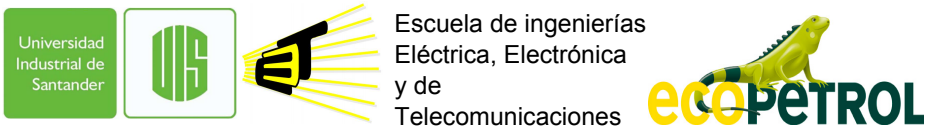
RENDIMIENTO COMPUTACIONAL EN UNA ARQUITECTURA DE ALTAS PRESTACIONES DE DOS ALTERNATIVAS DE DECODIFICACIÓN-CUANTIFICACIÓN INVERSA

CARLOS AUGUSTO OSORIO CADENA
LEONARDO FABIO RAMIREZ HERNANDEZ



Universidad Industrial de Santander
Facultad de Ingenierías Físico-Mecánicas
Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones
Bucaramanga
2014

RENDIMIENTO COMPUTACIONAL EN UNA ARQUITECTURA DE ALTAS PRESTACIONES DE DOS ALTERNATIVAS DE DECODIFICACIÓN-CUANTIFICACIÓN INVERSA

CARLOS AUGUSTO OSORIO CADENA

LEONARDO FABIO RAMIREZ HERNANDEZ

Trabajo de Investigación para optar al título de

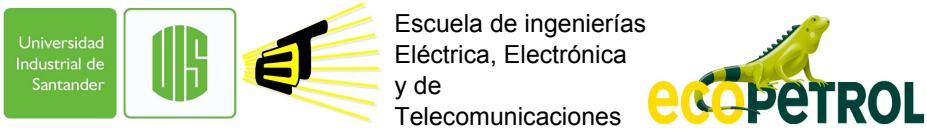
Ingeniero Electrónico

Director

CARLOS AUGUSTO FAJARDO ARIZA PhD(c)

Co-Director

Ing. CARLOS ARTURO BOADA QUIJANO



Universidad Industrial de Santander

Facultad de Ingenierías Físico-Mecánicas

Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones

Bucaramanga

2014

Agradecimientos

Agradecemos a nuestros padres por apoyar nuestras metas y acompañarnos en todas las etapas de nuestra vida.

A nuestro director Carlos Fajardo y a nuestro codirector Carlos Boada, quienes con sus ideas e importantes aportes permitieron alcanzar los objetivos planteados.

Al grupo de investigación CPS por su respaldo y poner a nuestra disposición todos los recursos necesarios en el proyecto. Este trabajo de grado fue financiado por ECOPETROL y COLCIENCIAS a través del proyecto de investigación 0266 del 2013.

A todos nuestros amigos, quienes compartieron con nosotros sus conocimientos e hicieron de nuestro paso por la universidad un tiempo muy agradable.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Resumen

TÍTULO: Rendimiento computacional en una arquitectura de altas prestaciones de dos alternativas de decodificación-cuantificación inversa. [1]

AUTORES: Carlos Augusto Osorio y Leonardo Fabio Ramirez. [2]

PALABRAS CLAVE: Cuantificación, Codificación, FPGA, VHDL.

Los métodos de codificación de longitud variable ofrecen buenos factores de compresión, pero es exactamente la variación de longitud en los códigos lo que hace que en un proceso de compresión sea la decodificación la etapa que presenta el mayor costo computacional, pues se requieren cálculos adicionales para reconocer el tamaño de los códigos. El presente proyecto tiene como objetivo evaluar el desempeño de dos algoritmos de decodificación de datos y su rendimiento computacional en una FPGA.

Se diseñó un algoritmo de compresión de datos compuesto por tres etapas: transformación, cuantificación y codificación. La etapa de transformación fue realizada con antelación y se utilizó transformada coseno enventanada. El objetivo de este proyecto es evaluar el rendimiento computacional en una FPGA de las dos últimas etapas. Para esto se implementaron una alternativa de cuantificación y dos métodos de codificación en una CPU. Para realizar la decodificación y la cuantificación inversa se implementaron los algoritmos en una FPGA.

Las pruebas fueron realizadas utilizando tres sets de datos, que corresponden a adquisiciones sísmicas terrestres. El rendimiento computacional se evaluó teniendo en cuenta los tiempos de cómputo en la FPGA. Además, se realizaron mediciones de SNR para las alternativas de cuantificación y bits promedio para los métodos de codificación. La implementación en CPU fue realizada utilizando lenguaje C++, la descripción de los decodificadores fue hecha en VHDL. Los resultados de la investigación muestran que para obtener medidas de SNR por encima de los 40 dB es necesario utilizar 10 bits o más al cuantificar. Por otra parte se obtuvieron mejores mediciones de bits promedio al utilizar codificación Shannon-Fano.

Se implementaron dos métodos de decodificación. El primero de ellos utiliza una bandera, la cual es usada para calcular el tamaño del código a decodificar, el segundo no emplea bandera. La medición de los tiempos de cómputo muestra que el decodificador sin bandera utiliza menos ciclos de reloj para poder decodificar los datos.

[1] Trabajo de investigación

[2] **Facultad:** Facultad de ingenierías Físico-Mecánicas. **Escuela:** Escuela de Ingenierías Eléctrica, Electrónica y de Telecomunicaciones. **Grupo de investigación:** CPS. **Director:** PhD(c). Carlos Augusto Fajardo

Abstract

TITLE: Computational efficiency of two decodification-inverse quantization alternatives in a high performance architecture. [3]

AUTHOR: Carlos Augusto Osorio y Leonardo Fabio Ramirez. [4]

KEY WORDS: Codification, Quantization , FPGA, VHDL.

Variable length coding methods allow sources to be compressed with high compression ratio (CR). But for decompression purposes the variation in lengths represents additional computational resources, in order to determine the length of each symbol. This project aims to test the computational performance of two data decoding algorithms.

It was designed a compression algorithm with three stages: Transformation, quantization and encoding. The transformation stage was made in advance. The Discrete Cosine Transform was used for this purpose, with a window of 32 samples.

The computational performance of the two last stages will be tested in an FPGA, so one quantization method and two coding methods were implemented in a CPU. And one inverse quantization method and two decoding methods were implemented in an FPGA.

Tests were performed using three data sets acquired by terrestrial sampling, and the decoding methods were compared by the computational times. Besides, SNR was measured for the quantization alternatives and average bits per code for the coding methods.

CPU implementation was made using C++ language, and the design of decoders was made in VHDL language. The results show that in order to obtain SNR measures above 40 dB, 10 or more bits are required for the quantization stage. On the other hand, better average bits per code were obtained when using Shannon-Fano coding.

Two decoding methods were implemented, one of them using a flag to determine the length of the codes, and the other one does not use any flag. Computational times measurements shows that the decoder without a flag needs fewer clock cycles to decode data.

[3] Research Work

[4] **Faculty:** Physical-Mechanical Engineering Faculty. **School:** School of Electrical, Electronics and Telecommunications Engineering. **Research group:** CPS. **Director:** PhD(c). Carlos Augusto Fajardo

INTRODUCCIÓN

La compresión de datos se usa con frecuencia en aplicaciones que requieren transmitir y procesar grandes cantidades de datos. La compresión consta de las etapas principales que se aprecian en la figura0.1. La codificación de datos se puede realizar a través de diferentes métodos, algunos de éstos llamados de longitud variable. Estos algoritmos brindan buenos factores de compresión [4], sin embargo la longitud variable de los códigos resultantes representa un desafío al intentar decodificarlos. Un solo error al decodificar la trama puede causar que todo el proceso sea interrumpido o que los datos decodificados sean incorrectos.

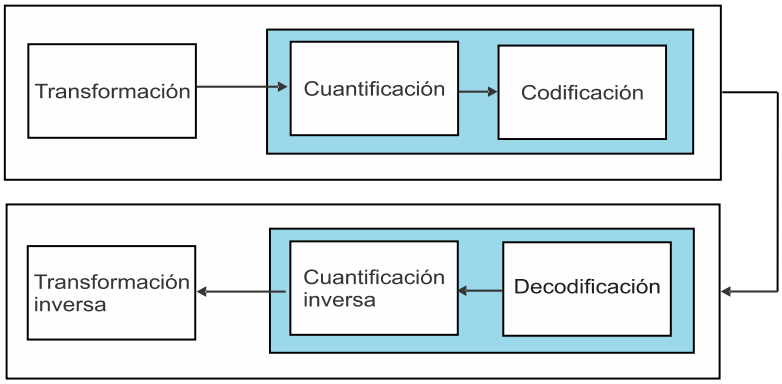


Figura 0.1: Etapas de la compresión de datos

Numerosos trabajos han abordado el tema de la compresión de datos. En [3] se usa codificación RLE y Huffman para comprimir trazas sísmicas, los resultados refieren buenos factores de compresión, pero altos tiempos de decodificación. Otro trabajos [1] [6] [7] se enfocan en implementar algoritmos de compresión de datos utilizando CPU's.

El presente trabajo centra su atención en las etapas de cuantificación, codificación, decodificación y

cuantificación inversa. Su objetivo es medir el rendimiento de dos alternativas de decodificación de datos al ser implementadas en una FPGA.

Las alternativas de cuantificación implementadas son comparadas a través de la relación de señal a ruido, los métodos de codificación se comparan mediante la medición de bits promedio, y los algoritmos de decodificación se confrontan en base a los recursos utilizados y parámetros de rendimiento computacional.

La estructura del informe es: Primero se presentan los métodos de cuantificación estudiados y los algoritmos de codificación que se deciden implementar. Seguidamente se explica la programación de los algoritmos de codificación, desarrollados en lenguaje C++. Posteriormente se presenta la descripción en VHDL de los algoritmos seleccionados, para su implementación en la FPGA. A continuación se presentan los resultados obtenidos. El informe termina con nuestras conclusiones y algunas sugerencias acerca de un posible trabajo futuro.

Capítulo 1

MARCO TEÓRICO

1.1 Métodos de cuantificación

Los métodos de cuantificación se pueden dividir en dos grupos principales: cuantificación uniforme y cuantificación no uniforme. La cuantificación no uniforme utiliza operaciones complejas computacionalmente (cálculo de logaritmos y potencias), por lo cual pensando en la implementación en una FPGA sería un punto en contra. Por lo tanto la investigación se centró en utilizar un método de cuantificación uniforme; para esto se probaron dos métodos: cuantificación escalar con zona muerta y cuantificación uniforme.

Cuantificación escalar con zona muerta

En [5] se explica un método de cuantificación escalar con zona muerta(*a mid-tread uniform threshold scalar quantizer*). El algoritmo de cuantificación está dado por:

$$i = \begin{cases} \frac{I}{2} + [\frac{y(k)+\frac{T}{2}}{\Delta}], & y(k) \leq \frac{-T}{2} \\ \frac{I}{2}, & \frac{-T}{2} < -y(k) < \frac{T}{2} \\ \frac{I}{2} + [\frac{y(k)-\frac{T}{2}}{\Delta}], & y(k) \geq \frac{T}{2} \end{cases}$$

Donde Δ indica la distancia entre los niveles de cuantificación (paso de cuantificación). T denota el ancho de la zona muerta alrededor de cero y esta definido por: $T = 2 \times \beta \times \Delta$. En compresión de imágenes $\beta = 0,5$ es usado normalmente. En datos sísmicos se ha utilizado $\beta = 0,6$ obteniendo buenos resultados[3]. I es una constante que simboliza la cantidad de palabras con las cuales queremos representar los datos.

Si decidimos utilizar 8 bits, se tendría $I = 2^8 = 256$.

La cuantificación inversa esta representada por:

$$\gamma(i) = (i - \frac{I}{2}) \times \Delta$$

Cuantificación uniforme

Los niveles de decisión en la cuantificación uniforme son igualmente espaciados, y pueden ser representados por una constante llamada *quantization step size*. Un cuantificador uniforme cubre un intervalo de $[X_{min}, X_{max}]$ de una variable aleatoria X, con una cantidad M de intervalos de decisión. El *quantization step size* esta definido por [9]:

$$\Delta = \frac{X_{min} - X_{max}}{M}$$

Los dos principales tipos de cuantificación uniforme son *Midtread* y *Midrise*. En el cuantificador *Midtread* el cero representa un nivel de cuantificación por lo tanto es apropiado para cuantificar trazas sísmicas. Este cuantificador toma como base que la señal de entrada es simétrica ($X_{min} = -X_{max}$), y puede ser implementado a través de la expresión:

$$q = round(\frac{x}{\Delta})$$

La correspondiente cuantificación inversa esta dada por:

$$x_q^{-1} = q\Delta$$

1.2 Métodos de codificación

Codificación Shannon-Fano

Es un algoritmo de longitud variable creado por Claude Shannon y Robert Fano[4]. Comienza con un conjunto de n símbolos con probabilidades conocidas (o frecuencias si se prefiere trabajar en formato entero). Para iniciar se ordenan los símbolos en orden decreciente en función de sus probabilidades. El conjunto de símbolos se divide en dos subgrupos, cada subgrupo posee igual probabilidad o aproximadamente (en cuanto el conjunto de datos lo permita). A los símbolos que pertenecen a un subgrupo se le asignan códigos que empiezan con 0 y a los del otro subgrupo códigos que empiezan con 1. Se realiza la misma operación

con los dos subgrupos resultantes, de esta forma hasta codificar todos los símbolos[4]. En la figura 1.1 se muestra la codificación Shannon-Fano para siete símbolos con sus respectivas probabilidades, los códigos generados se muestran en la tabla 1.1.

f(a)=25, f(b)=20, f(c)=15, f(d)=15, f(e)=10, f(f)=10, f(g)=5.

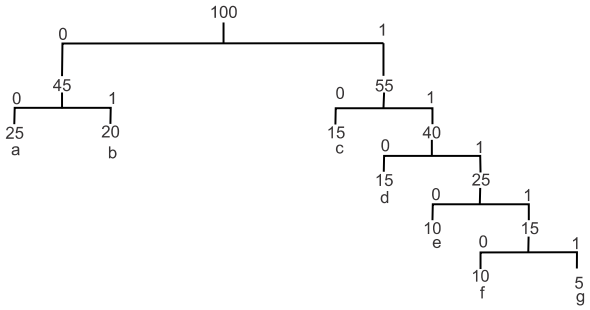


Figura 1.1: Diagrama codificación Shannon-Fano

Símbolo	**Frecuencia**	**Código**
a	25	00
b	20	01
c	15	10
d	15	110
e	10	1110
f	10	11110
g	5	11111

Tabla 1.1: Códigos algoritmo Shannon-Fano.

Codificación Huffman

A diferencia de la codificación Shannon-Fano que construye los códigos de arriba hacia abajo (genera los bits de izquierda a derecha), el método Huffman construye un árbol de abajo hacia arriba (genera los bits de derecha a izquierda).

Las probabilidades deben ordenarse en forma descendente, seguidamente se construye un árbol con un símbolo en cada hoja de abajo arriba. En cada etapa se deben seleccionar los dos símbolos que tienen menor probabilidad, se eliminan los símbolos de la lista y se reemplazan por un nuevo símbolo, cuya

probabilidad será igual a la suma de las probabilidades de los símbolos eliminados. Cuando todos los símbolos han sido reemplazados por un solo símbolo, el cual representa todo el alfabeto, se ha terminado la construcción del árbol. Por último se recorre todo el árbol para asignar los códigos[4]. En la figura 1.2 se muestra un ejemplo de la codificación Huffman, la tabla 1.2 muestra los códigos generados para cada símbolo.

f(a)=25, f(b)=20, f(c)=15, f(d)=15, f(e)=10, f(f)=10, f(g)=5.

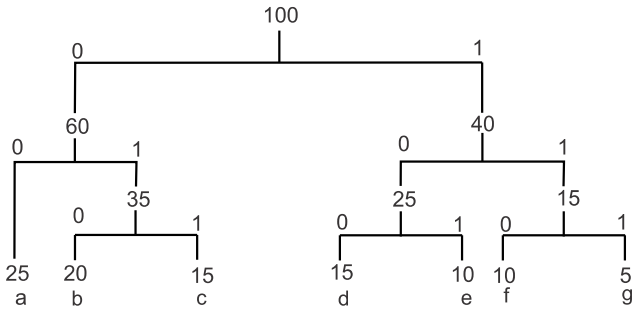


Figura 1.2: Árbol codificación Huffman

Símbolo	**Frecuencia**	**Código**
a	25	00
b	20	010
c	15	011
d	15	100
e	10	101
f	10	110
g	5	111

Tabla 1.2: Códigos algoritmo Huffman.

Capítulo 2

TRABAJO REALIZADO

2.1 Elección de los métodos de codificación

La tabla 2.1 muestra las principales características de cuatro métodos de codificación:

Método	**Ventajas**	**Desventajas**
Run length encoding	Códigos de tamaño fijo Bajo costo computacional	Bajo factor de compresión
Codificación Aritmética	Alto factor de compresión	Alto costo computacional Longitud de códigos variable
Codificación Huffman	Alto factor de compresión Bajo costo computacional	Longitud de códigos variable
Codificación Shannon-Fano	Alto factor de compresión Bajo costo computacional	Longitud de códigos variable

Tabla 2.1: Comparación métodos de codificación.

2.2 Programación de los algoritmos de codificación

Codificación Shannon-Fano

La figura 2.1 muestra el diagrama de flujo del algoritmo Shannon-Fano[4], haciendo mención especial en las funciones en las cuales se divide el algoritmo.

El programa inicia leyendo una traza guardada en un archivo .txt, estos datos son asignados a un vector y a continuación se realiza un histograma de los mismos. Los símbolos con sus respectivas frecuencias se ordenan descendentemente. A partir de esto ingresa en un ciclo “for”, dentro del cual se construyen los códigos respectivos para cada uno de los símbolos. Dentro de este ciclo aparecen tres funciones:

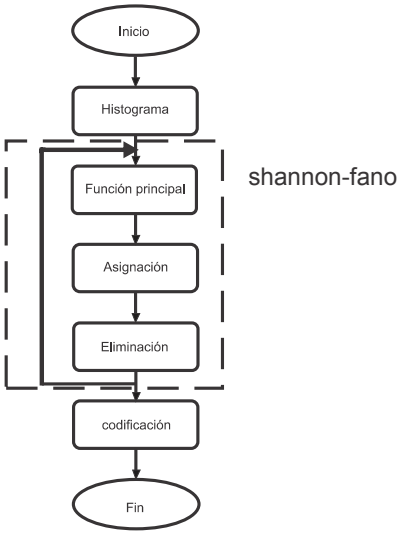


Figura 2.1: Diagrama de flujo codificación Shannon-Fano

- **Función principal**: para iniciar el programa la sumatoria de las frecuencias se guarda en un vector ("C"), este vector en un principio esta formado por un solo valor, para el ejemplo de codificación Shannon-Fano presentado en la figura 1.1 el valor almacenado seria 100, pero a medida que el programa se ejecuta son agregadas posiciones. Los valores guardados en C son divididos en 2, en el caso del ejemplo el resultado de esta división es 50, este valor es tomado como referencia para formar dos subgrupos, el primero de ellos compuesto por los elementos cuyas frecuencias sumadas dan un valor menor o igual a 50, el segundo, por las frecuencias restantes. Los subgrupos en el caso mostrado están formados por 2 y 5 elementos, los primeros 2 elementos suman una frecuencia de 45 y los restantes una frecuencia de 55, cumpliendose la condición descrita anteriormente. Estos dos valores de frecuencias se guardan en el vector C y son la base para la siguiente iteración en la cual cada una forma dos subgrupos, construidos de la misma manera y cuyos resultados son subgrupos de 1 elemento, para los tres primeros subgrupos y 4 elementos para el último subgrupo.

- **Asignación**: el número de elementos que conforma cada subgrupo, permite conocer los códigos a los cuales asignar "0" y a cuales asignar "1". Para la primera iteración en el ejemplo de la codificación Shannon-Fano, los dos primeros códigos inician con "0" y los restantes con "1". En la segunda iteración al código del primer elemento se le añade un "0", al segundo código un "1", al siguiente código un "0" y por último a los 4 restantes un "1". Dependiendo de la distribución de frecuencias algunos códigos terminan su construcción antes que otros, por lo tanto es necesario llevar una lista de las posiciones del diccionario a las cuales no se deben agregar bits, esto se ve claramente en el ejemplo, ya que a los códigos de los elementos a, b y c no se les deben agregar bits.

- **Eliminación**: los códigos anteriormente nombrados han terminado su construcción y no son necesarias sus frecuencias, por lo tanto se eliminan de la lista principal y se continúa la construcción con las frecuencias y símbolos restantes, quedando el histograma conformado por sólo 4 elementos. En otra lista se escribe el número de bits que conforma cada código, ya que en C++ es necesario declarar el número de columnas de una matriz, por lo tanto todos los códigos en el diccionario tendrán la misma longitud, para el ejemplo los 3 primeros códigos tienen una longitud de 2 bits.

El ciclo "for" termina cuando todas las frecuencias han sido eliminadas. La última parte del código consiste en la codificación de la traza:

- **Codificación**: inicia tomando el primer valor a codificar, este valor es buscado en los símbolos del histograma, de esta forma se conoce la posición del diccionario que le corresponde, y la cantidad de bits agregados a la trama de salida son los indicados por la lista número de bits. El proceso vuelve a iniciar tomando el siguiente dato de la traza, de esta forma hasta terminar la codificación de la traza.

Codificación Huffman

Al igual que en el algoritmo Shannon-Fano la traza es leída de un archivo de texto, para seguidamente realizar un histograma de los datos, estos deben ser ordenados en forma ascendente (el ordenar ascendentemente facilita el orden de las operaciones, si se ordenan descendentemente los punteros tendrían que retroceder en lugar de avanzar) en función de sus frecuencias.

La figura 2.2 muestra las principales funciones del programa. La función Huffman y Asignación son las encargadas de construir los códigos, mientras que la función codificación entrega la trama codificada. A continuación se describe el funcionamiento de cada una:

- **Huffman**: el árbol Huffman se contruye de abajo hacia arriba, ubicándose en la base las frecuencias de los diferentes símbolos. Inicia sumando las dos frecuencias menores, y comparándose esta suma con la frecuencia ubicada en la tercera posición. En el ejemplo de la codificación Huffman presentado en la figura 1.2 se suman 5+10=15 y se compara con la frecuencia ubicada en la tercera posición, que sería 10. En caso de que la suma sea mayor el resultado es guardado en un nuevo vector ("D"), continuando con la ejecución de la función asignación, de no ser mayor el valor no es guardado, pero igualmente se ingresa a la función asignación.

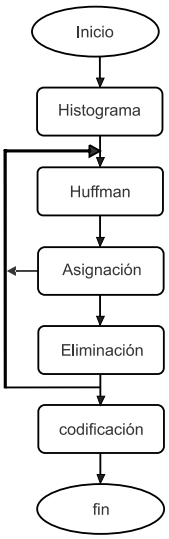


Figura 2.2: Diagrama de flujo codificación Huffman

- **Asignación**: adjudica los bits a los diferentes códigos. Se basa en conocer cuántas y cuáles frecuencias conforman los valores guardados en D. En el ejemplo de la figura 1.2 el primer valor guardado en D es 15 ya que cumple la condición descrita, por lo tanto los bits son asignados a los dos primeros códigos, un "1" al primero y un "0" al segundo. A medida que se sube en el árbol las sumas se hacen cada vez mayores, por ejemplo, para la tercera iteración una de las sumas es 40, este valor está formado por 15 y 25, es necesario observar qué valores conforman el 15 y el 25 para poder asignar los bits de manera correcta, en este caso se asigna "1" al primer y segundo código y "0" al tercero y cuarto código. Debido al tamaño fijo de los códigos en el diccionario es necesario conocer los bits que conforman cada código, para esto se lleva una lista que es actualizada a medida que se agregan los bits.

- **Eliminación**: sólo se ingresa a esta función en caso de que la condición no se haya cumplido. La función toma los dos valores que constituyen la suma, los elimina de la lista inicial e inserta el resultado de la suma en su posición, reemplazando dos posiciones por una sola. Los punteros son devueltos al valor que poseen al iniciar la ejecución de la función Huffman, y el programa vuelve a dicha función.

  Después de concluir la construcción de los códigos se puede codificar la traza. Para esto es necesaria la ejecución de la función codificación.

- **Codificación**: el valor a codificar se busca en la lista del histograma, esta posición es ubicada en el diccionario, y se agregan los bits indicados por la lista que contiene la cantidad de bits que conforma cada código. El proceso se realiza hasta codificar todos los valores de la traza.

2.3 Decodificación en VHDL

El proceso de decodificación descrito mediante VHDL tiene la limitación de que el almacenamiento de los códigos del diccionario en memoria ROM no permite que éstos tengan un tamaño variable, sino que todo el bloque ROM está limitado a tener el mismo tamaño del código más extenso. Es decir, aunque el diccionario esté compuesto por códigos de longitud variable, en el bloque ROM todos tienen el mismo tamaño.

De este modo, como se aprecia en la figura 2.3, la ROM usada para el diccionario contiene un código de mayor tamaño (en color verde) y otros de menor longitud a los que se les debe añadir bits de valor cero (en color rojo) para que posean el tamaño requerido. Si se recibe una trama de palabras codificadas y se desea comparar con los códigos presentes en el diccionario, es necesario asegurarse del tamaño real del código (color negro y azul) para retirarlo correctamente de la trama, pues en caso de ser retirada una cantidad incorrecta de bits, todo el proceso posterior arrojará resultados incorrectos.

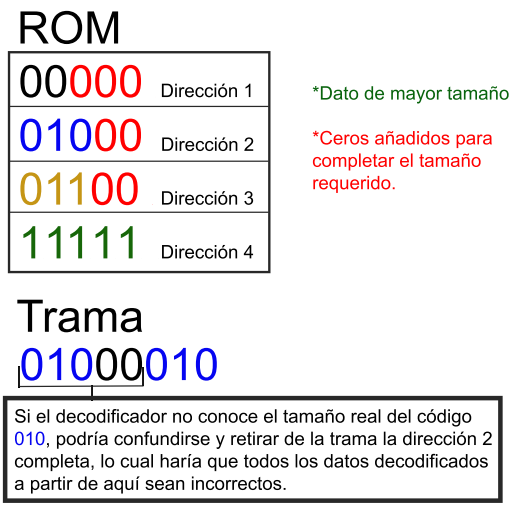


Figura 2.3: ROM con palabras codificadas sin bandera.

A partir de este planteamiento se han propuesto los siguientes modelos de decodificación en VHDL, que funcionan correctamente para códigos Shannon-Fano y Huffman, pues ambos son métodos estadísticos de codificación.

Decodificador con bandera

Para que el decodificador pueda detectar el tamaño correcto de los símbolos, se añade la bandera "1" al final de cada código presente en la ROM del diccionario, como se aprecia en la figura 2.4. La trama de datos codificados se transmite sin bandera, y al recorrer códigos en las direcciones del diccionario y comparar éstos con la trama entrante, se detectará un cambio al pasar por la bandera. En ese momento se almacena un registro del peso del bit de cambio. Se sigue recorriendo el código en el diccionario para comprobar que únicamente hay ceros delante de la bandera, si se detecta un bit de valor 1 se aumenta la dirección y se inicia de nuevo el proceso.

Al finalizar el recorrido del código en la dirección analizada el valor almacenado en el registro se compara con el valor calculado de acuerdo a los ceros presentes luego de la bandera. Si los valores son iguales se procede a decodificar el dato presente en la dirección analizada, de lo contrario se aumenta la dirección en uno y se inician de nuevo las comparaciones. Si se llega al final de los códigos sin decodificar dato alguno, se emite una señal de error. Todo este proceso se hace desde el bit más significativo hacia el menos significativo.

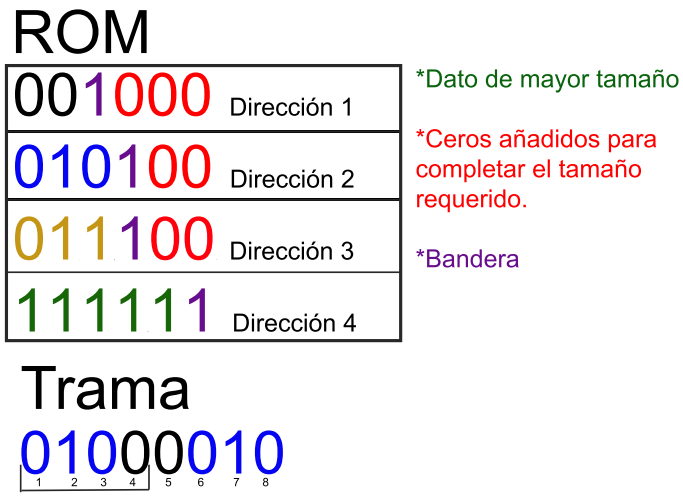


Figura 2.4: ROM con palabras codificadas y una bandera presente al final de la palabra.

Decodificador sin bandera

Los métodos de codificación Huffman y Shannon-Fano trabajan por pares y diferencian los códigos del diccionario mediante el último bit que le asignan a una determinada pareja, como se puede apreciar en la figura 2.5. Por lo tanto si el diccionario codificado está organizado en orden alfabético (iniciando con los bits en cero) y desde el bit más significativo al menos significativo, es posible para el decodificador detectar estos cambios entre parejas y decodificar si los bits de la trama codificada y del diccionario son iguales.

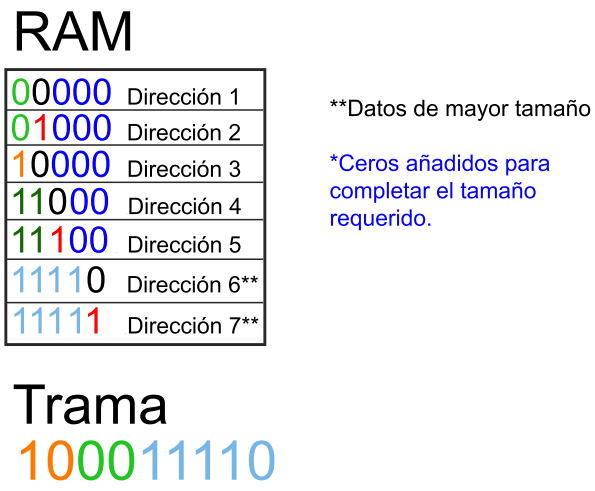


Figura 2.5: ROM con palabras codificadas ordenadas alfabéticamente.

En la figura 2.5 los símbolos están dispuestos en orden alfabético, los datos coincidentes tienen el mismo color, y el dígito que los diferencia tiene los colores negro y rojo, para los bits cero y uno respectivamente.

Si el decodificador se encuentra comparando el bit de peso x de la dirección n en el diccionario, y hay diferencia con el bit de peso x de la dirección n+1, e igualdad de bit respecto con la trama codificada, tiene las condiciones para decodificar el dato y retirar correctamente la palabra de la trama, pues conoce la cantidad de bits que componen el código, y la dirección en la cual se encuentra.

Del mismo modo si el decodificador detecta que los bits de peso x en la dirección n del diccionario y el buffer no coinciden, pero los bits de peso x en las direcciones n y $n+1$ son iguales, se aumenta la dirección hasta hallar aquella en la que sí coinciden los bits presentes en diccionario y buffer.

Esto también permite que la decodificación se realice en parejas, ya que si se detecta un cambio en las direcciones n y $n+1$, es posible decodificar inmediatamente, lo cual favorece la velocidad del proceso. Si se llega al final de los códigos en la ROM sin hallar el dato presente en la trama, se emite una señal de error.

Diagrama de Bloques

En las figuras 2.6 y 2.7 se aprecia el diagrama de bloques para los decodificadores sin bandera y con bandera, respectivamente.

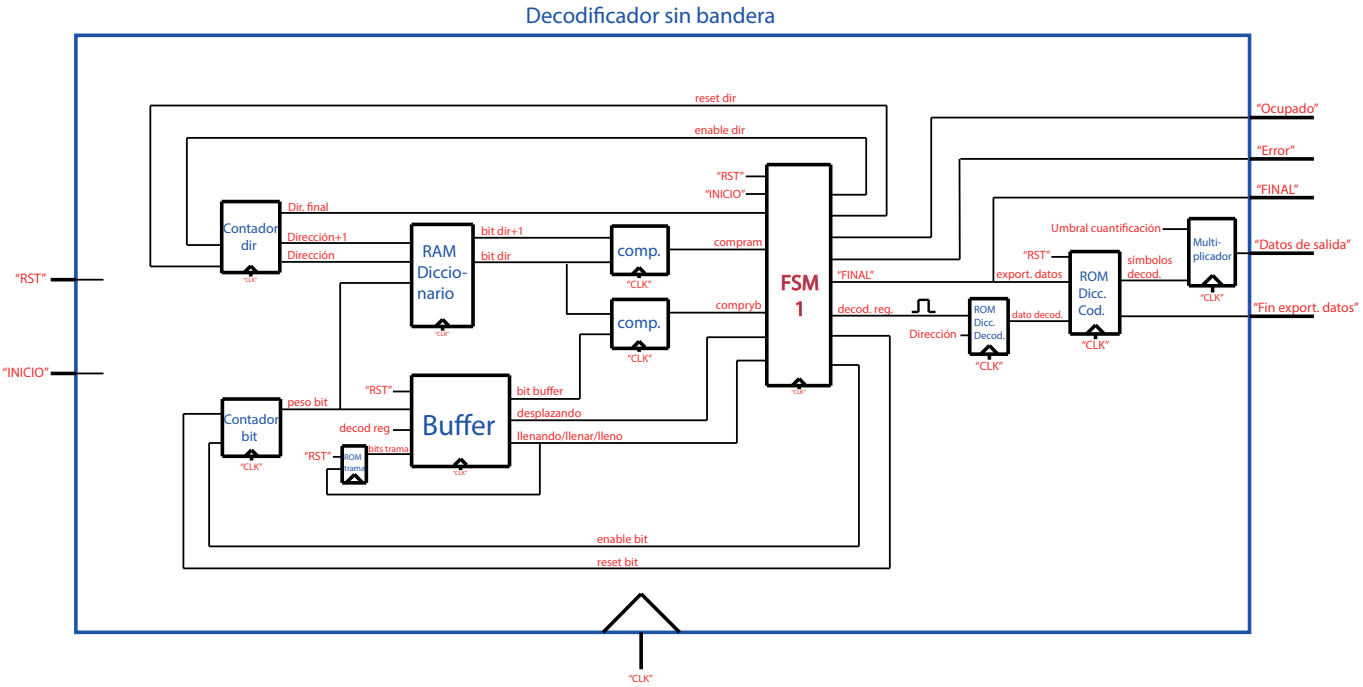


Figura 2.6: Diagrama de bloques del decodificador sin bandera.

Mapa genérico

Ambos decodificadores están definidos globalmente mediante valores genéricos que se entienden como se aprecia en la tabla 2.2, y que deben ser modificados de acuerdo a los datos cargados para la decodificación.

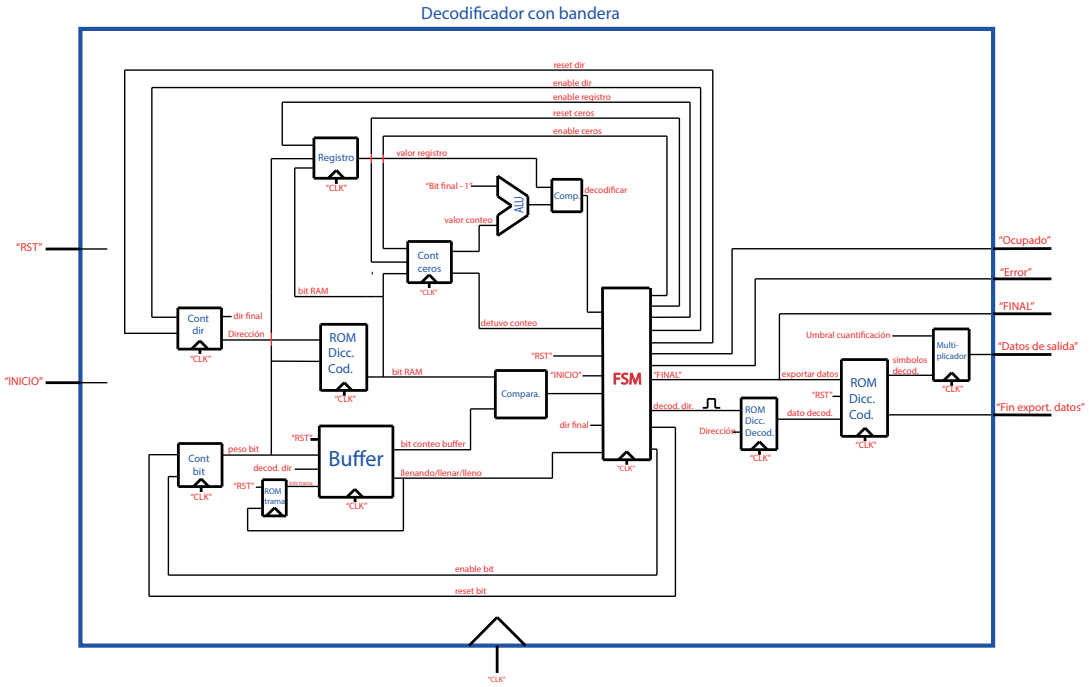


Figura 2.7: Diagrama de bloques del decodificador con bandera.

Mapa de puertos

Los puertos globales presentes en los decodificadores y sus significados son los apreciados en la tabla 2.3.

Buffer

Ambos decodificadores cuentan con un buffer compuesto de una memoria FIFO (*First In First Out*) y una máquina de estados que controla la escritura y el desplazamiento de los bits, el buffer recibe los datos desde un bloque RAM que contiene los bits de la trama de palabras codificadas. El buffer funciona como interfaz entre la RAM de la trama y el proceso de decodificación, y envía señales que indican al decodificador acerca de procesos de escritura o desplazamiento, así como también del final de la decodificación.

Bloques RAM y ROM

Los datos necesarios para la decodificación se encuentran almacenados en bloques ROM como el diccionario con las palabras codificadas y decodificadas, o la trama de símbolos a decodificar. El resultado de los procesos de decodificación se almacena en un bloque RAM que permite la escritura de los resultados y su posterior lectura y almacenamiento local en formato .txt al simular los resultados.

Variable genérica	**Función**
quant	Entero usado para cuantificación inversa.
sizequan	Tamaño del vector de cuantificación inversa.
strgdicc	Tamaño del vector de dirección en bloques ROM de diccionarios.
strgtr	Tamaño del vector de dirección en bloques ROM de trama.
strgdeco	Tamaño del vector de dirección en bloques RAM de datos resultantes.
sizecpb	Tamaño del vector usado para conteo de peso de bit.
sizebff	Tamaño del *buffer*
sizecode	Tamaño del mayor código codificado, bit más significativo.
sizedecode	Tamaño de la palabra decodificada
fintrama	Cantidad de bits presentes en la trama de palabras codificadas.
fincods	Cantidad de palabras en el diccionario

Tabla 2.2: Variables genéricas y sus funciones en el diseño de decodificación.

Puerto	**E/S**	**Función**
clkg	Entrada	Reloj interno
rstg	Entrada	Reset global, el decodificador espera señal de inicio.
startg	Entrada	Señal de inicio, permite iniciar la decodificación
errorg	Salida	Señal que indica error en la decodificación.
busyg	Salida	Señal que indica decodificación activa.
endendg	Salida	Indica el final de la decodificación.
endregg	Salida	Indica el final del registro de los datos de salida.
dataoutg	Salida	Exporta los datos de salida almacenados en RAM.

Tabla 2.3: Señales presentes en el mapa de puertos.

Las palabras en los bloques ROM son introducidas al diseño copiando archivos *.mif (Memory Initialization File)* en la carpeta local, y de allí son cargados a las respectivas instancias ROM mediante una función iterativa de lectura y escritura [8]. Esta función iterativa se usa asimismo para inicializar a cero todo el bloque RAM de datos finales decodificados.

2.4 Cuantificación inversa en VHDL

Ya que el proceso de cuantificación divide los datos sobre la norma, y los multiplica por $2^n - 1$, la cuantificación inversa debe multiplicar los símbolos de salida por la norma y dividirlos por $2^n - 1$. Esto se traduce en una etapa de salida que realiza la multiplicación del símbolo decodificado por un entero.

Debido a que los datos decodificados presentes en la ROM se encuentran en complemento a dos, la descripción en VHDL del multiplicador se realizó teniendo en cuenta este formato de representación.

El entero usado para la cuantificación inversa es almacenado en un vector binario, que será multiplicado por los datos provenientes de la ROM de datos decodificados. Es necesario declarar el tamaño del vector binario de modo que el entero se encuentre dentro del rango correspondiente a la longitud del vector. Los datos de salida resultantes de la cuantificación inversa se encuentran expresados en complemento a dos.

Capítulo 3

RESULTADOS

Los algoritmos de codificación y decodificación planteados fueron probados utilizando trazas sísmicas, las cuales corresponden a adquisiciones terrestres y pueden ser descargadas de [2]. Previamente las trazas fueron procesadas utilizando la transformada discreta del coseno DCT para una ventana de 32 elementos.

3.1 Cuantificación y codificación en CPU

Cuantificación

Para conocer cuál de los dos métodos de cuantificación proporciona una señal reconstruida, más semejante a la original, se midió la relación de señal a ruido (SNR). Una relación SNR por encima de 40dB permite conservar las características principales de las trazas sísmicas analizadas, por lo tanto este límite es tomado como referencia para las pruebas.

La tabla 3.1 muestra las mediciones de SNR realizadas al grupo de trazas después de aplicar los dos métodos de cuantificación antes descritos y aplicar cuantificación inversa. Inicialmente se utilizaron 9 bits de cuantificación. Para obtener estos valores fue utilizada una función en *Matlab*.

De cada set de datos se eligió aleatoriamente una traza para ser analizada. Se obtuvieron mayores mediciones de SNR al utilizar cuantificación uniforme, por lo tanto se usa este método para cuantificar las trazas; sin embargo los resultados para 9 bits son muy cercanos a la referencia (40 [dB]), por lo cual se decide cuantificar con un bit más, de esta forma se asegura que el SNR entre la señal original y la

Set de datos	**Trazas**	**Cuantificación escalar con zona muerta SNR [dB]**	**Cuantificación uniforme SNR [dB]**
$Data1$	(1,1568)	34,74	41,19
$Data2$	(1,1024)	35,31	41,72
$Data3$	(120,1088)	35,38	42,74
$Data4$	(24,1568)	32,16(promedio)	38,3(promedio)

Tabla 3.1: Comparación entre los dos métodos de cuantificación

reconstruida sea mayor a 40 [dB].

Codificación

Después de leer las trazas, el primer paso es cuantificarlas con 10 bits y seguidamente codificarlas utilizando alguno de los dos métodos antes descritos y programados en C++. La tabla 3.2 resume los resultados obtenidos al aplicar el proceso de codificación.

Traza	$Data1$	$Data2$	$Data3$
Cantidad de símbolos	254	239	115
Mayor probabilidad	0,25	0,16	0,43
Longitud código más corto Shannon-Fano [Bit]	2	3	1
Longitud código más largo Shannon-Fano [Bit]	11	11	11
Longitud código más corto Huffman [Bit]	7	7	6
Longitud código más largo Huffman [Bit]	8	8	8

Tabla 3.2: Tamaño diccionarios y códigos

Los datos originalmente se encuentran en formato entero *(int)* por lo que se toma como base una representación de 32 bits. En la tabla 3.3 se calculan los bits promedio utilizados para representar un código en alguno de los dos métodos.

La tabla 3.4 muestra los valores de SNR medidos entre los datos originales, datos con los cuales inicia el proceso, y los datos reconstruidos después de aplicar cuantificación, codificación, decodificación y

Traza [Datos]	**Bits representación normal**	**Bits promedio Shannon-fano**	**Bits promedio Huffman**
$Data1(1568)$	32	5,66	7,68
$Data2(1024)$	32	6,30	7,77
$Data3(1088)$	32	3,9	6,59

Tabla 3.3: Bits promedio

cuantificación inversa, esto desarrollado en lenguaje C++.

Traza [Datos]	**SNR [dB]**
$Data1(1568)$	48,60
$Data2(1024)$	48,78
$Data3(1088)$	42,14

Tabla 3.4: SNR traza inicial y reconstruida

3.2 Decodificación y cuantificación inversa en FPGA

El diseño e implementación de los decodificadores fueron realizados en el programa ISE 13.2, utilizando una FPGA XC3S50AN, la cual hace parte del sistema de desarrollo Xilinx Spartan 3AN. Las pruebas se iniciaron con diccionarios de 15 elementos, se introdujeron todos sus datos uno a uno, y se obtuvo una correcta decodificación de cada uno de los símbolos.

Posteriormente las pruebas se llevaron a cabo con los set de datos elegidos anteriormente. El tamaño de los diccionarios superó los 100 símbolos y las tramas decodificadas alcanzaron los 6900 bits. Los datos resultantes de la decodificación han sido escalados para que el tamaño de los vectores resultantes no interfiera con los tiempos de medida de desempeño computacional. Esto se traduce en un valor bajo de cuantificación inversa, de este modo, los vectores resultantes tienen un menor tamaño de acuerdo a las limitantes de memoria en la FPGA. Las tablas 3.5 y 3.6 presentan el desempeño de los decodificadores ante estas condiciones, se miden variables como *Average Throughput*, frecuencia máxima de operación, latencia y número de *slices* usados en la FPGA Xilinx Spartan 3AN (de 5888 slices disponibles).

Se utilizó el programa ChipScope para comprobar el correcto funcionamiento de los decodificadores, y comparar los resultados arrojados por dicha herramienta con los obtenidos utilizando el programa de

simulación ISim 13.2.

Decodificador sin bandera	**Av. Th. [Mbps]**	**Frec. Máx [MHz]**	**Latencia [ciclos]**	**N° *slices***
Data2(1024)	2,6414	50,246	154	569
Data3(1088)	5,1021	55,193	31	394

Tabla 3.5: Medición de desempeño computacional por parte del decodificador sin bandera.

Decodificador con bandera	**Av. Th. [Mbps]**	**Frec. Máx [MHz]**	**Latencia [ciclos]**	**N° *slices***
Data2(1024)	0,6707	81,752	346	460
Data3(1088)	1,3574	80,533	53	338

Tabla 3.6: Medición de desempeño computacional por parte del decodificador con bandera.

En las figuras 3.1, 3.2, 3.5 y 3.6 se aprecian los resultados arrojados por la herramienta de simulación ISim 13.2. En las figuras 3.3, 3.4, 3.7 y 3.8 se muestran los datos obtenidos en implementación mediante el análisis con la herramienta Chipscope Pro. .

Durante el proceso de decodificación los datos resultantes son almacenados en una memoria RAM, cuyo contenido es leído al finalizar la decodificación de toda la trama codificada.La señal que dispara el registro de los datos es *fin decod* (*Endedg*, que indica el final de la decodificación). En ese momento se activa la exportación de los datos decodificados presentes en la RAM (uno por cada ciclo de reloj), estos datos pasan por el módulo de cuantificación inversa y se presentan en la señal *sal cuant inv* (*Dataoutg*). Al finalizar la exportación, la señal *fin registro* (*Endregg*) se hace igual a 1, indicando el final del registro de datos.

En la figura 3.1 se presenta la exportación de los primeros 19 datos, para la traza Ozdata3 de 6973 bits, y en la figura 3.3 su equivalente en implementación analizado con Chipscope, que presenta la exportación de los primeros 15 datos, los cuales concuerdan con la simulación.

En la figura 3.2 se observa la exportación de los últimos 22 datos, para Data2, así como la indicación

del final de registro de los datos (*Endregg*), lo que indica que se ha llegado al final de los datos almacenados en la RAM final. Y en la figura 3.4 su equivalente en implementación mediante análisis con Chipscope.

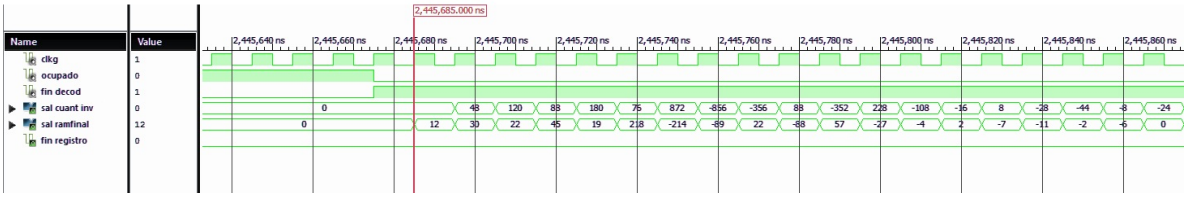


Figura 3.1: Simulación en ISim 13.2 para Data2. Decodificador con bandera.

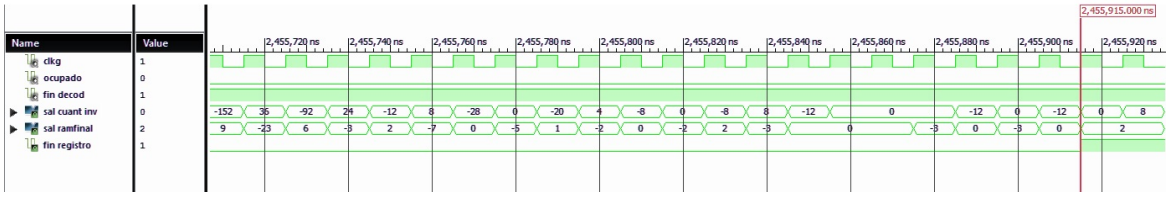


Figura 3.2: Simulación en ISim 13.2 para Data2. Decodificador con bandera.

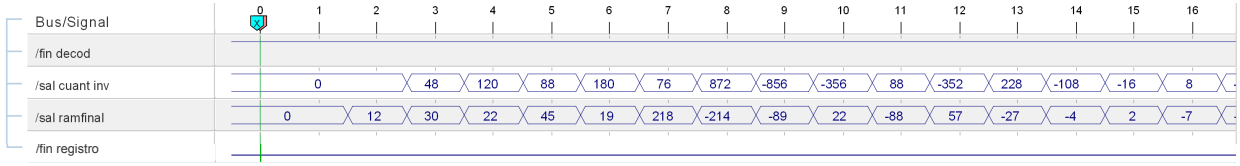


Figura 3.3: Análisis en ChipScope Pro para Data2. Decodificador con bandera.

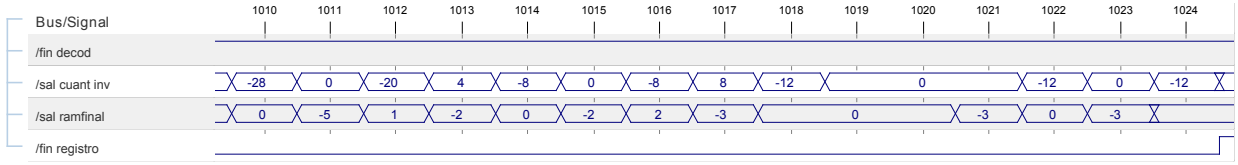


Figura 3.4: Análisis en ChipScope Pro para Data2. Decodificador con bandera.

Asimismo, las figuras 3.5, 3.6, 3.7 y 3.8 presentan las imágenes de simulación e implementación correspondientes a Data3, de 4858 bits.

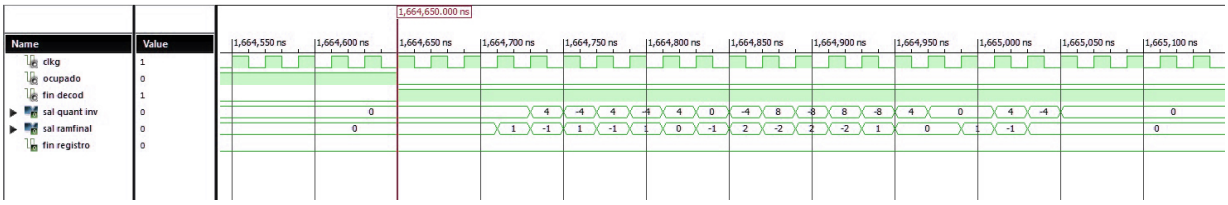


Figura 3.5: Simulación en ISim 13.2 para Data3. Decodificador sin bandera.

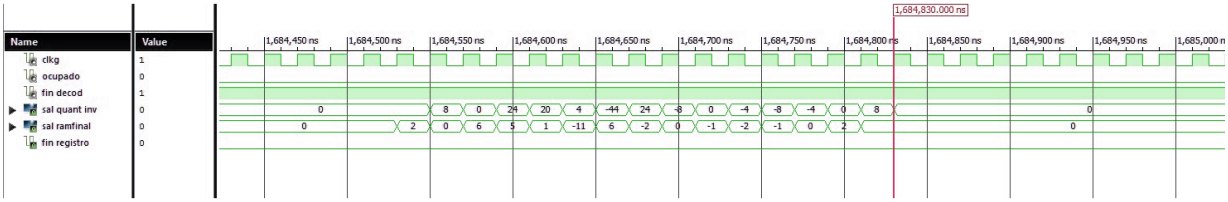


Figura 3.6: Simulación en ISim 13.2 para Data3. Decodificador sin bandera.

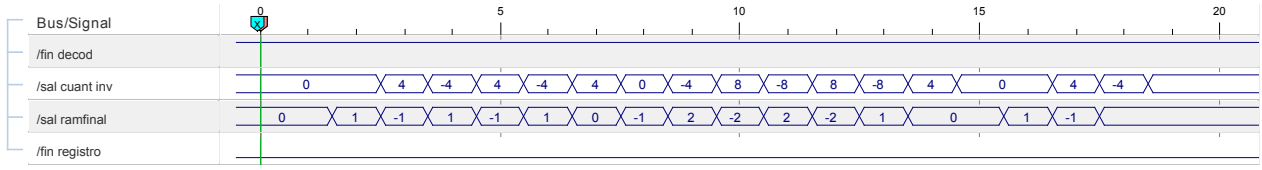


Figura 3.7: Análisis en ChipScope Pro para Data3. Decodificador sin bandera.

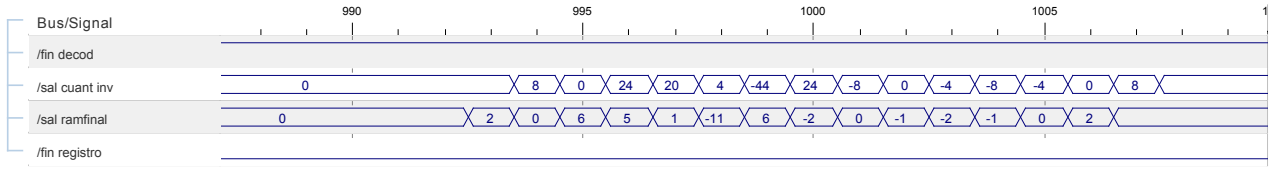


Figura 3.8: Análisis en ChipScope Pro para Data3. Decodificador sin bandera.

Capítulo 4

CONCLUSIONES

- La cuantificación uniforme permite obtener relaciones de señal a ruido (SNR) por encima de los 40 dB al usar 10 bits de cuantificación. El uso de menos bits al cuantificar representa una pérdida de información que en el caso de las trazas sísmicas utilizadas no es recomendable.
- La cuantificación uniforme presenta una baja complejidad matemática en sus operaciones, por lo cual al ser implementada en una FPGA, puede ser representada como una multiplicación por un valor entero.
- Como se muestra en la Tabla 3.2, la Codificación Shannon-Fano brinda un valor de bits promedio menor al producido por la Codificación Huffman. Esto se presenta cuando el valor que más se repite tiene una probabilidad cercana o mayor al 50%, lo cual implica que su código sea conformado por un solo bit. Si el dato que más se repite no tiene una probabilidad tan elevada, los dos métodos brindan valores de bits promedio cercanos, sin embargo, el menor valor siempre lo presenta la Codificación Shannon-Fano.
- El uso de una bandera en los elementos del diccionario implica la añadidura de un proceso de reconocimiento que resta tiempo a la decodificación, esto se aprecia en un mejor desempeño del decodificador sin bandera sobre el que sí tiene bandera.
- Siempre que se cumplan acertadamente las condiciones de orden alfabético en el decodificador sin bandera, o las condiciones en el diccionario de códigos del decodificador con bandera, ambos procesos de decodificación se llevan a cabo sin ningún error y los resultados son los datos equivalentes en la RAM de datos decodificados.

4.1 Recomendaciones

- Instanciar separadamente los componentes en módulos individuales de decodificación, almacenamiento y transmisión de datos, para que los procesos en cada módulo no interfieran directamente en el de-

sempeño del otro.

4.2 Trabajo futuro

- Tomar los procesos de decodificación y optimizar las etapas de transmisión y almacenamiento de datos, uniendo este trabajo a otros que mejoren su desempeño.
- Realizar el proceso de decodificación en paralelo, esto mejoraría el desempeño del decodificador notablemente.

BIBLIOGRAFÍA

[1] Amir Z Averbuch, F Meyer, J O Stromberg, R Coifman, and A Vassiliou. Low bit-rate efficient compression for seismic data. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, 10(12):1801–1814, jan 2001.

[2] CWP. Data oz.original @ONLINE. `http://www.cwp.mines.edu/cwpcodes/data/oz.original`, June 2014.

[3] Daniel Haugen. *Seismic Data Compression and GPU Memory Latency.* Master thesis, Norwegian University of Science and Technology, 2009.

[4] David Salomon. *Data Compression The Complete Reference*, pages 77, 78. 4 edition, 2007.

[5] Rosten Tage. *Seismic Data Compression using subband coding.* PhD thesis, Norwegian University of Science and Technology, 2000.

[6] W. Wu, Z. Yang, Q. Qin, and F. Hu. Adaptive Seismic Data Compression Using Wavelet Packets. *2006 IEEE International Symposium on Geoscience and Remote Sensing*, (3):787–789, July 2006.

[7] Xing Xie and Qianqing Qin. Fast lossless compression of seismic floating-point data. *International Forum on Information Technology and Applications*, 2009.

[8] Xilinx. *XST User Guide for Virtex-6, Spartan-6, and 7 series devices.* 2011.

[9] Y You. *Audio Coding*, pages 21, 22. 2010.