UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

MÓDULO HARDWARE DE COMPRESIÓN EN TIEMPO REAL

Jorge Rodríguez Ponce

Tutor: Raúl Esteve Bosch

Cotutor: Vicente Herrero Bosch

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2018-19

Valencia, 2 de septiembre de 2019

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS Gestión ambiental verificada REG.NO. ES-CV- 000030

Resumen

El proyecto abarca el proceso de desarrollo de un circuito digital para compresión. Su inclusión en el experimento NEXT busca la disminución del throughput que conlleva un deadtime menor, y, en consecuencia, una disminución del número de eventos perdidos. Comprenderá una fase inicial para el estudio de los diferentes algoritmos de compresión, comparación de los ratios obtenidos y selección del algoritmo óptimo. Posteriormente, en la fase de diseño se esquematizarán los diferentes módulos necesarios para la implementación, diferenciando el path de control asociado a la máquina de estados correspondiente. Una fase de simulación que muestre las formas de onda para una temprana corrección de errores y cambios en el diseño inicial. Finalmente, la fase de verificación, donde se estructurará un banco de pruebas para comprobar el correcto funcionamiento de forma amplia, mediante el procesado de un archivo de datos de un evento procedente del experimento, con su posterior decodificación y comparación con los valores iniciales. De esta manera, se completará un flujo de diseño completo que proporcione un módulo digital funcional y listo para su implementación.

Abstract

This project covers a whole development of a digital circuit for compression. Its incorporation seeks a throughput decrease that leads to a lower deadtime and, in consequence, less lost events in the NEXT experiment. It shall consist of a study phase, in which many compression algorithms will be considered, making comparisons of the different compression rates and selecting the optimum one. Subsequently, in the design phase, the different modules necessary for the implementation will be schematized, differentiating the control path associated to the corresponding state machine. A simulation phase that shows the waveforms for an early correction of errors and changes in the initial design. Finally, the verification phase, where a test bench will be structured to check the correct operation in a wide way, through the processing of a data file of an event coming from the experiment, with its later decoding and comparison with the initial values. Thus, a complete design flow which provides a functional digital module ready for implementation will be completed.

Resum

El projecte abasta el procés de desenvolupament d'un circuit digital per a compressió. La seua inclusió en l'experiment NEXT cerca la disminució del throughput que comporta un deadtime menor, i, en conseqüència, una disminució del nombre d'esdeveniments perduts. Comprendrà una fase inicial per a l'estudi dels diferents algorismes de compressió, comparació dels ràtios obtinguts i selecció de l'algorisme òptim. Posteriorment, en la fase de disseny s'esquematitzaran els diferents mòduls necessaris per a la implementació, diferenciant el path de control associat a la màquina d'estats corresponent. Una fase de simulació que mostre les formes d'ona per a una primerenca correcció d'errors i canvis en el disseny inicials. Finalment, la fase de verificació, on s'estructurarà un banc de proves per a comprovar el correcte funcionament de forma àmplia, mitjançant el processament d'un arxiu de dades d'un esdeveniment procedent de l'experiment, amb la seua posterior descodificació i comparació amb els valors inicials. D'aquesta manera, es completarà un flux de disseny complet que proporcione un mòdul digital funcional i llest per a la seua implementació.

Índice

Contenido

Capítulo 1. Introducción

NEXT es un programa experimental que desarrolla tecnología de alta presión de Xenón con amplificación electroluminiscente [1] para la búsqueda de decaimiento doble beta sin neutrinos. Se encuentra en funcionamiento desde octubre de 2016, en el Laboratorio Subterráneo de Canfranc.

La comprobación de esta hipótesis, la cual fue propuesta por el físico italiano Ettore Majorana, implicaría que el neutrino sea su propia antipartícula y serviría como explicación a la composición del universo por materia, y no por antimateria.

Para llevar a cabo el experimento se hace uso de un TPC (Time Projection Chamber), como la presente en la Figura 1, que consiste en un volumen cubierto por gas, en cuyo centro se encuentra un cátodo que lo divide en dos partes, cada una de las cuales posee un ánodo. Entre ellos se crea un fuerte campo eléctrico, provocando que una partícula cargada que atraviese el volumen ionice los átomos del gas. Los electrones resultantes de este proceso son empujados hacia el ánodo, en cuyo plano podrán ser detectados [2].

Específicamente para NEXT ha sido diseñado un plano de PMTs (photomultipliers) para la medida en energía de los distintos eventos. El sistema de adquisición de datos (DAQ), basado en una FPGA, registra continuamente los valores, que son almacenados en un buffer circular para cada uno de los PMTs. Posteriormente, a partir de un trigger se seleccionan diferentes eventos en función de la estimación inicial de energía.

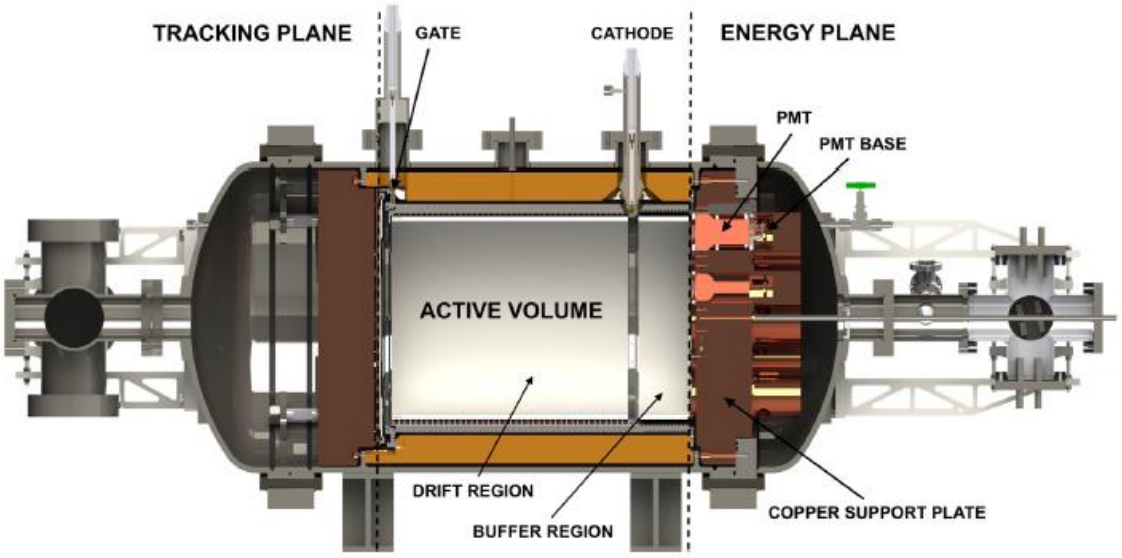

Figura 1. PTC for Next-White detector

Tras la digitalización y paquetización de la señal, el flujo generado es de aproximadamente 10 MByte/s para el total de los 12 PMTs utilizados para un muestreo de 10 Hz. Dicha información es enviada a una granja de ordenadores que la procesarán a través de un enlace GbE.

El sistema adquisición de datos [3] se encuentra compuesto por una serie de Front-end cards (FECs), unidas a dos diferentes tipos de front-ends. Por una parte existe la DTC card, que actúa como interfaz de FE-SiPM; mientras que por otro lado se encuentran las tarjetas ADC, que hacen de interfaz con tarjetas FE-PMT. A un nivel de jerarquía superior se encuentra DATE PC, que muestra los resultados enviados al mismo por un enlace de GbE.

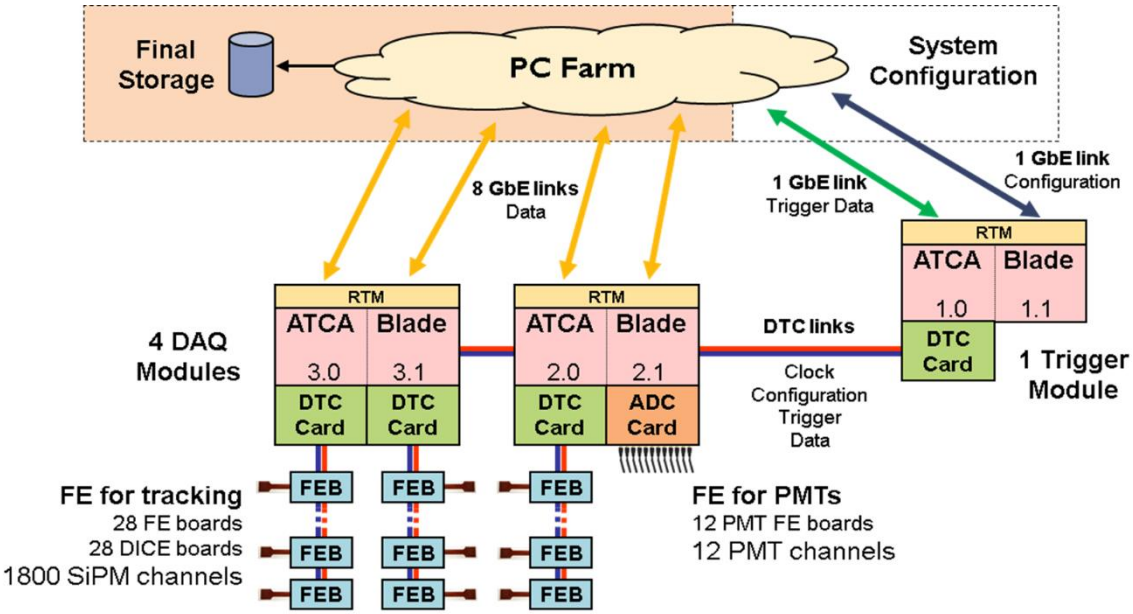

Figura 2. Esquema de Next

En el modelo actual para el experimento, la densidad de datos es aún administrable. Sin embargo, la evolución planeada para el mismo, NEXT-100, supondrá un incremento drástico en el número de PMTs a utilizar y, en consecuencia, será necesario una compresión de la información generada para hacer posible una transmisión directa y a tiempo real con la granja de ordenadores.

El proyecto buscará, por tanto, una implementación en FPGA, concretamente el modelo Xilinx Virtex-6 FPGA, de un módulo compresor previo a la interfaz de transmisión.

Capítulo 2. Objetivo del Proyecto

El enfoque dado al proyecto contempla el desarrollo completo de un sistema digital para su implementación en un modelo actual y en funcionamiento.

Comprenderá las fases de estudio, diseño, simulación y verificación

En primera instancia, el estudio consistirá en la búsqueda y valoración de los diferentes algoritmos en consideración, con sus respectivas comparaciones para la selección de aquel que mejor encaje con las características exigidas. Se considerará relevante la tasa de compresión, así como, la complejidad de implementación y la posible latencia en su diseño. Este estudio se llevará a cabo mediante la programación de los algoritmos, realizando variaciones para incluir la compresión con diferentes tamaños de palabra o, incluso, la combinación de varios de ellos para mejorar los resultados.

Una vez decidido se continuará con la fase de diseño. Mediante diagramas de flujos, se identificarán los diferentes pasos necesarios para llevar a cabo correctamente el algoritmo. Tras la inclusión de una máquina de estados adecuada, se definen las tareas a realizar en sus diferentes etapas.

Cada una de estas acciones se identificará con un módulo digital con un número determinado de salidas y entradas, así como de las señales de control pertinentes. Tras ello, se obtendrá un esquema completo del diseño y comenzará el proceso de programación de los distintos módulos en Verilog.

Mediante simulaciones, tanto individuales como globales, se buscará identificar errores tempranos apreciables a través de las formas de onda y su posterior corrección. Para la finalización es necesaria una fase de verificación consistente y completa. Por ello, la última fase se corresponderá con la construcción de un banco de pruebas que permita comprobar todo un archivo de datos propio de un evento, su compresión de acuerdo con el algoritmo seleccionado y su posterior descompresión y comparación con el archivo inicial. De esta forma, el flujo de diseño se habrá realizado en su totalidad y se podrá afirmar que el funcionamiento del mismo es correcto.

Capítulo 3. Metodología de trabajo

El flujo de trabajo seguido es el presente en la Figura 3. Las etapas no son excluyentes, puesto que la identificación de un error en cualquiera de ellas supondrá el retroceso a una previa y la modificación correspondiente.

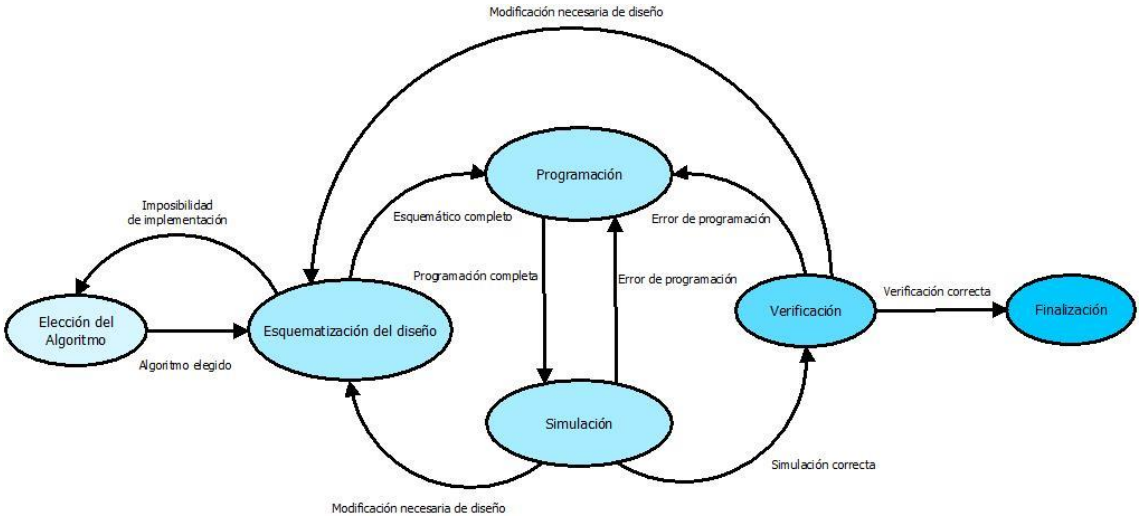

Figura 3. Flujo de Trabajo

Para la gestión del proyecto se ha utilizado la herramienta GitKraken. Consiste en una interfaz gráfica desarrollada por la compañía Electron que permite llevar un seguimiento de los diferentes repositorios y ramas creadas durante todo el proyecto. De forma adicional es posible la conexión con Github, por lo que es posible acceder de manera remota al mismo.

La versión empleada es 5.0.4 para Windows 10.

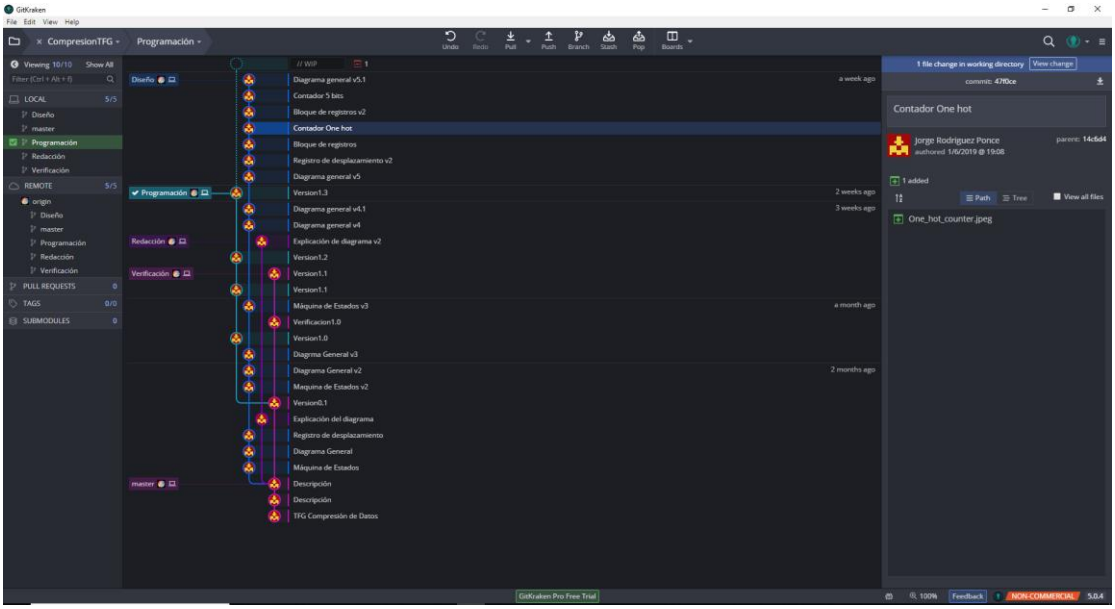

Figura 4. Interfaz GitKraken

Para el acceso al repositorio creado desde Github se debe acceder a la dirección presente en [4].

3.1 FPGA Virtex-6

El proyecto será implementado en una FPGA (Field programable Gate Array) Virtex-6. En concreto se trata del modelo XC6VLX240T.

Una Field Programmable Gate Array (FPGA) es un dispositivo semiconductor compuesto de matrices de bloques lógicos configurables (CLBs) cuyas conexiones son configurables. En su mayoría son reprogramables para adaptarse a una cierta aplicación o funcionalidad tras su fabricación.

A partir de los datos de catálogo se pueden obtener ciertas características del dispositivo a utilizar [5].

	Celdas Lógicas	CLBs		Bloques de RAM		DSP Slices
		Slices	RAM distribuida	18 Kb	36 Kb	
XC6VLX240T	241.152	37.680	3.650	832	416	768

Tabla 1. Características Virtex-6 XC6VLX240T

Se han incluido en la Tabla 1 solo aquellos elementos que tendrán influencia en el desarrollo del proyecto. Cabe resaltar la existencia de DSP Slices, puesto que cada uno de ellos contiene un multiplicador 25 x 18, un sumador y un acumulador; así como el hecho de que no se dispondrá de la FPGA en su totalidad, ya que el DAQ desarrollado en NEXT ocupa gran parte de la misma. En conclusión, será viable realizar ciertas operaciones aritméticas haciendo uso de los multiplicadores y sumadores embebidos, pero buscando una reducción del número total de elementos utilizados.

3.2 Elección del Algoritmo

3.2.1 HDF Viewer

Lector para los archivos de datos proporcionados por parte del experimento, usados tanto en la elección del algoritmo de compresión como la simulación funcional del proyecto.

Debido a la gran cantidad de información presente en cada uno de los eventos que serán evaluados se necesita un formato de archivo que presente una gestión eficiente de ellos; siendo en este caso la extensión h5.

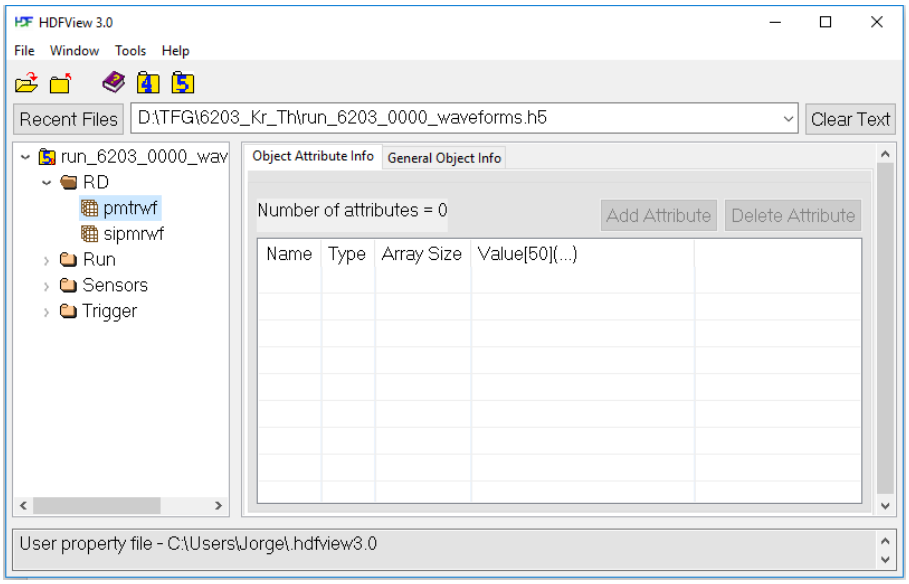

Figura 5. Gestor de archivos HDF Viewer

3.2.2 Jupyter Notebook

Plataforma web de código abierto que permite la creación de documentos de código. Haciendo uso de la herramienta Python es posible crear códigos modulares y ejecutarlos de manera secuencial.

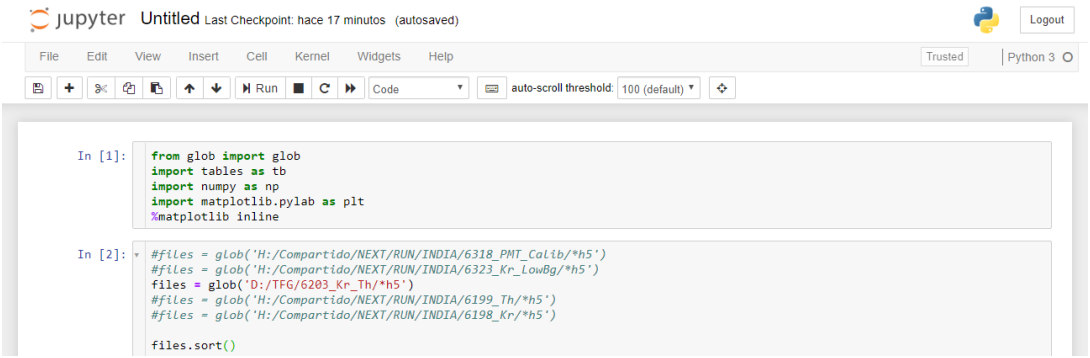

Figura 6. Interfaz Jupyter Notebook

Será utilizada para la simulación de los diferentes algoritmos de compresión utilizados, así como el cálculo de los ratios de compresión.

La principal ventaja que presenta dicha plataforma es la posibilidad de una ejecución separada e individual de cada uno de los bloques programados, como también la presentación de las diferentes gráficas de datos de manera sencilla.

3.3 Programación y simulación

3.3.1 Xilinx ISE

Para la programación y descripción del sistema digital propuesto se ha seleccionado la herramienta ISE Design Suite de la compañía Xilinx, con soporte para Virtex-6. La versión que se ha utilizado es la 14.7, actualizada para el sistema operativo Windows 10 el día 26 de febrero de 2018. Su uso se lleva a cabo a través de una máquina virtual, en concreto con la aplicación Oracle VM VirtualBox 5.1.30.

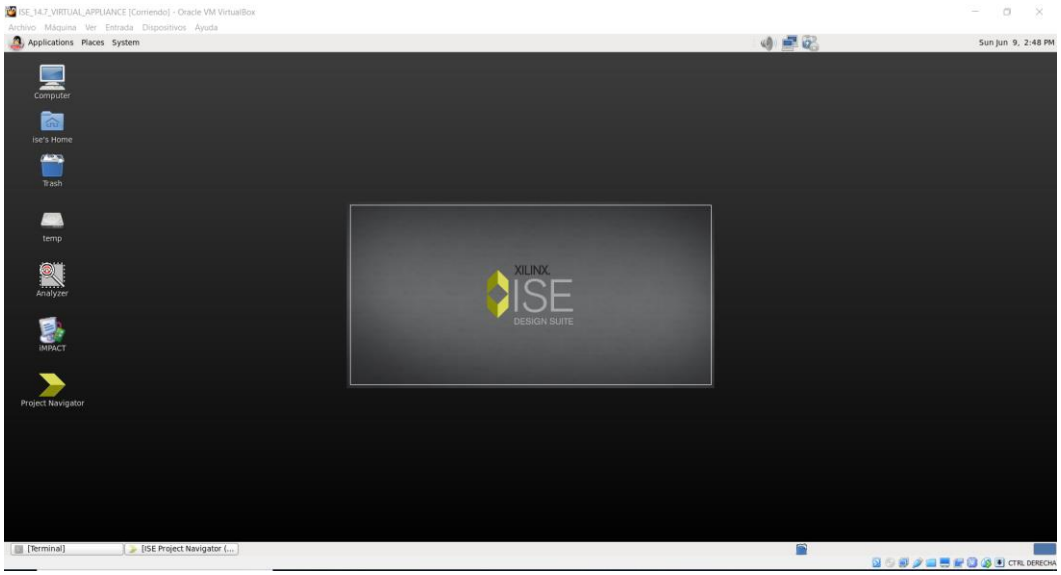

Figura 7. Escritorio Máquina Virtual

Esta elección se ha llevado a cabo para facilitar su implementación y reducir el número de incompatibilidades debido a que el resto de los elementos digitales diseñados para NEXT han sido también desarrollados con esta aplicación.

Dentro de la máquina virtual se encuentra Project Navigator, un gestor de archivos. En esta aplicación existen dos vistas para la jerarquía de archivos. Primeramente, la vista de implementación, la cual será utilizada para la creación y edición de los archivos del proyecto presente en la Figura 8.

Al seleccionar la opción de añadir una fuente a la jerarquía se permite escoger el formato a utilizar. Para el desarrollo de este proyecto se han usado principalmente los archivos de tipo “Verilog Modulo”; dichos archivos permiten una descripción en lenguaje Verilog de cada uno de los bloques lógicos descritos en el diseño.

En la creación de cada uno de los archivos que pueden tomar diferentes extensiones, como las indicadas en la Figura 9, se proporcionará un “Wizard” con opciones para seleccionar y nombrar las entradas y salidas de los módulos, así como para indicar si se corresponden o no con buses, y en dicho caso, su tamaño correspondiente, tal como se aprecia en la Figura 10.

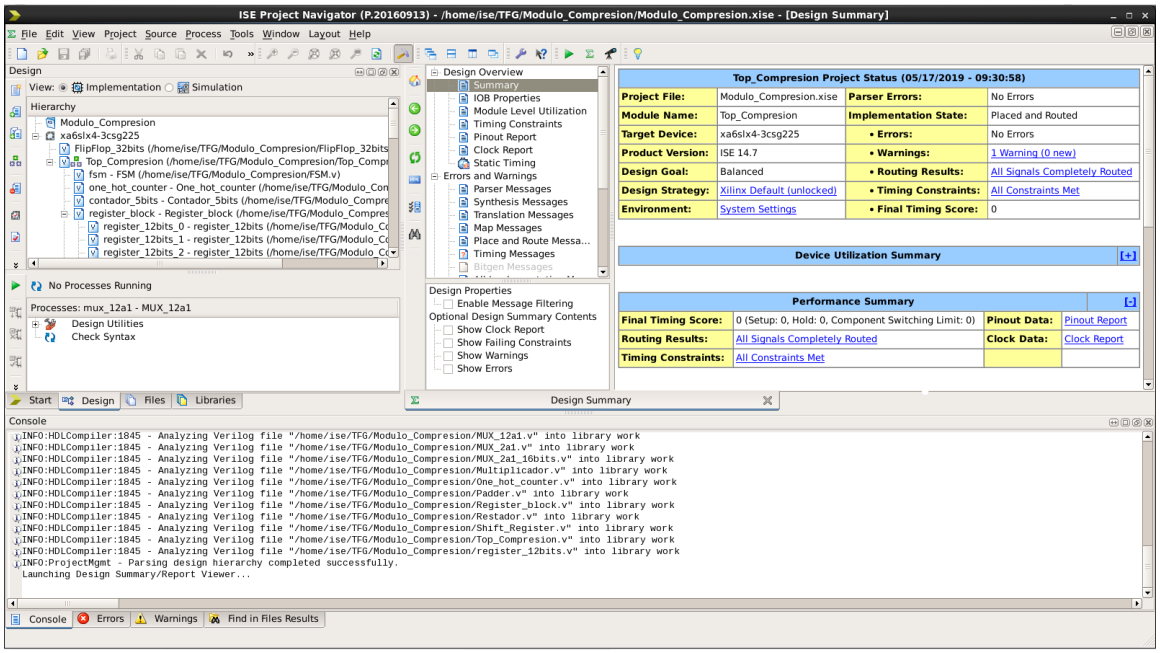

Figura 8. Interfaz Project Manager

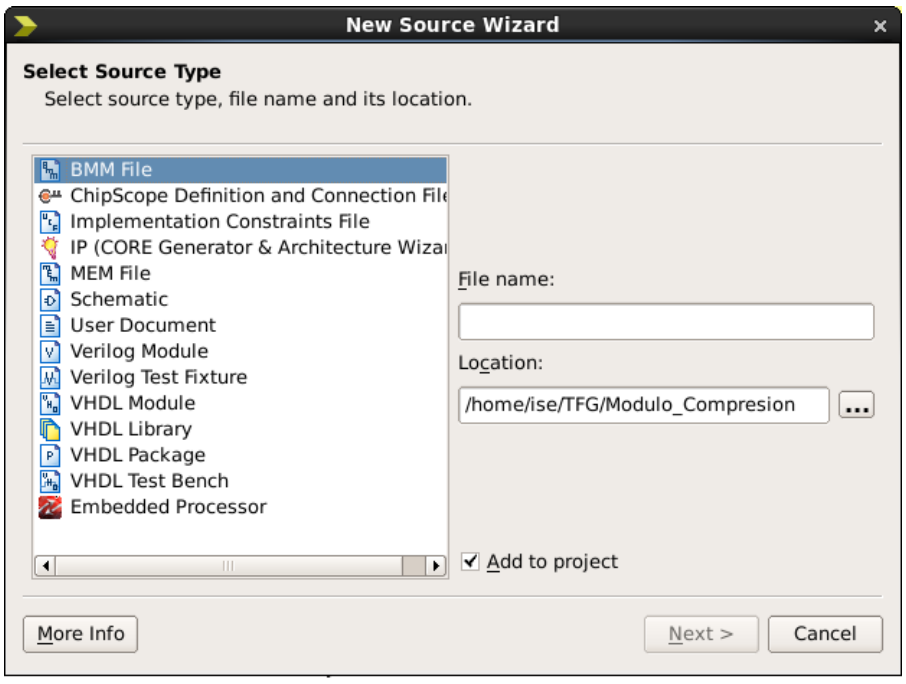

Figura 9. Source Wizard

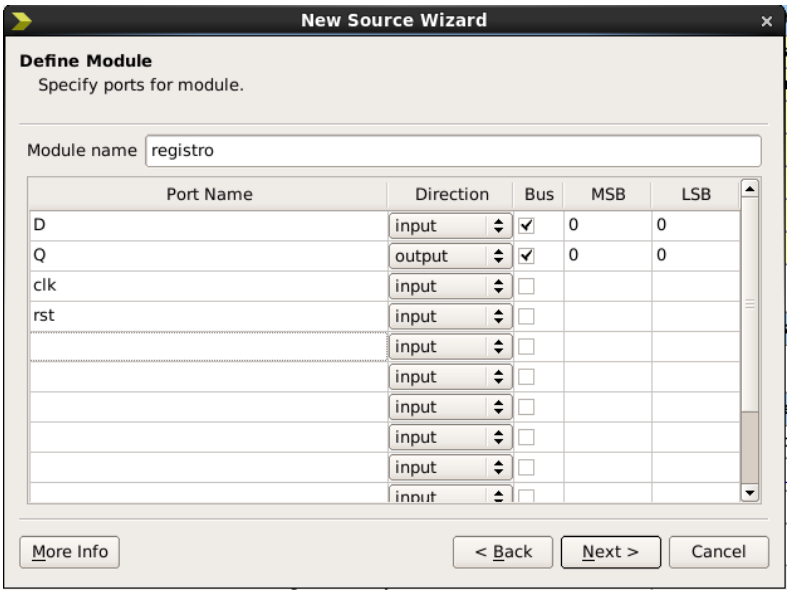

Figura 10. Configuración E/S de Source Wizard

Una vez completado el proyecto, es posible realizar la síntesis correspondiente del mismo. Con la consola inferior se muestran los errores, advertencias e información adicional.

La otra vista que se encuentra en Project Navigator es la de simulación. Esta permite la inclusión de un archivo adicional a los incluidos en la síntesis con propósito de descripción de una secuencia de estímulos, control de tiempos y mensajes en pantalla. Una vez se haya realizado el archivo de estímulos apropiado puede ser lanzada una simulación. Dicha simulación será llevada a cabo con la aplicación Isim, incluida en la herramienta de diseño adquirida, y cuya interfaz es visible en la Figura 11.

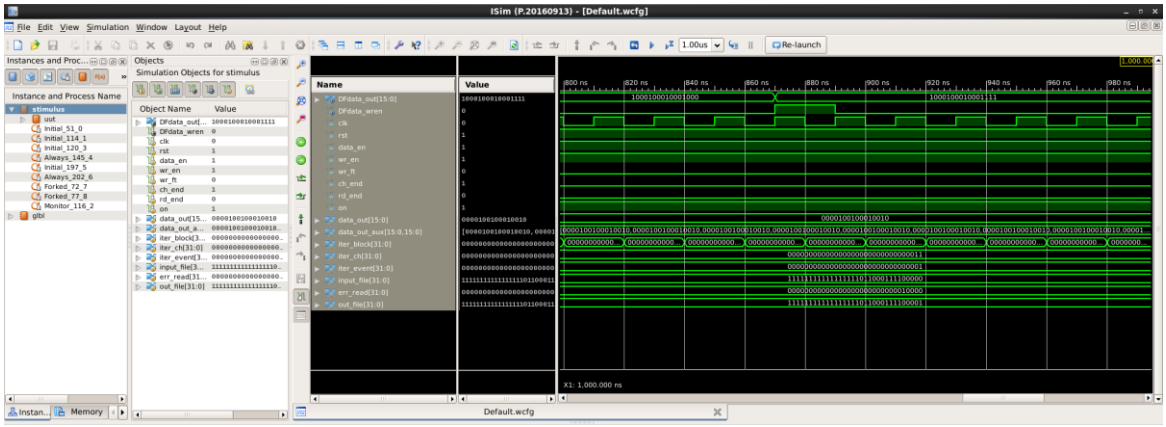

Figura 11. Interfaz Isim

Esta funcionalidad puede ser utilizada de manera previa a un testbench propiamente construido para corregir de manera manual posibles errores en la adquisición o sincronía de las señales, así como el correcto funcionamiento de los distintos módulos incluidos.

Previamente a cada una de las simulaciones se especifican qué señales internas del proyecto se quieren añadir a las formas de onda para su visualización.

3.4 Testbench

3.4.1 QuestaSim

Se ha utilizado QuestaSim, visible en la Figura 12, para la creación y ejecución del banco de pruebas, en concreto la versión 10.4 de 64 bits para Windows.

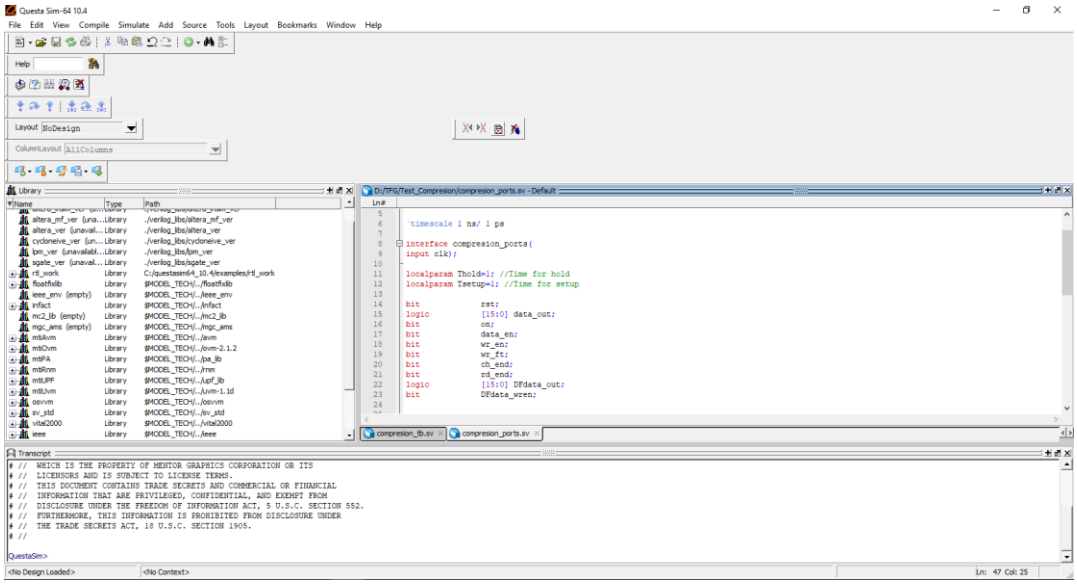

Figura 12. Interfaz QuestaSim

Mediante la creación y edición de archivos con extensión .sv, escritos en el lenguaje de programación SystemVerilog, es posible la simulación, recopilación de valores de las variables internas del módulo y comprobación de su correcto funcionamiento tras la comparación del resultado obtenido.

Por tanto, y haciendo un uso combinado de los archivos de texto y las funciones propias de verificación, el flujo de datos a la entrada y salida podrá ser monitorizado automáticamente e indicar mediante avisos o errores cualquier posible inexactitud.

Capítulo 4. Compresión de datos

Durante los últimos años, el desarrollo paulatino tanto de la velocidad de procesamiento en los ordenadores, así como de los tamaños de las memorias puede llevar a pensar que la compresión puede ser innecesaria. Sin embargo, en el uso intenso de las redes por parte de muchos usuarios, quienes exigen cada vez mejores prestaciones, y en concreto mayores velocidades de transmisión de datos, aparece el principal limitante de estas.

Es en ese punto donde la compresión es introducida como una solución óptima. El principal motivo para ello es que el proceso de compresión, transmisión y descompresión de datos es más rápido que un único proceso de transmisión sin compresión.

Es posible expresar dicha ganancia en velocidad a partir de la ecuación (4.1)

$$\boldsymbol{\rho = \frac{2 \cdot \frac{D}{c} + \frac{r \cdot D}{b}}{\frac{D}{b}} = \frac{\frac{2 \cdot b \cdot D + r \cdot c \cdot D}{c \cdot b}}{\frac{D}{b}} = \frac{2 \cdot b + r \cdot c}{c}} \qquad (4.1)$$

En la ecuación, *D* indica el número de bits que posee el mensajee que se desea transmitir, *b* hace referencia a la velocidad de transmisión en bps, *r* es el ratio de compresión del algoritmo que utilizamos, *c* es la velocidad tanto de compresión como de des compresión (puesto que las consideramos simétricas) y ρ representa la fracción de tiempo que se tardaría en enviar de forma completa el archivo comprimido con relación al archivo sin comprimir.

En el resultado final es posible observar cómo dicha tasa no depende de la cantidad de información a transmitir (*D*), sino de las diferentes compresiones utilizadas y de la velocidad de transmisión. En prácticamente una totalidad de los casos, la compresión supone una gran ventaja en muchos aspectos, ya que, aparte de la transmisión, también se deben tener en cuenta el almacenamiento de información.

4.1 Algoritmos de compresión

La compresión de datos puede dividirse en dos clases principales: compresión sin pérdidas y compresión con pérdidas [6]. En el caso sin pérdidas es posible obtener, a partir de un mensaje comprimido de *N'* datos, los *N* datos del mensaje original; mientras que con pérdidas solo es posible obtener *N** datos, siendo este una aproximación de *N*.

Los campos de uso de ambos tipos son marcadamente distintos. Por una parte, para compresión de datos presentes en el envío de texto o claves se exige la inexistencia de cualquier pérdida, puesto que esta es considerada inaceptable y perjudica la integridad del mensaje. De forma contraria, están presentes en situaciones donde una reducción en el flujo de datos no supone un problema para la recepción del mensaje, puesto que este es tolerante a pérdidas. En estos casos son incluidos el envío de vídeo o audio, así como imágenes, donde un archivo original puede ocupar grandes tamaños de memoria, y una compresión con pérdidas permite obtener archivos reducidos a costa de pérdidas de calidad.

Además de su clasificación en función de las pérdidas, también es posible distinguir la adaptabilidad de un método de compresión. Un mecanismo no adaptable es rígido y no permite la modificación de ninguno de sus parámetros de operación como resultado del análisis de los datos a comprimir; mientras que uno adaptable realiza un proceso de examinación y modificación de los parámetros de acuerdo con ellos.

4.1.1 Algoritmos con pérdidas

Este tipo de compresión consiste en la obtención de una copia de menor calidad que la entrada original, a partir de la elección por parte del algoritmo de la información a descartar. Posteriormente, el resultado obtenido puede ser comprimido por diversos métodos. A pesar de que el resultado descomprimido no sea idéntico al original, es considerado suficientemente parecido para el propósito de la aplicación y además obtiene altos ratios de compresión.

Entre los algoritmos presentes en esta clasificación se encuentran la transformada de coseno discreta (DCT) y la transformada de wavelet discreta (DWT). [7]

4.1.1.1 Transformada de coseno discreta

Mediante la expresión de una secuencia finita de datos representados como la suma de funciones coseno a diferentes frecuencias, la DCT es similar a una TF (Transformada de Fourier), pero a diferencia de esta aplica solo cosenos para aproximar la señal, lo cual implica una reducción de las funciones usadas y una mayor eficiencia.

Ejemplos de este tipo de compresión están presentes en MPEG-2, H264, HEVC o WebM.

4.1.1.2 Transformada de wavelet discreta

Una implementación de la transformada de wavelet discreta utiliza un grupo finito de escalas y translaciones basadas en ciertas reglas predefinidas. Descompone la señal en sets ortogonales entre sí.

4.1.2 Algoritmos sin pérdidas

La medida de la información surge a raíz de las probabilidades y, por tanto, existen compresores que utilizarán las características estasdísticas de la fuente para obtener una codificación óptima. A estos compresores se les denomina compresores estadísticos.

Parten una cantidad finita de mensajes, de los cuales son conocidas sus diferentes probabilidades, ya sea de forma experimental o fijas. Su objetivo es dar una codificación a los diferentes mensajes de la fuente de tal manera que se busque, haciendo uso de la redundancia de la información, obtener la mejor compresión posible.

Por otro lado, existen los algoritmos sustitucionales que poseen un diccionario con las cadenas de los diferentes mensajes. Cada cadena posee un índice y, en caso de que ya se haya enviado el mensaje con anterioridad, se envía únicamente el índice que precede a dicha cadena en el diccionario.

De forma adicional, y para casos concretos como es el presente de compresión de una señal, existen ciertos algoritmos que utilizan características propias como son su nivel de continua, la diferenciabilidad de la misma o incluso la similitud entre varias de ellas.

A continuación, se presentan varios de los algoritmos que se analizarán posteriormente.

4.1.2.1 Compresión Delta

Tanto en ingeniería como en matemáticas, la letra delta Δ ha denotado un cambio en una variable. La codificación delta hace referencia a técnicas que almacenan los datos como la diferencia de valores sucesivos en lugar de guardarlos directamente. El primer valor sí se corresponde con el original, mientras que los posteriores presentan su incremento o decremento con respecto al anterior.

Su uso está presente en multitud de compresiones de datos cuando estos presentan transiciones suaves, con un cambio pequeño entre los valores adyacentes.

4.1.2.2 Compresión Baseline

En muchas series de datos existe cierta correlación entre ellos, oscilando alrededor de un valor intermedio. En el caso de las señales, dicho valor se corresponde con el valor DC. La codificación de conjuntos de valores que poseen un alto valor medio puede conllevar un uso excesivo de bits para su almacenamiento. Sin embargo, pueden presentar diferencias entre sus valores máximo y mínimo muy inferiores al indicado nivel de continua.

La compresión baseline almacena por separado la moda o media del conjunto, de manera que todos los valores que se encuentren en él sean almacenados como una diferencia respecto a dicho baseline, consiguiendo así reducir el número de bits utilizados para cada uno de los valores.

4.1.2.3 Run Length Encoding

Algoritmo de compresión simple que permite el almacenamiento de series ordenadas de datos. Las secuencias de un mismo dato presente en la serie son codificadas como un solo valor y un contador indica las repeticiones de dicho valor de forma consecutiva. Está presente en la compresión de numerosas imágenes, animaciones y señales que mantienen valores durante un número considerable de intervalos de muestras.

Cabe considerar que es solo eficiente para grandes cantidades de datos, puesto que para archivos pequeños el overhead introducido será mayor a cualquier compresión aplicada.

En el caso presente de una señal proporcionará una compresión eficiente si en los tramos más monótonos de la señal aparecen de manera reiterada los mismos valores.

4.1.2.4 Compresión de Huffman y Fano

Este algoritmo es el encargado de construir un árbol que represente de forma inequívoca las diferentes codificaciones. Se comienza en la parte superior del árbol y, posteriormente se avanza por las diferentes bifurcaciones, asignando los valores binarios 0 ó 1. Es necesario que, tanto la fuente como el receptor posean la información del árbol.

Sin embargo, lo que verdaderamente caracteriza estos algoritmos es que la longitud de codificación llevada a cabo se asigna en función de la probabilidad de aparición del mensaje y de forma inversa. Esto implica que el mensaje más probable tenga la codificación más corta.

La diferencia entre ambos algoritmos, de Huffman y Fano, viene daba por la manera de construir el árbol. Mientras que Huffman (Figura 13) agrupa los nodos con menor probabilidad y, a continuación, construye un nodo superior a ambos con una probabilidad igual a la suma de probabilidad de ambos nodos, repitiendo de nuevo la búsqueda de los nuevos dos nodos con menor probabilidad y dejando los nodos con menor probabilidad más al fondo en el árbol. Por otra parte, el algoritmo de Shannon-Fano (Figura 14) agrupa todos los mensajes en un solo conjunto, posteriormente los divide en dos conjuntos que posean casi la misma probabilidad, y a cada uno de estos conjuntos se les asigna el valor 0 ó 1. Este proceso se repite hasta obtener conjuntos de un solo elemento.

A diferencia del algoritmo de Huffman, este último ya no es óptimo.

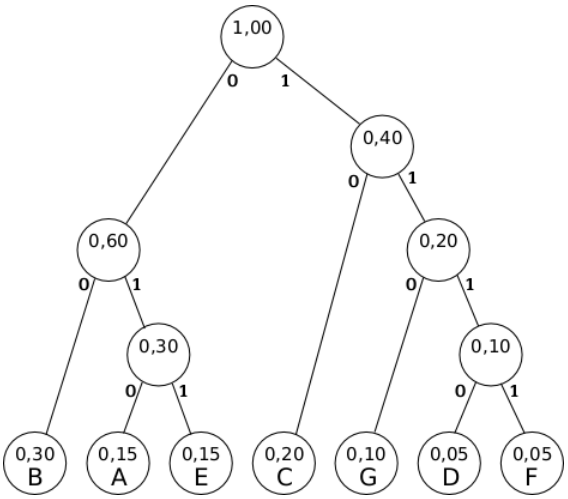

Figura 13. Ejemplo árbol de Huffman

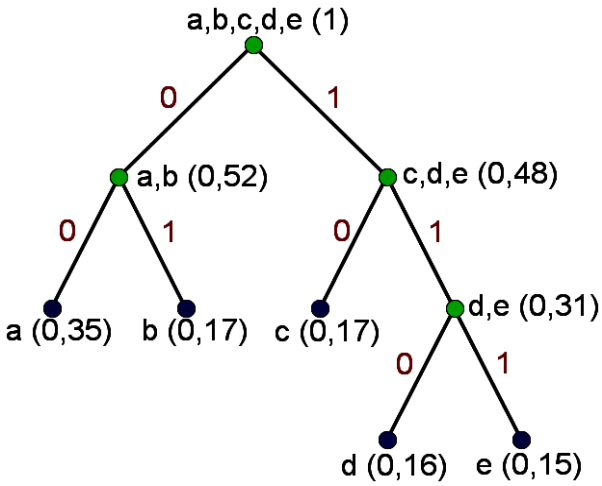

Figura 14. Ejemplo árbol de Shannon-Fano

4.2 Análisis de los Algoritmos

Se decidirá entre los algoritmos anteriormente expuestos cuál será implementado. La compresión con pérdidas ha sido descartada debido a las condiciones expuestas para el desarrollo.

Para decidir qué algoritmo implementar, se han realizado diversos programas en lenguaje de programación Python. Estos han permitido evaluar las diferentes tasas obtenidas por los diferentes algoritmos. Para el análisis se ha hecho uso de los datos proporcionados por el propio experimento en varios de sus eventos.

Constan de nueve archivos diferentes en formato .hdf. El peso aproximado de cada uno de los archivos es de 215KB. Internamente los datos se distribuyen en tablas de la forma en la que se muestra en Figura 15.

En cada archivo se encuentran 52 000 tablas, representando cada una de ellas un instante temporal. Las columnas por su parte representan los 12 sensores existentes, mientras que las filas los diferentes eventos registrados, en total, 346.

Por tanto, para la obtención de los datos de un evento, se selecciona una columna y fila, y se concatenan cada uno de sus valores presentes a lo largo de las diferentes tablas.

Como ejemplo se han tomado dos de los eventos en un pmt y se han representado en las Figura 17 y Figura 16.

pmtrwf at /RD/ [run_6203_0000_waveforms.h5 in D:\TFG\6203_Kr_Th]

Table Import/Export Data Data Display

0 51999

0-based

	0	1	2	3	4	5	6	7	8	9	10	11
0	2322	2310	2291	2315	2281	2272	2233	2270	2284	2299	2333	2313
1	2322	2309	2291	2315	2281	2271	2233	2272	2284	2300	2333	2313
2	2321	2309	2292	2315	2281	2271	2232	2271	2283	2299	2333	2313
3	2323	2309	2292	2314	2282	2271	2233	2271	2283	2299	2332	2312
4	2322	2310	2293	2314	2282	2271	2234	2272	2283	2299	2332	2313
5	2322	2310	2291	2314	2282	2271	2233	2270	2284	2299	2332	2312
6	2321	2311	2293	2313	2282	2271	2233	2271	2284	2300	2331	2313
7	2322	2309	2292	2315	2283	2272	2233	2270	2282	2300	2332	2312
8	2321	2310	2292	2314	2281	2271	2232	2271	2283	2299	2332	2312
9	2322	2310	2291	2314	2281	2271	2234	2270	2285	2299	2331	2311
10	2323	2310	2293	2315	2283	2272	2233	2270	2284	2299	2333	2313
11	2322	2310	2292	2314	2282	2273	2233	2272	2284	2299	2332	2312
12	2321	2311	2292	2315	2281	2271	2233	2270	2283	2298	2332	2311
13	2322	2309	2292	2316	2282	2271	2233	2271	2284	2301	2332	2312
14	2322	2310	2292	2314	2282	2273	2232	2272	2280	2299	2332	2312
15	2321	2309	2292	2315	2282	2272	2234	2272	2283	2300	2331	2312
16	2321	2310	2293	2314	2282	2271	2232	2271	2284	2298	2333	2311
17	2321	2308	2293	2314	2282	2272	2232	2270	2284	2298	2332	2311
18	2322	2311	2293	2315	2281	2271	2234	2271	2283	2299	2332	2310
19	2322	2311	2292	2315	2282	2271	2233	2271	2283	2298	2332	2313
20	2322	2309	2293	2314	2281	2271	2232	2270	2284	2299	2332	2311
21	2322	2310	2292	2315	2282	2272	2233	2271	2283	2298	2332	2312
22	2322	2311	2293	2315	2281	2272	2233	2270	2285	2300	2333	2311
23	2322	2309	2292	2315	2283	2272	2232	2272	2283	2297	2331	2312
24	2322	2308	2292	2315	2282	2271	2234	2272	2285	2300	2332	2312
25	2321	2308	2292	2315	2282	2272	2234	2270	2283	2299	2332	2311
26	2323	2311	2293	2315	2281	2271	2233	2271	2283	2300	2332	2312
27	2320	2310	2293	2315	2283	2273	2234	2270	2284	2299	2332	2311
28	2322	2310	2293	2314	2282	2271	2232	2270	2284	2300	2332	2314
29	2322	2310	2292	2315	2283	2272	2233	2270	2285	2300	2332	2312
30	2321	2310	2291	2314	2281	2272	2233	2272	2284	2299	2332	2313
31	2322	2310	2292	2316	2282	2270	2232	2271	2283	2300	2331	2313

Figura 15. Archivo de datos de una Run NEXT

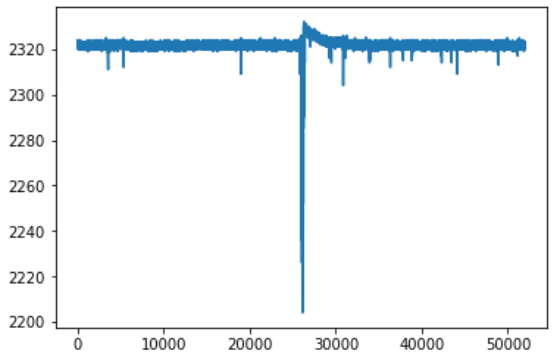

Figura 17. Ejemplo señal pmt 1

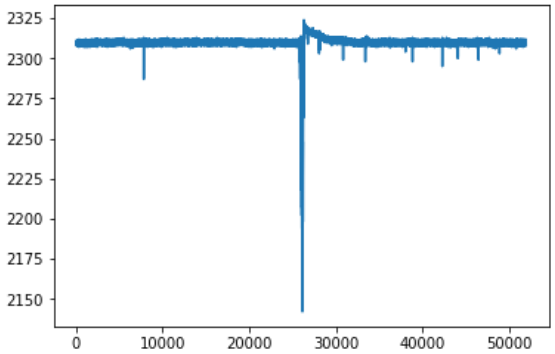

Figura 16. Ejemplo señal pmt 2

En relación con la información importante presente en el evento, se debe destacar la bajada brusca que existe a mitad de este. Sin embargo, también son importantes ambas colas, tanto la que precede a la bajada como la posterior a la misma; es por ese motivo por el cual no se pueden recortar del evento más datos.

Un factor importante para destacar es la monotonía al principio y al final del evento, y por tanto para tener en cuenta, es la compresión respecto al valor medio o repeticiones de datos.

En la programación de los diferentes algoritmos de compresión se ha utilizado el lenguaje Python, incluyendo los módulos.

Glob: módulo que permite encontrar todas las rutas que concuerden con un patrón determinado de acuerdo con las reglas de Unix, devolviendo los resultados en un orden aleatorio.

Tables: permite la administración jerárquica de conjuntos de datos para su procesado con grandes cantidades de datos. Además, permite tratar con archivos de extensión .hdf.

Numpy: extensión para Python que permite trabajar con vectores y matrices de manera eficiente. También posee una biblioteca con una gran variedad de funciones matemáticas para operar con ellos.

Matplotlib: biblioteca para la representación de figuras detalladas con una gran variedad de formatos y ecosistemas interactivos.

Heapq: módulo que provee la implementación del algoritmo de colas de prioridad. Su uso permite, entre otros, la programación del algoritmo de Huffman.

Los distintos algoritmos han sido creados en la plataforma Jupyter Notebook en distintos archivos.

4.2.1 Baseline

La compresión baseline consiste en calcular la moda de la señal (el valor más repetido), y realizar una resta de dicho valor a toda la señal en formato .raw. Como la señal característica de los PMTs oscila entre 2000 y 2400, en lugar de ser necesarios 12 bits para expresar todos y cada uno de los valores, se puede reducir a menos de 9 bits, puesto que con 512 valores se cubren todos los estados presentes a lo largo de un evento.

Para su programación ha sido necesaria la definición de una función mode que calcule el valor de la moda de la señal y lo almacene. El valor de baseline obtenido es propio de cada sensor y evento, por tanto, es necesario ejecutar numerosas veces esta función.

Posteriormente se recorren todos los valores presentes en todos los eventos y se les restan su valor de baseline correspondiente por evento y sensor. El primer evento en el primer sensor obtenido se corresponde con el observado en la Figura 18.

La distribución en frecuencia de los diferentes valores se muestra gracias a la función de histograma presente en el módulo matplotlib.pylab, y el resultado es el apreciable en la Figura 19. El valor más repetido se corresponde con 0, así como los valores negativos son más frecuentes debido principalmente a que el pico pronunciado de la señal es un mínimo global, mientras que el resto de la señal se mantiene estable en su mayoría.

Una vez obtenidos todos los valores codificados con respecto a la baseline, se procede al cálculo del ratio de compresión. Para ello se utiliza el código desarrollado incluido en la Figura 20.

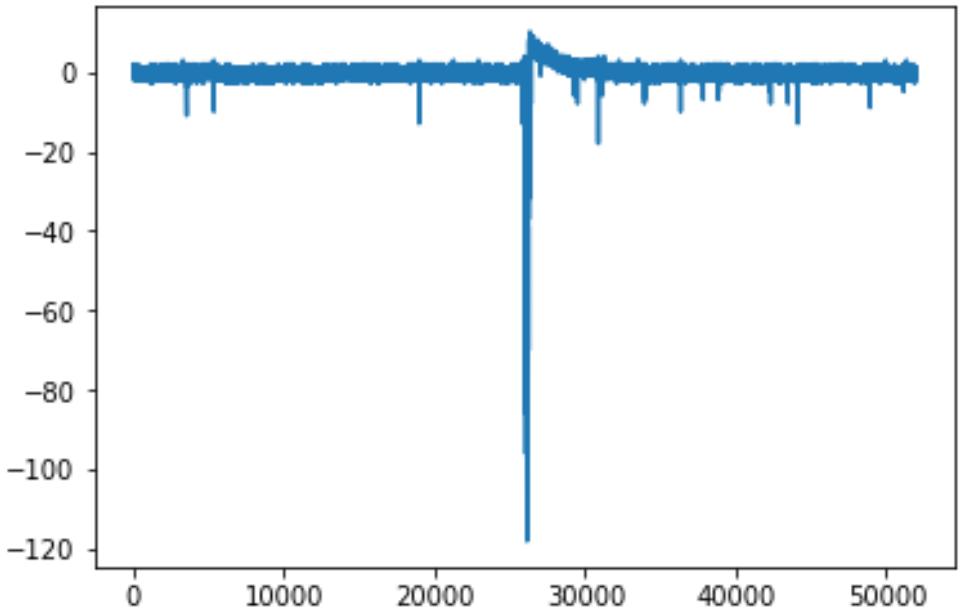

Figura 18. Señal sin baseline

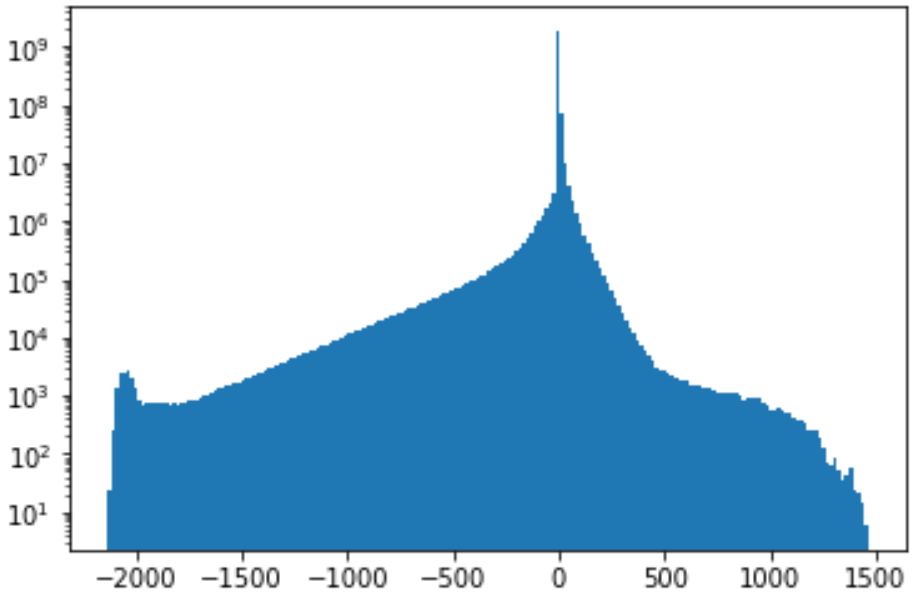

Figura 19. Histograma de valores de la señal pmt

```
for i in range(1,11):

    dif_in  = ((diffs[0:limit] >= (-1*((2**(i-1))))) & (diffs[0:limit] <= ((2**(i-1)-1)))).sum(dtype='int64')
    dif_out = ((diffs[0:limit] < (-1*((2**(i-1))))) | (diffs[0:limit] > ((2**(i-1)-1)))).sum(dtype='int64')

    dif_porcentaje.append((dif_in * (i+1) + (dif_out) * (12+1)) / (limit * 12))
    print(dif_in/(dif_out + dif_in))
    print("Dif respecto al baseline: Num bits:", i," Rango:[", (-1*((2**(i-1)))),":",((2**(i-1)-1)),"]"," % = ",
            dif_porcentaje[i-1])
```

Figura 20. Cálculo porcentaje de compresión

Se ha recurrido a un bucle for en el cual se ha calculado en cada iteración para un número diferente de bits.

El algoritmo consiste en enumerar la cantidad de valores que cumplen las cotas impuestas por dicho número de bits. Como ejemplo están los 4 bits. Podrán ser codificados todos los valores que cumplan la expresión (4.2).

$$(valor >= -2^{4-1}) \,\&\, (valor <= 2^{4-1} - 1) \tag{4.2}$$

Es decir, que se encuentren en el intervalo [-8, 7]. Posteriormente se multiplican el número palabras que cumplen la condición por el número de bits establecido y uno adicional, puesto que se necesita un bit que indique el estado de compresión, y, por tanto, incluye cierto overhead. Los resultados obtenidos son los presentes en la

Tabla 2.

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	65 %	51.25 %
2	86.9 %	64.08 %
3	92.47 %	61.018 %
4	95.7 %	55.47 %
5	97.64 %	48.627 %
6	98.6 %	40.969 %
7	99.23 %	33.01 %
8	99.64 %	24.88 %
9	99.8 %	16.63 %
10	99.94 %	8.32 %

Tabla 2. Compresión sobre baseline

Como se observa, la mejor compresión posible se obtiene utilizando 2 bits y uno extra para indicar el estado de codificación con un 64.08 %. La principal desventaja presente en este tipo de codificación es la necesidad de conocer previamente el valor de la moda en cada señal siendo, por tanto, peores los resultados a los aquí mencionados, puesto que se tendrían que predecir dichos valores de baseline con sus consecuentes ineficiencias.

4.2.2 Delta o Diferencial

En la codificación diferencial se calcula la resta entre un valor y su valor posterior, de manera que conociendo el valor inicial sea posible averiguar el resto simplemente añadiendo los incrementos o decrementos con respecto a este. La señal de un evento obtenida tras realizar el proceso diferencial es como la mostrada en la Figura 21.

Aplicando la función de histograma se observa en la Figura 22 como el valor más repetido se corresponde con 0 y su frecuencia de aparición decrece conforme se aleja de este.

En el código utilizado en la Figura 23 se ha empleado un bucle for dentro del cual se ha realizado la diferencia entre los arrays por evento y sensor. Para ello se ha restado a dicho vector él mismo desplazado una posición a la derecha.

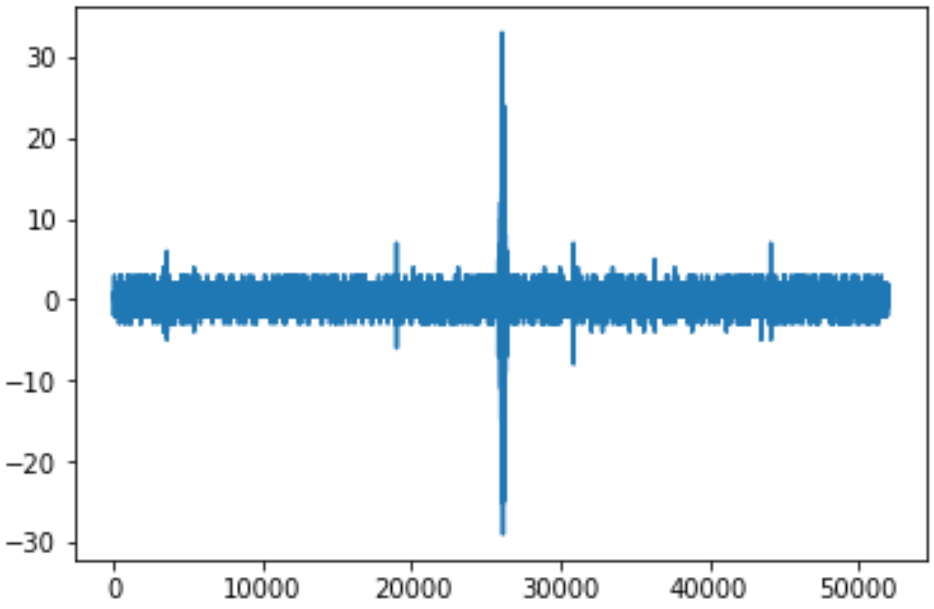

Figura 21. Señal diferencial

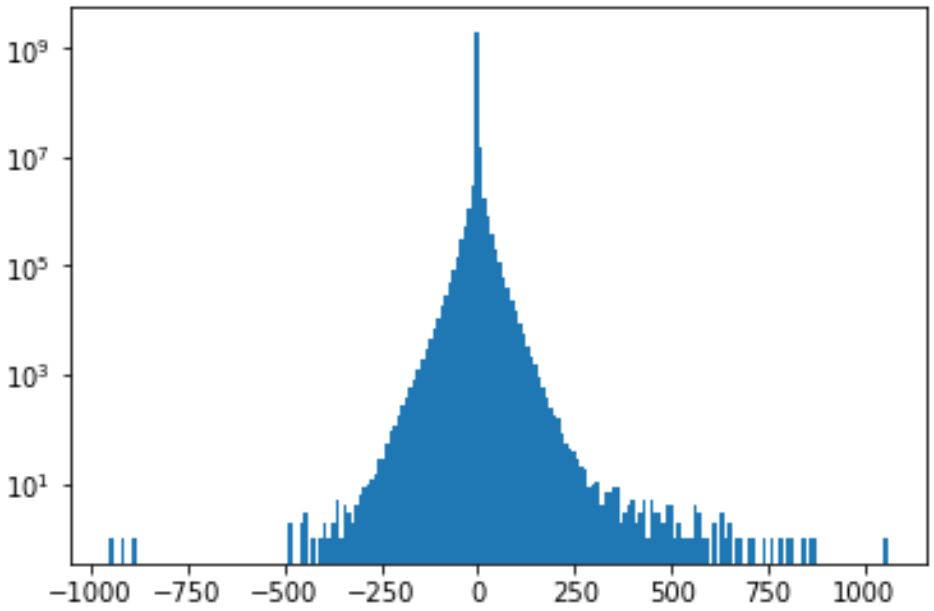

Figura 22. Histograma de las diferencias de la señal pmt

Tras la ejecución del código se han recogido los valores obtenidos y han sido presentados en la Tabla **3**.

```
for fname in files:
    # Read file
    h5in = tb.open_file(fname)
    data = h5in.root.RD.pmtrwf[:]

    #Compute baselines
    #baselines = np.apply_along_axis(mode, 2, data)

    #Sustract baselines
    for evt in range(data.shape[0]):
        for sns in range(data.shape[1]):
            diffs.append(data[evt][sns][:data.shape[2]-1] - data[evt][sns][1:])

diffs = np.concatenate(diffs)#.astype('int64')
```

Figura 23. Cálculos de las diferencias entre muestras

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	64 %	50.39 %
2	92.9 %	69.153 %
3	99.09 %	65.986 %
4	99.47 %	57.98 %
5	99.72 %	49.84 %
6	99.9 %	41.62 %
7	99.9838 %	33.32 %
8	99.99912 %	24.99 %
9	99.99998 %	16.66 %
10	99.999998 %	8.33 %

Tabla 3. Compresión delta

4.2.3 Doble diferencial

Como extensión a la compresión simple diferencial se ha empleado un algoritmo de doble diferencia. El objeto del algoritmo es calcular las variaciones de las diferencias entre valores de las señales, buscando aprovechar y comprimir aumentos monótonos e invariantes.

La señal procesada mostrada en la Figura 24 posee unos picos inferiores y superiores menos pronunciados que los extraídos del algoritmo diferencial simple.

El histograma visible en la Figura 25 posee, a su vez, los valores máximos y mínimos más cercanos al cero central.

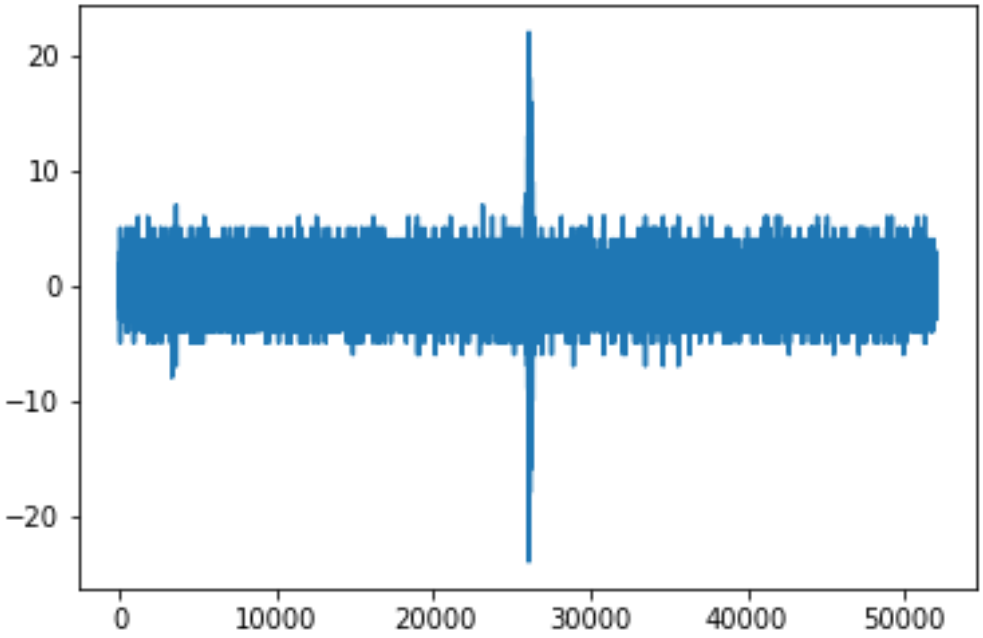

Figura 24. Señal doble diferencial

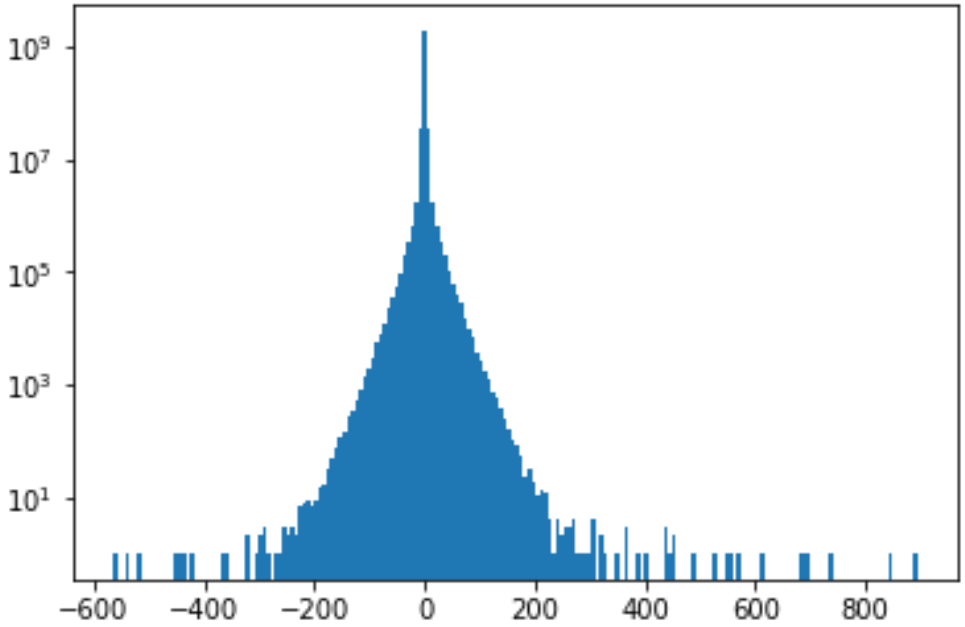

Figura 25. Histograma de las dobles diferencias de la señal pmt

En términos de programación, simplemente se ha incluido una variable auxiliar que permite guardar el valor de la diferencia inicial para aplicar sobre ella otra resta entre sus valores, como se observa en la Figura 26.

```
diffs = []
for fname in files:
    # Read file
    h5in = tb.open_file(fname)
    data = h5in.root.RD.pmtrwf[:]

    #Compute baselines
    #baselines = np.apply_along_axis(mode, 2, data)

    #Sustract baselines
    for evt in range(data.shape[0]):
        for sns in range(data.shape[1]):
            aux = np.array(data[evt][sns][:data.shape[2]-1] - data[evt][sns][1:])
            diffs.append(aux[:(aux.shape[0]-1)]-aux[1:])

diffs = np.concatenate(diffs)#.astype('int64')
```

Figura 26. Cálculo de dobles diferencias

Según la Tabla 4. Compresión doble delta este algoritmo muestra un máximo de compresión haciendo uso de 3 bits. Cabe destacar que en el cálculo de compresión no se han tenido en cuenta el valor inicial con respecto al que se realiza la sustracción, por tanto, los valores finales serán ligeramente inferiores.

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	43.9 %	31.94 %
2	75.1 %	54.296 %
3	97.4 %	64.7168 %
4	99.57 %	58.05 %
5	99.814 %	49.89 %
6	99.95 %	41.64 %
7	99.994 %	33.331 %
8	99.99985 %	24.99 %
9	99.9999675 %	16.66 %
10	99.99999933 %	8.33 %

Tabla 4. Compresión doble delta

4.2.4 Con respecto a un canal

Debido a la similitud de un mismo evento entre los diferentes canales se ha considerado la posibilidad de enviar el primer canal completo, mientras que el resto envían únicamente las diferencias con respecto a este.

La señal obtenida tiene cierta similitud con la conseguida con los métodos diferenciales anteriores, pero con una mayor variación entorno a la zona en la que se produce el pico del evento, tal como se muestra en la Figura 27.

El histograma (Figura 28) por su parte muestra dos picos claramente diferenciados, los cuales se corresponden con el valor 0 debido al envío de las diferencias y al valor de la moda en los eventos del primer canal con respecto al cual se calculan las diferencias.

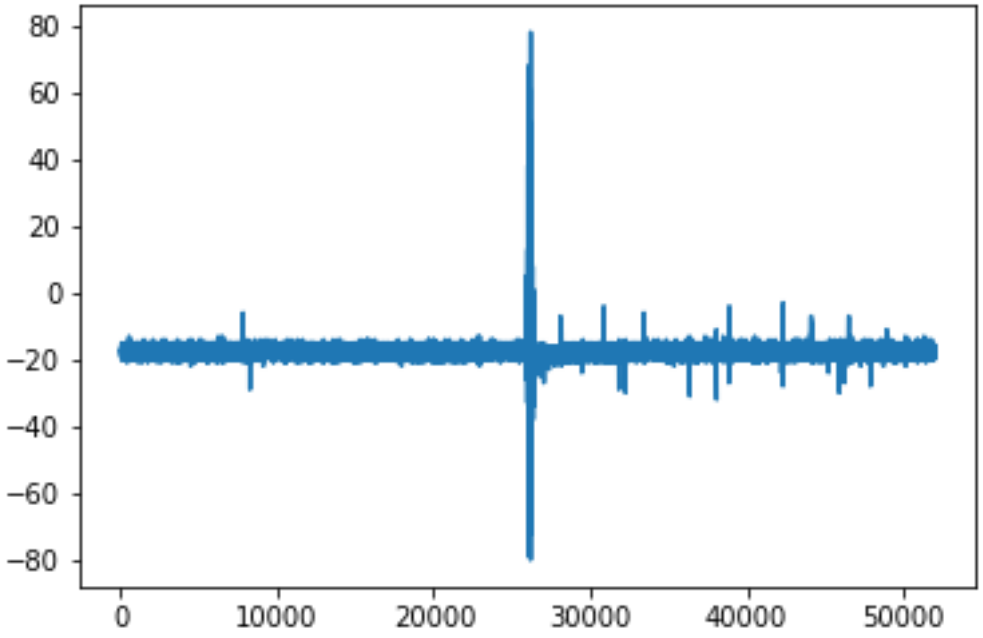

Figura 27. Diferencia canal 2 respecto al canal 1 de la señal pmt

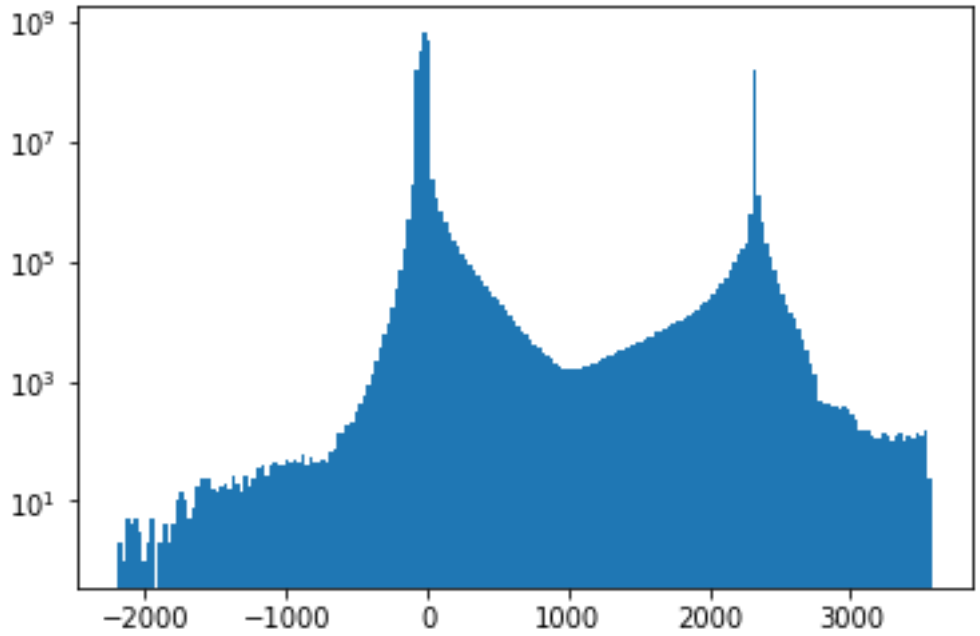

Figura 28. Histograma valores compresión respecto al canal 1

En el código empleado se ha sustraído a cada evento de cada sensor el correspondiente evento del primer canal en cada iteración del bucle for. También se ha tenido que añadir de manera previa el evento de este primer canal sin comprimir, según el código incluido en la Figura 29.

```
diffs = []

for fname in files:
    # Read file
    h5in = tb.open_file(fname)
    data = h5in.root.RD.pmtrwf[:]

    #Compute baselines
    #baselines = np.apply_along_axis(mode, 2, data)

    #Sustract baselines
    for evt in range(data.shape[0]):
        diffs.append(data[evt][1][:])
        for sns in range(2,data.shape[1]):
            diffs.append(data[evt][sns][:] - data[evt][1][:])

diffs = np.concatenate(diffs)#.astype('int64')
```

Figura 29. Codificación respecto a un canal

En la Tabla 5 se muestra como para pocos bits no solo no existe una compresión considerable, sino que dicha compresión es negativa. Esto es debido a que el porcentaje de valores codificados es muy pequeño e inferior al 1 % y, por tanto, se introduce un overhead al 99 % restante de los datos. Esto provoca que la cantidad total de datos sea muy superior.

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	0.979 %	-7.435 %
2	3.002 %	-5.831 %
3	9.382 %	-1.29 %
4	18.1 %	3.73 %
5	28.5 %	8.29 %
6	63.11 %	23.22 %
7	81.55 %	25.646 %
8	90.785 %	21.928 %
9	90.874 %	14.385 %
10	90.9 %	6.81 %

Tabla 5. Compresión respecto a un canal

El mejor resultado se obtiene con 7 bits de codificación, con 81.55 % de los datos codificados y un porcentaje de compresión del 25.646 %

4.2.5 Con respecto a un canal sobre baseline

Repitiendo el proceso previo, pero restando el baseline propio a cada uno de los sensores se pueden optimizar los resultados. El nuevo histograma de la Figura 30 consigue eliminar ese segundo pico, ya que se ha restado el baseline al primer canal de cada evento. Los nuevos valores de compresión son los presentes en la Tabla 6.

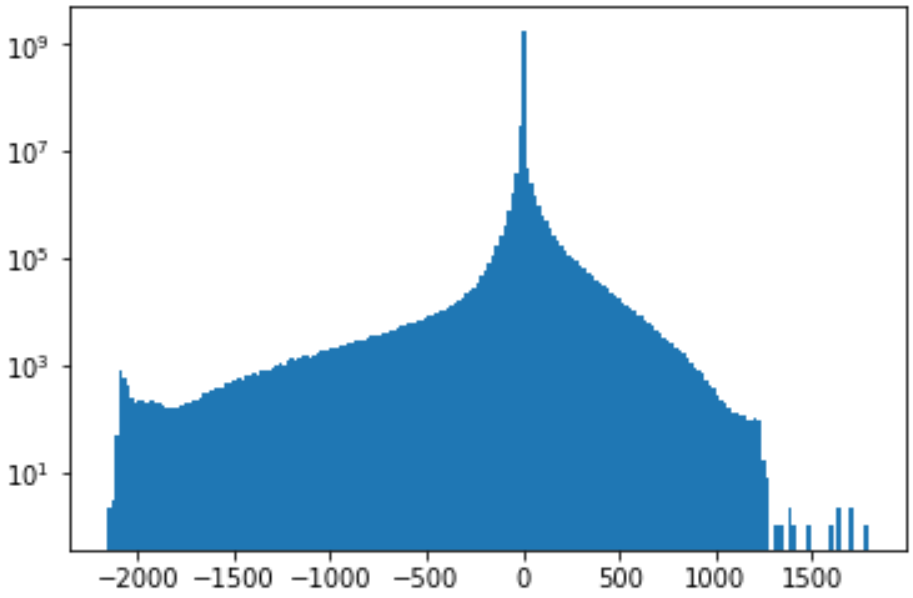

Figura 30. Histograma valores señales sobre baseline respecto al canal 1

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	54.36 %	41.49 %
2	84.05 %	61.71 %
3	95.80 %	63.519 %
4	97.53 %	56.568 %
5	98.61 %	49.194 %
6	99.21 %	41.271 %
7	99.60 %	33.169 %
8	99.83 %	24.94 %
9	99.94 %	16.65 %
10	99.98 %	8.33 %

Tabla 6. Compresión respecto a un canal sobre baseline

Incrementando el ratio de compresión a un 63 % esta vez, con casi un 96 % de los datos comprimidos, el resultado es sustancialmente mejor que sin tener en cuenta el baseline propio de cada señal.

4.2.6 Valores Consecutivos sobre baseline

Debido a la monotonía de las colas inicial y final de cada evento es posible que se mantengan valores iguales durante una gran cantidad de muestras. Por ello se ha incluido el análisis de una compresión basada en el cálculo de repeticiones continuas de un mismo valor. Básicamente, lo que se propone es enviar al decodificador el valor junto con su número de repeticiones.

```
consecutivos = []
nconsecutivos = []*0

ncon = 0

for i in range(diffs.shape[0]):#1000000000):#
    #print(diffs[i] - diffs[i-1])
    if (diffs[i] - diffs[i-1]==0):

        ncon = ncon + 1 #contador que acumula la cantidad de diferencias
    else:
        if (ncon>0):
            consecutivos.append(diffs[i-1]) #Almaceno el valor de la diferencia
            nconsecutivos.append(ncon) # Digo cuantas veces se ha repetido la ultima diferencia
            print(i)
        ncon = 0
consecutivos.append(nconsecutivos)
```

Figura 31. Codificación de valores consecutivos

Las repeticiones más frecuentes de datos seguidos iguales son 0 y 1, mientras que el valor que más veces aparece es de nuevo el 0. Se observa el histograma en la Figura 33, mientras que una ampliación de los valores más repetidos en la Figura 32.

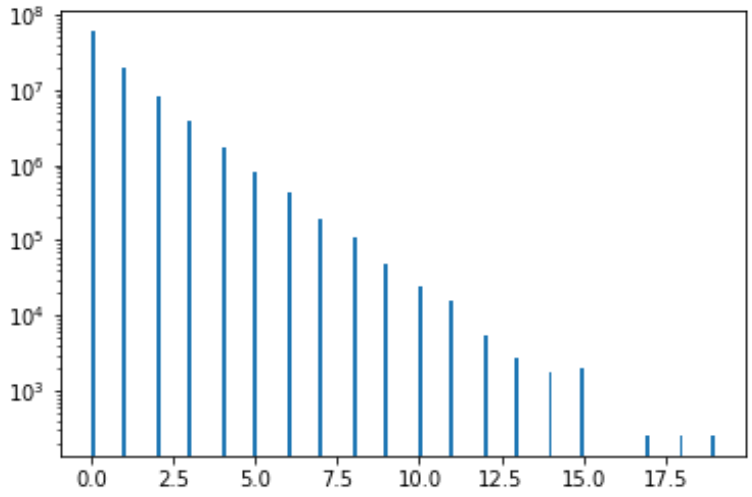

Figura 32. Repeticiones de las diferencias

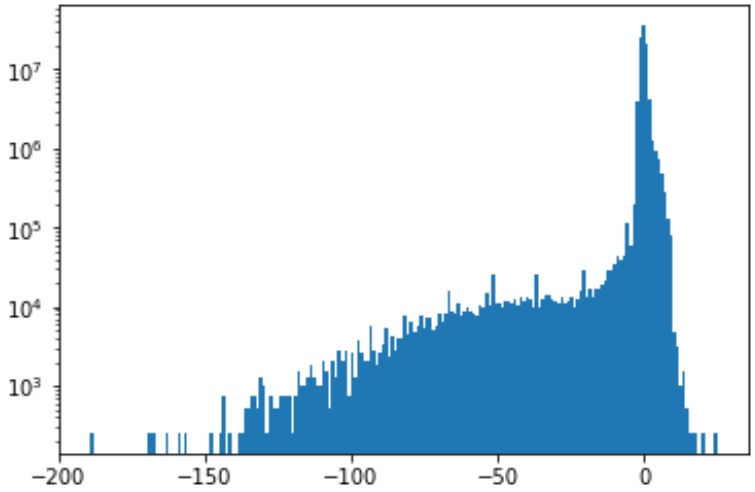

Figura 33. Histograma de valores de las repeticiones

Se trata de una compresión RLE simplificada que consta únicamente de dos elementos. El primero de ellos indica el propio valor de la muestra actual de la señal, mientras que el segundo indica la cantidad de veces que se repite dicho valor de forma consecutiva

Tal y como se muestra en la Tabla 7, el mejor ratio de compresión obtenido es de 64.05 % para 2 bits.

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	63.15 %	49.56 %
2	86.86 %	64.05 %
3	96.21 %	63.82 %
4	99.25 %	57.83 %
5	99.59 %	49.76 %
6	99.7 %	41.518 %
7	99.89 %	33.28 %
8	99.995 %	25 %
9	100 %	16.66 %
10	100 %	8.33 %

Tabla 7. Compresión por valores consecutivos sobre baseline

4.2.7 Valores consecutivos sobre diferencias

Se ha repetido el algoritmo previo aplicándolo sobre los valores de las diferencias entre muestras, en lugar de sobre los valores de estas. La distribución en el histograma es simétrica entre los valores positivos y negativos, tal como se muestra en la Figura 34

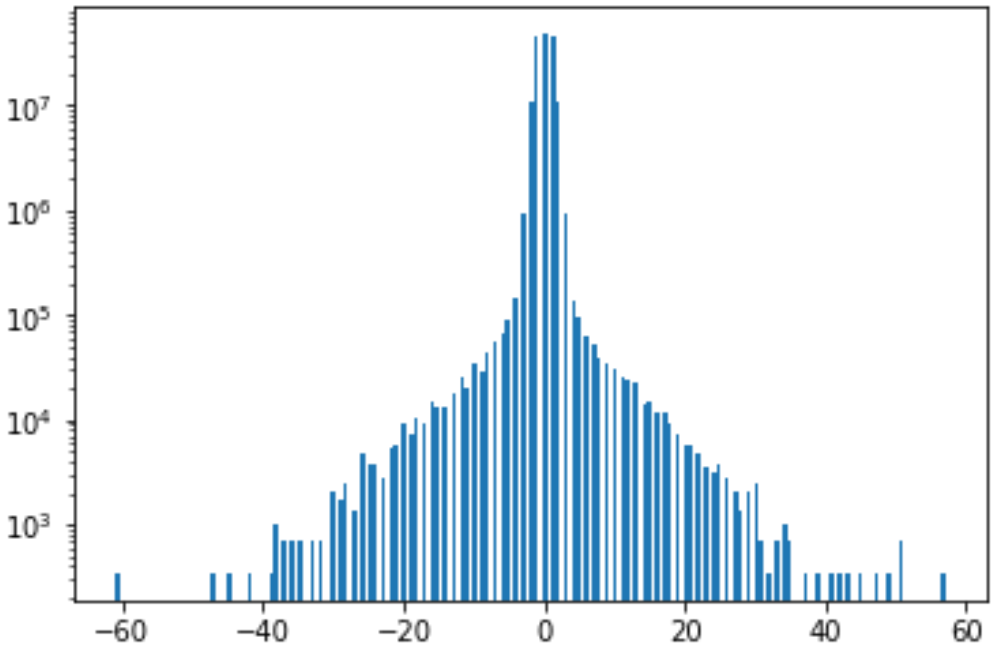

Figura 34. Histograma de los valores consecutivos sobre diferencias

Se consigue alcanzar hasta un 68 % de compresión con tan solo 2 bits (Tabla 8).

Número de bits	Porcentaje de valores comprimidos	Porcentaje de compresión
1	68.48 %	54.44 %
2	92.557 %	68.798 %
3	98.97 %	65.90 %
4	99.79 %	58.19 %
5	99.9499 %	49.97 %
6	99.996 %	41.66 %
7	100 %	33.33 %
8	100 %	25.0 %
9	100 %	16.66 %
10	100 %	8.33 %

Tabla 8.Compresión de valores consecutivos sobre diferencias

4.2.8 Huffman respecto a un canal sobre baseline

Debido a los valores de compresión obtenidos, insuficientes para el objetivo de la implementación, se ha programado una doble compresión utilizando sobre las codificaciones anteriormente mencionadas una compresión con el algoritmo de Huffman.

A partir de los datos obtenidos previamente en la compresión sobre baseline, se ha realizado un cálculo en las probabilidades de aparición de los diferentes valores mediante una función definida como aparece en la Figura 35.

```
def get_probabilities(content):
    total = len(content) + 1 # Agregamos uno por el caracter FINAL
    c = collections.Counter(content)
    res = {}
    #sortres = {}

    for ch,count in c.items():
        res[ch] = float(count)/total
    #res[10000] = 1.0/total

    #sortres = res.items()
    #print(sortres)
    #sortres.sort(sortres, key=lambda x:x[1])

    sortres = sorted(res.items(), key=operator.itemgetter(1))
    #sortres = sortres.reverse()
    return sortres, res
```

Figura 35. Cálculo de las probabilidades

Posteriormente, haciendo uso de dichas probabilidades, se crea el árbol correspondiente. De nuevo, se ha creado una función para su realización, visible en la Figura 36.

```
def make_tree(probs):
    q = []
    # Agregamos todos los símbolos a la pila
    for ch,pr in probs:# .items():
        # La fila de prioridad está ordenada por
        # prioridad y profundidad
        heapq.heappush(q,(pr,0,ch))

    # Empezamos a mezclar símbolos juntos
    # hasta que la fila tenga un elemento
    while len(q) > 1:
        e1 = heapq.heappop(q) # El símbolo menos probable
        e2 = heapq.heappop(q) # El segundo menos probable

        # Este nuevo nodo tiene probabilidad e1[0]+e2[0]
        # y profundidad mayor al nuevo nodo
        nw_e = (e1[0]+e2[0],max(e1[1],e2[1])+1,[e1,e2])
        heapq.heappush(q,nw_e)
    return q[0] # Devolvemos el arbol sin la fila
```

Figura 36. Creación del árbol

Aplicando el cálculo de los diferentes ratios de compresión obtenemos los mostrados en la Tabla 9.

A diferencia de las compresiones anteriores, la tasa de compresión aumenta conforme aumenta el número de códigos aplicados. El motivo radica en que cada símbolo se comprime siempre con el mismo número de bits, establecido por el árbol, y no como sucedía previamente donde se comprimían todos con el mismo número establecido. El motivo por el cual es posible aplicarlo en este caso viene dado por la lectura inequívoca de cada código Huffman, conociendo el final de este sin necesidad de tener previamente la información de su longitud.

Número de bits	Número de códigos Huffman	Porcentaje de compresión
2	2	56.061 %
3	3	70.4278 %
4	4	76.17 %
6	5	79.859 %
7	6	80.9527 %
8	8	81.875 %
9	11	82.209 %

Tabla 9. Compresión Huffman respecto a un canal sobre baseline

4.2.9 Huffman sobre diferencial

Imitando los pasos previos, pero modificando la compresión inicial por una diferencial, podemos obtener los valores resultantes de aplicar el árbol de Huffman a la codificación diferencial o delta.

Los resultados obtenidos son muy buenos, llegando a alcanzar un 85 % de compresión en para tan solo 7 códigos Huffman (Tabla 10).

Número de bits	Número de códigos Huffman	Porcentaje de compresión
2	1	45.196 %
3	2	60.86 %
4	3	77.979 %
5	4	81.34 %
7	6	84.878 %
9	7	85.234 %
10	10	82.461 %

Tabla 10. Compresión Huffman sobre diferencial

Presenta unos valores de compresión máximos entorno a los 9 bits. Con alrededor del 85 % es, hasta ahora, el mejor resultado de compresión obtenido.

4.2.10 Huffman sobre baseline

Finalmente, se comprueba la aplicación de la codificación Huffman sobre una compresión previa sobre baseline. Según la Tabla 11, los resultados de compresión son buenos; sin embargo, no alcanzan a los obtenidos previamente aplicando Huffman sobre diferencias.

Número de bits	Número de códigos Huffman	Porcentaje de compresión
2	1	46.64 %
3	2	61.69 %
5	3	73.9166 %
6	5	78.25 %
7	7	79.935 %
8	9	80.834 %
9	10	80.89 %

Tabla 11. Compresión Huffman sobre baseline

4.3 Comparación y Elección del Algoritmo

Una vez todos los algoritmos de compresión han sido analizados, se debe elegir uno para su implementación.

A partir de la gráfica de la Figura 37 se observan las diferentes codificaciones, utilizando para cada una de ellas los colores indicados en la leyenda inferior.

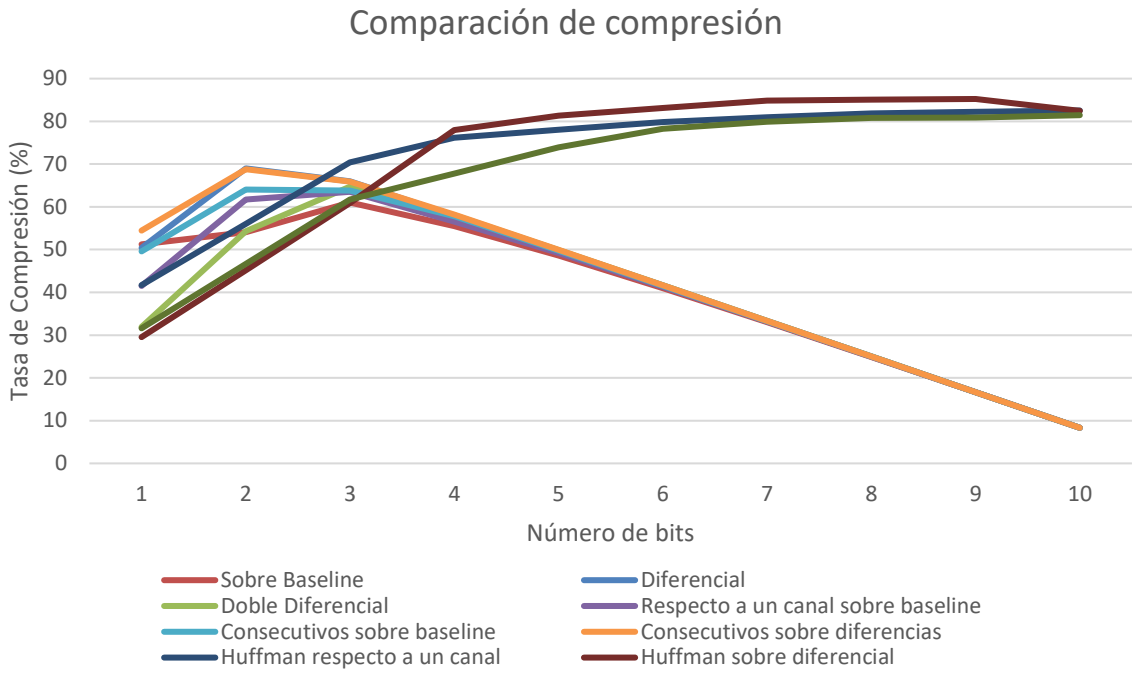

Figura 37. Comparativa de las compresiones

Todos aquellos en los que se ha implementado la codificación Huffman han mostrado unos ratios de compresión bastante superiores a las compresiones de una sola etapa, por tanto, el algoritmo a implementar se corresponderá con una doble etapa de compresión.

Entre los tres que contienen esta compresión Huffman, la compresión con respecto a la baseline presenta el problema del cálculo de la moda de la señal de cada canal. A pesar de ser posible una aproximación previa a su valor, no es posible conocerlo hasta que todos los datos hayan sido procesados y, por tanto, se deberá usar dicha estimación. El problema viene dado por su precisión, puesto que, si la moda difiere en más de 2 valores con respecto a su valor real, la posterior compresión Huffman puede reducir de manera considerable su eficiencia.

De la misma manera, la codificación con respecto a un canal necesita conocer la baseline de, al menos, el primer canal.

Por tanto, el algoritmo más sencillo para implementar, además de ser aquel que obtiene unas mayores tasas de compresión, se corresponde con la codificación Huffman sobre una etapa previa diferencial.

Capítulo 5. Diseño digital

El desarrollo del proyecto se ha llevado a cabo de manera jerárquica, permitiendo así la creación de módulos individuales para su posterior implementación en un archivo superior con instancias de estos.

Se encuentran representados en la Figura 41, el Control Path con bordes de azules, mientras que el Data Path se encuentra indicado mediante bordes de color negro. El módulo obtiene a su salida DFdata_out los datos codificados, así como una señal DF_wren que indica cuándo los datos son válidos.

Para ello, en su creación se ha tomado una visión global de las diferentes funciones. A grandes rasgos, la funcionalidad del diseño se debe encargar de permitir realizar una compresión delta, sobre la cual se utilizará otra compresión Huffman. Debe, a su vez, decidir si el dato puede ser comprimido y en caso negativo mandar el dato entero sin comprimir. Finalmente debe de concatenar los valores y juntarlos en bloques de 16 bits, incluyendo la función de padding en caso de llegar al final del evento que está siendo enviado y no ajustarse al tamaño determinado a su salida.

Se explicarán cada uno de los módulos de manera ordenada, cubriendo según la imagen del diseño todo el diagrama de izquierda a derecha. Se debe tener en cuenta que las señales de control empleadas provienen de la máquina de estados que no aparece en dicho diagrama.

5.1 Máquina de Estados

Las señales de control que aparecen como entrada del módulo no son directamente las proporcionadas por el resto del diseño digital del experimento, sino que, por el contrario, son generadas internamente por medio de una máquina de estados finita. Dicha máquina se corresponde con la representada en la Figura 38.

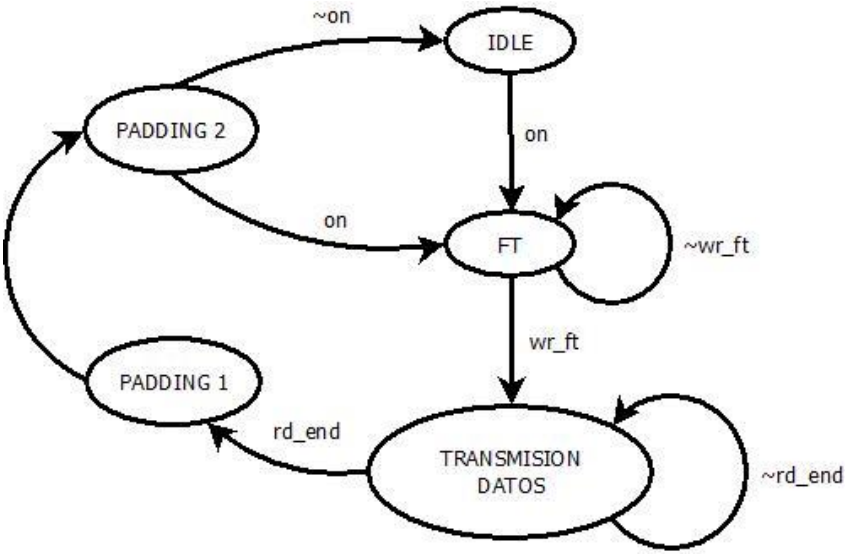

Figura 38. Diagrama máquina de estados

Las señales de salida de la máquina se indican mediante la adición de la letra ‘R’ previamente al nombre de estas.

Se distinguen hasta 5 estados diferentes. Se procederá a explicar cada uno de ellos.

- **IDLE**: estado inicial tras la aplicación de la señal de reset. La máquina se encuentra en un estado vacío esperando la llegada de activación de la señal on, la cual indica que se aproxima

un evento. Una vez recibida esta señal se pasa al estado FT. El resto de las entradas enmascaradas se encuentran a nivel bajo.

- **FT**: se busca recibir la huella de tiempo propia del evento, que se encuentra al principio de este. Su llegada se conoce debida a la señal wr_ft. Al igual que en el estado anterior, solo la señal de wr_ft se corresponde con su salida enmascarada, mientras que el resto se encontrarán a nivel bajo.
- **TRANSMISIÓN DE DATOS**: estado principal en el que permanecerá la máquina durante una mayor cantidad de ciclos. En este estado se realiza una conexión directa entre las entradas y las salidas enmascaradas, a excepción de rd_end. Una vez recibida la señal de fin de evento se avanza hasta el primer estado para realizar el padding correspondiente.
- **PADDING 1**: primera etapa de padding. Se anulan todas las señales de salida de la máquina de estados permitiendo que, en caso de haber completado el buffer de concatenación en la iteración anterior, sea posible vaciarlo y desplazarlo a la espera de realizar el padding necesario.
- **PADDING 2**: segunda etapa de padding. Se envía la señal de rd_end que permite hacer el padding al buffer.

El principal motivo para crear una máquina de estados viene dado por el hecho de que la señal que indica el final del evento viene acompañada del último bloque de datos, siendo necesarios ciclos adicionales para poder hacer el padding final.

En su programación se ha utilizado la estructura típica de una máquina Mealy, siendo sus salidas dependientes tanto de su estado actual como de las propias señales de entrada a la máquina.

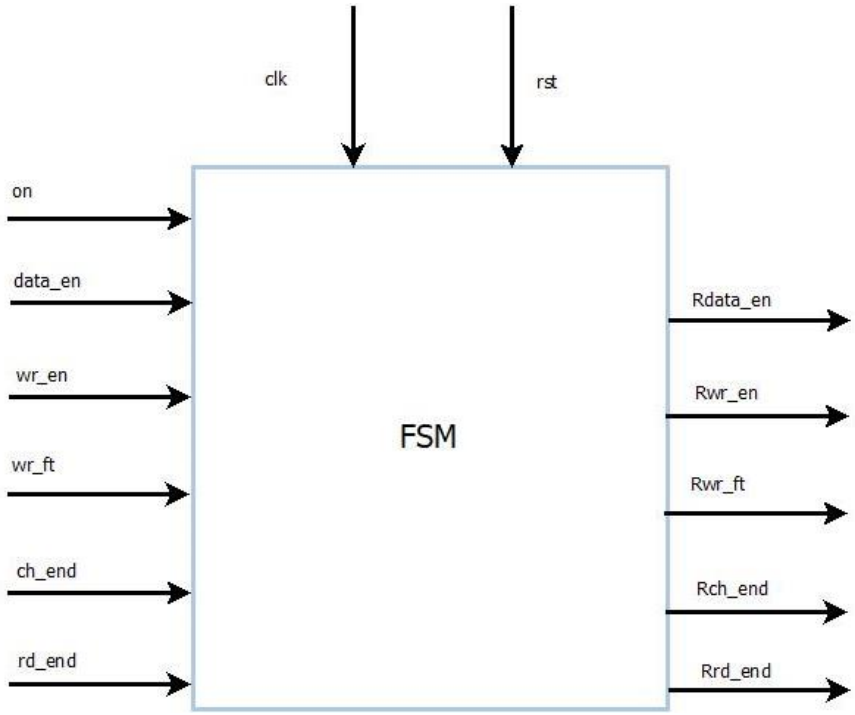

Figura 39. Módulo máquina de estados

5.2 Bloque de Registros

Al comienzo del módulo se encuentra un bloque de registros utilizado para guardar los valores de muestras anteriores y poder efectuar la compresión delta.

A su entrada se encuentra el bus de datos data_out en el cual aparecerán los valores de cada uno de los canales habilitados de manera sucesiva. Posteriormente, y tras la señal de ch_end, aparecerán de nuevo y en el mismo orden los valores de los canales en el siguiente instante de muestreo. Su diagrama se encuentra en la Figura 40.

Son necesarios al menos 12 registros que guarden cada uno de los canales anteriores. La señal de habilitación de cada uno de los registros se corresponde a la función lógica AND de la señal wr_en y del bit correspondiente del count_1. Este último bus de datos se corresponde con la salida del contador One hot creado. También tienen una señal de clr activa a nivel bajo mediante la señal de rd_end negada.

```
register_12bits register_12bits_0(.clk(clk),.clr(clr),.rst(rst),
.enable(count_1[0] & wr_en), .in(data_out[11:0]), .out(data_out_register_0));
```

A la salida del bloque de registros encontramos un multiplexor que se encargará de seleccionar la salida del registro correspondiente para poder realizar la resta entre el valor actual y el previo almacenado, con un valor predeterminado de 0 para las situaciones de llegada de wr_ft.

Son en total 12 registros de 12 bits, que es el tamaño del dato data_out válido y se corresponden a sus 12 bits más bajos.

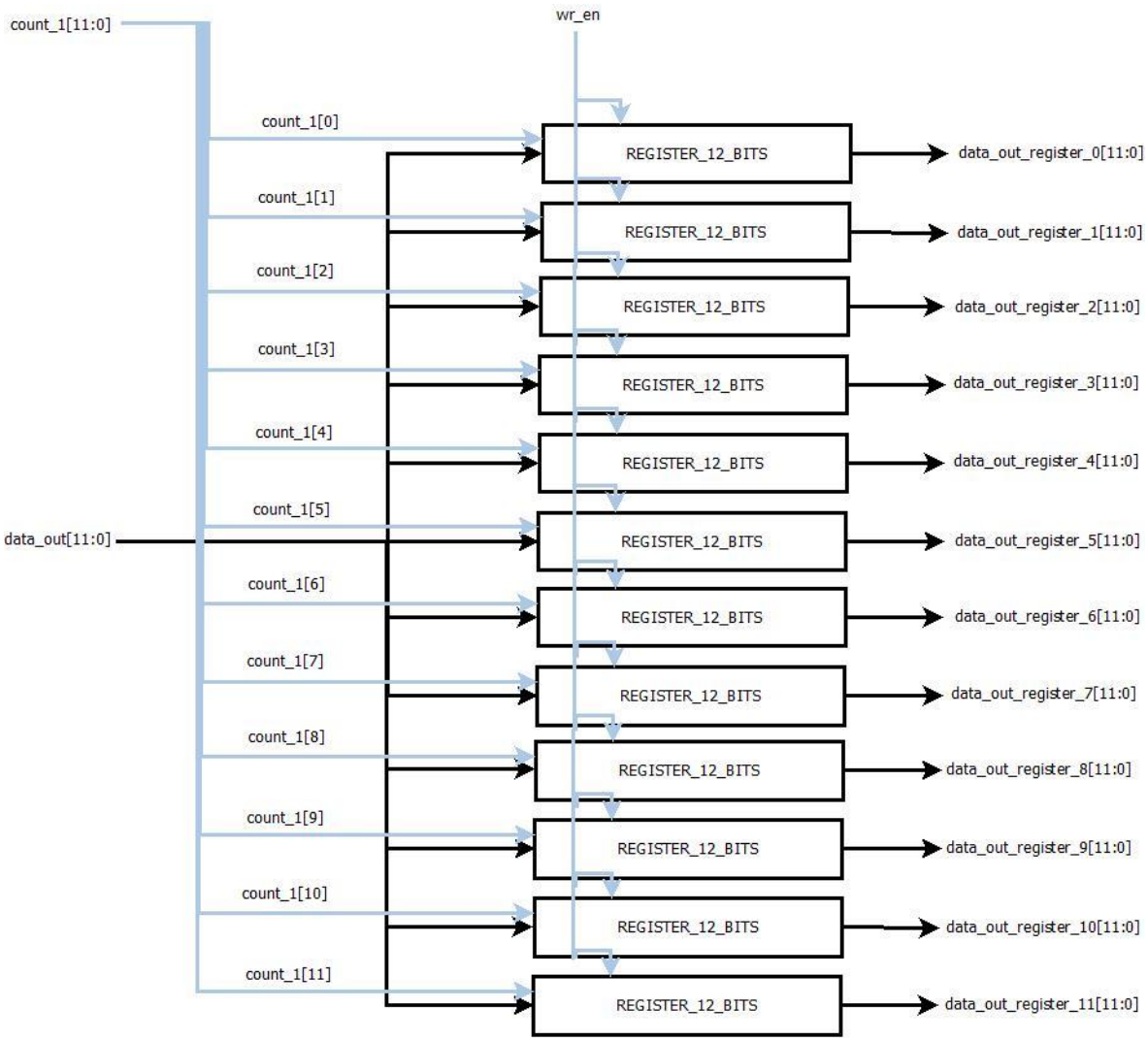

Figura 40. Diagrama bloque de registros

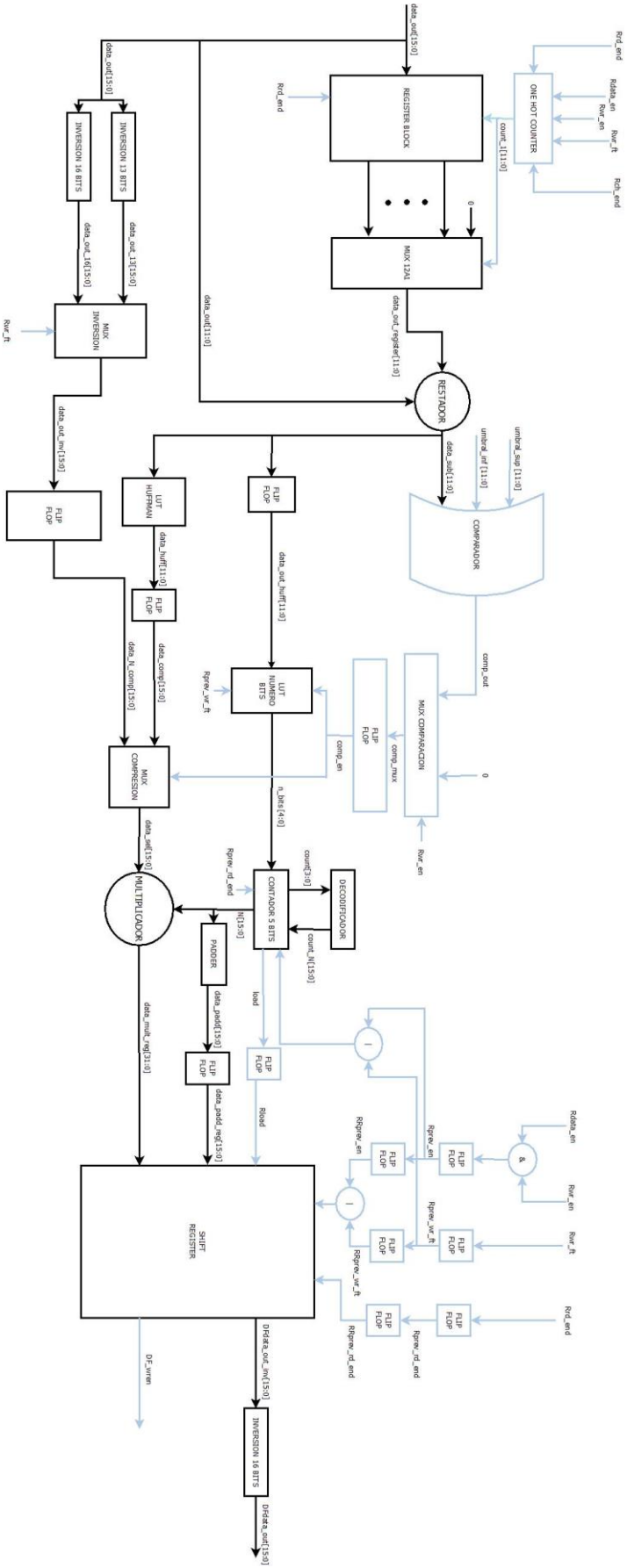

Figura 41. Diagrama módulo de compresión

5.3 Contador One Hot

Para poder seleccionar el dato válido en el bloque de registros es necesario controlar en qué canal se encuentra en cada momento. Para ello se ha utilizado un contador One hot, que tiene la característica de poseer únicamente un bit activo. Dicho bit se moverá en cada ciclo de reloj hacia una posición más alta de su bus hasta llegar a la posición 12.

Cada bit del contador se encuentra conectado como parte de la señal de enable de cada uno de los registros que se encuentran dentro del bloque, habiendo solo un registro habilitado para escritura al mismo tiempo.

A su vez, la señal count_1 que acumula la cuenta del contador se corresponde con la señal de selección del multiplexor posterior y permite seleccionar aquel registro (que es el mismo que se está habilitando para su escritura), tomando su valor data_out_register (Figura 42).

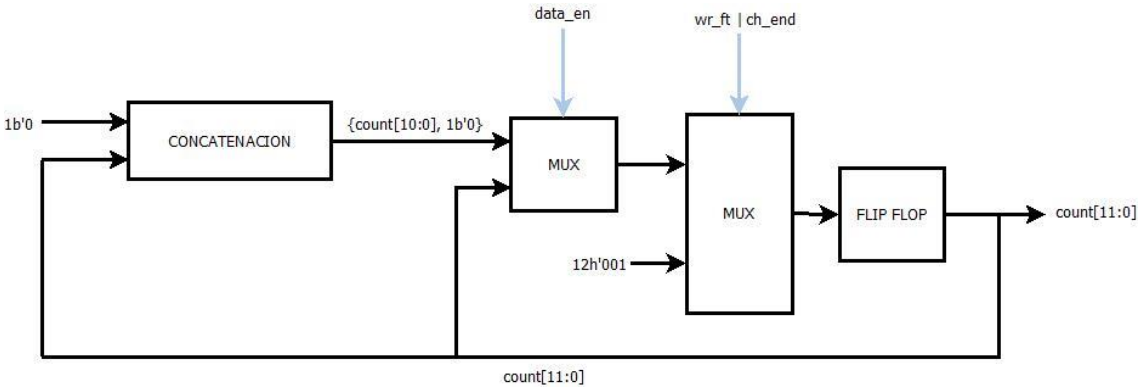

Figura 42. Diagrama contador One-Hot

Para el reinicio del contador se utilizada la señal ch_end, que indica la llegada al canal 12, poniendo el valor del contador a 1. También toma dicho valor si se encuentra después de activar la señal wr_ft.

El avance del contador depende de la señal data_en, que indica la existencia de un dato válido en data_out.

```
if(data_en)
    count <= {count[10:0],1'b0};
```

Figura 43. Avance del contador One-hot

5.4 Restador y comparador

Tras haber obtenido el dato previo del canal seleccionado se realiza la resta entre ambos. Como el dato guardado es de 12 bits mientras que el de entrada tiene 16 bits, utilizaremos solo sus 12 bits inferiores. La salida se corresponde con la señal data_sub, la cual actúa como entrada del comparador.

En el comparador se decidirá si el valor encontrado en data_sub se encuentra dentro del umbral determinado por dos valores, umbral_inf y umbral_sup. Ambos deben ser incluidos de manera manual y variarán en función de la tabla de Huffman que se utilice para la compresión. El comando utilizado para la comparación aparece en la Figura 44.

```
assign comp_out = ((data_sub[11]==umbral_inf[11]) & (data_sub[10:0]>umbral_inf[10:0]))
 | ((data_sub[11]==umbral_sup[11])&(data_sub[10:0]<umbral_sup[10:0]));
```

Figura 44. Comparación de umbrales para compresión

Es necesario que el valor de umbral_inf sea negativa, puesto que en caso contrario solo tomará en consideración el umbral superior.

En caso de encontrarse entre dichos valores, la señal comp_out será puesta a nivel alto, indicando que el valor de la diferencia se encuentra dentro de los codificables.

Tras registrar la salida del comparador su valor se utilizará como entrada en la look up table, que determina el número de bits, y en el multiplexor, que seleccionará si escoger el dato comprimido o sin comprimir.

5.5 LUT Huffman y LUT número de bits

Para codificar el valor se ha tomado la decisión de implementar una look up table, haciendo uso de una estructura case. La creación y actualización debe ser realizada de forma manual, ya que no se ha implementado la lectura de un fichero de texto para su descripción automática.

Para el caso de LUT Huffman se han codificado siguiendo la tabla definida, añadiendo un 1 delante de dicho valor e invirtiéndolo. Posteriormente se ha copiado el valor en los bits más bajos de su señal de salida de 16 bits, data_huff.

Como ejemplo tenemos la codificación del valor -1. Según está indicado, su codificación es 3'b000. A ello se le añade un bit que indica que dicho valor se encuentra codificado, obteniendo 4'b1000. Finalmente se invierte su orden y se copian en 16 bits, obteniendo la siguiente línea de código en la Figura 45.

```
12'hfff: //-1
    data_out = 16'b0000_0000_0000_0001;
```

Figura 45. Ejemplo de codificación

El motivo de que se utilicen 16 bits es para la simplificación del multiplexor posterior, puesto que esta codificación es de extensión variable.

Por otra parte, en el caso de LUT número de bits no aparecerán los valores codificados, sino la cantidad de bits que son necesarios para cada uno de ellos, incluyendo ya el bit adicional que indica si se encuentra codificado.

Este módulo es necesario porque la concatenación posterior necesita conocer cuántos de los bits que se encuentran en el dato deben ser enviados.

Además de indicar la extensión de los valores codificados, también se indican la de los valores sin codificar, que son de 13 bits, y de la huella correspondiente de tiempo, de 16 bits. Para estas situaciones se utilizan las señales de comp_en y de wr_ft respectivamente.

La salida del contador solo puede tomar valores de 0 a 16 y, en consecuencia, se necesitará un bus de 5 bits, presente en el esquema como n_bits.

5.6 Contador de 5 bits

Una vez conocida la cantidad que se escribirán en el buffer, es necesario mantener una cuenta que permita saber en cada momento la situación de este. Para ello se ha creado un contador que realice una suma acumulativa de los bits totales en el buffer (Figura 47).

Primeramente, es necesario conocer la señal que indique si el dato es válido. Dicha señal se corresponde con la entrada 'en'.

```
Contador_5bits contador_5bits( .clk(clk),.clr(~Rprev_rd_end), .rst(rst), .load(load),
.en(Rprev_en | Rprev_wr_ft), .n_bits(n_bits), .count(count), .count_N(count_N), .N(N));
```

Figura 46. Instancia contador 5 bits

Tal y como se aprecia en el diagrama general, las señales en las que se encuentra incluida la expresión "prev" han sido registradas. Por tanto, la habilitación del sumador depende de la validez

de los datos (data_en), de su enmascaramiento (wr_en) y de la llegada de la huella de tiempo (wr_ft).

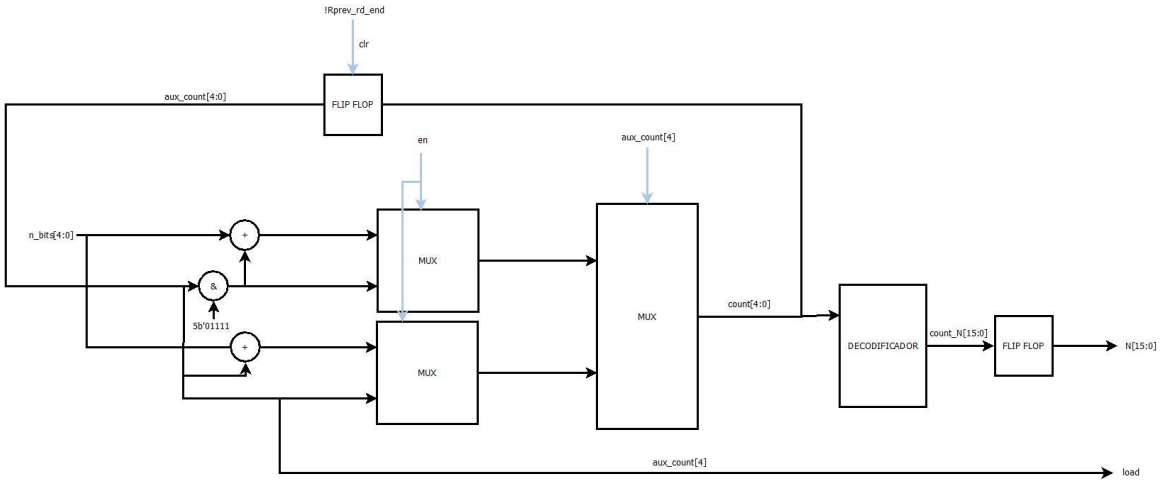

Figura 47. Diagrama de contador de 5 bits

En caso de ser válido se suma a la cuenta total n_bits. Pero si la cuenta total antes de ser sumada posee su bit más alto activo, lo cual indica que hay al menos 16 bits en el buffer, se enmascara la cuenta con el valor 5'b01111. Esta operación es el equivalente a restar el valor 16 a la cuenta, y se realiza porque en el ciclo previo se habría habilitado la señal de load (Figura 48), definida como su quinto bit.

```
assign load = aux_count[4];
```

Figura 48. Señal de carga en contador de 5 bits

Y por lo tanto se habrían enviado 16 bits, reduciendo en dicha cantidad los bits en el buffer (Figura 49).

```
assign count = (aux_count[4])? ((en)? ((aux_count & 5'b01111) + n_bits)
:(aux_count & 5'b01111)): ((en)?(aux_count + n_bits):aux_count);
```

Figura 49. Asignación interna del contador acumulable

Por último, el contador de 5 bits se encuentra conectado con un decodificador de 16 bits. Su funcionamiento se corresponde con una conversión binaria a termómetro. La característica principal de la decodificación es que solo tendrá un bit activo, cuya posición dependerá del valor siguiente al descrito por los 4 bits más bajos del contador. Como ejemplo de decodificación se presenta la Figura 50.

```
4'b0011:
    count_N = 16'h0008;
```

Figura 50. Ejemplo decodificador de 16 bits

La salida del contador volverá a este, el cual se encargará de registrarla y permitirá su salida hacia el multiplicador y el padder.

5.7 Multiplexor de compresión

Para decidir si se selecciona el dato comprimido o el dato sin comprimir se implementa un multiplexor. Cabe resaltar que ambas entradas al multiplexor están registradas. La primera de ellas proveniente del LUT Huffman explicado anteriormente, mientras que la segunda se trata del dato sin comprimir invertido.

En el caso del dato comprimido ya se encuentra invertido por la definición intrínseca del módulo, mientras que son necesarios 2 inversores para el dato sin comprimir. El primero de ellos invierte los 12 bits más bajos, de manera que el bit 0 pase a la posición del bit 12 y viceversa. Finalmente se concatena un 0 en la parte baja, que será el bit que indique que el dato no está comprimido (Figura 51).

De la misma manera, pero sin tener en cuenta la adición del bit del estado de la compresión, se ha necesitado un inversor de 16 bits para poder invertir la huella de tiempo (Figura 52).

```
assign data_out_13 = {1'b0,1'b0,1'b0, data_out[0],data_out[1],data_out[2],data_out[3],data_out[4],
data_out[5],data_out[6],data_out[7],data_out[8],data_out[9],data_out[10],data_out[11],1'b0};
```

Figura 51. Inversor de 13 bits

```
assign data_out_16 = {data_out[0],data_out[1],data_out[2],data_out[3],data_out[4],data_out[5],data_out[6],
data_out[7],data_out[8],data_out[9],data_out[10],data_out[11],data_out[12],data_out[13],data_out[14],data_out[15]};
```

Figura 52. Inversor de 16 bits

Ambos son llevados a un multiplexor que decidirá en función de la señal wr_ft qué dato tomar como data_out_inv.

Tras todo ello, el multiplexor de compresión elegirá, según la señal registrada generada por el comparador, el data_comp o el data_N_comp, tomando el valor respectivo a su salida data_sel.

5.8 Multiplicador

Mediante el uso de una IP de Xilinx se ha instanciado un multiplicador dedicado que presenta su salida ya registrada. Al ser el path crítico del diseño se ha decidido reducir el camino de sus entradas sin registrarlas.

El objeto del multiplicador es realizar una concatenación de datos. Por un lado, se encuentra como entrada el dato a concatenar data_sel, mientras que por otro está la señal N.

Suponiendo que el buffer se encuentra vacío, N será igual a 16'h0001, y por tanto la salida de la multiplicación será igual a data_sel. Por otro lado, si el buffer tuviera 8 bits ocupados, N sería 16'h0100. El resultado de este último caso daría como salida de la multiplicación el valor de data_sel desplazado tantos bits como tenga ocupados el buffer. El multiplicador actúa por tanto como un desplazador de log2(N) bits.

La salida de este, data_mult_reg, deberá ser de 32 bits para evitar un overflow en caso de poseer ambas entradas con su MSB activo.

5.9 Padder

Tras la indicación del final del evento es necesario rellenar los bits sobrantes en el último segmento enviado de 16 bits. Para seguir la estructura creada para la concatenación se ha implementado un módulo externo denominado Padder que, a partir del valor de N, creará una salida data_padd que debe poseer activos sus bits más altos, de manera que al realizar una suma con el valor del registro interno del módulo Shift Register ponga a 1 todos aquellos bits que no contienen información (Figura 53).

Como N poseerá un bit activo, y realizando el complemento a 2 conseguimos un bus en el que todos los bits a partir del que tenía originalmente N a 1, se encuentren a nivel alto, incluido este.

```
assign data_padd = -N;
```

Figura 53. Asignación valor de padding.

5.10 Shift Register

Este módulo se encuentra al final del diseño y es el encargado de concatenar los valores y enviarlos por el bus DFdata_out_inv de 16 bits, así como de activar la señal DFdata_wren cuando eso suceda. La descripción se asemeja a la de un registro normal, pero incluye un sumador interno que permite realizar la acumulación de los valores provenientes de data_mult_reg (Figura 54).

Como se ha detallado anteriormente, a la salida del multiplicador encontramos el siguiente dato que se debe añadir al buffer desplazado log2(N) posiciones. Como la posición más alta en la que el buffer tendrá un valor válido será log2(N)-1, la suma del registro interno del módulo Shift Register con data_mult_reg será el equivalente a concatenar dichos datos.

El tamaño del registro interno tiene 32 bits, puesto que el caso crítico se da cuando hay escritos 16 bits y se quieren concatenar otros 16.

A partir de la señal load registrada, generada inicialmente por el contador de 5 bits, la señal de salida DFdata_wren será activada, así como DFdata_out tomará el valor de los 16 bits más bajos del registro interno (Figura 55). A su vez los 16 bits superiores del registro serán desplazados a la parte baja para permitir que se sigan concatenando valores, mientras que pone a nivel bajo la parte superior.

Este procedimiento se llevará a cabo solo cuando los datos a la entrada sean válidos, que vendrá indicado por la combinación de señales data_en, wr_en y wr_ft tras ser registradas en dos ocasiones.

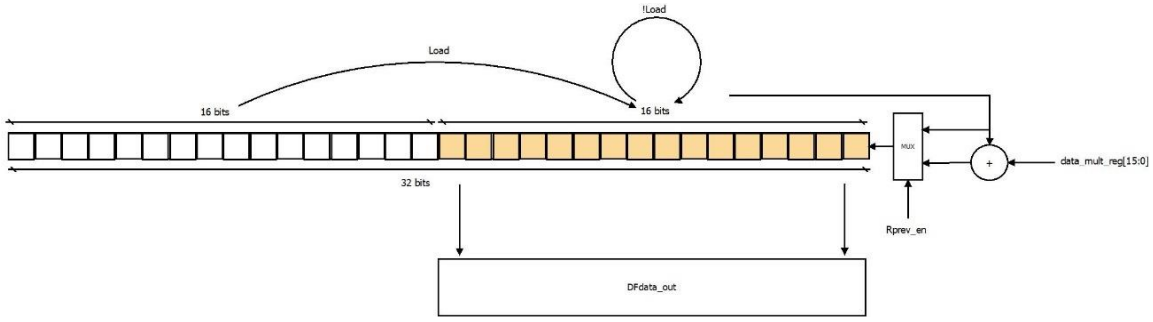

Figura 54. Diagrama shift register

```
shift_register[31:0] <= {16'h0000,shift_register[31:16]} + data_mult;
```

Figura 55. Habilitación de load en shift register

```
Shift_Register shift( .clk(clk), .rst(rst), .en(RRprev_en | RRprev_wr_ft), .rd_end(RRprev_rd_end),
.load(Rload), .data_padd(data_padd_reg), .data_mult(data_mult_reg),
.DFdata_out(DFdata_out_inv), .DFdata_wren(DFdata_wren));
```

Figura 56. Instancia shift register

El caso restante se da cuando la señal de rd_end es detectada por el módulo. En dicha situación la suma no se realiza con la salida del multiplicador, sino con la salida del padder. A su salida se encuentra un inversor de 16 bits que cambia el orden de recepción de los datos.

Haciendo uso de dos bloques always combinacionales en los que se asignarán el próximo estado de la máquina y las salidas respectivamente, así como de un tercer bloque always síncrono en el que se asigna a su actual su estado próximo.

Son necesarios 3 bits para poder definir los cinco estados diferentes utilizados siendo posible, por tanto, la existencia de estados espurios. Ante esta situación se ha decidido implementar la estructura case con el comentario //synopsis full_case.

El diseño final, diferenciado en las diferentes etapas de segmentación, es el representado en la Figura 57.

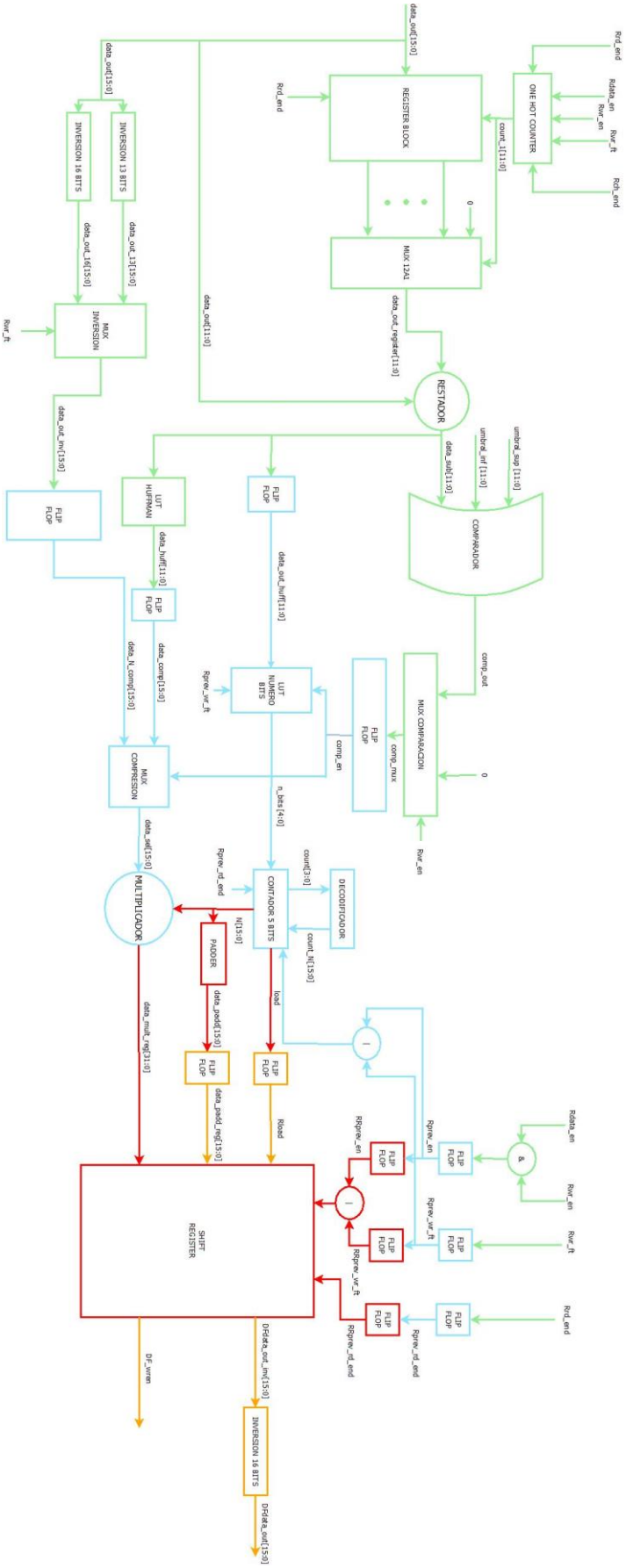

Figura 57. Diagrama módulo de compresión por ciclos

Capítulo 6. Simulación

La simulación inicial realizada para la corrección de los primeros errores ha sido llevada a cabo observando las formas de onda a partir de la función ISim incluida en la suite de diseño de ISE.

Primeramente, se ha debido crear un archivo de estímulos denominado stimulus.v. En este se describen los valores de las distintas señales de entrada al módulo, así como un control temporal utilizando delays entre los diferentes estados de estas.

Es necesaria la instanciación del módulo top de la jerarquía del diseño. Posteriormente se especifican dos bucles anidados. En el primero de ellos se especifica el número de muestras temporales que habrá por cada uno de los 12 canales. El segundo bloque detalla el barrido en cada uno de los 12 canales con su muestra correspondiente.

Para cada muestra de cada canal se debe especificar el estado de las señales data_en y wr_en, indicando si el dato se escribirá o no. En el caso de la última iteración también se debe activar la señal ch_end, indicando que es el último canal.

Una vez todas las muestras del evento hayan sido enviadas se activa la señal rd_end para finalizar la simulación y realizar el padding. Para todo ello se ha programado el bucle de la Figura 58

```
for(iter_ch = 1; iter_ch<=400000; iter_ch = iter_ch + 1)
begin
   for(iter_block = 0; iter_block<12; iter_block = iter_block + 1)
   begin
      #20
      ch_end <= 0;
      data_en <= 1;
      wr_en <= 1;
      data_out <= data_out_aux[iter_block];
   end
ch_end <= 1;
end
rd_end <= 1;
```

Figura 58. Ciclo de señales por evento

Los datos utilizados en la simulación han sido cogidos del archivo 5471_0000_0_0to5.txt (Figura 61), producto de una ejecución real del experimento para el que será utilizado. La lectura de dicho archivo se lleva a cabo de forma reiterada en cada llegada de la señal ch_end, como se indica en la Figura 60.

El formato de lectura se realiza con la función $fscan(), conociendo previamente la distribución de los datos en el archivo. Esta consiste en 16 columnas con 400 000 muestras representadas en formato decimal, se ha incluido el modo de lectura binario, guardando cada bloque de 16 bits en array de la matriz data_out_aux.

Esta escritura se lleva a cabo por una función llamada dentro de un bloque always en cada ciclo de reloj con DFdata_wren como señal de habilitación (Figura 59).

```
always  @(posedge clk)
begin
   if(DFdata_wren)
   $fwrite(out_file,"%b",DFdata_out[15:0]);
end
```

Figura 59. Trigger de escritura en el archivo de salida

```
always @(posedge clk)
begin
   //every clock cycle an input value is stored in D reg
   if (ch_end)
   begin
      #4 err_read=$fscanf(input_file,
     "%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
      data_out_aux[0],
      data_out_aux[1],
      data_out_aux[2],
      data_out_aux[3],
      data_out_aux[4],
      data_out_aux[5],
      data_out_aux[6],
      data_out_aux[7],
      data_out_aux[8],
      data_out_aux[9],
      data_out_aux[10],
      data_out_aux[11],
      data_out_aux[12],
      data_out_aux[13],
      data_out_aux[14],
      data_out_aux[15]
      );
    end
end
```

Figura 60. Bucle de lectura del archivo de entrada

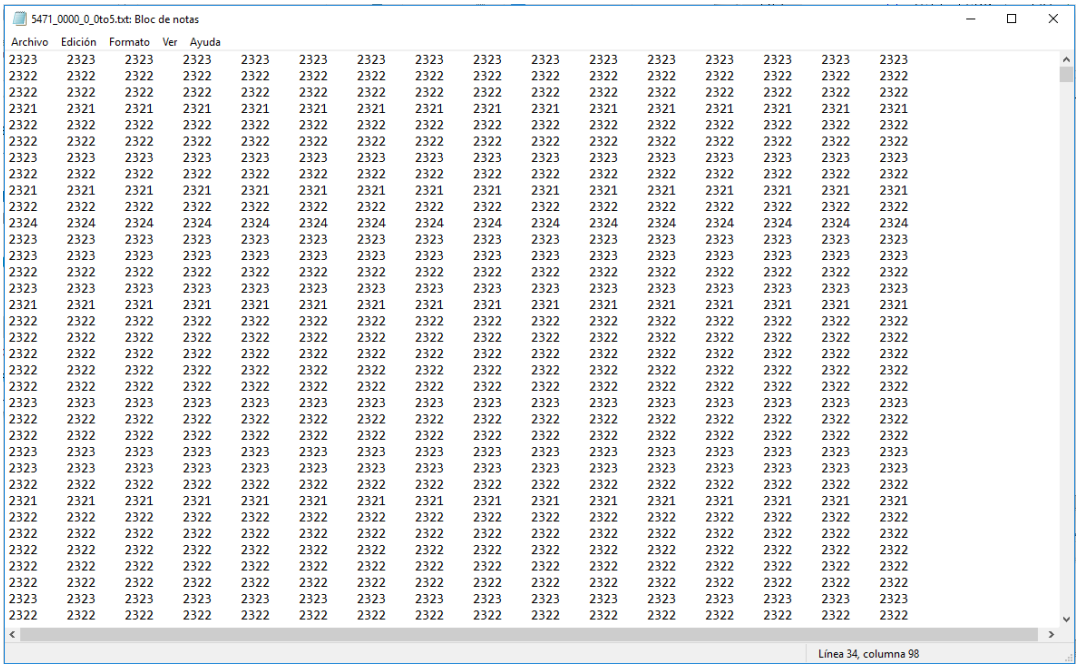

Figura 61. Archivo de entrada de eventos

La simulación se ha hecho sobre las 12 primeras columnas, obteniendo a la salida un archivo llamado output.txt (Figura 62), el cual incluirá todos los datos comprimidos y concatenados en forma binaria.

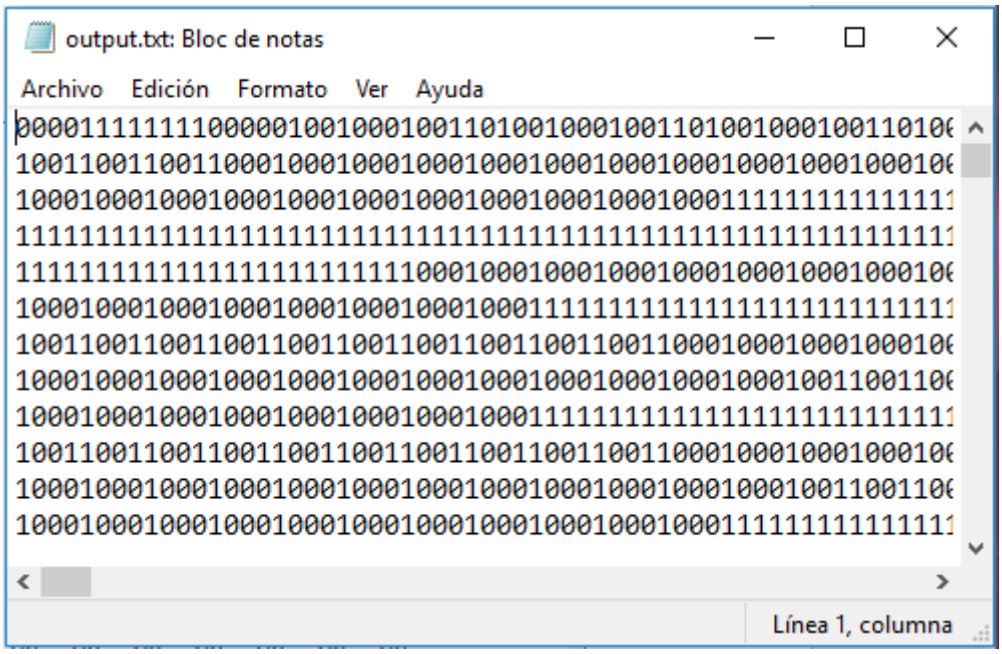

Figura 62. Archivo de salida codificado

Una vez el archivo stimulus.v ha sido configurado correctamente se procede a su compilación. Tras ello se cambia a la vista de simulación en Project Navigator y se lanza desde la opción "Simulate Behavioral Model", la cual abrirá ISim y se podrá pulsar la opción "Run all", que permitirá obtener todos los cambios de señales hasta la instrucción $stop() incluida dentro del archivo.

Previamente se añaden aquellas señales intermedias que se quieran visualizar (Figura 63), puesto que de forma predeterminada solo se muestran las entradas y salidas del módulo instanciado, así como las variables definidas dentro del archivo de simulación. La configuración permite cambiar la base de representación según aquello que facilite la compresión, como es el valor de la variable n_bits a decimal.

Figura 63. Simulación del diseño

Este método, al margen de su sencillez, resulta útil para detectar grandes errores iniciales en el desarrollo, así como para modificar rápidamente las condiciones de la simulación leyendo solo determinados canales.

6.1 Inversión de datos

Uno de los principales problemas que surgieron a mitad del desarrollo fue el orden de concatenación de los datos.

Asumiendo una lectura de izquierda a derecha en el archivo comprimido se detectó una incapacidad para reconocer y distinguir cada uno de los canales en orden. Por tanto, y teniendo como objetivo simplificar el decodificador, se estudió la posibilidad de alterar en cierta medida la forma de concatenación de los datos.

Para comprender el problema se debe tener en cuenta que el orden natural de los canales comienza en 1 y termina en 12, mientras que en la concatenación el primer dato se encuentra en la parte baja del bloque de 16 bits y los bloques de datos posteriores se colocarán encima del mismo en el registro utilizado en el módulo “Shift Register”.

Dato 3 (LSB)	Dato 2	Dato 1

Dato 5	Dato 4	Dato 3(MSB)

En el lado del receptor llega el dato 1, el cual posee la estructura básica de un dato comprimido.

1	0	1	1	1

Por definición, la compresión de Huffman no es igual para todos los valores y, por tanto, puede tomar longitudes desde 2 bits, si se encuentra comprimido e incluyendo el bit extra de estado, hasta 13 si no se encuentra codificado. Partiendo de ello, no es posible averiguar donde comienza el dato.

Una solución rápida que aparece inicialmente es cambiar la posición del bit de estado a la parte baja de cada uno de los datos, de tal forma que el dato 1 se muestre de la siguiente manera.

0	1	1	1	1

Inicialmente sí sería valido para datos sin comprimir, puesto que estos mantienen una longitud constante. Sin embargo, una peculiaridad de Huffman anula esta solución, la secuencia unidireccional del árbol. Para saber el valor de un dato comprimido con este algoritmo se debe comenzar desde el primer bit, siendo cada uno de ellos una desviación por las distintas ramas del árbol generado para la compresión. Como las ramas son de distinta longitud, el orden de lectura de los bits es fundamental para el conocimiento de la longitud del dato, puesto que indica su final.

Surge aquí la necesidad de encontrar una variante en la concatenación. Una consideración podría ser realizarla de la parte alta a la baja, pero esto implicaría conocer tanto la ocupación del buffer de 16 bits como el tamaño del dato actual que se quiera concatenar y, en consecuencia, el uso de sumadores adicionales que empeorarían el path crítico.

La solución adoptada ha sido realizar una doble inversión de los datos. Con ello se consigue por un lado tener el dato 1 en la parte alta del buffer de 16 bits, así como permite una descompresión inequívoca en el receptor debido a que se conoce el origen de cada dato comprimido y, por tanto, su final.

Para implementar la solución se han añadido 3 inversores en el diseño. Primeramente, en los datos de la huella de tiempo son necesarios los 16 bits y es necesario invertir el orden de todos ellos. El

caso restante se da cuando aparece un dato que no va a ser comprimido, el cual necesita una inversión de sus 12 bits más bajos y añadir un bit de estado a 0 en la parte más baja.

Para los datos comprimidos ya se han definido en el módulo de Huffman las secuencias correspondientes de compresión invertidas y con un bit de estado en la parte baja a 1.

Capítulo 7. TestBench

Un banco de pruebas se utiliza para verificar el correcto funcionamiento de un diseño o modelo. En el caso presente se trata de una verificación de un módulo digital y, para ello, se ha hecho uso del software QuestaSim.

El testbench diseñado sigue una estructura jerárquica de archivos. El objetivo final del banco de pruebas será la recopilación de los datos a la salida del módulo digital creado, y su guardado en un archivo de texto en formato binario. Posteriormente se procesarán los datos guardados con la función de decodificación software empleada en el proyecto, en el cual será incluido dicho módulo. Finalmente se compararán los datos que se encontraban en el archivo original con los datos nuevamente decodificados. En caso de comprobar la completa igualdad entre ambos archivos, se puede considerar concluida de forma exitosa la verificación.

Las partes que componen el testbench son variadas, por tanto, se explicarán detalladamente de manera individual, así como su funcionalidad en el total de la jerarquía.

7.1 Compresion_tb

Top de la jerarquía del testbench. En él se establece la escala de tiempos y presión que se utilizarán en el resto de la simulación interna del módulo en la verificación. La única variable descrita de forma única en este módulo es la señal de reloj, nombrada *clk* (Figura 64), y se ha utilizado el tipo de variable bit, puesto que esta señal solo puede tomar dos valores.

```
forever begin

 #12.5 clk  = ~clk;

end
```

Figura 64. Creación de la señal de reloj

Dentro del bloque initial de este archivo se incluye una comitiva forever que definirá los ciclos de reloj. Como se observa en la imagen, se ha establecido un cambio del valor de la variable clk cada 12.5 unidades de tiempo. Como las unidades han sido definidas con la instrucción timescale al comienzo del módulo en nanosegundos, un ciclo total de reloj tendrá una duración de 25 ns.

Otra de las funcionalidades del módulo tb es la de hacer uso de dumpfile y dumpvars (Figura 65). La primera de ellas deposita los cambios en los valores de los registros y conexiones en un archivo cuyo nombre es especificado como parámetro. Este archivo tiene la extensión VCD y guardará los cambios especificados por dumpvars. En el caso de este testbench se ha decidido copiar los cambios de las variables presentes en el módulo tb, así como de todos los instanciados internamente en este.

```
 $dumpfile("test_01.vcd");
 $dumpvars(0, compresion_tb.sv);
clk = 0;
```

Figura 65. Registro de las variables

Es necesaria la instancia del resto de módulos del banco de pruebas, para ello, previamente se importan al comienzo del archivo como la instrucción include. Posteriormente se instancian como se muestra en la Figura 67.

```
`include "compresion_ports.sv"
`include "Modulo_Compresion_adaptado.sv"
`include "Compresion_top.sv"
```

Figura 66. Módulos incluidos para el testbench

```
compresion_ports interfaz(.clk(clk)); //Interfaz for establishing modports and creating conexions
compresion_top top(.ports (interfaz.test)); //Module that instantiates the monitor and scoreboard for comparison purposes
Modulo_Compresor_adaptado dut(.dports(interfaz.dut)); //The device under verification, Modulo_Compresion
```

Figura 67. Instancia de los módulos del testbench

7.2 Compresion_interfaz

Una de las mayores innovaciones incluidas en el lenguaje System Verilog es la posibilidad de crear archivos de interfaces. Básicamente, una interfaz es una o varias recopilaciones de puertos de entrada y salida nombrados bajo un cierto nombre. De esta manera, es posible acceder a una lista completa de señales sin la necesidad de nombrar todas y cada una de ellas en todos los módulos en los que sean utilizadas.

Primeramente, es necesario definir todas las señales que se utilizarán como variables internas en el archivo (Figura 68).

Es importante destacar la ausencia de la señal clk, puesto que esta no es definida en dicho archivo sino, como se mencionó anteriormente, en el top de la jerarquía.

A continuación, se definen los bloques de modports, que son las listas de señales. En este testbench se han creado dos modports diferentes (Figura 69 y Figura 70).

```
bit             rst;
logic           [15:0] data_out;
bit             on;
bit             data_en;
bit             wr_en;
bit             wr_ft;
bit             ch_end;
bit             rd_end;
logic           [15:0] DFdata_out;
bit             DFdata_wren;
```

Figura 68. Variables de la interfaz

```
modport dut( //Modport conexions for Modulo_Compresion
        input       rst,
        input       clk,
        input       data_out,
        input       on,
        input       wr_ft,
        input       data_en,
        input       wr_en,
        input       rd_end,
        input       ch_end,
        output      DFdata_wren,
        output      DFdata_out
        );
```

Figura 69. Modport para el diseño

```
modport test   (input clk,clocking tx, //Modport for creating input signals and save codified data
output rst);
```

Figura 70. Modport para el test

Un modport asociado al dispositivo bajo verificación o DUT. Son todas las señales de entrada y salida del módulo que se utilizarán en la verificación.

El segundo modport es el correspondiente al test. A diferencia del otro, en este se ha empleado un bloque clocking que sincronizará las señales de entrada y salida respecto al reloj, presente en la Figura 71. Se ha incluido la señal de reset fuera de dicho bloque puesto que se trata de una señal asíncrona.

```
clocking tx @(posedge clk);

        output #Thold    data_en;
        output #Thold    wr_en;
        output #Thold    wr_ft;
        output #Thold    rd_end;
        output #Thold    on;
        output #Thold    ch_end;
        output #Thold    data_out;
        input  #Tsetup   DFdata_wren;
        input  #Tsetup    DFdata_out;
```

Figura 71. Bloque de clocking

En el bloque de clocking se especifica el tiempo de hold y setup de las diferentes señales. En el caso de las salidas el momento de actualización de su valor se llevará a cabo Thold unidades de tiempo posteriores al flanco activo de reloj, mientras que las señales de entrada se actualizarán y cogerán el valor correspondiente Tsetup unidades de tiempo previas al flanco. De esta manera se evitará causar metaestabilidad en la simulación que realiza el banco de pruebas. Cabe destacar que las señales definidas en este bloque clocking como outputs se corresponden con los inputs señalados en el bloque modport previo y viceversa.

7.3 Compresion_top

Es el encargado de estructurar las diferentes tareas o tasks a las que se someterá al dispositivo DUT. Consiste en un bloque program que permite la entrada del modport de test. Internamente se crea una instancia del driver, al cual se introducirá dicho modport de manera virtual (Figura 72).

```
`timescale 1 ns/ 1 ps

program compresion_top (compresion_ports.test ports);

`include "compresion_driver.sv"

compresion_driver driver = new(ports);

        initial
        begin

                driver.Write_All();
                driver.Decodifica_Archivo();
                driver.Compara_Archivos();
        end


endprogram
```

Figura 72. Program Compresión_top

Para la ejecución de las diferentes tasks que se llevarán a cabo en el desarrollo del testbench existe un bloque initial dentro del cual se llaman mediante el uso del driver.

7.4 Compresión_drivers

Archivo utilizado para la definición de las tasks llamadas por el archivo top. Al tratarse de un banco de pruebas sencillo, no ha sido necesaria la creación de estímulos ni aserciones o covergroups.

Las señales con las que trabaja el controlador son una interfaz virtual del modport test, con su correspondiente bloque de clocking.

Se han definido distintas variables para el desempeño de las tareas. Primeramente, los enteros utilizados como contadores en los diferentes bloques, i, i_block, i_comparacion e i_control. También se han definido enteros para la llamada a los diferentes archivos de texto que se utilizarán, input_file, out_file y decode_file; así como para los errores de lectura de dichos archivos, err_read, err_inicial y err_decodificado (Figura 73).

```
integer input_file;
integer out_file;
integer decode_file;
integer err_read, err_inicial, err_decodificado;
integer i, i_block, i_comparacion, i_control;
```

Figura 73. Variables de contadores, archivos y errores

Como variables de tipo reg se han definido Channel_Mask de 12 bits. Cada uno de los bits se corresponde con el valor que tomará la señal data_en en los distintos canales, actuando como una máscara.

Para la carga de los datos del archivo original se ha utilizado data_out_aux, constituido por 16 variables de 16 bits cada una de ellas. Se han tomado 16 en lugar de 12 bits debido a que en el archivo procedente existen 16 columnas de datos y el módulo desarrollado trabaja con solo 12. El objetivo de esta variable es obtener los datos del archivo de texto para que puedan ser copiados posteriormente a la señal de entrada data_out presente en el DUT (Figura 74).

```
reg [15:0] data_out_aux [15:0];
reg [15:0] data_inicial [15:0];
reg [11:0] data_decodificada [15:0];
reg [11:0] Channel_Mask;
```

Figura 74. Variables auxiliares

También ha sido necesario definir data_inicial y data_decodificada para permitir las lecturas simultáneas del archivo inicial y el decodificado que se generará. De esta manera será posible su comparación para comprobar el correcto funcionamiento.

A continuación, se presentarán las diferentes tasks definidas en el archivo.

7.4.1 Init_Sequence

Esta task (Figura 75) es la encargada de establecer todas las señales a un valor bajo inicial a la vez que ejecuta un reset del módulo completo.

```
task Init_Sequence();
        begin

                ports.tx.rd_end         <= 1'b0;
                ports.tx.data_en        <= 1'b0;
                ports.tx.wr_ft          <= 1'b0;
                ports.tx.on             <= 1'b0;
                ports.tx.wr_en          <= 1'b0;
                ports.tx.ch_end         <= 1'b0;
                ports.rst               <= 1'b0;
                ports.tx.data_out       <= 16'h0000;

                @(posedge ports.clk);
                @(posedge ports.clk);
```

Figura 75. Task de inicio de secuencia de un evento

7.4.2 Enable_On

Posteriormente a la inicialización, se debe proporcionar una señal de on activa a la máquina de estados para que avance hasta el estado FT (Figura 76).

```
task Enable_On();
        begin
                ports.rst       <= 1'b1;
                ports.tx.on     <= 1'b1;
                Channel_Mask    <= 12'hfff; //All channels enabled

                @(posedge ports.clk);
```

Figura 76. Task de habilitación de la máquina de estados

También se anula la señal de reset poniéndola a nivel alto de nuevo, así como se define la máscara que se utilizará con los distintos canales en la señal data_en.

7.4.3 Write_FT

Task encargada de habilitar la señal wr_ft y dar un valor de 16 bits a data_out para la escritura de la huella de tiempo en el buffer (Figura 77).

```
task Write_FT();
        begin

                ports.tx.wr_ft          <= 1'b1;
                ports.tx.data_out       <= 16'h5d6a; //random value of FT

                @(posedge ports.clk);

                ports.tx.wr_ft          <= 1'b0;
                ports.tx.data_out       <= 16'h0000;

                @(posedge ports.clk);

        end
endtask
```

Figura 77. Task de escritura de la FT

7.4.4 Read_file

La lectura del archivo con la información del evento se lleva a cabo mediante la función $fscanf(). Se deben incluir como parámetros el archivo, el formato de lectura y las variables sobre las que se escribirán los datos leídos, como aparece en la Figura 78.

El formato de lectura presente indica las 16 columnas presentes dentro del archivo, así como su base, en este caso, decimal.

7.4.5 Write_Block

En cada bloque de datos se envían al módulo de compresión los valores en formato de 12 bits de cada uno de los diferentes canales desde el primero hasta el último, y solo aquellos indicados por la máscara.

Para ello, en esta tarea se encuentra un bucle for que permite modificar data_out, mantiene wr_en activa y enmascara data_en para cada canal. En caso de llegar al último canal se activa además la señal de ch_end para indicar que se encuentra al final del bloque. Si además se ha producido un error en la lectura del archivo de texto, implica que se ha llegado al final del evento y, por tanto, se activa la señal de rd_end también (Figura 79).

```
task Read_file();
        begin
                err_read = $fscanf(input_file,
                "%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
                data_out_aux[0],
                data_out_aux[1],
                data_out_aux[2],
                data_out_aux[3],
                data_out_aux[4],
                data_out_aux[5],
                data_out_aux[6],
                data_out_aux[7],
                data_out_aux[8],
                data_out_aux[9],
                data_out_aux[10],
                data_out_aux[11],
                data_out_aux[12],
                data_out_aux[13],
                data_out_aux[14],
                data_out_aux[15]
                );
                if(err_read == -1)
                        begin
                                $display("Hay un err_read %d", err_read);
                                $fclose(input_file);
                        end
        end
endtask
```

Figura 78. Task de lectura del archivo de entrada

```
task Write_Block(); //Write a full block of data before reaching ch_end
        begin
                for(i_block = 0; i_block < 12; i_block++)
                        begin
                                ports.tx.wr_en <= 1'b1;
                                ports.tx.data_en <= Channel_Mask[i_block]; //Valid data only inside the mask.
                                ports.tx.data_out <= data_out_aux[i_block];
                                if(i_block == 11)
                                        begin
                                                ports.tx.ch_end <= 1'b1;
                                                Read_file();
                                                if(err_read != 16)
                                                        ports.tx.rd_end <= 1'b1;
                                        end
                                else
                                        begin
                                                ports.tx.ch_end <= 1'b0;
                                                ports.tx.rd_end <= 1'b0;
                                        end

                                @(posedge ports.clk);

                        end


        end
endtask
```

Figura 79. Task de escritura de una muestra por canal

7.4.6 Write_All

La task principal desde la que se llaman a todas las previamente explicadas. Tras haber escrito el FT, abre los archivos input_file y out_file en modo de lectura y escritura respectivamente. Se ejecuta una primera llamada a Read_file() para inicializar los valores de data_out_aux. Posteriormente comienza un bloque fork, que se utilizará para ejecutar en paralelo dos bloques secuenciales. El primero de ellos trata de un bucle while que realiza llamadas a Write_Block siempre y cuando no se haya llegado al final del archivo de entrada, indicado por la variable err_read. Cuando esto sucede, pone todas las señales a nivel bajo (a excepción del reset) y espera 5 ciclos de reloj, permitiendo al módulo terminar la escritura en el buffer y realizar el padding correspondiente.

El segundo bloque secuencial consiste en un monitor que en cada ciclo de reloj comprueba el estado de la señal de salida DFdata_wren y, en caso de que se encuentre activa, añade al archivo de salida out_file el valor presente en DFdata_out (Figura 82).

Una vez finalizado todo el proceso se cierran ambos archivos con la instrucción $fclose().

7.4.7 Decodifica_Archivo

Tras haber rellenado el archivo de salida con los valores comprimidos, se debe proceder a su decodificación para poder comparar ambos archivos, el original con el decodificado. Para este propósito se ha utilizado una función programada en C++ y proporcionada por el informático José María Benlloch, miembro del proyecto en el que se incluirá el módulo.

Debido a los diversos problemas para ejecutar la función de manera interna con el DPI (Direct Program Interface), se ha tomado la decisión de convertirlo en un ejecutable y, de esta manera, poder ser llamada mediante la instrucción $system() presente en SystemVerilog.

Para ello ha sido necesaria la adquisición del software MinGW, que añade una implementación de compiladores GCC a Windows. Tras ello se ha ejecutado el comando de la Figura 80 en el terminal.

Una vez obtenido el ejecutable, es posible llamar a la función e introducir los parámetros de entrada que son <filein> <tree_file> <fileout> <npmts> <offset>, como se hace en la Figura 81.

Se corresponden con el archivo codificado, el archivo de texto que define el árbol de Huffman empleado, el archivo de salida sobre el que se escribirán los valores decodificados, el número de pmts con el que se trabaja y la cantidad de bits de offset que presenta al inicio (en este caso el FT), respectivamente.

Ahora existe un archivo en el directorio especificado que tiene el mismo formato y distribución de los datos que el archivo original, restando solo compararlos para finalizar el banco de pruebas.

```
C:\Users\Jorge\Desktop\compile>gcc -o decode_huffman.exe decode_huffman_standalone.cpp
```

Figura 80. Comando de decodificación

```
task Decodifica_Archivo();
$system("C:/Users/Jorge/Desktop/compile/decode_huffman.exe \"D:/TFG/Test_Compresion/output.txt\" \
\"D:/TFG/Test_Compresion/tree.txt\" \"D:/TFG/Test_Compresion/5471_0000_0_0to5_decompressed.txt\" 12 16");
endtask
```

Figura 81. Task de decodificación

```
task Write_All();
	begin

		Init_Sequence();
		Enable_On();
		Write_FT();

		input_file = $fopen("D:/TFG/Test_Compresion/5471_0000_0_0to5.txt", "rb");
		out_file = $fopen("D:/TFG/Test_Compresion/output.txt", "w");

		i_control = 0;
		Read_file();
		fork
			begin
				while (err_read != -1)
					begin
						i_control = i_control + 1;
						Write_Block();
						$display("Bloque: %d", i_control);
					end

				ports.tx.rd_end		<= 1'b0;
				ports.tx.data_en	<= 1'b0;
				ports.tx.wr_ft		<= 1'b0;
				ports.tx.on		<= 1'b0;
				ports.tx.wr_en		<= 1'b0;
				ports.tx.ch_end		<= 1'b0;
				ports.tx.data_out	<= 16'h0000;

				@(posedge ports.clk);
				@(posedge ports.clk);
				@(posedge ports.clk);
				@(posedge ports.clk);
				@(posedge ports.clk);
				$display("Terminado");
			end
			begin
				while(err_read != -1)
					begin
						if(ports.tx.DFdata_wren)
							$fwrite(out_file,"%b",ports.tx.DFdata_out[15:0]);
						@(posedge ports.clk);
					end
				for (i = 0; i<7; i++)
					begin
						if(ports.tx.DFdata_wren)
							$fwrite(out_file,"%b",ports.tx.DFdata_out[15:0]);
						@(posedge ports.clk);
					end
			end

		join
		$fclose(out_file);
		$fclose(input_file);
	end
endtask
```

Figura 82. Task de escritura de las muestras de un evento

7.4.8 Compara_Archivos

Para la comparación se ha creado una task compuesta por un gran bucle do while dentro del cual se leen ambos archivos y se comparan los valores presentes en ellos. Cabe destacar que, mientras que del archivo inicial se leen 16 columnas, como ya se ha mencionado anteriormente, del archivo decodificado se leen solo 12.

En caso de no haber recibido ningún error de lectura en el archivo inicial se continúa comparando. Y si se da la situación en la que dos datos no coinciden se muestra un error en pantalla indicando el dato que se esperaba obtener y el presente en el archivo decodificado (Figura 83).

```
task Compara_Archivos();
begin
        decode_file = $fopen("D:/TFG/Test_Compresion/5471_0000_0_0to5_decompressed.txt", "rb");
        input_file = $fopen("D:/TFG/Test_Compresion/5471_0000_0_0to5.txt", "rb");
        do
        begin
                err_inicial = $fscanf(input_file, "%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
                data_inicial[0], data_inicial[1], data_inicial[2], data_inicial[3],
                data_inicial[4], data_inicial[5], data_inicial[6], data_inicial[7],
                data_inicial[8], data_inicial[9], data_inicial[10], data_inicial[11],
                data_inicial[12], data_inicial[13], data_inicial[14], data_inicial[15]
                );
                err_decodificado = $fscanf(decode_file, "%d %d %d %d %d %d %d %d %d %d %d %d",
                data_decodificada[0],data_decodificada[1],data_decodificada[2],data_decodificada[3],
                data_decodificada[4],data_decodificada[5],data_decodificada[data_decodificada[7],
                data_decodificada[8],data_decodificada[9],data_decodificada[10],data_decodificada[11]
                );

                if(err_inicial != -1)
                        begin
                        for(i_comparacion = 0; i_comparacion < 12; i_comparacion = i_comparacion + 1)
                                begin
                                if(data_inicial[i_comparacion] != data_decodificada[i_comparacion])
                                        //$display("Original: %d  Decodificado: %d", data_inicial[i_comparacion], data_decodificada[i_comparacion]);
                                        $error("Original: %d  Decodificado: %d", data_inicial[i_comparacion], data_decodificada[i_comparacion]);
                                end
                        if((err_inicial != -1) & (err_decodificado == -1))
                                $display("La extensión de los archivos es diferente" );
                        end
                end
                while((err_inicial!=-1));
                $display("Testbench finalizado");
                $stop();
end
endtask
```

Figura 83. Task de comparación

Por la ventana de comandos indica la finalización del testbench (Figura 84).

```
# Testbench finalizado
# ** Note: $stop    : D:/TFG/Test_Compresion/compresion_driver.sv(238)
#    Time: 120000262500 ps  Iteration: 1  Instance: /compresion_tb/top
# Break in Task compresion_top/compresion_driver::Compara_Archivos at D:/TFG/Test_Compresion/compresion_driver.sv line 238
```

Figura 84. Finalización del testbench

Capítulo 8. Conclusión y futuros trabajos propuestos

Una vez el proceso de diseño ha sido finalizado y verificado para su correcto funcionamiento se ha implementado dentro de la FPGA utilizada junto al resto de elementos presentes para el desarrollo del experimento. La adaptación ha sido correcta y no han existido errores posteriores en el mismo.

Para la obtención de valores más precisos de la compresión media que se llegará a conseguir, Raúl Esteve Bosch ha introducido sobre el hardware una gran cantidad de eventos diferentes, teniendo como resultados los valores presentes en la Figura 85.

RUN 7298 y 1299, 10000 eventos
- RAW: Data Rate 17 - 22MB/s en GDC, Trigger rate 20.6 Hz, 40 ficheros, 19.047.979 kBytes de datos
- DComp: Data Rate 3.5 - 4MB/s en GDC, Trigger rate 20.6 Hz, 8 ficheros, 3.362.285 kBytes de datos

En fin, lo esperado, en torno al 82.3 % de compresión en NEW.

Figura 85. Comparación flujo de datos con compresión

Se han alcanzado valores superiores al 80 % de compresión, un resultado satisfactorio para el proyecto de compresión creado.

Como extensiones al trabajo aquí presentado, se propone un módulo digital paralelo que permite realizar cálculos estadísticos de los valores de las diferencias entre muestras, así como la creación de un árbol de Huffman que sea actualizable y se integre en el LUT de Huffman, de esta forma se optimizarán más los resultados y será extensible a una mayor cantidad de señales. También deben de ser modificables los umbrales de entrada al comparador.

Capítulo 9. Bibliografía

[1] Álvarez, V.; Herrero-Bosh, V.; Esteve, R.; Laing, A.; Rodríguez, J.; Querol, M.; Monrabal, F.; Toledo, J. F. and Gómez-Cadenas, J.J. *The electronics of the energy plane of the NEXT-White detector*. Nuclear Inst. And Methods in Physics Research, 2019.

[2] LC TPC, "Working Principle of a TPC," https://www.lctpc.org/e8/e57671 [Online].

[3] Esteve, R. *Data Format and GUI*.

[4] Github, "Desarrollo de un módulo digital de compresión," https://github.com/JorgePonce97/CompresionTFG [Online].

[5] Xilinx, "Virtex-6 Family Overview", 2015.

[6] Morales Sandoval M. *Notas sobre Compresión*. INAOE, 2003.

[7] Gupta R.; Kumar M. and Bathla R. *Data Compression – Lossless and Lossy Techniques. International* Journal of Application or Innovation in Engineering & Management.