


Figura 4.20: Descodificador sin multiplicaciones. El funcionamiento del modelo se puede acelerar gracias a la desaparación del divisor.

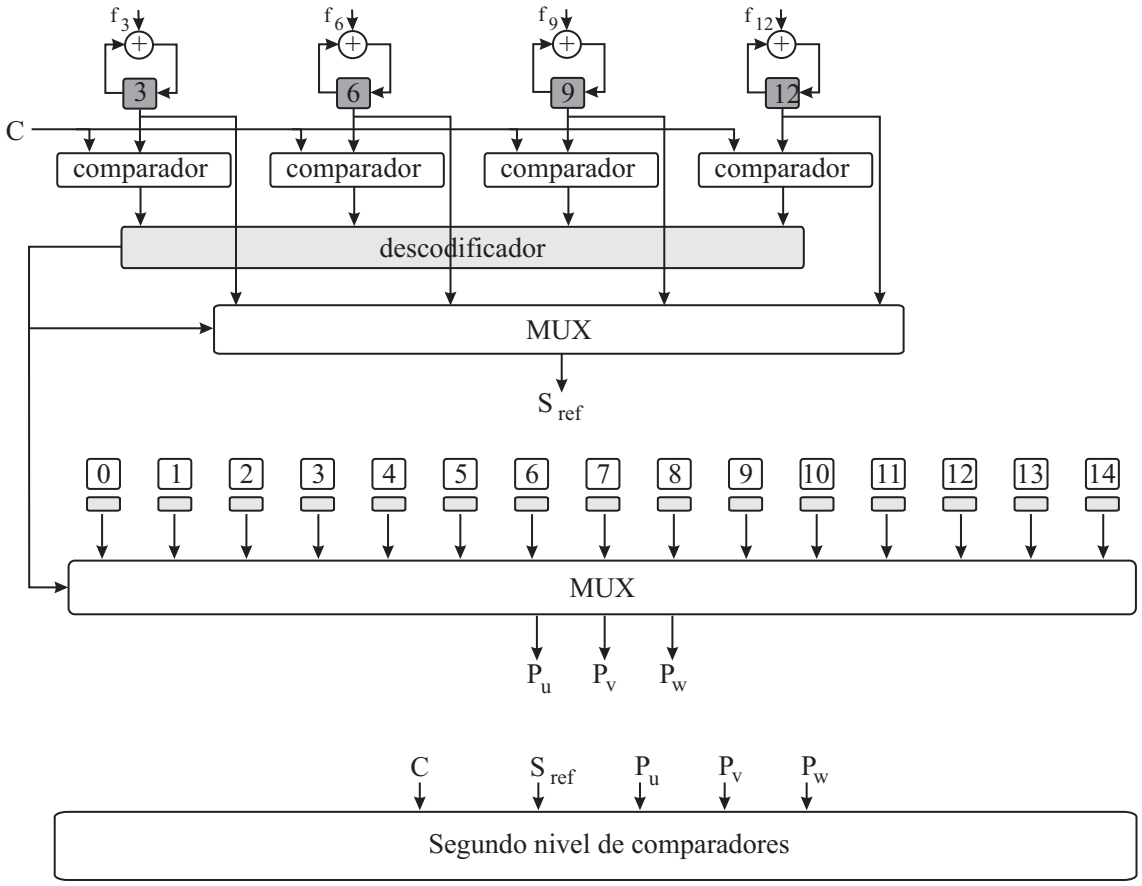


Figura 4.21: Modelo del descodificador con dos símbolos por línea.

En un primer nivel comparamos con las probabilidades acumulativas de referencia. Como resultado, las opciones posibles se reducen a tres líneas, o seis símbolos. El esquema resultante se muestra en la figura 4.21.

Iteración sobre el rango

Poco hay que decir al respecto. Ahora la actualización consiste simplemente en normalizar A_i, sumar C_i y $S_i(k)$ y normalizar también este resultado. En el descodificador la suma se convierte en una resta. La actualización del rango no era una parte crítica en las otras arquitecturas vistas y ahora todavía lo es menos. La verdadera importancia de eliminar las multiplicaciones reside en haber eliminado el divisor del descodificador. En la figura 4.22 mostramos el nuevo hardware de la iteración, que naturalmente es más rápido y también menos costoso.

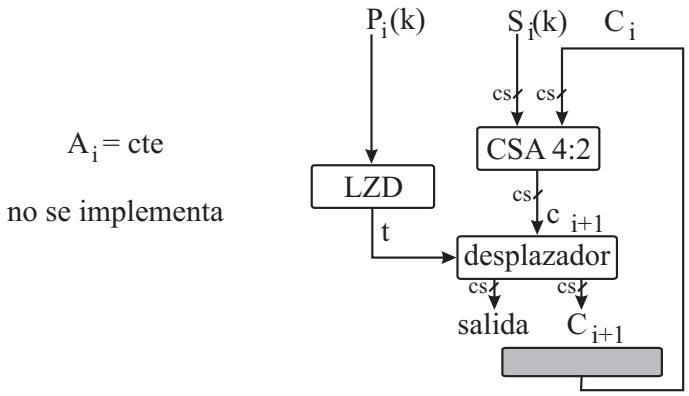


Figura 4.22: Actualización del rango sin multiplicaciones.

4.2.5 Evaluación

Desde el punto de vista de la eficiencia en la compresión hemos visto que el nuevo modelo supone un avance importante. Ahora debemos ratificar la conveniencia desde el punto de vista del área y tiempo de computación de la arquitectura asociada. El *timing* de las distintas tareas lo presentamos en la figuras 4.23 y 4.24 tal y como hemos hecho en los casos anteriores. También se incluyen los tiempos con segmentación.

El descodificador vuelve a ser el gran beneficiado. La reducción del ciclo es notable, debido principalmente a la eliminación de las multiplicaciones. Esta reducción es doble, por un lado la actualización del rango es más sencilla y por otro lado se ha eliminado la división. Por otra parte el codificador ve ligeramente aumentada la duración del ciclo. Esto no es un problema importante ya que esta parte estaba ya muy optimizada.

Las diferencias en consumo de área son favorables a esta nueva implementación. Presentamos la tabla 4.4 con los resultados obtenidos. Sólo mostramos los capítulos en los que hay diferencias. En términos generales diremos que la velocidad mejora apreciablemente en el descodificador y empeora ligeramente en el codificador, debido a que resulta más complicado gestionar dos probabilidades por línea. El área se reduce ligeramente en el codificador, y de forma significativa en el descodificador.

4.3 Codificador no adaptativo

Si bien hemos puesto énfasis desde un principio en la importancia de un modelo adaptativo, existen aplicaciones en las cuales la adaptabilidad no es necesaria. Ésta dificulta la implementación y además, en los casos en los que no es necesaria puede

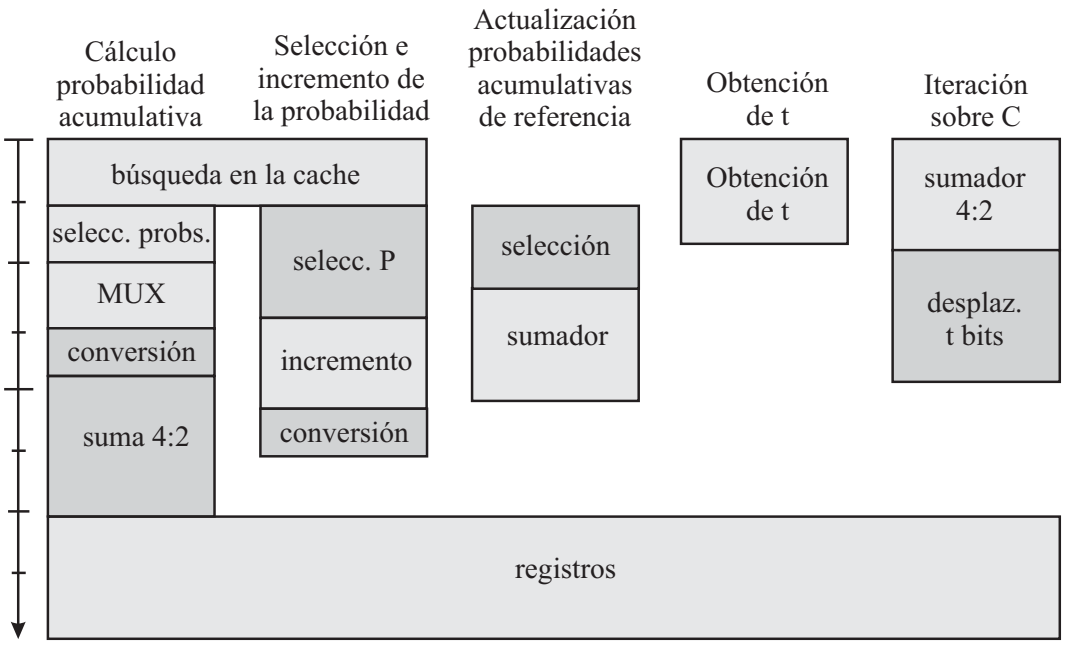


Figura 4.23: Diagrama de tiempos en el codificador. La duración del ciclo es de 30 t_{nand}.

suponer un empeoramiento de los resultados. Esto se debe a que el modelo adaptativo sufre perturbaciones debidas a los escalamientos y el rango disponible no está aprovechado en todo momento. En estos casos, podemos simplificar la arquitectura de forma importante, puesto que buena parte del hardware de nuestras arquitecturas está dedicado a gestionar el modelo.

En primer lugar, y dada la cantidad de modificaciones que ya hemos realizado respecto al modelo original, es necesario definir cuales son las características que ha de tener el nuevo modelo no adaptativo.

- Arquitectura sin memoria RAM. Si el modelo no es adaptativo estamos en el caso de un histograma con forma conocida.

- Mantendremos las multiplicaciones. La eliminación de las multiplicaciones fue un efecto secundario de una optimización que tenía por objeto reducir el número de escalamientos del modelo, pero ahora éstos ya no se producen.

- Quedan abiertas las opciones de introducir uno o dos símbolos por línea y de codificar los fallos en uno o dos ciclos. Estas posibilidades dependen de las características de los datos a comprimir.

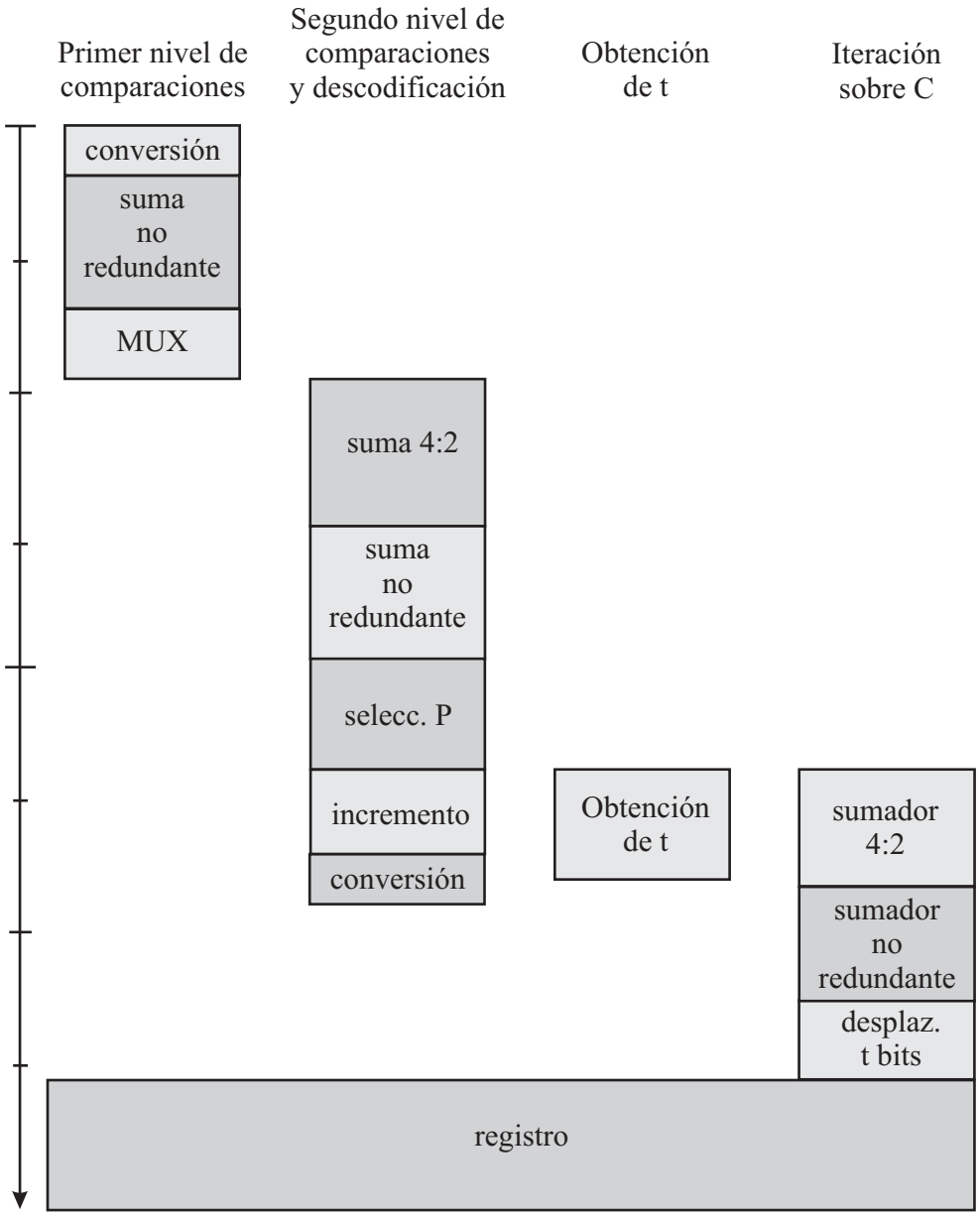


Figura 4.24: Diagrama de tiempos en el descodificador.La duración del ciclo es de 62 t_{nand}.

Codificador			
Concepto	Medida	Sin RAM	Nueva arquitectura
ciclo	t_{nand}	28	30
ciclo segm.	t_{nand}	24	30
Mutiplicadores	sumadores completos	40	0
Descodificador			
Concepto	Medida	Sin RAM	Nueva arquitectura
ciclo	t_{nand}	72	62
ciclo segm.	t_{nand}	52	52
divisor	sumadores completos	110	0
comparaciones	sumadores completos	350	150

Tabla 4.4: Resumen de mejoras obtenidas con la nueva configuración de la cache.

Así pues, se mantiene el hardware de actualización del rango en el codificador y descodificador, y en este último se mantiene también el divisor. Los cambios están en el modelo. No es necesario incrementar las probabilidades de los símbolos, ni actualizar las probabilidades acumulativas ni controlar las saturaciones del modelo. Ni siquiera es necesario almacenar las probabilidades ya que en lugar de registros se puede utilizar lógica cableada que es ocupa menos área. Esta última posibilidad sin embargo no es tan interesante como parece ya que no permite introducir nuevas tablas de datos, realizar codificación dependiente del contexto, etc, que serían posibilidades muy atractivas.

4.3.1 El modelo

Con el fin de minimizar el coste y el tiempo de computación, en este caso echaremos manos de una solución que hemos desechado en secciones anteriores: almacenar las probabilidades acumulativas en lugar de las probabilidades. Ya que el modelo no evoluciona, no es necesario realizar ninguna operación de actualización con ellas, así que es la solución perfecta para este caso.

El codificador

Para el codificador el modelo ahora se reduce a detectar en que línea se encuentra el símbolo pedido y utilizar un multiplexor y un sumador para obtener su probabilidad y su probabilidad acumulativa. Si se introducen dos símbolos en cada línea, el valor de la probabilidad será un medio del obtenido restando las probabilidades acumulati-

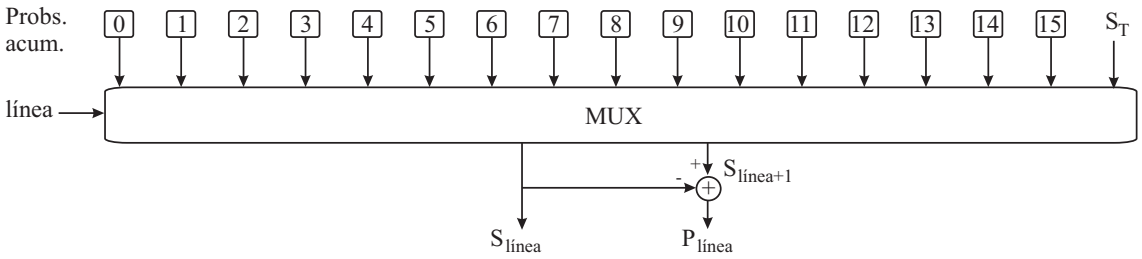


Figura 4.25: Modelo del codificador para el caso no adaptativo.

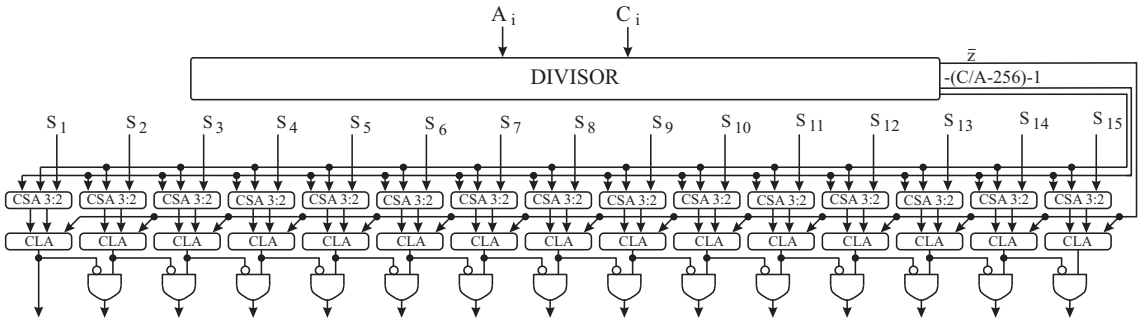


Figura 4.26: Modelo del descodificador para el caso no adaptativo.

vas adyacentes, y la probabilidad acumulativa se corregirá sumando la probabilidad en el caso de que el símbolo requerido no encabece su línea. En cualquier caso, es una operación rápida y sencilla. El esquema es el que se muestra en la figura 4.25.

El descodificador

En el descodificador existe la dificultad añadida de las comparaciones. En principio esto no tiene que afectar al modelo, ya que éste se limita a facilitar las probabilidades acumulativas y dejar que las comparaciones se realicen en uno o más niveles hasta descodificar el símbolo. En este momento se seleccionan una probabilidad y una probabilidad acumulativa.

El esquema utilizado es el que se muestra en la figura 4.26. Las diferencias con la figura 3.26 son notables. Las probabilidades acumulativas ya no necesitan ser calculadas. Además, están disponibles en formato no redundante y se suman a la salida del divisor con un sumador 3:2, en lugar de un 4:2 que es más costoso. Los comparadores, sin embargo, permanecen.

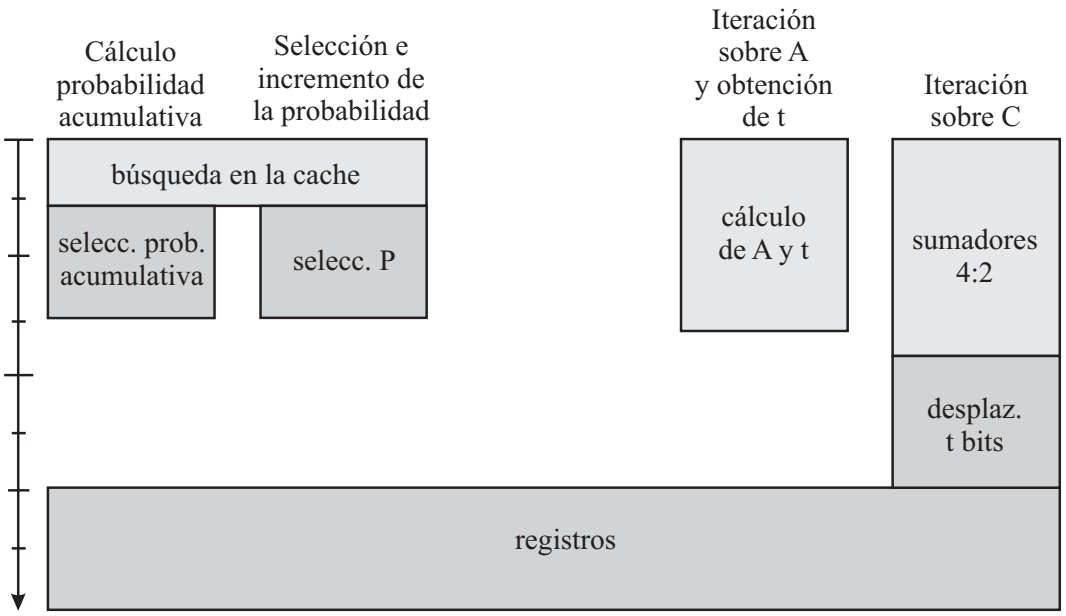


Figura 4.27: Diagrama de tiempos para el codificador no adaptativo. La duración del ciclo es de aproximadamente 28 t_{nand}.

4.3.2 Evaluación

Al ser pocas las diferencias de esta arquitectura con respecto a la mostrada en la sección 4.1, no existen cambios importantes en la arquitectura, si bien el coste del modelo desciende. Los resultados los resumimos en la tabla 4.5.

En el capítulo de velocidad de procesamiento el codificador no experimenta ningún cambio. El motivo es que si bien la gestión del modelo es trivial, la vía crítica está marcada por la iteración sobre el rango. Vemos esto en la figura 4.27, que podemos contrastar con la figura 4.9. La gestión del modelo ha quedado muy reducida y deja de ser un factor crítico.

El descodificador si experimenta una ligera mejora al simplificarse las comparaciones, pero no es importante. Lo podemos apreciar en la figura 4.28, donde los sumadores 4:2 utilizados en las comparaciones (figura 4.10) han sido sustituidos por sumadores 3:2.

Con respecto al área podemos prescindir de toda la cirtuitería relacionada con la actualización de las probabilidades, cálculo de probabilidades acumulativas, gestión de la saturación, mantenimiento de probabilidades acumulativas de referencia, etc. La simplificación es notoria si comparamos el número de columnas que aparecen en las figuras 4.27 y 4.28 con las anteriores 4.9 y 4.10.

Las diferencias no son muy grandes pero si significativas, especialmente en el

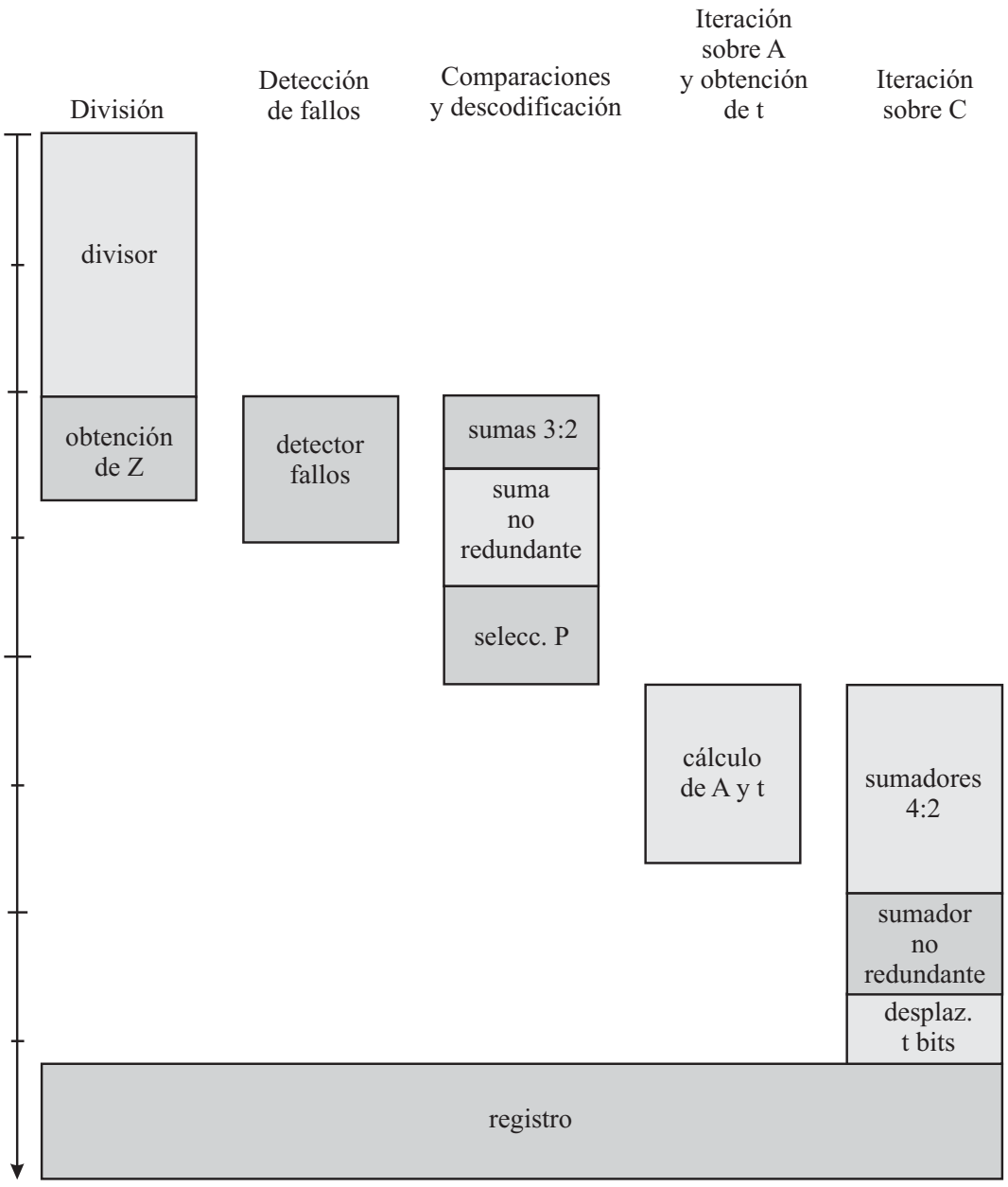


Figura 4.28: Diagrama de tiempos para el descodificador no adaptativo. La duración del ciclo es de aproximadamente 68 t_{nand}.

Codificador			
Concepto	Medida	Sin RAM	Modelo no adaptativo
ciclo	t_{nand}	28	28
registros	bits	168	160
modelo	sumadores completos	70	15
Descodificador			
Concepto	Medida	Sin RAM	Modelo no adaptativo
ciclo	t_{nand}	72	68
registros	bits	168	160
comparaciones	sumadores completos	350	300

Tabla 4.5: Resultados con un modelo no adaptativo.

número de sumadores que necesita el modelo. Fuera de este análisis queda, por ser difícilmente evaluable, la circuitería necesaria para gestionar las pequeñas tareas del modelo, como incrementadores de probabilidades, actualización y control de la saturación, etc. que contribuyen también a aumentar el área de la arquitectura adaptativa. Sin embargo, como ya hemos dicho, el análisis es muy complicado y su contribución nunca será comparable al coste de los registros de la cache o los comparadores del descodificador.

4.4 Conclusiones

Este capítulo ha estado dedicado a presentar arquitecturas que no almacenan el modelo completo. La idea de utilizar una memoria cache ha evolucionado dando lugar a arquitecturas en las cuales el protagonismo es de la cache y de la tabla virtual, desapareciendo la memoria principal. Las aplicaciones de estas arquitecturas son muchas ya que la mayor parte de los datos son susceptibles de ser trasformados (mediante predicción, codificación subbanda, wavelets, DCT, etc.), eliminando redundancia de tal forma que el histograma de los datos resultantes se adapte a nuestros modelos.

Hemos propuesto tres arquitecturas diferentes, cada una con características diferentes. La primera de ellas implementa el modelo sin RAM tal y como fue descrito en el capítulo 2. La segunda está orientada a mejorar la relación de compresión aumentando la capacidad de la cache y permite eliminar las multiplicaciones y el divisor. Por último la arquitectura con modelo no adaptativo es sencilla y eficiente cuando las probabilidades estadísticas de los datos no varían. En las siguientes tablas (4.6 y 4.7) resumimos las características principales de todas las arquitecturas.

Concepto	Medida	Sin cache	Con cache con RAM	Sin RAM	Sin productos	No adaptativa
ciclo	t_{nand}	41	37	28	30	28
RAM	Kbits	3	2	0	0	0
registros	bits	192	128	168	168	160
modelo	sumadores	192	99	70	70	15
iteración	sumadores	150	50	50	0	50

Tabla 4.6: Resumen de resultados para los codificadores.

Concepto	Medida	Sin cache	Con cache con RAM	Sin RAM	Sin productos	No adaptativa
ciclo	t_{nand}	141	77	72	62	68
RAM	Kbits	3	2	0	0	0
registros	bits	192	128	168	168	160
modelo	sumadores	1800	614	460	260	300
división	sumadores	960	110	110	0	110
iteración	sumadores	150	50	50	0	50

Tabla 4.7: Resumen de resultados para los descodificadores. En el apartado división, el coste de la arquitectura sin cache se refiere a los multiplicadores implementados para la descodificación.

Capítulo 5

Evaluación

Finalizamos esta memoria aplicando los algoritmos y arquitecturas desarrollados en los capítulos anteriores a la compresión de distintos tipos de datos. Si hasta ahora nos hemos centrado en la compresión de tipos de datos sencillos con el fin de no perder generalidad, procederemos ahora a comprimir datos de aplicaciones ampliamente utilizadas.

No pretendemos mejorar los resultados, ya que algunas de estas aplicaciones constituyen el estado del arte en su campo. Por lo contrario, presentamos nuestros codificadores como una alternativa rápida y flexible para compresión de distintos tipos de datos, con la ventaja adicional de permitir implementaciones rápidas y poco costosas como también veremos al final del capítulo.

Por otra parte, y a la vista de los resultados obtenidos en los capítulos 3 y 4, sabemos que nuestros codificadores superan en eficiencia a la codificación aritmética convencional, de manera que estamos en condiciones de asegurar que se implantarían con éxito en cualquier aplicación que utilice codificación ritmética multinivel.

5.1 Compresión de imágenes sin aplicar predicción

Las imágenes son, por naturaleza, datos con una alta redundancia espacial, en las que se puede aplicar con facilidad algoritmos de predicción. Son, así mismo, datos que contienen en muchos casos gran cantidad de información irrelevante.

Estos dos aspectos definen el campo de actuación de este tipo de compresión:

- Imágenes con alto grado de detalle en las cuales no se pueda aplicar compresión con pérdidas. Ejemplos típicos son las imágenes de radioscopía o microscopía electrónica que tienen una relación señal ruido muy pobre. Otros casos de

interés con imágenes con líneas definidas y textos, tales como esquemas y diagramas, en los cuales la pérdida de información provoca una degradación intolerable de la calidad (efecto Gibb).

- Dentro de la compresión sin pérdidas debemos centrarnos en imágenes en las que sea difícil aplicar predicción. Imágenes de este tipo son, por ejemplo, imágenes con paleta de colores reducida, en las cuales no existe ninguna relación entre el color de un pixel y el número que se le asigna dentro de la paleta.

Para este tipo de datos hemos desarrollado una arquitectura con jerarquía de memoria en dos niveles (capítulos 2 y 3). El modelo completo se almacena en una memoria RAM, y se dispone de una cache de 16 líneas para almacenar los símbolos que se estima tienen mayor probabilidad de ser referenciados. Esta estimación se realiza de forma natural sin seguir otro criterio que la lógica de los reemplazos en una cache de asignación directa. Es por ello un sistema de compresión que no necesita de más elementos que el propio codificador entrópico.

El tipo de imágenes al que es aplicable este sistema es reducido, ya que en la mayor parte de los casos existirá un sistema que ofrezca mejores resultados mediante predicción o codificación con pérdidas. El conjunto por el que hemos optado han sido diversas imágenes de escala de grises que son estándares en compresión de datos, versiones en color de algunas de estas imágenes e imágenes de cartografía.

El método de compresión con los que nos disponemos a comparar es el clásico utilizado para imágenes de color reducido, el denominado GIF [Bor95], creado por la compañía Compuserve. Este formato se basa en el algoritmo LZW [ZL77] patentado por Unisys. Además de su popularidad, el motivo principal por el que elegimos este algoritmo para las comparaciones es que no incluye ningún método elaborado para mejorar la compresión. Es decir, no se aplica ningún tipo de predicción, codificación en planos de bits, etc. Es un método sencillo con el que podemos comparar nuestros resultados sin tener que desarrollar un método de compresión de imágenes completo.

Los resultados los mostramos en la tabla 5.1. Para cada imagen mostramos el tamaño original en bytes, y los porcentajes de compresión alcanzados para cada una de ellas. Las tabla está dividida en dos partes. La primera parte comprende imágenes en escala de grises que son estándares en compresión de imágenes. La segunda parte comprende imágenes en color reducido (8 bits por pixel). Algunas de ellas son versiones en color de imágenes estándar, otras son imágenes topográficas que presentan interés como imágenes de calidad.

El balance es muy positivo a favor de nuestro codificador. En casi todos los casos obtenemos mejores relaciones de compresión. Resulta contraproducente intentar comprimir una imagen como *baboon*, ya que se llega a una expansión del archivo. En los casos en los que el algoritmo con cache sale perdiendo, las diferencias pueden

Nombre de la imagen	Tamaño original (bytes)	Número de semitonos	Compresión (%)	
			Cache	GIF
aerial0	262144	251	90.85	97.52
airfield	262144	176	97.11	102.13
airplane	262144	186	76.94	76.97
baboon	262144	194	103.99	112.01
barbara	417600	238	91.88	108.33
barbara2	417600	220	92.38	107.55
boats	417600	219	81.35	87.10
bridge	262144	64	90.19	65.91
camera	65536	247	82.18	83.39
couple	262144	196	89.42	94.60
crowd	262144	223	86.15	85.86
fchest	172800	254	78.10	84.20
frog	309258	102	63.64	52.62
goldhill	414720	220	88.96	95.93
lax	262144	225	98.94	107.74
lena	262144	180	83.14	95.32
man	262144	200	88.51	98.40
min21-6	77860	256	85.51	89.98
peppers	262144	196	81.91	97.48
teapot	115600	126	72.38	61.41
tiffany	262144	157	76.94	82.45
woman2	262144	218	76.30	82.18
xa1-1	262144	256	77.16	89.42
xa2-1	262144	178	73.57	85.88
zelda	417600	208	79.83	93.29
coupleColor	65536	231	52.54	35.27
lenaColor	262144	256	74.31	77.04
mapa_balcanes	3401808	216	89.13	96.05
mapa_oriente_medio	3221344	216	88.80	95.49
mapa_golfo_persico	3583080	216	89.26	96.65
mapa_mar_baltico	3584000	216	92.65	98.42
mapa_china	3738592	216	97.40	103.86
mapa_cantabrico	3350976	216	90.18	95.13
peppersColor	262144	256	66.89	62.00
mapa_crawford	256793	218	93.53	93.66

Tabla 5.1: Comparación de relaciones de compresión entre nuestro codificador con cache y el formato GIF. Menos es mejor.

llegar a ser altas. En las imágenes *bridge*, *frog*, *teapot* y *coupleColor* la desventaja es importante. El motivo es la naturaleza de estas imágenes, bien porque contienen pocos semitonos (*bridge* y *frog*) o bien porque contienen amplias áreas con colores planos.

En resumen, hemos comprobado que nuestro algoritmo supera al método LZW incorporado en el formato GIF cuando la imagen es compleja, con un elevado número de semitonos y sin grandes áreas de colores planos. Por contra, no es adecuado para comprimir diagramas, imágenes planas con textos ni imágenes en blanco y negro puro. Para este tipo de datos, sería necesario utilizar previamente un esquema de codificación basado en encontrar carreras de elementos repetidos o búsqueda de patrones antes de aplicar el codificador entrópico.

5.2 Compresión de imágenes monocromas sin pérdidas

Después del acercamiento a la compresión sin pérdidas de la sección anterior nos centraremos ahora en los algoritmos utilizados modernamente para este propósito. La compresión sin pérdidas es un campo de gran utilidad ya que si bien buena parte de las imágenes que consumimos (fotografía, televisión, etc.) admiten un elevado porcentaje de pérdidas sin distorsión aparente, desde el punto de vista de la producción y edición es importante contar con un formato eficiente de almacenamiento de imágenes sin perdidas.

Son muchos los formatos de compresión sin pérdidas aparecidos en los últimos años. Algunos buscan la compresión a cualquier precio, sin importar el coste, otros están orientados a implementaciones eficientes en términos de complejidad y compresión. Haremos un breve resumen de algunos de ellos y nos centraremos a continuación en los elegidos para compararlos con nuestros algoritmos.

5.2.1 Algoritmos de compresión sin pérdidas

La compresión de imágenes es un campo muy amplio [Jai81] en el que se distinguen claramente dos disciplinas, una es la conversión de las imágenes en un nuevo tipo de datos de los cuales se ha extraído la redundancia asociada a las características de localidad espacial de las imágenes y otra es la compresión de los datos resultantes. Nuestro trabajo está centrado en esta última parte, el último eslabón de la cadena.

Aunque resulta de vital importancia tener un codificador entrópico eficiente, la mayor parte de las mejoras en compresión provienen de la primera parte del proceso. La diferencia entre aplicar un sistema de predicción u otro puede llegar a ser considerable. Por otro lado, los datos resultantes de un determinado predictor pueden

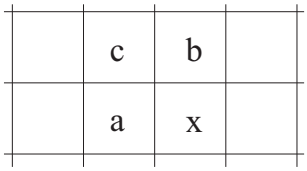


Figura 5.1: Vecinos de un pixel para un predictor. JPEG sin pérdidas.

Selección	Predicción
1	a
2	b
3	c
4	a+b-c
5	a+(b-c)/2
6	b+(a-c)/2
7	(a+b)/2

Tabla 5.2: Predictores utilizados en JPEG sin pérdidas.

definir cual es el codificador entrópico más adecuado para comprimirlos, ya que la distribución de datos puede adaptarse mejor a las características de un codificador u otro. Así, en una distribución cuyas probabilidades están muy próximas a potencias de 2 el método de Huffman será muy apropiado, cuando las diferencias entre probabilidades sean muy marcadas, de forma que algún símbolo genere códigos de menos de un bit por símbolo, la codificación aritmética será la mejor opción.

Los métodos de mayor impacto aparecidos en los último años ha sido la primera implementación del estándar JPEG [Gro94] sin pérdidas y los algoritmos FELICS [How93], CALIC [WM97] y LOCO [WSS96], el último de los cuales será, muy posiblemente, el elegido para el nuevo estandar JPEG 2000. De estos métodos nos interesan dos aspectos: el predictor y el compresor entrópico utilizados.

5.2.2 Compresión sin pérdidas en JPEG

El codificador sin pérdidas de JPEG [PM93] puede utilizar 7 predictores distintos. En la figura 5.1 mostramos la nomenclatura utilizada para referirnos a los pixeles del entorno del que está siendo procesado, x. Se utilizan los pixeles que llamamos a, b y c, y por el mismo nombre nos referiremos a los valores de luminancia de los mismos. Los 7 predictores los mostramos en la tabla 5.2.

La primera línea de cada imagen ha de ser codificada utilizando el predictor número 1. De los restantes, el número 7 es el más eficiente en términos generales.

Sin embargo, esto puede variar de forma importante de una imagen a otra.

JPEG es un estándar fomentado por ISO del cual existen múltiples implementaciones. Generalmente sólo se implementa la variedad denominada *baseline*, que no incluye la codificación sin pérdidas. El codificador que hemos utilizado es una versión realizada en las universidades de California y Cornell [HS94] y se distribuye gratuitamente y con libertad para ser modificado con la única condición de reconocer la autoría original del código. Es posible seleccionar el conjunto de predictores de los que se puede hacer uso, y la codificación entrópica se realiza utilizando un codificador Huffman, semiadaptativo o fijo.

5.2.3 El codificador FELICS

FELICS (*Fast, Efficient, Lossless Image Compression System*) es un método de compresión de imágenes sin pérdidas desarrollado por Paul G. Howard y Jeff S. Vitter [How93]. Es un método que consigue una eficiencia notable combinando un predictor sencillo y utilizando un codificador de Rice 1.2.2 como compresor entrópico.

El predictor difiere del visto para JPEG sin pérdidas en que no es lineal. Esta es una característica que se repite en otros predictores que veremos. La eficiencia de los predictores reside en la capacidad para detectar la tendencia de evolución de los pixeles de una imagen. Cuando esa tendencia se rompe, debido por ejemplo a un borde, la predicción lineal deja de ser adecuada y es preferible realizar otro tipo de estimación. Conservando la notación de la figura 5.1 para nombrar los pixeles, el predictor utilizado es el siguiente:

- Dados los vecinos a y b de un pixel x se calculan el máximo y el mínimo y la diferencia entre estos. $L = min(a, b)$, $H = max(a, b)$ y $\Delta = H - L$.
- Si el pixel x está comprendido entre H y L, $L \leq x \leq H$, entonces se emite un bit que indica que x se encuentra dentro del rango, y se codifica a continuación la diferencia, $x - L$.
- Si por el contrario, x está fuera del rango, se emite un bit para indicar este evento. A continuación se emite un nuevo bit que indica si x es mayor que H o bien menor que L. La diferencia, $L - x - 1$ o $x - H - 1$ según el caso, se codifica.
- En el caso en el que x esté dentro del rango la codificación se realiza sin comprimir, ya que al estar su valor acotado (conocemos Δ) se puede utilizar el número mínimo de bits para expresar su valor.
- Cuando x está fuera de rango la situación es diferente, el valor no está acotado y se utiliza un codificador de Rice.

La eficiencia del codificador de Rice depende enormemente de la adecuada elección del parámetro k (sección 1.2.2). Para estimar el valor óptimo se utiliza un esquema basado en contextos. Para cada contexto se calcula la longitud del código que se produciría para cada valor del parámetro k, y se escoge el que genere un código de menor longitud. El mantenimiento de los datos necesarios para calcular estas estimaciones tiene un cierto coste en memoria que en [How93] se estima vagamente en *unas pocas cuentas para unos pocos cientos de contextos.*

5.2.4 CALIC

CALIC [WM97] es el acrónimo de Context-based, Adaptive, Lossless Image Codec, realizado en las universidades de Western Ontario y Northern Illinois. Es uno de los métodos propuestos para el nuevo estándar JPEG 2000. Su eficiencia es muy alta aunque es un método relativamente lento lo que le hace perder terreno respecto a sus competidores.

Se utiliza un predictor no lineal, que además modela el error de la predicción bajo distintos contextos. Dado el error de la predicción, es realimentado obteniendo una mejor estimación para el contexto en curso. El número de pixeles involucrados en la predicción es alto, se recurre a los 7 vecinos más próximos, de forma que se detectan bordes de distinta intensidad en todas las direcciones. El número de contextos también es alto, más de 500, y se han escogido a partir de un complicado estudio de las características de las imágenes.

Contempla dos modos de operación, uno denominado binario para imágenes que contienen pocos semitonos, y el modo de tono continuo para imágenes en general. Aunque es compatible con cualquier codificador entrópico, la versión presentada utiliza codificación aritmética en los dos modos. En modo binario el alfabeto tiene 3 símbolos, y el modo de tono continuo el número de símbolos es variable.

5.2.5 LOCO-I

LOCO-I (LOw COmplexity LOssless COmpression for Images) [WSS96] ha sido desarrollado en los laboratorios de Hewlett-Packard en Palo Alto y propuesto a ISO como codificador sin pérdidas del futuro estándar JPEG 2000. Hoy por hoy es el candidato mejor situado para suceder al obsoleto codificador descrito en 5.2.2.

Se utiliza un predictor no lineal muy sencillo pero eficiente. El número de pixel implicados es de sólo 3. El resultado de la predicción es comprimido utilizando códigos de Rice con contextos. Para determinar el contexto adecuado se utilizan más pixeles del entorno hasta un total de 5. En la figura 5.2 mostramos los pixeles involucrados. El predictor es el siguiente:

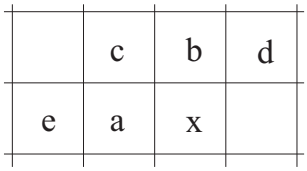


Figura 5.2: Vecinos de un pixel para un predictor. LOCO-I.

- Si $c \geq max(a, b)$ entonces se estima el valor de x como $min(a, b)$.
- En el caso en que $c \leq min(a, b)$ la estimación es $max(a, b)$.
- En los demás casos, la predicción es $a + b - c$.

5.2.6 Comparación

A continuación compararemos los métodos anteriores con nuestro codificador. En primer lugar evaluaremos los codificadores anteriores, a fin de elegir aquel que obtiene mejores resultados. En la tabla 5.3 mostramos los resultados para distintas imágenes expresados en porcentaje respecto al tamaño original de la imagen.

Los mejores resultados son los que se obtienen con CALIC, seguidos de cerca por LOCO-I. Los restantes métodos se separan de forma importante en muchos casos. En particular, JPEG demuestra su ineficiencia. Debemos decir que el codificador de JPEG que hemos utilizado admite muchos parámetros de configuración (en realidad un número excesivo dada la simplicidad del codificador) que nos ha llevado a tener que variarlos hasta encontrar los mejores resultados. Añadimos una columna para JPEG fijo, esto es, que utiliza un conjunto de tablas de códigos Huffman por defecto. La diferencia de compresión debida a la adaptabilidad se hace evidente. FELICS es un codificador que pierde terreno de forma importante respecto a los codificadores más modernos, CALIC y LOCO-I, que incorporan mejores predictores y estimación de contextos.

Ahora aplicaremos nuestro codificador aritmético a esquemas de compresión similares a los vistos. Nos centraremos en el predictor de JPEG y el de LOCO-I. Hemos elegido estos por ser los que con mayor facilidad se adaptan a las características de nuestro codificador. No ha sido posible, sin embargo, utilizar los predictores de FELICS ni CALIC. El predictor de FELICS introduce bits ad-hoc en la secuencia de salida para indicar eventos como *dentro o fuera del rango*, que no es fácilmente compatible con la codificación aritmética. En cuanto a CALIC, es un codificador complicado del que no se ha encontrado el código fuente.

Nombre de la imagen	Tamaño original (bytes)	JPEG	JPEG fijo	FELICS	CALIC	LOCO-I
aerial0	262144	68.27	74.83	67.68	62.94	64.41
airfield	262144	70.93	77.31	71.93	68.39	69.57
airplane	262144	74.79	83.61	49.36	44.24	45.09
baboon	262144	68.98	74.56	75.87	70.69	72.69
barbara	417600	69.56	75.28	65.05	54.90	58.31
barbara2	417600	68.59	73.63	64.92	56.75	58.62
boats	417600	67.70	74.32	54.45	47.72	48.89
bridge	262144	69.73	75.58	71.05	67.11	68.76
camera	65536	61.34	65.45	57.97	52.37	53.92
couple	262144	68.95	74.47	62.41	57.22	58.46
crowd	262144	68.41	75.12	54.88	47.03	48.93
fchest	172800	68.96	75.04	40.95	32.69	33.94
frog	309258	69.00	74.05	76.62	73.16	75.61
goldhill	414720	68.94	75.01	59.33	54.93	55.96
lax	262144	68.87	75.31	74.46	70.32	72.03
lena	262144	69.41	75.16	61.00	54.90	56.60
man	262144	69.41	75.40	59.77	54.49	56.31
min21-6	77860	66.57	71.13	59.81	53.82	51.53
peppers	262144	70.60	75.89	58.37	52.54	53.61
teapot	115600	65.75	69.07	47.72	36.92	40.92
tiffany	262144	72.27	81.50	52.26	47.40	48.86
woman2	262144	69.37	77.08	44.69	39.97	41.25
xa1-1	262144	70.32	78.86	56.85	52.27	53.89
xa2-1	262144	69.29	75.98	56.31	52.00	53.98
zelda	417600	69.20	75.43	52.02	46.66	48.35

Tabla 5.3: Comparación entre distintos métodos de compresión sin pérdidas. Menos es mejor.

En cualquiera de estos casos hemos implementado únicamente el predictor, no el resto del modelo. En LOCO-I existe un modelado de contextos adicional que permite mejorar la compresión pero no hemos entrado en él. Cuando se introduce codificación de orden superior el número de contextos se dispara y buena parte del trabajo de diseño pasa por definir que contextos se utilizan, como se agrupan varios para formar uno único, etc. Nosotros, dado que no estamos involucrados en el desarrollo de ninguno de estos codificadores, nos limitamos a aplicar nuestro codificador sobre los datos utilizando únicamente los predictores. En este sentido apostamos por una arquitectura sencilla que pueda ser introducida en cualquier desarrollo y adaptada en muy poco tiempo.

Los resultados para ambos predictores y dos codificadores distintos los mostramos en la tabla 5.4. El primer codificador es el descrito en la sección 4.1, y es la implementación básica sin RAM. El segundo codificador lo hemos visto en la sección 4.2, no utiliza RAM, almacena dos símbolos en cada línea, no utiliza multiplicaciones y codifica los fallos en dos ciclos.

Los mejores resultados son los del segundo codificador, como ya comprobamos para un predictor básico en el capítulo 4. Observando la tabla 5.4 con detenimiento podemos ver que las mayores diferencias son debidas al codificador, mientras que el predictor influye en un porcentaje alrededor del 2 o el 3% en la mayoría de los casos. Los buenos resultados obtenidos en el capítulo 4 frente a los codificadores sin cache y con precisión completa [WNC87] se reproducen con estos nuevos predictores tal y como vemos en la tabla 5.5.

Finalmente, compararemos los resultados obtenidos con el segundo codificador y el predictor de LOCO-I, con los datos de la tabla 5.3. Añadimos también resultados obtenidos tras aplicar múltiples contextos. Hemos escogido un caso sencillo, con sólo 5 contextos basados en el último error de predicción calculado. Los 5 contextos son los siguientes:

- error $\in [-2, 2]$
- error $\in [-6, 3]$
- error $\in [3, 6]$
- error ≥ 7
- error ≤ -7

El número de contextos es bajo y no utiliza más datos que los ya disponibles. Es posible mejorar el rendimiento utilizando pixeles del entorno que no están incluidos en el predictor. Esta solución es utilizada en CALIC y en LOCO-I pero no la hemos considerado. En la tabla 5.6 mostramos los resultados.

Nombre de la imagen	Codificador 1		Codificador 2	
	P. JPEG	P. LOCO-I	P. JPEG	P. LOCO-I
aerial0	80.00	78.53	75.23	72.43
airfield	87.15	89.41	76.16	76.85
airplane	58.53	56.74	53.87	52.20
baboon	91.27	91.44	80.12	80.12
barbara	76.55	73.09	71.34	67.85
barbara2	78.60	74.72	71.99	67.91
boats	64.44	60.93	58.61	55.16
bridge	83.27	74.95	76.86	68.43
camera	66.44	65.00	61.91	60.60
couple	74.91	69.26	67.10	62.32
crowd	66.62	62.87	61.39	58.00
fchest	48.79	44.07	44.89	38.82
frog	99.54	100.51	82.71	85.65
goldhill	70.20	68.23	62.12	60.44
lax	89.04	91.06	78.19	78.88
lena	70.15	69.11	63.94	62.62
man	70.26	68.56	64.14	62.88
min21-6	73.31	62.62	68.58	58.42
peppers	66.37	69.00	60.53	61.21
teapot	58.05	44.44	53.68	49.63
tiffany	60.27	60.16	55.65	54.86
woman2	52.77	51.90	47.90	47.60
xa1-1	66.04	68.55	57.60	59.53
xa2-1	64.98	67.90	56.69	59.02
zelda	59.06	59.03	53.69	53.05

Tabla 5.4: Relaciones de compresión para dos codificadores con cache aplicados a los predictores de JPEG y LOCO-I. El primer codificador es el básico sin memoria RAM, el segundo codifica los fallos en dos ciclos, almacena dos símbolos por línea y no utiliza multiplicaciones. Menos es mejor.

Nombre de la imagen	Predictor JPEG			Predictor LOCO		
	Cache	Sin cache	Witten	Cache	Sin cache	Witten
aerial0	75.23	78.82	71.74	72.43	76.65	69.50
airfield	76.16	79.65	72.24	76.85	79.93	72.50
airplane	53.87	58.42	51.05	52.20	56.72	49.25
baboon	80.12	82.29	74.91	80.12	82.45	75.05
barbara	71.34	74.52	67.26	67.85	71.59	64.30
barbara2	71.99	75.40	68.09	67.91	72.15	64.78
boats	58.61	63.23	55.94	55.16	60.35	52.99
bridge	76.86	73.61	66.25	68.43	60.81	53.39
camera	61.91	67.05	59.21	60.60	65.80	57.91
couple	67.10	71.79	64.34	62.32	67.50	60.01
crowd	61.39	65.51	58.05	58.00	62.13	54.65
fchest	44.89	49.52	41.86	38.82	44.28	36.54
frog	82.71	81.74	74.41	85.65	75.36	68.03
goldhill	62.12	67.28	60.03	60.44	65.67	58.30
lax	78.19	81.43	74.02	78.88	82.10	74.70
lena	63.94	68.83	61.36	62.62	67.80	60.33
man	64.14	68.88	61.41	62.88	67.71	60.22
min21-6	68.58	73.02	65.01	58.42	61.74	53.57
peppers	60.53	65.74	58.25	61.21	66.79	59.26
teapot	53.68	58.57	51.15	49.63	42.70	35.11
tiffany	55.65	60.64	53.17	54.86	60.41	52.91
woman2	47.90	52.98	45.51	47.60	52.25	44.70
xa1-1	57.60	62.38	55.22	59.53	63.43	56.24
xa2-1	56.69	61.65	54.46	59.02	62.95	55.74
zelda	53.69	59.34	51.97	53.05	58.94	51.53

Tabla 5.5: Nuestro comparador frente al codificador sin cache y al codificador de Witten con precisión completa. Los resultados con los predictores de JPEG y LOCO siguen la misma tónica vista en el capítulo 4.

Nombre de la imagen	FELICS	CALIC	JPEG	LOCO-I	Cache 1 contexto	Cache 5 contextos
aerial0	67.68	62.94	68.27	64.41	72.43	70.81
airfield	71.93	68.39	70.93	69.57	76.85	76.65
airplane	49.36	44.24	74.79	45.09	52.20	50.59
baboon	75.87	70.69	68.98	72.69	80.12	79.74
barbara	65.05	54.90	69.56	58.31	67.85	65.09
barbara2	64.92	56.75	68.59	58.62	67.91	66.07
boats	54.45	47.72	67.70	48.89	55.16	53.95
bridge	71.05	67.11	69.73	68.76	68.43	68.03
camera	57.97	52.37	61.34	53.92	60.60	59.76
couple	62.41	57.22	68.95	58.46	62.32	61.89
crowd	54.88	47.03	68.41	48.93	58.00	55.48
fchest	40.95	32.69	68.96	33.94	38.82	37.98
frog	76.62	73.16	69.00	75.61	85.65	85.78
goldhill	59.33	54.93	68.94	55.96	60.44	59.91
lax	74.46	70.32	68.87	72.03	78.88	78.40
lena	61.00	54.90	69.41	56.60	62.62	62.01
man	59.77	54.49	69.41	56.31	62.88	61.70
min21-6	59.81	53.82	66.57	51.53	58.42	58.15
peppers	58.37	52.54	70.60	53.61	61.21	59.49
teapot	47.72	36.92	65.75	40.92	49.63	47.22
tiffany	52.26	47.40	72.27	48.86	54.86	54.03
woman2	44.69	39.97	69.37	41.25	47.60	46.39
xa1-1	56.85	52.27	70.32	53.89	59.53	58.89
xa2-1	56.31	52.00	69.29	53.98	59.02	58.24
zelda	52.02	46.66	69.20	48.35	53.05	51.57
promedio	59.83	54.06	69.01	55.62	62.18	61.11

Tabla 5.6: Comparativa entre FELICS, JPEG sin pérdidas, CALIC y LOCO-I con nuestro codificador aplicado con 1 y 5 contextos.

De esta tabla podemos extraer varias conclusiones. En primer lugar, JPEG es un codificador obsoleto. No hemos mostrado de nuevo los resultados de nuestro codificador con el predictor de JPEG, pero volviendo al vista a la tabla 5.4 nos daremos cuenta de que nuestro codificador mejora a JPEG utilizando el mismo predictor. Por tanto se hace patente que nuestro codificador mejora al utilizado en JPEG, y las diferencias no se deben exclusivamente al predictor.

La introducción de contextos consigue una ligera mejora que nos permite acercarnos a los resultados de FELICS con un incremento de coste muy bajo. Sin embargo la compresión en LOCO-I y en CALIC es mucho más eficiente. Los motivos no deben ser el codificador entrópico, ya que LOCO-I tiene el mismo codificador de Rice que FELICS, salvo que la estimación del código más adecuado es distinta. Por otro lado, al codificador aritmético de CALIC no puede ser más eficiente que el de Witten, al que como sabemos nos hemos aproximado mucho (tabla 5.5 y sección 4.2). Por tanto consideramos que la diferencia de compresión se debe a la utilización de gran cantidad de contextos que se evalúan utilizando mucha información del entorno del pixel que se codifica. En cambio, nuestra utilización de contextos es mucho más sencilla, ya que no se hace uso de más información que la utilizada en la predicción.

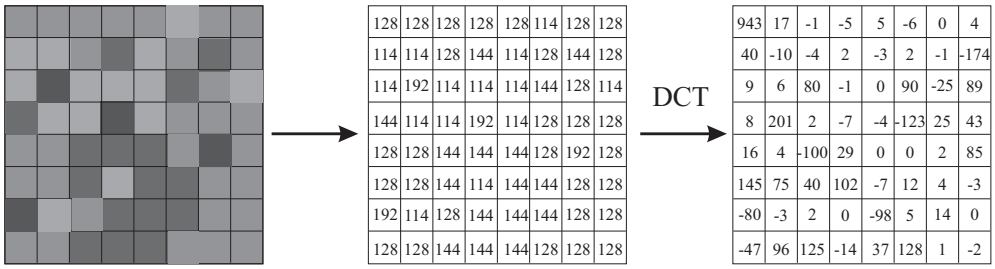
5.3 Compresión con pérdidas. Aplicación a JPEG

JPEG es el estándar actual de compresión de imágenes con pérdidas [Gro94, PM93, Wal91]. Ya hemos comentado que el estándar está siendo revisado y en breve aparecerá la nueva versión JPEG 2000. Esta nueva versión responde a los avances de nuevas técnicas de compresión que si bien mejoran a JPEG no están extendidas por no estar estandarizadas.

El algoritmo de compresión con pérdidas de JPEG se basa en la transformada coseno discreta (DCT). La imagen es dividida en bloques de $n \times n$ pixeles (figura 5.3.(a)) a los que se aplica la transformada, obteniendo tantos coeficientes como pixeles. Los coeficientes representan variaciones en distintas frecuencias y direcciones espaciales.

La imagen puede ser reconstituida a partir de los coeficientes, y la calidad subjetiva de la reconstrucción se ve escasamente afectada si los coeficientes se sustituyen por aproximaciones. Mediante el proceso denominado cuantización (figura 5.3.(b)), la precisión de estas aproximaciones puede ser reducida al nivel de calidad deseado [Jai81]. A menor precisión, menor calidad, pero mayor compresión, ya que el número de posibles valores de los coeficientes se reduce.

Debido a las características de la percepción visual humana, la importancia de los coeficientes decrece a medida que aumenta la frecuencia. Por ello es posible expresar con menor precisión los coeficientes asociados a frecuencias altas, y ob-



128	128	128	128	128	114	128	128
114	114	128	144	114	128	144	128
114	192	114	114	114	144	128	114
144	114	114	192	114	128	128	128
128	128	144	144	144	128	192	128
128	128	144	114	144	144	128	128
192	114	128	144	144	144	128	128
128	128	144	144	144	128	128	128

DCT

943	17	-1	-5	5	-6	0	4
40	-10	-4	2	-3	2	-1	-174
9	6	80	-1	0	90	-25	89
8	201	2	-7	-4	-123	25	43
16	4	-100	29	0	0	2	85
145	75	40	102	-7	12	4	-3
-80	-3	2	0	-98	5	14	0
-47	96	125	-14	37	128	1	-2

(a)

943	17	-1	-5	5	-6	0	4
40	-10	-4	2	-3	2	-1	-174
9	6	80	-1	0	90	-25	89
8	201	2	-7	-4	-123	25	43
16	4	-100	29	0	0	2	85
145	75	40	102	-7	12	4	-3
-80	-3	2	0	-98	5	14	0
-47	96	125	-14	37	128	1	-2

Cuantización

58	1	0	0	0	0	0	0
3	0	0	0	0	0	0	-3
0	0	5	0	0	1	0	1
0	13	0	0	0	1	0	0
0	0	2	0	0	0	2	1
5	0	0	1	0	0	0	0
-1	0	0	0	0	0	0	0
0	1	1	0	0	1	0	0

(b)

Figura 5.3: Aplicación de la DCT en JPEG con pérdidas.

tener así mejores relaciones de compresión. Codificando los coeficientes en forma de zig-zag (figura 5.3.(b)) la tendencia que se observa es que el valor absoluto de los coeficientes desciende de forma monotona (idealmente). Aparece entonces un fenómeno muy positivo para la compresión, y es existen largas carreras de 0's que pueden ser codificadas de forma especial, e incluso algunas llegan hasta el final del bloque con lo que el número de coeficientes a codificar se reduce de forma significativa utilizando un símbolo especial para este evento que se denomina EOB (fin de bloque).

La versión básica (baseline) del algoritmo trabaja con profundidades de color de 8 bits por pixel y componente. El tamaño de los bloques sobre los que se aplica la transformada es de 8×8, y los coeficientes resultantes son recodificados y comprimidos mediante un codificador de Huffman [Huf52].

Respecto a la recodificación [Wal91], la describiremos brevemente en estas líneas. En primer lugar, se codifican por separado los coeficientes de continua (DC, frecuencia 0) y los coeficientes de alterna (AC). Los coeficientes DC (uno por cada bloque) se codifican como la diferencia con respecto al último coeficiente procesado y se convierten en dos números. El primero representa el signo y el número de bits que ocupa el coeficiente, y el segundo el coeficiente en valor absoluto expresado con el número mínimo de bits. El primer número se comprime por Huffman, y el segundo se introduce tal cual en el código de salida.

Los coeficientes AC (63 por cada bloque de 8×8 pixeles), se comprimen de forma ligeramente distinta. Dada la gran cantidad de compresión que se puede alcanzar codificando de forma especial las carreras de ceros y los EOB, cada coeficiente no nulo se recodifica en dos números. El primero de ellos indica el número de bits necesarios para expresar el coeficiente así como el número de coeficientes nulos que lo preceden hasta un máximo de 15. El segundo número es el valor absoluto del coeficiente expresado con la mínima cantidad de bits. El EOB y las carreras de ceros de longitud mayor que 15 se codifican con sólo el primer número.

5.3.1 Una recodificación alternativa

La recodificación descrita en los párrafos anteriores es muy poco conveniente para nuestros propósitos ya que los histogramas resultantes no están lo suficientemente concentrados. Podemos apreciar esto en la figura 5.4. Por tanto optaremos por una representación distinta de los coeficientes para aplicar nuestro compresor basado en cache.

Buscando siempre la sencillez y tras realizar distintas pruebas nos hemos decantado por codificar los coeficientes sin recodificación alguna con la siguiente configuración del codificador.

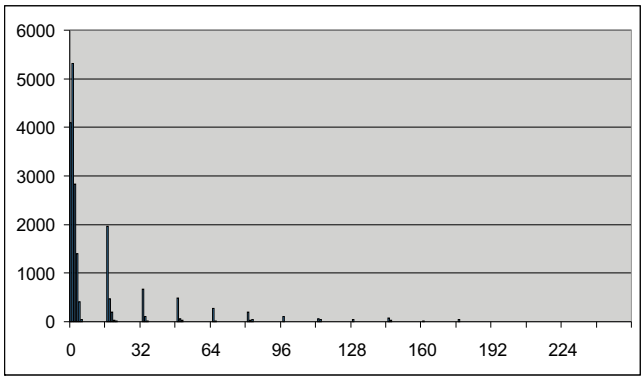


Figura 5.4: Número de apariciones de cada símbolo resultante de la recodificación de los coeficientes AC en JPEG.

- Utilizaremos un cache de 16 líneas, que almacena 2 símbolos por línea salvo las excepciones que comentaremos.
- Se utiliza el algoritmo sin reemplazos ni multiplicaciones, y los fallos se codifican en dos o tres ciclos.
- Se utilizan cuatro contextos. El primero para los coeficientes DC. Los tres restantes para los coeficientes AC, el primero para los 7 primeros coeficientes del bloque, el segundo para los 8 siguientes y el último para los restantes.
- El símbolo '0', comparte línea y probabilidad con el símbolo doble '0', para sacar partido de forma sencilla de las carreras de ceros existentes.
- Los símbolos (coeficientes) del '1' al '12' ocupan cada uno una línea que comparten con su respectivo valor negado, del '-1' al '-12'.
- El símbolo EOB ocupa una línea sin compartirla.
- Lo mismo sucede con dos símbolos utilizados para codificar los fallos. Uno de ellos codifica los primeros 8 fallos y el segundo los 64 siguientes, todos ellos en dos ciclos.
- Los fallos debidos a coeficientes con valores absolutos mayores se codifican en un ciclo adicional, utilizando símbolos adicionales incluidos entre los 64 de los que se habló en el punto anterior.

No se han codificado carreras de ceros de longitud mayor que 2 porque estas son escasas cuando la calidad de la imagen aumenta, de forma que podría resultar contraproducente. Los coeficientes almacenados en la cache se codifican igualando

las probabilidades de coeficientes con signos opuestos ya que la diferencia de probabilidades entre estos es usualmente pequeña, mucho menor que los resultantes de un predictor de los que ya hemos visto. Finalmente, y dado que los coeficientes pueden tomar valores entre -4095 y 4095, hemos optado por codificar los coeficientes de mayor valor absoluto en 3 ciclos. De esta forma se consideran todos los casos posibles y el tiempo de procesamiento no se incrementa demasiado ya que estos casos son poco probables.

5.3.2 Resultados obtenidos

El esquema de codificación descrito ha sido aplicado al conjunto de imágenes en escala de grises con las que hemos venido trabajando. Los resultados se muestran en la tablas 5.7 a 5.10. Corresponden a distintos niveles de calidad tal y como se indica al codificador del IJG, que ha sido el codificador utilizado.

De estos resultados destacaremos tres aspectos. En primer lugar, nuestra implementación ofrece, en general, relaciones de compresión similares pero inferiores. La tendencia que se aprecia es a empeorar a medida que se aumenta el nivel de calidad. Finalmente, los mejores resultados se obtienen para las imágenes fotográficas más comunes como *barbara*, *lena* o *boats* o imágenes médicas como *xa1-1* y *xa2-1*, mientras que los peores ocurren para imágenes de fotografía aérea como *aerial0*, *airfield* o *lax*, y para imágenes de tamaño reducido (*camera*) o con escaso número de semitonos (*camera*).

El hecho de que exista un comportamiento dispar según la naturaleza de la imagen y el nivel de calidad sugiere que el sistema de elección de contextos debiera ser sensible al tipo de imagen y al nivel de compresión aplicado. Efectivamente, distintas elecciones de los parámetros utilizados para aplicar la codificación por contextos consiguen mejorar la compresión de algunas imágenes en detrimento de otras y del promedio de todas ellas. La elección que hemos hecho ha sido la que mejores valores obtenía promediando sobre todas las imágenes y niveles de calidad.

La versión de JPEG con codificación aritmética binaria soluciona este problema definiendo todos los contextos posibles (varios centenares), de forma que se mejora de forma ostensible la relación de compresión al coste de aumentar la complejidad del codificador. En nuestro caso, la aplicación de múltiples contextos es contraproducente no sólo por su coste sino por el hecho de que el número de símbolos codificados en cada contexto puede llegar a ser tan bajo que no justifica un codificador adaptativo.

Diremos finalmente que uno de los motivos que hemos observado que influyen negativamente en las prestaciones de nuestro codificador es el hecho de que dependiendo del nivel de calidad el número de símbolos distintos procesados es muy bajo, hasta el punto de que la cache se hace demasiado grande. Se podría proponer usar

Nombre de la imagen	Tamaño original	JPEG (bytes)	Cache (bytes)	JPEG (%)	Cache (%)
aerial0	262144	28650	30405	10.93	11.60
airfield	262144	23925	25665	9.13	9.79
airplane	262144	13482	14017	5.14	5.35
baboon	262144	25862	27092	9.87	10.33
barbara	417600	30988	32960	7.42	7.89
barbara2	417600	30581	31996	7.32	7.66
boats	417600	22610	23052	5.41	5.52
bridge	262144	25922	27042	9.89	10.32
camera	65536	4373	4962	6.67	7.57
couple	262144	18466	19546	7.04	7.46
crowd	262144	18890	19970	7.21	7.62
fchest	172800	6578	6926	3.81	4.01
frog	309258	26736	27846	8.65	9.00
goldhill	414720	24631	25616	5.94	6.18
lax	262144	22545	24167	8.60	9.22
lena	262144	14999	16009	5.72	6.11
man	262144	17591	18778	6.71	7.16
min21-6	77860	6936	7768	8.91	9.98
peppers	262144	12040	12666	4.59	4.83
teapot	115600	6837	7401	5.91	6.40
tiffany	262144	10716	11143	4.09	4.25
woman2	262144	9097	9218	3.47	3.52
xa1-1	262144	8273	8266	3.16	3.15
xa2-1	262144	7311	7056	2.79	2.69
zelda	417600	16205	16392	3.88	3.93

Tabla 5.7: Relaciones de compresión para JPEG. Nivel de calidad 25.

Nombre de la imagen	Tamaño original	JPEG (bytes)	Cache (bytes)	JPEG (%)	Cache (%)
aerial0	262144	43776	46310	16.70	17.67
airfield	262144	38521	41323	14.69	15.76
airplane	262144	20359	21371	7.77	8.15
baboon	262144	41236	43670	15.73	16.66
barbara	417600	46681	49747	11.18	11.91
barbara2	417600	47029	49498	11.26	11.85
boats	417600	34083	35692	8.16	8.55
bridge	262144	40995	42886	15.64	16.36
camera	65536	6845	7604	10.44	11.60
couple	262144	28871	30508	11.01	11.64
crowd	262144	28486	30236	10.87	11.53
fchest	172800	9872	10758	5.71	6.23
frog	309258	46338	48521	14.98	15.69
goldhill	414720	39615	41587	9.55	10.03
lax	262144	37867	40657	14.45	15.51
lena	262144	23450	25450	8.95	9.71
man	262144	27783	29662	10.60	11.32
min21-6	77860	10377	11435	13.33	14.69
peppers	262144	18916	20476	7.22	7.81
teapot	115600	10097	10939	8.73	9.46
tiffany	262144	17226	18520	6.57	7.06
woman2	262144	14014	14911	5.35	5.69
xa1-1	262144	16252	17040	6.20	6.50
xa2-1	262144	14858	15512	5.67	5.92
zelda	417600	24953	26319	5.98	6.30

Tabla 5.8: Relaciones de compresión para JPEG. Nivel de calidad 50.

Nombre de la imagen	Tamaño original	JPEG (bytes)	Cache (bytes)	JPEG (%)	Cache (%)
aerial0	262144	64963	68267	24.78	26.04
airfield	262144	59810	63459	22.82	24.21
airplane	262144	30676	32505	11.70	12.40
baboon	262144	62807	65346	23.96	24.93
barbara	417600	68209	72678	16.33	17.40
barbara2	417600	71522	75961	17.13	18.19
boats	417600	50657	53692	12.13	12.86
bridge	262144	62601	64798	23.88	24.72
camera	65536	10395	11594	15.86	17.69
couple	262144	43503	46051	16.60	17.57
crowd	262144	41540	43773	15.85	16.70
fchest	172800	14728	16079	8.52	9.30
frog	309258	81995	86775	26.51	28.06
goldhill	414720	60657	63798	14.63	15.38
lax	262144	60967	64594	23.26	24.64
lena	262144	36781	40127	14.03	15.31
man	262144	42586	45607	16.25	17.40
min21-6	77860	15219	16631	19.55	21.36
peppers	262144	30223	32968	11.53	12.58
teapot	115600	14650	15799	12.67	13.67
tiffany	262144	28059	30476	10.70	11.63
woman2	262144	21913	23684	8.36	9.03
xa1-1	262144	28470	29516	10.86	11.26
xa2-1	262144	26636	27566	10.16	10.52
zelda	417600	39294	42082	9.41	10.08

Tabla 5.9: Relaciones de compresión para JPEG. Nivel de calidad 75.

Nombre de la imagen	Tamaño original	JPEG (bytes)	Cache (bytes)	JPEG (%)	Cache (%)
aerial0	262144	106243	113992	40.53	43.48
airfield	262144	105268	110300	40.16	42.08
airplane	262144	53321	57205	20.34	21.82
baboon	262144	105548	108518	40.26	41.40
barbara	417600	112872	120744	27.03	28.91
barbara2	417600	121349	128331	29.06	30.73
boats	417600	87908	93793	21.05	22.46
bridge	262144	104520	107025	39.87	40.83
camera	65536	17649	19441	26.93	29.66
couple	262144	74587	78718	28.45	30.03
crowd	262144	68743	73114	26.22	27.89
fchest	172800	25378	27547	14.69	15.94
frog	309258	138832	137100	44.89	44.33
goldhill	414720	107666	112310	25.96	27.08
lax	262144	106374	110921	40.58	42.31
lena	262144	66907	72358	25.52	27.60
man	262144	73151	77159	27.90	29.43
min21-6	77860	25186	27562	32.35	35.40
peppers	262144	59214	64772	22.59	24.71
teapot	115600	24321	26561	21.04	22.98
tiffany	262144	52482	56570	20.02	21.58
woman2	262144	40449	43432	15.43	16.57
xa1-1	262144	58740	62042	22.41	23.67
xa2-1	262144	56786	60018	21.66	22.90
zelda	417600	76376	81786	18.29	19.58

Tabla 5.10: Relaciones de compresión para JPEG. Nivel de calidad 90.

de forma selectiva, en función del estado de la codificación, caches más pequeñas para estos casos, si bien esto se aparta del enfoque hardware que hemos querido dar en todo momento a nuestro trabajo. Insistimos en la filosofía de nuestro codificador como un compresor de propósito general y bajo coste que puede ser aplicado a cualquier tarea con un esfuerzo de adaptación mínimo.

5.4 Comparación de coste con otras arquitecturas

Evaluaremos ahora la arquitectura descrita en la sección 4.2 y que hemos utilizado a lo largo de este capítulo. La compararemos con otras presentadas recientemente en la literatura. Por un lado consideraremos el codificador aritmético multinivel presentado en [HW98] que utiliza un buffer para almacenar la historia reciente del modelo y que describimos en la sección 1.3.7. Por otro, consideraremos una arquitectura de alto rendimiento para la codificación y descodificación Huffman [WM95]. El método de Huffman ha sido el más implementado en codificación entrópica debido a su sencillez. Veremos como nuestra arquitectura tiene un coste similar con la ventaja de ser adaptativa.

La arquitectura descrita en [HW98] está basada en un buffer (cola fifo) que contiene la historia reciente del modelo. Los símbolos contenidos en el buffer ven ponderada su probabilidad. El esquema utiliza memoria RAM para almacenar el alfabeto, pero con la ventaja de que las probabilidades se almacenan con pocos bits (entre 5 y 7). La iteración no utiliza multiplicaciones, pero el tamaño de palabra utilizado es de 16 bits, complicando las sumas y las operaciones de renormalización. Tiene la ventaja de no necesitar reescalado de probabilidades ya que se incrementa la probabilidad de los símbolos que entran en el buffer pero se decrementa la de los símbolos que son expulsados. En contrapartida son necesarios dos ciclos para procesar cada símbolo. Esta arquitectura utiliza ciclos de lectura-modificación-escritura en el acceso a memoria, que son aproximadamente un 30% más largos que los ciclos sencillos [Str92].

La arquitectura para la descodificación Huffman presentada en [WM95] simplifica la búsqueda del símbolo al introducir una comparación en dos niveles. En el primero se obtiene la longitud del código, y en el segundo se obtiene el código en sí. Tiene la ventaja adicional de ser fácilmente segmentable. Las arquitecturas tradicionales necesitaban costosas búsquedas a lo largo de grandes tablas de códigos hasta dar con el correcto, lo que representaba una gran desventaja. La complejidad de un codificador Huffman es muy pequeña, limitándose a un registro de desplazamiento y memoria.

Compararemos codificadores y descodificadores prestando atención a los siguientes aspectos: tamaño de la memoria, número de registros, duración del ciclo y th-

	RAM (bits)	Registros (bits)	Modelo (sumad.)	Ciclo (t_{nand})	Frecuen. (MHz)	(ciclos/ símbolo)
Cache	–	168	70	30	23	≈ 1.1
Buffer [HW98]	$256 \cdot 5$	200	80	40	17	2
Huffman [WM95]	$256 \cdot 20$	–	–	20	34	1

Tabla 5.11: Comparación entre codificadores

	RAM (bits)	Registros (bits)	Modelo (sumad.)	Ciclo (t_{nand})	Frecuen. (MHz)	(ciclos/ símbolo)
Cache	–	168	260	62	11	≈ 1.1
Buffer [HW98]	$256 \cdot 5$	200	≈ 600	50	14	2
Huffman [WM95]	$272 \cdot 8$	–	100	50	14	1

Tabla 5.12: Comparación entre descodificadores

roughput (símbolos procesados por ciclo). La estimación del tiempo de computación es siempre difícil, y más cuando no se dispone de una descripción completa de la arquitectura. Para este análisis hemos optado por una descripción cualitativa utilizando. Los resultados, que a continuación comentaremos, se muestran en las tablas 5.11 y 5.12.

En primer lugar comentaremos la forma en que se han hecho las estimaciones de coste. Los datos de nuestra arquitectura han sido obtenidos en el capítulo 4 y son bastante precisos, pero los de las restantes arquitecturas hemos debido estimarlos. Los datos de memoria RAM son exactos, ya que se mencionan en los trabajos originales. Para [HW98] el número de registros se ha calculado sumando los del modelo y los utilizados para el buffer.

El número de sumadores utilizados en el modelo es una estimación. Para [HW98] se ha contabilizado a partir de la arquitectura vista en la sección 1.4, aplicando a continuación la proporción entre los tamaños de palabra utilizados. Para [WM95] se han despreciado la mayor parte de ellos como tampoco se contabilizan los utilizados en la iteración de las arquitecturas de codificación aritmética. En el descodificador si se han contabilizado los utilizados en las comparaciones, que son una cantidad relativamente pequeña.

Los tiempos de computación están calculados de forma aproximada, tomando como referencia el tiempo de acceso a memoria RAM, que es un componente que forma parte de las dos arquitecturas. Respecto a la latencia, en [HW98] se necesitan dos ciclos por símbolo y en [WM95] sólo 1. Nuestra arquitectura necesita aproximadamente 1.1 ciclos por símbolo, ya que hemos estimado en un 10% el número de ciclos extra debidos a fallos.

	Codificador		Descodificador	
	Área (sumadores)	Tiempo (t_{nand})	Área (sumadores)	Tiempo (t_{nand})
Cache	238	33	428	68
Buffer [HW98]	493	80	≈ 1000	100
Huffman [WM95]	853	20	463	50

Tabla 5.13: Consumo de área y tiempo en las tres arquitecturas.

La comparación entre áreas ocupadas por las arquitecturas es favorable a nuestra arquitectura en cuanto a consumo de área ya que desaparece la memoria RAM. Una memoria RAM tiene un coste en área que es aproximadamente 6 veces inferior al de su equivalente en registros, así que podemos hacer una estimación unitaria del área sumando la memoria RAM, los registros, y los sumadores. También calculamos el tiempo necesario para procesar un símbolo. Estos valores los resumimos en la tabla 5.13.

Podemos comprobar que nuestra arquitectura es la menos costosa de las consideradas, y la de Huffman [WM95] la más rápida. En conjunto consideramos que nuestra arquitectura es la más ventajosa ya que reúne tres características muy importantes: la codificación es adaptativa, es la menos costosa y la velocidad es buena. La desventaja de la de Huffman reside en no ser adaptativa y que el codificador es excesivamente costosa. Hacemos notar que el modelo de ésta se implementa utilizando memorias RAM en lugar de PLAs, que serían menos costosas, ya que se pretende que se puedan cargar diferentes tablas según la aplicación.

5.5 Conclusiones

En este capítulo hemos dado el paso de integrar nuestras arquitecturas en aplicaciones ampliamente extendidas. En primer lugar propusimos nuestro codificador dos niveles de jerarquía para comprimir imágenes sin aplicar predicción. En este caso dedujimos que este codificador sería adecuado para comprimir imágenes de calidad y propusimos utilizarlo en imágenes con mapas de color reducido.

Consideramos a continuación la compresión de imágenes sin pérdidas. Introdujimos varios compresores, alguno de los cuales se erigirá en estándar en breve. El resultado fue que nuestro codificador sin memoria RAM ni multiplicaciones puede obtener resultados próximos a los de algunos de los mejores codificadores. Si bien la diferencia existe, hacemos énfasis en el hecho de que hemos conseguido estos resultados con escaso esfuerzo en la configuración de los contextos.

Resultados similares hemos obtenido al aplicar este mismo codificador al estándar

de compresión con pérdidas JPEG. Si bien no hemos logrado más que acercarnos a sus relaciones de compresión, la metodología utilizada fue, nuevamente, muy elemental.

Finalmente, hemos escogido dos de las arquitecturas más interesantes aparecidas recientemente en la literatura. Una de ellas implementa, como la nuestra, codificación aritmética multinivel, y la otra es posiblemente la arquitectura más rápida y eficiente para la codificación Huffman. En una comparación con nuestra arquitectura hemos llegado a excelentes resultados, ya que no sólo es la menos costosa, sino que tiene una velocidad muy competitiva. La otra arquitectura para codificación aritmética resultó, en cambio, excesivamente costosa y lenta.

Conclusiones y principales aportaciones

La codificación aritmética constituye el estado del arte en compresión entrópica. Por ello, es de gran interés en todo tipo de aplicaciones de compresión en las cuales alcanzar la mayor cantidad de compresión sea la principal motivación. Por otro lado, su implementación es compleja y costosa, particularmente en hardware, de forma que el rango de aplicación se ha visto siempre muy limitado. En los últimos años sólo la versión binivel ha gozado de popularidad entre los desarrolladores, mientras que las versiones multinivel aparecen con escasa frecuencia en la literatura y habitualmente se obvia la descodificación, la parte más compleja del algoritmo.

El objetivo de esta memoria ha sido desarrollar nuevos algoritmos y arquitecturas para la codificación aritmética adaptativa para alfabetos multinivel que redujesen la complejidad de implementaciones anteriores. Los tres puntos básicos que se han tenido presentes en todo momento han sido: aumentar la velocidad, disminuir el área en las arquitecturas y mantener o mejorar la relación de compresión. Estos tres objetivos han sido cumplidos a lo largo de este trabajo.

Hemos propuesto una idea novedosa, modificar el modelo de probabilidades utilizando una memoria con dos niveles de jerarquía. El nuevo nivel, que funciona como una memoria cache, contiene la información más importante para el codificador, y a él se restringen todas las operaciones de gestión del modelo. Cuando, al utilizar un predictor, se prescinde de la memoria principal, la cache pasa a ser el único nivel de memoria junto con la tabla virtual, con la consiguiente economía de área.

La implementación básica del modelo con memoria cache soluciona la mayor parte de los inconvenientes de la codificación aritmética adaptativa para alfabetos multinivel. En primer lugar, el cálculo y actualización de las probabilidades acumulativas se ha simplificado en gran medida. Ya sólo es necesario utilizar las probabilidades almacenadas en la memoria cache, y la utilización de aritmética redundante acelera las operaciones. En el descodificador el beneficio es aún mayor, ya que también las comparaciones se reducen al contenido de la cache. Siendo la

descodificación la parte más costosa de la codificación aritmética, esta simplificación tiene un gran impacto en las prestaciones de la arquitectura.

Se ha establecido la precisión necesaria para mantener buenas relaciones de compresión y disminuir de forma sustancial el coste de las operaciones aritméticas. Tanto el tamaño de palabra utilizado como la precisión en las multiplicaciones han sido estudiadas para llegar a balancear eficiencia y coste. Estas simplificaciones han influído notablemente en el descodificador, al permitir que los multiplicaciones previas a las comparaciones hayan sido sustituídas por un sencillo divisor.

Se ha solucionado además un importante inconveniente de los modelos adaptativos, la corrección de las probabilidades cuando estas crecen por encima de la precisión manejable. La forma habitual de realizar esta operación pasa por detener la codificación y proceder al escalado de todas las probabilidades. Nosotros hemos demostrado que esto no es necesario en nuestro modelo ya que el contenido de la cache se puede escalar sin detener la operación, y esto es suficiente para corregir el modelo sin que la compresión se vea afectada negativamente.

Cuando los datos tienen un histograma con perfil conocido, es posible llegar a mayores simplificaciones. Esto resulta de gran importancia ya que las aplicaciones más importantes trabajan con este tipo de datos. En este caso es posible prescindir de la memoria principal, mejorando al mismo tiempo la relación de compresión. Las implicaciones de eliminar la memoria RAM son aumentar la velocidad, disminuir el área y simplificar el mantenimiento del modelo.

Esta arquitectura admite otras mejoras. Hemos llegado así a una arquitectura que aumenta la capacidad efectiva de la cache sin aumentar su tamaño. El nuevo modelo resultante ha sido gestionado de forma diferente dando como resultado que resulta conveniente eliminar las multiplicaciones de la iteración. Ello conlleva no sólo la simplificación de la iteración en el codificador y el descodificador, sino la eliminación del divisor en el descodificador. Finalmente, se ha mejorado la relación de compresión codificando los fallos de la cache en dos ciclos. La eficiencia de la compresión es, entonces, comparable a la obtenida con algoritmos con precisión completa implementados en software.

Por último, hemos hecho sencillas aproximaciones a la compresión de distintos tipos de imágenes, comparando nuestros codificadores con métodos ampliamente utilizados. Los resultados han sido en general inferiores, pero hacemos énfasis en el hecho de que se han utilizado configuraciones muy sencillas que han respondido de forma aceptable a problemas muy diferentes. En cuanto al punto de vista de la implementación hardware, nuestras arquitecturas mejoran ampliamente a otras aparecidas recientemente en la literatura.

Si bien la codificación aritmética ha sido vista siempre como la forma más eficiente pero también más costosa de comprimir datos, nosotros proponemos otra lectura. A la vista de las importantes simplificaciones y mejoras en todos los aspectos, que

hemos introducido, la codificación aritmética puede ser vista como un método de compresión relativamente rápido y poco costoso, apreciándose entonces la que siempre ha sido una de sus virtudes: la facilidad para comprimir todo tipo de datos, tal y como hemos demostrado, gracias a la separación existente entre el modelo y la codificación.

Bibliografía

[Abr63] Abramson. *Information theory and coding.* McGraw-Hill, New York, 1963.

[AHH88] A. Agarwal, J. Hennessy y M. Horowitz. Cache performance of operating systems and multiprogramming. *ACM Trans. Computer Systems*, 6(4):393–431, Noviembre 1988.

[ATL+88] R. B. Arps, T. K. Truong, D. J. Lu, R. C. Pasco y T. D. Friedman. A multi-purpose vlsi chip for adaptative data compression of bilevel images. *IBM Journal of Research and Development*, 32(6):775–795, 1988.

[Avi61] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Trans. on Elect. Comput.*, EC-10:389–400, 1961.

[BBL97] M. Bóo, J. D. Bruguera y T. Lang. A VLSI architecture for arithmetic coding of multi-level images. *IEEE Transactions on Circuits and Systems-II: Analog and Digital Signal Processing.*, 45(1):163–168, 1997.

[BL] J. D. Bruguera y T. Lang. Leading–one prediction with concurrent position correction. *IEEE Transactions on Computers.* Pendiente de publicación.

[Bor95] G. Born. *The file format handbook.* International Thomson Computer Press, 1995. ISBN 1-850-32128-0.

[BRM88] M. A. Bassiouni, N. Ranganathan y A. Mukherjee. Software and hardware enhancement of arithmetic coding. En *International Working Conference SSDBM Statical and Scientific Database Management*, páginas 120–132, 1988.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan y V. K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, Abril 1986.

[CKW91] D. Chevion, E. D. Karnin y E. Walach. High efficiency multiplication free approximation of arithmetic coding. En *Data Compression Conference*, páginas 43–52, 1991.

[CY91] Y Chen y Y. Yasuda. Highly efficient entropy coding of multi-level images using a modified arithmetic code. En *Visual communications and image processing*, 1991.

[Eli75] P. Elias. Universal codeword sets and representation of the integers. *IEEE Trans. on Inf. Theory*, 21(2):194–203, Marzo 1975.

[Fal73] N. Faller. An adaptive system for data compression. En *Record of the 7th Asilomar Conference on Circuits, Systems and Computers*, páginas 593–597, Noviembre 1973.

[Fan49] R. M. Fano. *Transmission of Information.* M.I.T. Press, Cambridge, Mass., 1949.

[Fen95] P. Fenwick. A new data structure for cumulative probability tables : an improved frequency-to-symbol algorithm. Informe Técnico 110, University of Auckland, Department of Computer Science, Febrero 1995.

[FP95] B. Fu y K. K. Parhi. Two vlsi design advances in arithmetic coding. En *Proc. Int. Symp. Circuits Syst. (ISCAS'95)*, páginas 1440–1443, 1995.

[Gal78] R. G. Gallager. Variations on a them by huffman. *IEEE Trans. on Inf. Theory*, 24(6):668–674, Noviembre 1978.

[Ger79] A. Gersho. Asymptotically optimal block quantization. *IEEE Trans. on Information Theory*, 4(IT-25), Julio 1979.

[Gol66] S. W. Golomb. Run-length encodings. *IEEE Trans. on Inf. Theory*, 12:399–401, Julio 1966.

[Gro93] Moving Pictures Ecxperts Group. *Documentos del estandard MPEG. Partes 1, 2 y 3.* Número ISO/IEC 11172-1,2,3. ISO, 1993.

[Gro94] Joint Photographic Experts Group. *Digital Compression and Coding of Continuous-tone Still Images, Part 1: Requirements and guidelines.* Número ISO/IEC 10918-1. Information Technology, 1994.

[GT88] H. Gharavi y A. Tabatabai. Sub-band coding of monochrome and color images. *IEEE Trans. on circuits and systems*, 35(2):207–214, Febrero 1988.

[Han93] J. Handy. *The cache memory book*. Hartcourt Brace and Company. Academic Press, 1993. ISBN 0-12-322985-5.

[Hil88] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, Diciembre 1988.

[HM90] E. Hokenek y R.K. Montoye. Leading–zero anticipator (lza) in the ibm risc system/6000 floating–point execution unit. *IBM Journal Research and Development*, 34(1):71–77, 1990.

[How93] P. G. Howard. *The design and analysis of efficient lossless data compression systems*. Tesis Doctoral, Department of Computer Science. Brown University, Providence, Rhode Island 02912, Junio 1993.

[HS94] K. Huang y B. Smith. Lossless jpeg codec. ftp://ftp.cs.cornell.edu/pub/multimed/ljpg.tar.Z, Junio 1994. Department of Computer Science. Cornell University.

[Huf52] D. Huffman. A method for the construction of minimum redundancy codes. *Proc. IRE*, 40:1098–1101, 1952.

[HV94] P. G. Howard y J. S. Vitter. Arithmetic coding for data compression. *Proc. of the IEEE*, 82(6), 1994.

[HW98] M. H. Hsieh y C. H. Wei. An adaptive multialphabet arithmetic coding for video compression. *IEEE Transactions on Circuits an Systems for Video Technology*, 8(2):130–137, Abril 1998.

[Jai81] A. K. Jain. Image data compression: a review. *Proceedings of the IEEE*, 69(3):349–389, Enero 1981.

[Jia95] J. Jiang. Novel design of arithmetic coding for data compression. *IEE Proc-Comput. Digit. Tech.*, 142(6):419–424, 1995.

[Jon81] C. B. Jones. An efficient coding system for long source sequences. *IEEE Transactions on Information Theory*, IT-27(3), 1981.

[Jou90] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. En *17th ISCA*, páginas 364–373, Mayo 1990.

[KCZ+94] G. Kurpanek, K. Chan, J. Zheng, E. DeLano y W. Bryg. Pa7200: A pa-risc processor with integrated high performance mp bus interface. En *CompCon'94*, páginas 375–382, Febrero 1994.

[Knu85] D. E. Knuth. Dynamic huffman coding. *J. Algorithms*, 6(2):163–180, Junio 1985.

[Knu98] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volumen 3. Addison-Wesley Pub Co, 2 edición, Junio 1998. ISBN 0201896850.

[LB99] T. Lang y J.D. Bruguera. Multilevel reverse–carry computation for comparison and for sign and overflow detection in addition. En *Proc. Int. Conf. Computer Design. ICCD'99. Austin (Texas). USA.*, 1999.

[LBS98] R. Llados-Bernaus y R. L. Stevenson. Fixed-length entropy coding for robust video compression. *IEEE Trans. on Circuits and Systems for Video Technology*, 8(6):745–755, Octubre 1998.

[LR81] G. G. Langdon y J. Rissanen. Compression of black-white images with arithmetic coding. *IEEE Trans. on Communications*, 29(6):858–867, 1981.

[LST92] S. M. Lei, M. T. Sun y K. H. Tzou. Design and hardware architecture of high-order conditional entropy coding for images. *IEEE Transactions on Circuits an Systems for Video Technology*, 2(2):176–186, Junio 1992.

[MF90] M. W. Marcellin y T. R. Fischer. Trellis coded quantization of memoryless and gauss-markov sources. *IEEE Trans. on Communications*, 38(1):82–93, Enero 1990.

[MH95] J. C. McKinney y R. Hopkins. Digital audio compression standard. Informe técnico, Advanced Television System Committee, 1995.

[Mof90] A. Moffat. Linear time adaptative arithmetic coding. *IEEE Transactions on Information Theory*, 36(2):401–406, Marzo 1990.

[Nol91] T. Noll. Carry-save architectures for high speed signal processing. *Journal of VLSI Signal Processing*, 3:121–140, 1991.

[OB97] R. R. Osorio y J. D. Bruguera. New arithmetic coder/decoder architectures based on pipelining. En *Proc. IEEE International Conference On Application Especific Systems, Architectures, And Processors*, páginas 106–115, 1997.

[OB98] R. R. Osorio y J. D. Bruguera. Arithmetic coding/decoding architecture based on a cache memory. En *Proceedings of the 24th Euromicro Conference*, páginas 139–146, Agosto 1998.

[OB99] R. R. Osorio y J. D. Bruguera. New model for arithmetic coding/decoding of multilevel images based on a cache memory. En *IEEE International Conference on Electronics, Circuits and Systems (ICECS'99)*, September 1999.

[Okl94] V. G. Oklobdzija. An algorithmic and novel design of a leading zero detector circuit: comparison with logic synthesis. *IEEE Transactions on very large scale integration (VLSI) systems*, 2(1):124–128, 1994.

[PH94] Patterson y Hennesy. *Computer organization and design. The hardware/software interface.* Morgan Kauffmann Publishers, Inc., 1994. ISBN 1-55860-281-X.

[PM93] W. B. Pennebaker y J. L. Mitchell. *JPEG Still Image Data Compression Standard.* Van Nostrand Reinhold, 1993. ISBN 0-442-01272-1.

[PMJA88] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon Jr y R. B. Arps. An overview of the basic principles of the q-coder adaptative binary arithmetic coder. *IBM Journal of Research and Development*, 32(6), 1988.

[POB97] M. Peon, R. R. Osorio y J. D. Bruguera. A VLSI implementation of an arithmetic coder for image compression. En *Proc. Euromicro'97*, páginas 591–598, 1997.

[Ric79] R. F. Rice. Some practical universal noiseless coding techniques. Jet propulsion Laboratory, JPL Publication, 79-22, Marzo 1979.

[Ric83] R. F. Rice. Some practical universal noiseless coding techniques - part ii. Jet propulsion Laboratory, JPL Publication, 83-17, Marzo 1983.

[Ric91] R. F. Rice. Some practical universal noiseless coding techniques - part iii. Jet propulsion Laboratory, JPL Publication, 91-3, Noviembre 1991.

[Ris83] J. J. Rissanen. A universal data compression system. *IEEE Trans. on Inf. Theory*, 29(5):656–664, Septiembre 1983.

[Sha93] J. M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Trans. on Signal Processing*, 41:3445–3462, Diciembre 1993.

[Spa78] F. Sparacio. Data processing system with second level cache. *IBM Tech. Disc.*, 21(6):2468–2469, Noviembre 1978.

[SS82] J. A. Storer y T. G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, Octubre 1982.

[Str92] European Silicon Structures. *ES2 ECPD10 Library Databook*. European Silicon Structures, 1992.

[SW49] C. E. Shanon y W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Ill., 1949.

[Vit87] J. S. Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34(4):825–845, Marzo 1987.

[Wal91] G. K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):31–44, Abril 1991.

[WM95] B. W. Y. Wei y T. H. Meng. A parallel decoder of programmable huffman codes. *IEEE Transactions on circuits and systems for video technology*, 5(2):175–178, Abril 1995.

[WM97] X. Wu y N. Memon. Context-based, adaptive, lossless image coding. *IEEE Transactions on Communications*, 45(4):437–444, Abril 1997.

[WNC87] I. H. Witten, R. M. Neal y J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6), 1987.

[WSS96] M. Weinberger, G. Seroussi y G. Sapiro. Loco-i: A low complexity, context-based, lossless image compression algorithm. En *IEEE Data Compression Conference, Snowbird, Utah*, Marzo-Abril 1996.

[ZL77] J. Ziv y A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Inf. Theory*, 23(3):337–343, Mayo 1977.