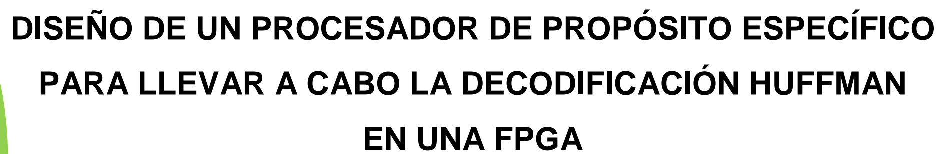
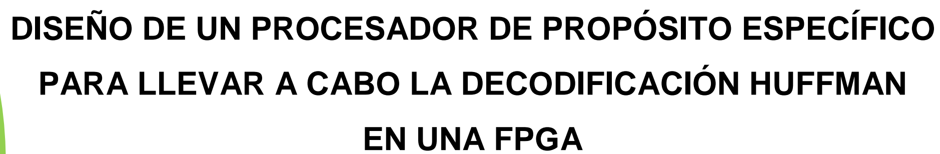


DISEÑO DE UN PROCESADOR DE PROPÓSITO ESPECÍFICO PARA LLEVAR A CABO LA DECODIFICACIÓN HUFFMAN EN UNA FPGA

MARCOS ANDRES CAICEDO MATEUS

MANUEL FERNANDO PÉREZ LAYTÓN

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
BUCARAMANGA
2016



MARCOS ANDRES CAICEDO MATEUS
MANUEL FERNANDO PÉREZ LAYTÓN

Trabajo de grado para optar al título de
Ingeniero Electrónico

Director
PhD(c) Carlos Augusto Fajardo Ariza
Co-director
PhD Óscar Mauricio Reyes Torres
Co-director
Ing. Carlos Andrés Angulo Julio

UNIVERSIDAD INDUSTRIAL DE SANTANDER
FACULTAD DE INGENIERÍAS FÍSICO-MECÁNICAS
ESCUELA DE INGENIERÍA ELÉCTRICA, ELECTRÓNICA Y DE TELECOMUNICACIONES
BUCARAMANGA
2016

AGRADECIMIENTOS

Este trabajo fue apoyado por la compañía Colombiana de Petróleos ECOPETROL y COLCIENCIAS (Departamento Administrativo de Ciencia, Tecnología e Innovación) como parte del proyecto de investigación No. 0266 de 2013. Los autores agradecen a la Universidad Industrial de Santander y a los miembros del grupo de investigación CPS.

CONTENIDO

Pág.

INTRODUCCIÓN 11

1. MARCO TEÓRICO 12

1.1. COMPRESIÓN DE DATOS 12

1.2. CUANTIFICACIÓN 13

1.3. ALGORITMO HUFFMAN 13

1.4. DECODIFICACION DE DATOS 16

1.5. PUERTO PCIe (Peripheral Component Interconect Express) 16

2. TRABAJO DESARROLLADO 17

2.1. DISEÑO DEL DECODIFICADOR HUFFMAN 17

2.1.1. Determinación de los parámetros del proceso de compresión 17

2.1.2. Almacenamiento de datos 19

2.1.3. Decodificador Huffman 21

2.2. IMPLEMENTACIÓN DEL DECODIFICADOR HUFFMAN 24

2.2.1. Proceso de Comunicación CPU-FPGA 25

2.2.2. Implementación de un core 26

2.2.3. Implementación paralela del proceso de decodificación 27

3. RESULTADOS 30

4. CONCLUSIONES 35

5. RECOMENDACIONES 37

REFERENCIAS BIBLIOGRAFICAS 38

BIBLIOGRAFÍA 40

LISTA DE FIGURAS

Pág.

Figura 1: Árbol binario..15
Figura 2: Plataforma de co-procesamiento. Tomada de [1] 16
Figura 3: Memoria de datos codificados 20
Figura 4: Memoria de símbolos..20
Figura 5: Memoria de códigos..21
Figura 6: Arquitectura del decodificador Huffman 22
Figura 7: Implementación de un *core*..27
Figura 8: Implementación de los decodificadores en forma paralela 28
Figura 9: Línea de tiempo para la decodificación en forma paralela......................29
Figura 10: Variación del factor de compresión al cambiar los bits de cuantificación 30
Figura 11: Ciclos de reloj para la decodificación de datos 32
Figura 12: Variación del tiempo de decodificación al cambiar el factor de compresión 33
Figura 13: Tiempo de decodificación debido a la cantidad de decodificadores implementados...34

LISTA DE TABLAS

Pág.

Tabla 1: Frecuencia de los caracteres 15
Tabla 2: Asignación de códigos 15
Tabla 3: Códigos con longitud entre 1 y 5 bits de los 12 disparos con 12 bits de cuantificación 18
Tabla 4: Diccionario *Huffman* 19

RESUMEN

Título: Diseño de un procesador de propósito específico para llevar a cabo la decodificación Huffman en una FPGA[1]

Autores: Marcos Andrés Caicedo Mateus[2], Manuel Fernando Pérez Laytón[2]

Palabras clave: FPGA, PCIe, Huffman, Decodificación

La exploración sísmica produce gran cantidad de datos que pueden exceder el centenar de Terabytes, lo que dificulta su transmisión y almacenamiento para su respectivo procesamiento y análisis. Los algoritmos de compresión son muy utilizados ya que ofrecen una reducción en términos de capacidad de almacenamiento y ancho de banda de transmisión. El algoritmo de codificación *Huffman* ofrece uno de los mejores factores de compresión. Este algoritmo comprime los datos mediante la asignación de palabras de código más cortas para los símbolos más frecuentes, mientras que a los otros símbolos se les asigna palabras de código más largas. Sin embargo, es difícil disminuir el tiempo del proceso de decodificación ya que es un proceso altamente secuencial debido a la longitud variable de los códigos. Se ha diseñado un decodificador *Huffman* que permite decodificar en solo un ciclo de reloj los datos representados con códigos de hasta 5 bits de longitud. El diseño fue desarrollado en lenguaje VHDL en el software *ISE Design Suite* 13.2 de *Xilinx*, se implementó en una FPGA Virtex 5 XC5VFX70T y la comunicación CPU-FPGA se realizó por medio del bus PCIe. La estrategia desarrollada permite disminuir el tiempo de decodificación al paralelizar más del 50% del proceso. Los resultados sugieren que el diseño es superior a las versiones anteriores.

[1] Trabajo de grado modalidad en investigación

[2] Facultad de Ingenierías Físico Mecánicas. Escuela de Ingeniería Eléctrica, Electrónica y de Telecomunicaciones. Director: PhD(c) Carlos A. Fajardo. Codirectores: PhD Óscar Mauricio Reyes, Ing. Carlos A. Angulo

ABSTRACT

Title: Specific purpose processor design to carry out the Huffman decoder in a FPGA[3]

Authors: Marcos Andrés Caicedo Mateus[4], Manuel Fernando Pérez Laytón[4].

Key words: FPGA, Pcle, Huffman, Decoder.

A seismic survey produces a huge amount of data, which is in the order of hundreds of terabytes. This amount of data requires high capacities both storage and transmission. Thus, compression algorithms are desired to make the storage and transmission more efficient. The Huffman algorithm is commonly used to compress seismic data because it offers high compression ratios. The algorithm compresses the data by assigning shorter codewords to the most frequent symbols while the other symbols use longer codewords. The decoding process is a highly sequential algorithm because of the variable length of the Huffman codewords. For this reason, the decodification of a Huffman codeword requires several clock cycles. It is designed and implemented a Huffman decoder that decodes, in a just one clock cycle, code words up to five bits of length. The design was developed in VHDL by using the ISE Design Suite 13.2 software from Xilinx. The design was implemented in an FPGA Virtex 5 XC5VFX70T, which was communicated to the CPU by the PCI Express bus. The results showed that the implementation can decode about the 50% of seismic data in a parallel fashion, which allowed us to reduce the decoding time. The results suggest that the design is superior to previous versions.

[3] Degree project

[4] Faculty of Physics Mechanics Engineering. Electrical, Electronics Engineering and Telecommunication School. Advisor: PhD(c) Carlos A. Fajardo. Co-Advisors: PhD Óscar Mauricio Reyes, Ing. Carlos A. Angulo.

INTRODUCCIÓN

La industria del petróleo utiliza la reflexión sísmica como una de las herramientas principales para la exploración y evaluación de yacimientos de petróleo. Al mismo tiempo, la exploración sísmica produce gran cantidad de datos que pueden exceder el centenar de Terabytes [1], los cuales son transmitidos entre las estaciones locales para su respectivo procesamiento. Debido al tamaño de los datos sísmicos, las empresas del sector petrolero generalmente utilizan algoritmos de compresión, con el fin de hacer más manejable los procesos de almacenamiento y transmisión. Actualmente se utilizan diversos algoritmos para comprimir los datos sísmicos. Entre los algoritmos de compresión que más se utilizan se encuentra el algoritmo de codificación *Huffman* [2], pues este ofrece uno de los mejores factores de compresión. Sin embargo, el proceso de decodificación es un proceso altamente secuencial debido a la longitud variable de los códigos. Esto hace difícil paralelizar el algoritmo de decodificación.

En este trabajo se presenta el diseño y la implementación en una FPGA de un decodificador *Huffman* de datos sísmicos. El diseño decodifica en paralelo los códigos con longitudes entre 1 y 5 bits, empleando tan sólo un ciclo de reloj. Gracias a esta técnica se logra disminuir el tiempo del proceso de decodificación al paralelizar más del 50% de los datos, reduciendo significativamente la cantidad de datos decodificados de forma secuencial. Los resultados sugieren que nuestro diseño es superior a las versiones anteriores. Este documento está organizado de la siguiente manera: El primer capítulo muestra el marco teórico. El segundo capítulo muestra la arquitectura propuesta para este proyecto. En el tercer capítulo se presenta una descripción de los pasos empleados en la implementación y los resultados obtenidos en las pruebas. En el cuarto capítulo se exponen las conclusiones y finalmente en el quinto capítulo se presentan las recomendaciones.

1. MARCO TEÓRICO

1.1. COMPRESIÓN DE DATOS

La estrategia de compresión de datos permite enviar la información utilizando un menor número de bits. De forma muy general podemos observar la estrategia de la siguiente manera: dentro de una memoria se encuentran almacenados los datos originales o cadena de información original y se busca que la salida sea una cadena de datos de menor tamaño respecto a la original. Existen dos tipos de algoritmos de compresión: compresión sin pérdida y compresión con pérdida. En el primero, la información del dato original se preserva completamente. En los algoritmos de compresión con pérdida, la salida es una aproximación de la entrada y se busca preservar la información más relevante [3]. En este trabajo nosotros utilizamos un algoritmo de compresión con pérdida.

La acción de compresión aplicada sobre la información fuente puede ser ejecutada mediante diferentes estrategias o métodos que pueden generar diferentes resultados dependiendo del tipo de dato de entrada. Básicamente todos los métodos tienen el mismo objetivo, comprimir el tamaño de los datos reduciendo la redundancia de los datos del archivo fuente [4].

Generalmente la compresión de los datos sísmicos se realiza en tres etapas [4]:

- **Transformación:** Esta etapa permite presentar los datos de una forma más compacta. En términos de la teoría de la información, esta etapa ayuda a disminuir la entropía en los datos.

- **Cuantificación:** El principal objetivo de esta etapa es reducir el número de bits requeridos para representar cada muestra obtenida durante el proceso de discretización.

- **Codificación:** Es en este paso donde se aplica la estrategia para reducir la redundancia en la información y así reducir el tamaño del archivo original.

Con el propósito de reducir los tiempos de descompresión, en este trabajo nosotros utilizamos un algoritmo compuesto por dos etapas: cuantificación y codificación. Específicamente, utilizamos cuantificación uniforme y codificación *Huffman* [4].

1.2. CUANTIFICACIÓN

La etapa de cuantificación es el proceso de mapear un conjunto de datos dentro de un intervalo a un conjunto finito de niveles de salida. Esta aproximación reduce el número de bits necesarios para representar cada coeficiente, pero conlleva pérdida de información. Para aplicaciones de datos sísmicos esta pérdida es aceptable. Un cuantificador uniforme se utiliza generalmente en la compresión de datos sísmicos, ya que mejora la SNR sobre otros tipos de cuantificadores [5] y se alcanza la entropía mínima en aplicaciones sísmicas [6], [7]. Una ganancia adicional de usar un cuantificador uniforme es su baja complejidad computacional.

1.3. ALGORITMO HUFFMAN

El algoritmo Huffman es un procedimiento que permite codificar cierta información con símbolos estadísticamente independientes y de longitud variable, logrando representar los caracteres más frecuentes con un reducido número de bits y los caracteres menos frecuentes con un mayor número de bits; por esta razón, este tipo de codificación es más eficiente que aquellos de longitud fija, ya que para representar cierta información utilizará menos bits en promedio [8].

El algoritmo *Huffman* se puede explicar de la siguiente manera: dado un conjunto de símbolos, el primer paso a realizar es una lista de todos los símbolos con su respectiva frecuencia de repetición, luego la lista se debe organizar de forma descendente de arriba hacia abajo según las frecuencias. Cada uno de los símbolos de la lista conformará un nodo inicial.

Posteriormente, se conectan los dos símbolos con menor repetición para formar un nuevo nodo y se suman sus frecuencias de repetición; organizaremos de nuevo la lista ubicando el nodo en su lugar respectivo según su frecuencia de aparición. Se aplicará este procedimiento las veces necesarias hasta obtener un único nodo principal y los demás como un despliegue de este nodo. A esta representación se le conoce como árbol de codificación *Huffman.*

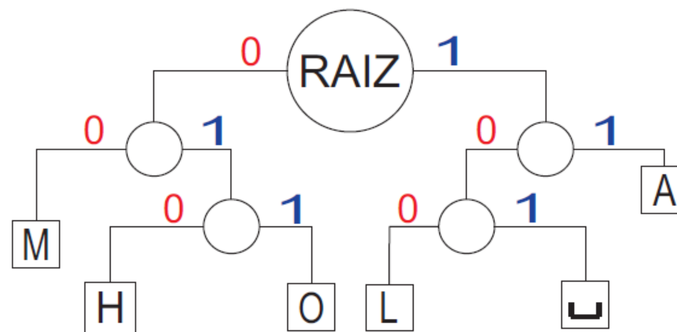
Ahora, se asigna un valor digital, es decir, “1” o “0”, para los caminos que se despliegan hacia la izquierda y el otro valor digital a los caminos que se despliegan hacia la derecha. De esta forma, dependiendo del camino que se recorra para llegar a cada nodo inicial, se obtendrá un código binario único para representar cada uno de los símbolos. La lista de códigos con su respectivo símbolo es conocida como diccionario y es fundamental para el proceso de decodificación.

Dada la forma en que funciona el algoritmo, se clasifica como un codificador semi adaptativo ya que crea un diccionario según los datos que va a comprimir, es decir, crea el diccionario mientras analiza el archivo y después lo comprime. Mediante un ejemplo se explicará la forma de obtener el diccionario. En la Tabla 1 se muestra la frecuencia de los caracteres que conforman la frase “HOLA MAMA”. En la Figura 1 se observa el árbol binario producto de la Tabla 1. En este ejemplo hemos asignado el valor “0” a las ramificaciones hacia la izquierda y “1” hacia la derecha.

Tabla 1: Frecuencia de los caracteres

Símbolo	Frecuencia
H	1
O	1
L	1
A	3
M	2
Espacio	1

Figura 1: Árbol binario



Los símbolos con menor frecuencia están ubicados en los nodos más inferiores y los de mayor frecuencia en los nodos superiores. A partir del árbol se generan caminos desde el nodo superior hacia los nodos inferiores, creando los códigos que serán asignados a cada símbolo. En la Tabla 2 se muestran los códigos asignados cada uno de los símbolos.

Tabla 2: Asignación de códigos

Símbolo	Frecuencia	Código
A	3	11
M	2	00
H	1	010
O	1	011
L	1	100
Espacio	1	101

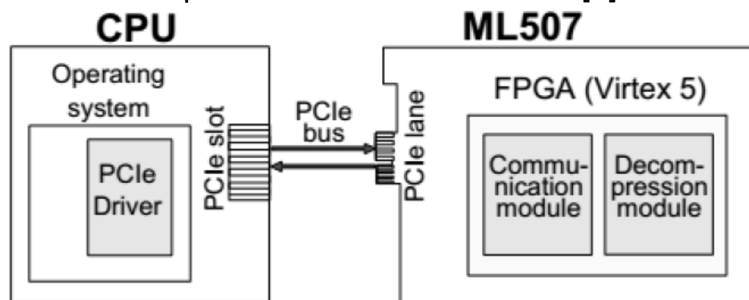
1.4. DECODIFICACION DE DATOS

La decodificación de datos es el proceso inverso a la codificación. Aquí se toman los códigos en el orden de llegada y son comparados con el diccionario para entregar el símbolo respectivo. Al igual que en el proceso de codificación, se pueden aplicar distintas estrategias para su realización, pero básicamente la elección depende en gran parte de la forma en que se hayan codificado los datos. En un sentido amplio, se pretende transformar los datos codificados en información asimilable a la máxima velocidad posible, es decir, decodificar los datos para mostrarlos de un modo comprensible rápidamente.

1.5. PUERTO PCIe (Peripheral Component Interconect Express)

PCIe es una interconexión de alto rendimiento para la conexión de dos dispositivos a alta velocidad. Utiliza un protocolo que permite una comunicación simultánea, mediante la implementación de caminos unidireccionales entre los dos dispositivos [9]. En la Figura 2 se observa la arquitectura de la plataforma utilizada en este proyecto para establecer la comunicación CPU-FPGA. El sistema consta de una CPU Pentium 4 de 3 GHz y una tarjeta de Desarrollo LXT ML507 que incluye un FPGA Virtex 5 XC5VFX70T.

Figura 2: Plataforma de co-procesamiento. Tomada de [1]



2. TRABAJO DESARROLLADO

2.1. DISEÑO DEL DECODIFICADOR HUFFMAN

2.1.1. Determinación de los parámetros del proceso de compresión. Con el fin de determinar la longitud de los códigos *Huffman* que pueden generar al comprimir los datos sísmicos, se realizó un estudio preliminar a doce disparos. Cada disparo está compuesto por 96 trazas sísmicas en formato punto fijo y cada traza cuenta con 3584 datos.

Antes de realizar la codificación *Huffman* fue aplicado un proceso de cuantificación a los datos sísmicos. La etapa de cuantificación se realizó utilizando la siguiente ecuación:

$$Dato_{cuantificado} = ROUND\left[(X - min) * \frac{2^n - 1}{max - min}\right] \quad (1)$$

donde *max* y *min* corresponden a los valores máximos y mínimos del disparo, *X* es el dato de entrada y *n* indica la cantidad de bits de cuantificación. Este proceso se desarrolló usando 8, 9, 10, 11 y 12 bits de cuantificación.

El estudio arrojó como resultado la cantidad de códigos con longitud de uno, dos, tres, cuatro y cinco bits. Este dato se utilizó para determinar la cantidad de registros necesarios para guardar los códigos más cortos y poder implementar la estrategia de decodificación en paralelo.

La literatura sugiere, que para preservar la información geofísica relevante se debe obtener una Relación Señal a Ruido (SNR por sus siglas en inglés) superior a 40dB (entre la traza original y la traza descomprimida) [10]. Para este estudio utilizamos 12 bits de cuantificación, pues nuestras pruebas preliminares (análisis con 8, 9, 10, 11 y 12 bits de cuantificación) mostraron que con este número de bits

aseguramos una SNR superior a los 40dB para cada uno de los 12 disparos. Un número menor de bits de cuantificación aumentaría el factor de compresión, pero implicaría una SNR menor a 40 dB.

La Tabla 3 muestra los resultados del estudio. En ella se puede observar el Factor de Compresión (FC), la SNR, el número de códigos cuya longitud está entre 1 y 5 bits y el porcentaje de datos representados con dichos códigos. Por ejemplo para el Disparo 1, tenemos 3 códigos de 3 bits, 2 códigos de 4 bits y 4 códigos de 5 bits.

Tabla 3: Códigos con longitud entre 1 y 5 bits de los 12 disparos con 12 bits de cuantificación

DISPARO	FC	SNR	1 bit	2 bits	3 bits	4 bits	5 bits	% Datos Representados
DISPARO 1	5,79	43,13	0	0	3	2	4	58,26
DISPARO 2	5,71	43,41	0	0	2	3	6	61,48
DISPARO 3	5,21	45,08	0	0	0	5	6	53,16
DISPARO 4	5,16	43,83	0	0	0	5	7	51,45
DISPARO 5	6,58	45,69	0	1	2	2	2	65,09
DISPARO 6	7,73	40,10	0	1	2	2	3	75,72
DISPARO 7	6,16	45,86	0	1	2	2	2	61,51
DISPARO 8	7,87	40,18	0	1	2	2	2	75,14
DISPARO 9	8,21	44,12	1	0	2	0	2	73,87
DISPARO 10	10,03	40,19	1	0	2	0	2	80,65
DISPARO 11	7,51	44,87	0	1	2	2	2	76,12
DISPARO 12	8,02	44,08	0	2	1	2	1	77,71

En base a estos resultados, el decodificador debe almacenar un máximo de 18 códigos: 1 de un bit, 2 de dos bits, 3 de tres bits, 5 de cuatro bits y 7 de cinco bits.

2.1.2. Almacenamiento de datos. En la Tabla 4 se encuentra un ejemplo de una codificación *Huffman*, con el cual se describirá la forma de almacenar los datos para su almacenamiento en las memorias implementadas en la FPGA. El diccionario se ordena en forma descendente de acuerdo a la frecuencia de los símbolos. Nótese que en este ejemplo el dato más frecuente es "00", al cual le es asignado el código "1".

Tabla 4: Diccionario *Huffman*

SÍMBOLO	CÓDIGO
00	1
01	00
10	011
11	010

Para almacenar la información codificada es necesario contar con tres memorias, cada una de ellas con un tamaño de palabra de 32 bits. A continuación se describe el contenido de cada una de estas memorias:

- **Memoria de datos codificados:** Esta memoria contiene el *string* de los datos codificados. Estos códigos se organizan uno a continuación de otro y se ubican desde la posición más significativa a la menos significativa. En la Figura 3 se observa la forma en la que se almacenan los datos codificados. Si un código no se puede guardar por completo en una posición de memoria, los bits que hagan falta se almacenarán en la siguiente posición de memoria. Por ejemplo, el código "010" resaltado en la Figura 3, no se puede guardar por completo en la Dirección 1 ya que superaría los 32 bits, por lo tanto, el último bit es almacenado en la siguiente dirección de memoria.

Figura 3: Memoria de datos codificados

→ DIRECCION DE ESCRITURA Y LECTURA DE LINEA

Datos	Dirección
010 1 00 1 011 010 1 00 1 00 010 1 00 1 1 010 1 1	DIRECCION 0
011 1 1 1 010 00 00 1 00 1 1 00 00 1 1 1 1 00 1 1 01	DIRECCION 1
0 1 010 1 1 00 1 1 011 010 1 1 0 1 011 010 00 1 01	DIRECCION 2
1 011 011 1 011 010 011 010 011 010 1 011 00	DIRECCION 3

- **Memoria de Símbolos:** Cada símbolo del diccionario se almacena en posiciones de memoria distintas ordenadas de acuerdo a la longitud de su código *Huffman* tal como se muestra en la Figura 4.

Figura 4: Memoria de símbolos

← 32 BITS →

Símbolo	Dirección
000000000000000000000000000000**00**	DIRECCION 0
000000000000000000000000000000**01**	DIRECCION 1
000000000000000000000000000000**10**	DIRECCION 2
000000000000000000000000000000**11**	DIRECCION 3

- **Memoria de Códigos:** En la Figura 5 se observa la forma en la que se almacenan los códigos del diccionario *Huffman*. Los 3 bits más significativos de cada posición de memoria se emplean como bandera para indicar la diferencia de longitud entre dos códigos almacenados en posiciones de memoria consecutivas. Para la Dirección 0, la bandera indica el tamaño de palabra del primer código del diccionario. Esta bandera se utiliza para indicarle al decodificador un cambio en el tamaño del código y determinar cuántos bits del código deben compararse con los datos

codificados. En los 29 bits restantes de cada posición de memoria se escribe el código con el que se representa cada símbolo.

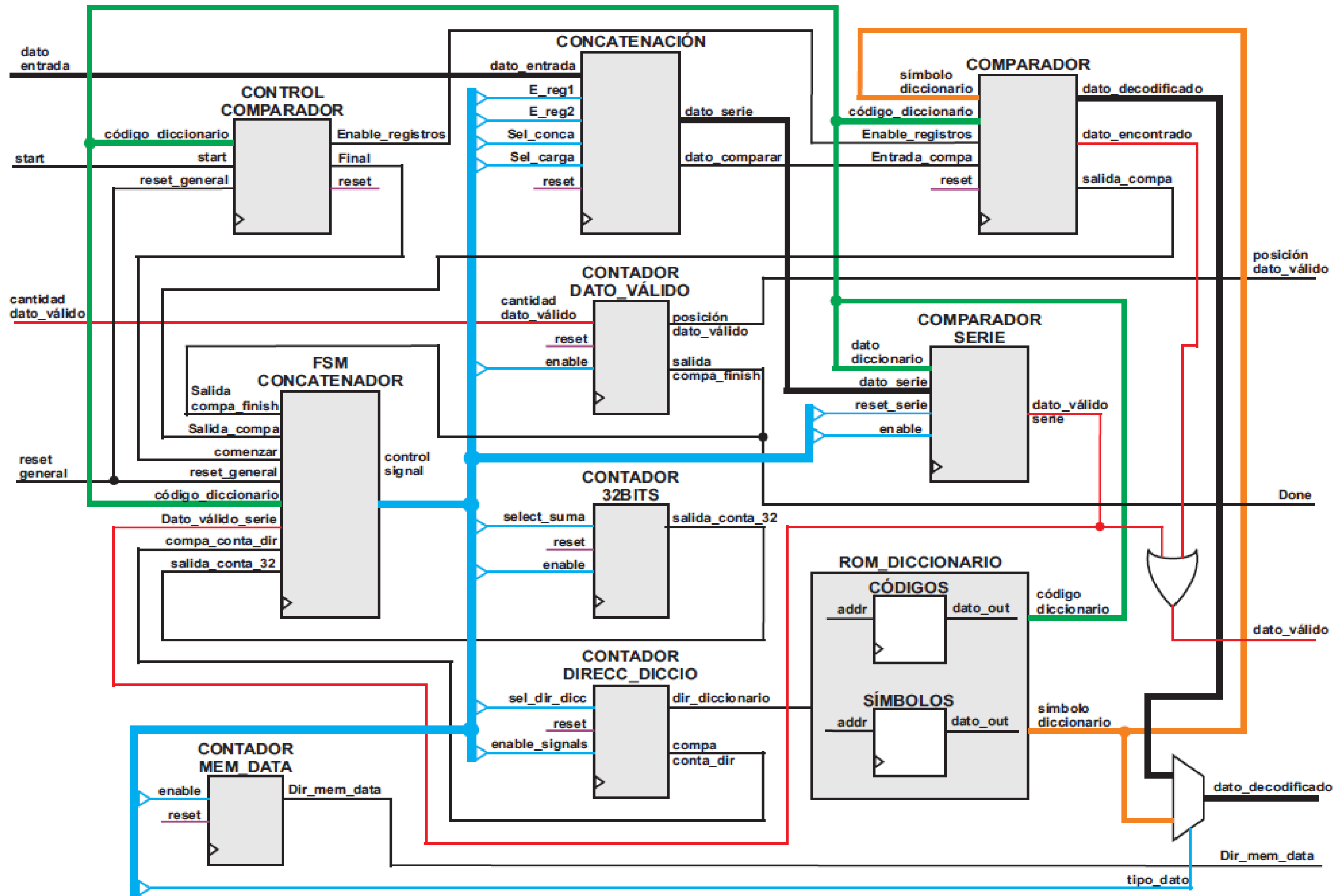
Figura 5: Memoria de códigos

BANDERA	CODIGO (32 BITS)	
001	00000000000000000000000000001	DIRECCION 0
001	00000000000000000000000000000	DIRECCION 1
001	00000000000000000000000000011	DIRECCION 2
000	00000000000000000000000000010	DIRECCION 3

2.1.3. Decodificador Huffman. Una de las desventajas del algoritmo de decodificación es su naturaleza altamente secuencial. Una versión anterior de un decodificador *Huffman* [2] procesa solo los datos representados con un solo bit cada ciclo de reloj mientras que el resto de datos son procesados de forma secuencial.

Para acelerar la decodificación y teniendo en cuenta que más del 50% de los datos codificados se representan con códigos de entre 1 y 5 bits de longitud, nuestro diseño almacena los códigos de uno, dos, tres, cuatro y cinco bits de longitud con sus respectivos símbolos, con el fin de efectuar la decodificación de cada uno de estos datos en un ciclo de reloj. La estrategia de decodificar estos datos cada ciclo de reloj representa uno de los aportes más importantes de este trabajo, ya que la cantidad de datos que se decodifican de forma secuencial se reduce considerablemente. Por esta razón, este diseño presenta una mejora significativa con respecto a la versión diseñada por Fajardo, Angulo, Reyes y Castillo en 2014 [2]. La arquitectura del decodificador se observa en la Figura 6. Cada uno de los módulos se describe a continuación:

Figura 6: Arquitectura del decodificador Huffman



- **Comparador:** Se encarga de comparar los códigos que posean una longitud de 1 a 5 bits con el *string* de datos codificados. Este módulo, además de estos códigos, también guarda en un conjunto de registros los símbolos correspondientes a cada código. Si el Comparador encuentra que uno de los códigos concuerda con el dato estudiado, éste proporciona el dato decodificado.

- **Control_comparador:** Esta unidad de control se encarga de controlar el almacenamiento en el módulo comparador de los códigos que presentan una longitud de entre 1 y 5 bits junto con sus respectivos símbolos. Para esto, lee los 3 bits más significativos de la memoria de códigos, comprueba la longitud del código y determina cuáles registros del módulo comparador se deben escribir y cuáles no.

- **Concatenación:** Este módulo tiene dos funciones específicas. La primera función es cargar una línea de la memoria de datos al inicio del proceso de decodificación o cuando se terminan de procesar los 32 bits que contiene cada línea de la memoria. La segunda función es la de concatenar (dependiendo de la longitud del código estudiado) 1, 2, 3, 4 o 5 bits de la memoria de datos a los datos que se están comparando.

- **Comparador_serie:** Este módulo toma los datos con longitudes mayores a 5 bits, suministrados por el módulo concatenación, los encadena con caracteres iguales a cero para formar una palabra de 29 bits y los compara con los 29 bits menos significativos de la memoria de códigos.

- **Contador_32bits:** Este módulo le indica a la unidad de control FSM_concatenador si es posible efectuar la concatenación de más bits de la línea de datos actual de la memoria de datos o si es necesario cargar una nueva línea.

- **Cont_mem_data:** Este módulo controla la dirección de la memoria de datos.

- **Contador_dato_valido:** Este módulo cuenta la cantidad de datos decodificados y permite finalizar el proceso de decodificación.

- **Contador_direcc_diccio:** Este módulo controla la dirección de las memorias de códigos y de símbolos.

- **FSM_concatenador:** Este módulo es una máquina de estados que se encarga de controlar las formas de comparación de los códigos y se encuentra dividida en dos etapas. La primera etapa se encarga de la decodificación de códigos de longitud 1, 2, 3, 4 o 5 bits manteniendo siempre 5 bits en el código a procesar para compararlo con los códigos guardados en el módulo comparador. Esta parte de la máquina de estados le indica al módulo concatenación que debe realizar este mismo número de concatenaciones. La segunda etapa entra a trabajar cuando el código a decodificar presenta una longitud mayor a 5 bits realizando directamente la comparación del código a procesar con el código suministrado por la memoria de códigos. Este módulo realiza un sondeo constante sobre los 3 bits más significativos de la memoria de códigos para determinar la longitud del código y si es necesario realizar una concatenación o simplemente se debe cambiar la dirección de la memoria para seguir la comparación.

2.2. IMPLEMENTACIÓN DEL DECODIFICADOR HUFFMAN

La implementación se realizó en la tarjeta de desarrollo de Xilinx ML507 la cual cuenta con una FPGA Virtex 5 XC5VFX70T y por medio del puerto PCIex1 se estableció la comunicación entre la FPGA y la CPU.

Esta etapa tuvo como punto de partida el trabajo desarrollado por Obregón y Mantilla en 2014 [11]. Se modificó el bloque *Endpoint* para PCIe, el cual está compuesto por un núcleo que contiene todo el hardware necesario para la transferencia de datos a través del bus PCIe y fueron diseñados 4 bancos de memorias a medida. Cada banco tiene 11 memorias de doble puerto con ancho de palabra de 32 bits. El primer banco de memoria se utilizó para almacenar los resultados del proceso de decodificación, cada memoria dentro de este banco cuenta con 4096 posiciones. El segundo banco se utilizó para almacenar los símbolos del diccionario *Huffman* y cada memoria posee 2048 posiciones. El tercer banco se utilizó para almacenar el *string* de los datos codificados en memorias de 4096 posiciones. El cuarto banco se utilizó para almacenar los códigos del diccionario *Huffman* y cada memoria de este banco cuenta con 2048 posiciones. Además se cuenta con un banco de 32 registros, por el cual se almacenan algunos parámetros importantes para el proceso de decodificación y que son enviados desde la CPU a través del bus PCIe.

2.2.1. Proceso de Comunicación CPU-FPGA. El proceso de comunicación CPU-FPGA se ejecuta por medio de un script desarrollado en lenguaje C++, el cual permite la interacción entre el usuario y el decodificador. Fué necesario el uso de un controlador que permitiera al usuario la escritura en posiciones determinadas de la memoria. Al igual que en [11], se implementó un mecanismo de compensación, llamado *llseek*, que permitió escribir datos en una posición de memoria particular.

Originalmente, el módulo de comunicación solo permite la escritura en una memoria de 4096 posiciones. El decodificador diseñado requiere de cuatro bancos de memorias, por lo cual se tomó la decisión de dividir la memoria original en ocho partes de 512 posiciones cada una. Se tomó una parte de la primera partición para crear el banco de registros. Se usaron dos registros para almacenar los

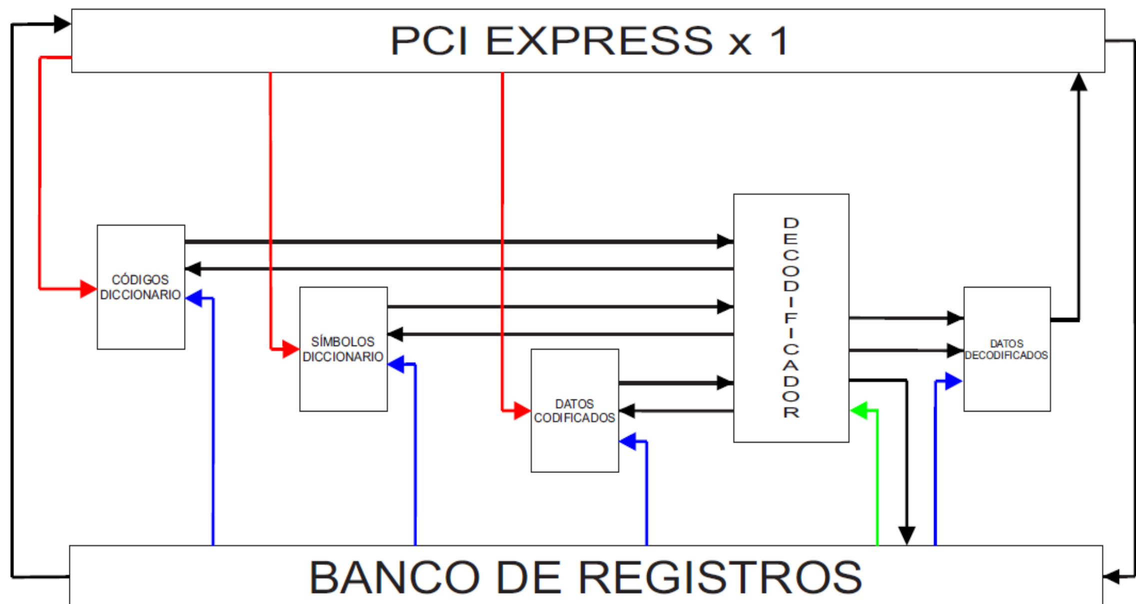
parámetros *Sel_bank* (selector del banco) y *Sel_mem* (selector de memoria), que permiten acceder a todas las memorias. Internamente se ejecutan un conjunto de operaciones lógicas que permiten acceder a los diferentes bancos y memorias del decodificador. Para solucionar el problema de acceso a la nueva cantidad de posiciones, se almacena en un registro el parámetro llamado *Offset* el cual se suma con la dirección física de PCIe logrando alcanzar todas las posiciones.

Otra serie de registros se usaron para guardar parámetros necesarios en la decodificación. Entre estos parámetros se encuentran: la cantidad de datos a descomprimir, señal de *reset*, señal de *start*. Los demás registros se encuentran diseñados para solo lectura, en ellos el decodificador escribe señales como *Done* que indica la finalización del proceso y una señal de error, por si no se encuentra un código durante la comparación.

2.2.2. Implementación de un core. Inicialmente se implementó un decodificador o *core* con el cual se realizaron diferentes pruebas, que permitieron verificar que no existieran errores en la transmisión de datos y el correcto funcionamiento del proceso de decodificación. En la Figura 7 se observa la implementación del decodificador diseñado.

Como primera medida se procedió a enviar los parámetros y los datos a las memorias por el puerto PCIe. En segundo lugar se procedió a la lectura del banco de registros y de las memorias con el fin de corroborar el correcto funcionamiento del enlace CPU-FPGA. En tercer lugar se dio inicio al *core* y se esperó a que finalizara el proceso de decodificación. Por último se procedió a leer nuevamente el banco de registros y la memoria de resultados, con el fin de comprobar la terminación del proceso de decodificación y corroborar la correcta decodificación de los datos.

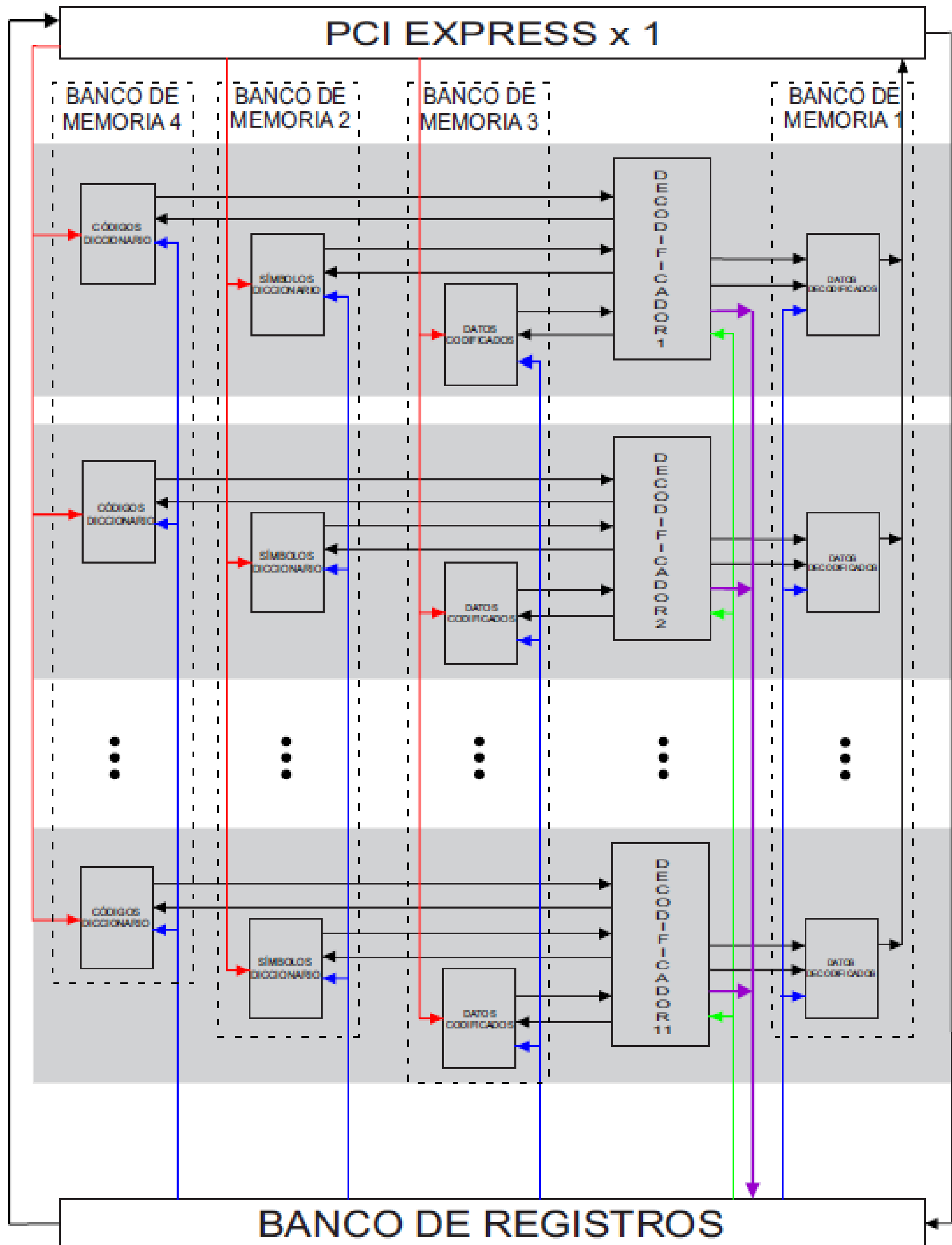
Figura 7: Implementación de un *core*



En una segunda prueba se alteraron los datos almacenados en las memorias con el fin de observar el comportamiento del decodificador ante errores en el proceso de codificación. Al momento de realizar la lectura del banco de registros y de la memoria de resultados, se observó que el decodificador manifestó un error en el proceso de decodificación, comprobando su correcto diseño.

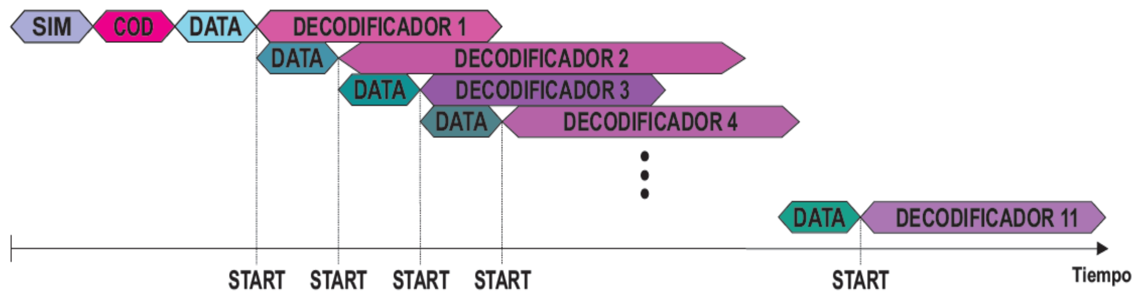
2.2.3. Implementación paralela del proceso de decodificación. Para esta etapa se implementó un circuito que tiene varios decodificadores que trabajan en paralelo. Se logró implementar un máximo de once decodificadores alcanzando el límite de los recursos de almacenamiento de la FPGA. En la Figura 8 se puede observar la comunicación entre los decodificadores, los 4 bloques de memoria, el banco de registros y el puerto PCIe.

Figura 8: Implementación de los decodificadores en forma paralela



En la Figura 9 se observa la línea de tiempo en la que se especifica el proceso de escritura de los bloques de memoria y la decodificación de los datos. En primer lugar se envían los símbolos y códigos correspondientes al diccionario *Huffman* a los bancos de memoria 2 y 4 respectivamente. En segundo lugar, se envía un *string* de datos codificados a la primera memoria de datos codificados para el primer decodificador. En tercer lugar se envía otro *string* de datos codificados a la segunda memoria de datos codificados para el segundo decodificador y así sucesivamente para todos los decodificadores.

Figura 9: Línea de tiempo para la decodificación en forma paralela



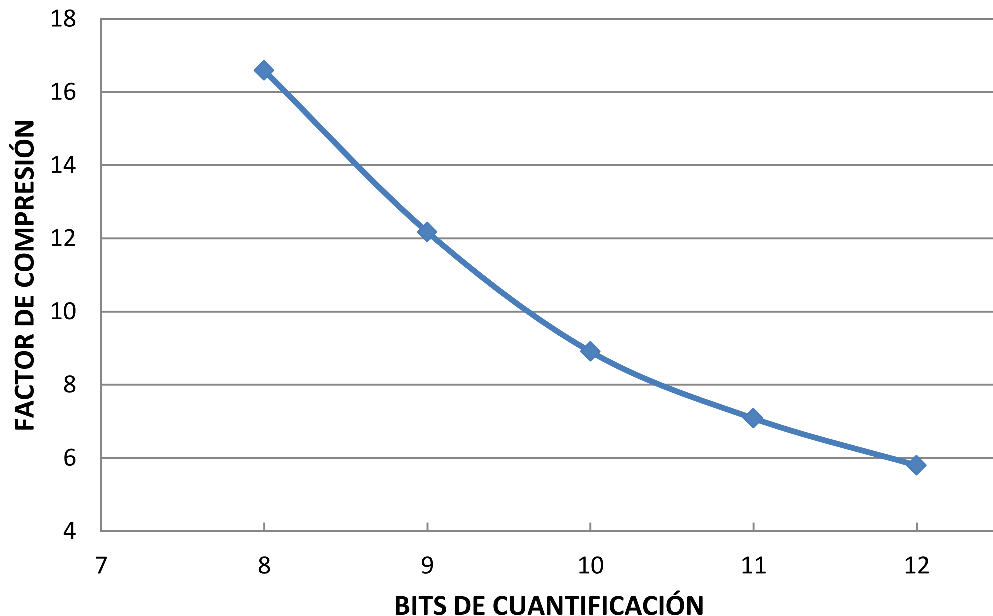
Como se puede observar en la Figura 9, una vez se envía y se almacena la memoria de datos, el decodificador respectivo comienza a operar. El tiempo de decodificación depende de la cantidad de datos a decodificar y de su naturaleza (es decir, dependiendo de la cantidad de ciclos de reloj necesarios para decodificar cada dato), por lo que es posible que en un momento determinado, no se utilicen todos los decodificadores implementados.

Luego de culminar el proceso de decodificación, se procede a la lectura de las memorias de resultados para corroborar el correcto funcionamiento del decodificador. Gracias a una serie de registros encargados de almacenar algunos resultados escritos desde los decodificadores, se puede obtener algunos datos tales como la cantidad de ciclos de reloj que toma cada decodificación o si ocurrió algún error durante dicho proceso.

3. RESULTADOS

Uno de los parámetros a tener en cuenta al comprimir información es obtener un factor de compresión alto y una relación señal a ruido superior a los 40 dB, con el fin de no tener una pérdida de información importante. En la Figura 10 se observa el comportamiento del factor de compresión con respecto a la cantidad de bits de cuantificación. A medida que se incrementan los bits de cuantificación, se reduce el factor de compresión.

Figura 10: Variación del factor de compresión al cambiar los bits de cuantificación



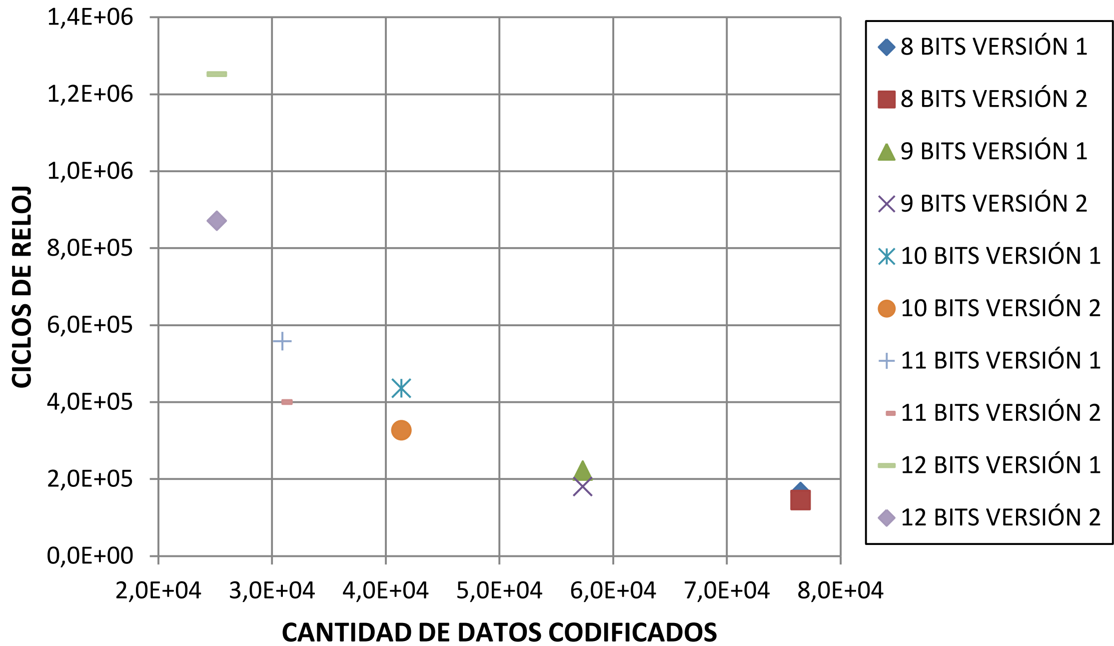
Como se mencionó anteriormente, nuestro diseño permite decodificar en un solo ciclo de reloj, los datos representados hasta con 5 bits de longitud. Si el dato a decodificar presenta una longitud mayor a 5 bits, es necesario realizar la comparación directamente con la memoria de códigos del diccionario, lo que implica aumentar la dirección de la memoria cada vez que sea necesario realizar una comparación.

El decodificador diseñado requería de al menos 3 ciclos de reloj para cambiar la dirección de la memoria y comparar un nuevo dato, y de un cuarto ciclo de reloj cuando se presenta un cambio en la longitud del código. Durante las pruebas realizadas, se logró mejorar el decodificador diseñado, al reducir a 2 la cantidad de ciclos de reloj para cambiar la dirección de la memoria y comparar un nuevo dato. A simple vista no parece importante, pero esta mejora representa un gran significado a medida que se aumenta la dirección de la memoria.

La Figura 11 muestra la cantidad de ciclos de reloj utilizados por las dos versiones del decodificador diseñado, para la decodificación de los datos con 8, 9, 10, 11 y 12 bits de cuantificación. A medida que se aumentan los bits de cuantificación se generan mayores diferencias entre las dos versiones diseñadas. Esto se debe a que al aumentar los bits de cuantificación, una mayor cantidad de datos se representan con códigos de longitudes mayores a 5 bits, aumentando la cantidad de datos que se decodifican de forma secuencial.

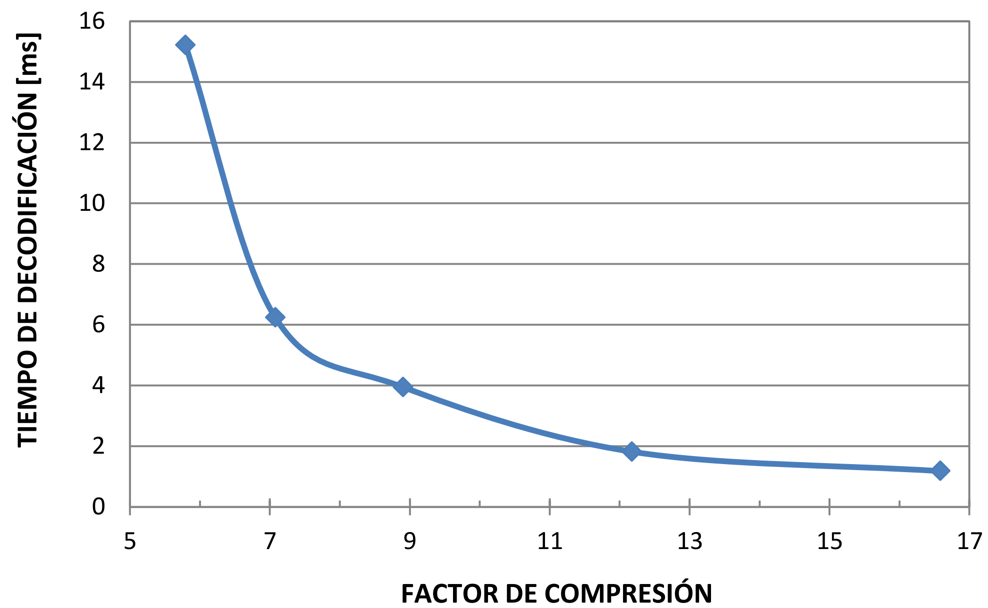
Con 12 bits de cuantificación, la segunda versión del decodificador reduce en un 30% con respecto a la primera versión, la cantidad de ciclos de reloj necesarios para la decodificación de los datos. Para 11 bits de cuantificación, la reducción es del 28%. Para 10 bits de cuantificación, la reducción es del 25%. Para 9 bits de cuantificación, la reducción es del 19%. Para 8 bits de cuantificación, la reducción es del 12%.

Figura 11: Ciclos de reloj para la decodificación de datos



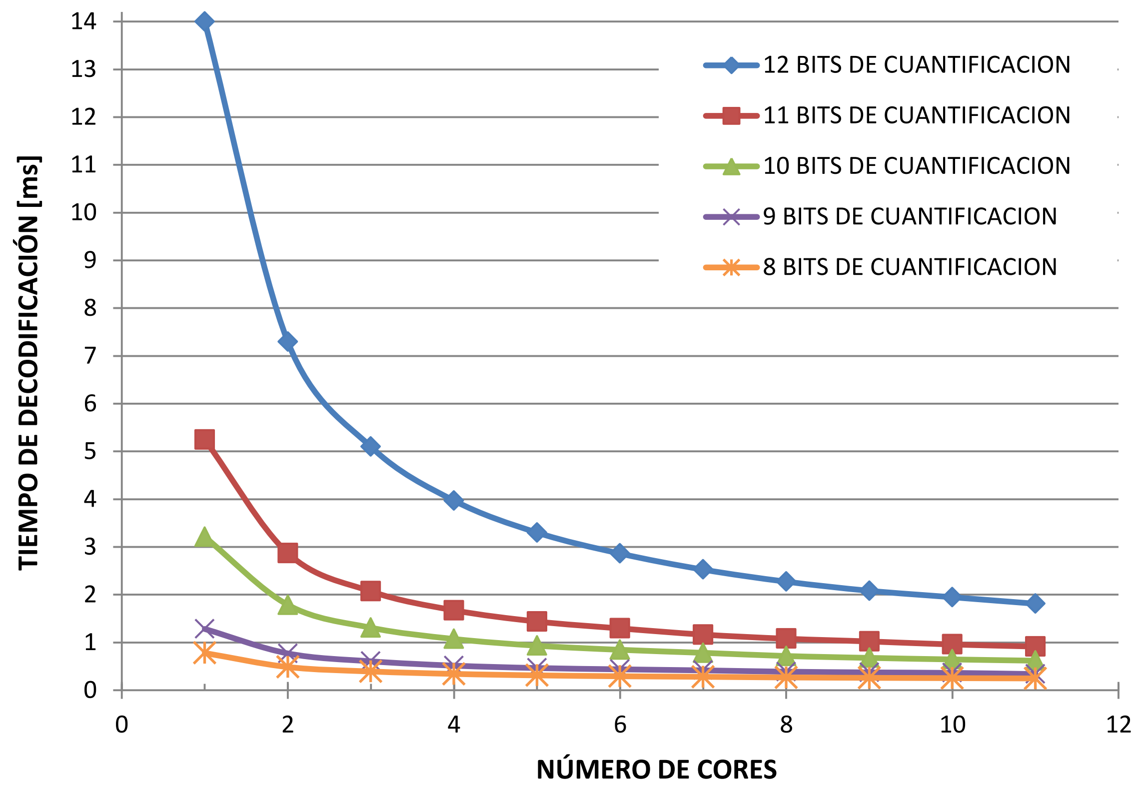
En la Figura 12 se observa la variación del tiempo (en milisegundos) empleado en el proceso de decodificación debido al cambio en el factor de compresión. Para este análisis se tomó como base la decodificación de 25171 datos. Se observa una reducción en el tiempo de decodificación a medida que se aumenta el factor de compresión. Este comportamiento se debe a que al aumentar el factor de compresión, una mayor cantidad de datos se representan con un mismo símbolo, aumentando la frecuencia de repetición y por ende, aumenta la cantidad de datos que se decodifican en solo un ciclo de reloj.

Figura 12: Variación del tiempo de decodificación al cambiar el factor de compresión



En la Figura 13 se observa el tiempo de decodificación dependiendo la cantidad de decodificadores trabajando de forma paralela para diversos bits de cuantificación. Para este análisis se tomó como base la decodificación de 25171 datos. Se puede apreciar una gran reducción en los tiempos de decodificación al aumentar la cantidad de decodificadores trabajando en forma paralela.

Figura 13: Tiempo de decodificación debido a la cantidad de decodificadores implementados



4. CONCLUSIONES

Se ha diseñado un procesador de propósito específico para la decodificación *Huffman* de datos sísmicos en una FPGA. El procesador fue descrito en lenguaje VHDL y el software ISE de XILINX entregó como frecuencia máxima de operación 62.613 MHz.

El diseño implementado presenta una solución al cuello de botella generado en la etapa de decodificación del proceso de trasmisión de datos sísmicos al entregar un dato decodificado cada ciclo de reloj, de esta forma se procesa más del 50% del total de los datos; este porcentaje puede variar de acuerdo al factor de compresión. El factor de compresión se ve directamente afectado por la cantidad de bits de cuantificación.

La frecuencia máxima de operación puede ser mejorada aplicando segmentación al diseño. En este trabajo de grado la frecuencia de todo el sistema estuvo determinada por la frecuencia del IP-*core* del PCI. Sin embargo debido a que la frecuencia del *core* del PCI es menor a la frecuencia del *core* de decodificación, se consideró que no era necesario aplicar segmentación a nuestro *core* de decodificación.

Es difícil tener un promedio de la cantidad de datos decodificados por segundo ya que el tiempo de decodificación depende de la forma de los datos. La decodificación de los datos de forma secuencial es el proceso que utiliza una mayor cantidad de ciclo de reloj, ya que es necesario cambiar la dirección de la memoria de códigos para comparar los datos codificados.

El decodificador está diseñado para almacenar un máximo de 18 códigos (1 de un bit, 2 de dos bits, 3 de tres bits, 5 de cuatro bits y 7 de cinco bits) con sus respectivos símbolos. Esta elección está basada en las pruebas preliminares que

se realizaron a nuestros data sets. Es posible que al intentar decodificar otro tipo de datos se haga necesario modificar la arquitectura del procesador, ya que la cantidad de códigos y la longitud de los mismos pueden variar dependiendo de la naturaleza de los datos. Este tipo de modificaciones también pueden ser producto de usar un número mayor a 12 bits de cuantificación, ya que se pueden generar cambios en la cantidad de códigos con longitudes de 1 a 5 bits.

5. RECOMENDACIONES

Pruebas futuras podrían desarrollarse aplicando algún tipo de transformada a los datos antes de la codificación, por ejemplo la transformada *WAVELET*; esta etapa mejoraría el factor de compresión sin tener que modificar la cantidad de bits de cuantificación. Este proceso implicaría introducir a la arquitectura propuesta una etapa de transformación inversa, la cual tendría que ser diseñada de forma eficiente.

REFERENCIAS BIBLIOGRAFICAS

[1] 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*, Jul. 2006, 787–789.

[2] C. Fajardo, C. Angulo, O. Reyes, J. Castillo, “Fpga implementation of a Huffman decoder for high speed seismic data decompression”, in 2014 Data Compression Conference, No. 0266. Salt Lake, United States.: IEEE Comput. Soc, 2014.

[3] D. Salomon, “Variable-length Cosdes for Data compression”. Springer, 2007.

[4] D. Salomon, “Data Compression The Complete Reference”, 4th ed. Springer, 2007.

[5] T. Chen, “Seismic Data compression: a tutorial," 1995. [Online]. Available: http://www.cwp.mines.edu/Documents/cwpreports/cwp-180.pdf

[6] A. Gercho and R. Gray, “Vector quantization and signal compression”. Kluwer Academic Publiser, 1992.

[7] T. Chen, “Seismic Data compression: a tutorial,” 1995. [Online]. Available: http://www.cwp.mines.edu/Documents/cwpreports/cwp-180.pdf 232, 233

[8] D. Huffman. “A method for the construction of minimum redundancy codes”. Proceedings of the I.R.E, 1098-1101.

[9] Wilen, A. H., Schade, J. P., & Thornburg, R. (n.d.). “Introduction to PCI Express”.

[10] C. Fajardo, O. Reyes, and A. Ramirez, “Seismic Data Compression Using 2D Lifting-Wavelet Algorithms,” vol. 11, no. 21, pp. 53–70, 2015.

[11] I. Obregón y J. Mantilla, “Viabilidad de acelerar la transmisión de datos entre una CPU y una FPGA a través del puerto PCIe (Pheriperal Component Interconnect Express) usando compresión de datos”, Universidad Industrial de Santander, 2014.

BIBLIOGRAFÍA

CHEN, T. Seismic Data compression: a tutorial. 1995. [Online]. Available: http://www.cwp.mines.edu/Documents/cwpreports/cwp-180.pdf 232, 233

FAJARDO, Carlos; ANGULO, Carlos; REYES, Óscar y CASTILLO, Javier. Fpga implementation of a Huffman decoder for high speed seismic data decompression. in 2014 Data Compression Conference, No. 0266. Salt Lake, United States.: IEEE Comput. Soc, 2014.

FAJARDO, Carlos; REYES, Óscar y RAMIREZ Ana. Seismic Data Compression Using 2D Lifting-Wavelet Algorithms. Vol. 11, No. 21, pp. 53–70, 2015.

GERCHO, A. y GRAY, R. Vector quantization and signal compression. Kluwer Academic Publiser, 1992.

HUFFMAN, D. A method for the construction of minimum redundancy codes. Proceedings of the I.R.E, 1098-1101.

OBREGÓN, Iván y MANTILLA, Julián. Viabilidad de acelerar la transmisión de datos entre una CPU y una FPGA a través del puerto PCIe (Pheriperal Component Interconnect Express) usando compresión de datos. Universidad Industrial de Santander, 2014.

SALOMON David. Variable-length Cosdes for Data compression. Springer, 2007.

SALOMON David. Data Compression The Complete Reference, 4th ed. Springer, 2007.

WILEN, A. H.; SCHADE, J. P. & THORNBURG, R. (n.d.). Introduction to PCI Express.

WU, W.; YANG, Z.; QIN, Q. and HU, F. Adaptive Seismic Data Compression Using Wavelet Packets. *2006 IEEE International Symposium on Geoscience and Remote Sensing*, Jul. 2006, 787–789.