_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESTUDIO DE IMPLEMENTACIÓN DEL ALGORITMO LZW (Lempel-Ziv-Welch) EN UNA PLATAFORMA ZYNQ 7000 DE XILINX

Autor: Borja Gastaldo Ramon

Tutor: 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 2015-2016

Valencia, 24 de julio de 2014

Resumen

El trabajo fin de grado propuesto tiene como objetivo explorar las opciones de implementación del algoritmo de compresión sin pérdidas LZW sobre una plataforma programable System-On-Chip llamado “Red Pitaya” de la empresa Xilinx.

Dicha plataforma consta de un procesador “*Dual Core ARM Cortex A9*” embebido en una FPGA *Xilinx Zynq 7010* de complejidad media. Habitualmente la implementación sobre el ARM se realiza empleando un lenguaje de programación software (C o similares) siendo más flexible pero más lenta. La implementación hardware sobre el tejido de la FPGA es más compleja y se suelen emplear lenguajes de descripción como Verilog o VHDL, alcanzando las mayores velocidades y el menor consumo.

El Trabajo Fin de Grado se centrará primeramente en implementar el algoritmo de compresión LZW en la plataforma “RedPitaya” para su funcionamiento en tiempo real sobre señales biomédicas obtenidas con la misma plataforma seguido del desarrollo de otra aproximación del algoritmo que se adecua más a las necesidades reales del trabajo.

Resum

El treball final de grau proposat té com a objectiu explorar les opcions d'implementació de l'algoritme de compressió sense pèrdues LZW sobre una plataforma programable System-On-Chip anomenat "RedPitaya" de l'empresa Xilinx.

Aquesta plataforma consta d'un processador "Dual Core ARM Cortex A9" embegut en una FPGA Xilinx Zynq 7010 de complexitat mitjana. Habitualment la implementació sobre l'ARM es realitza emprant un llenguatge de programació software (C o similars) sent més flexible però més lenta. La implementació hardware sobre el teixit de la FPGA és més complexa i se solen emprar llenguatges de descripció com Verilog o VHDL, aconseguint les majors velocitats i el menor consum.

El Treball Fi de Grau es centrarà primerament en implementar l'algoritme de compressió LZW a la plataforma "RedPitaya" per al seu funcionament en temps real sobre senyals biomèdiques obtingudes amb la mateixa plataforma seguit del desenvolupament d'una altra aproximació de l'algoritme que s'adequa més a les necessitats reals del treball.

Abstract

The final project proposed aims to explore options for implementing lossless compression LZW algorithm on a programmable System-On-Chip platform called "Red Pitaya" from Xilinx.

This platform consists of a "Dual Core ARM Cortex A9" embedded processor in an medium complexity Xilinx Zynq 7010 FPGA. Usually, ARM implementations are performed using a software programming language (C or similar), being more flexible but slower. The hardware implementation on an FPGA tissue is more complex and description languages such as Verilog or VHDL are often used, reaching higher speeds and lower power consumption.

The final project will focus primarily on implementing the LZW compression algorithm in the "Red Pitaya" platform for real-time operation on biomedical signals obtained with the same platform followed by the development of another approach of the algorithm that fits over the real needs of the project.

Índice

Capítulo 1. Introducción 2

1.1 Entropía y Complejidad 2

1.2 Algoritmo Lempel-Ziv-Welch 3

1.3 Hardware y Software 5

Capítulo 2. Objetivos del TFG 7

Capítulo 3. Metodología 8

3.1 Distribución de tareas 8

3.2 Diagrama temporal 9

Capítulo 4. Algoritmo Compresión LZW 10

4.1 Programación 10

Capítulo 5. Diferente aproximación al algoritmo LZW 12

5.1 Introducción 12

5.2 Programación 12

5.3 Resultados 14

Capítulo 6. Conclusión y propuesta de trabajo futuro 15

6.1 Mejora con la ayuda de al FPGA 16

6.2 Adaptación y mejora del algoritmo para un tipo de datos concretos. 17

Capítulo 7. Bibliografía 18

Capítulo 8. Anexos 19

Capítulo 1. Introducción

1.1 Entropía y Complejidad

La entropía de un sistema se define como el ratio de generación de nueva información. Una medida de lo rápido que una información relacionada a un estado actual se vuelve irrelevante.

Cuanto más aleatoria o irregular sea la información a analizar más alta será la entropía mientras que cuanto más regular y estable sea esta la entropía será baja llegando a 0 para señales continuas.

La entropía de Shannon se define como la probabilidad P(x) cuando un sistema está en el estado x para un estado A

$$H(A) = -\int_{x \in A} P(x) \log_2\big(p(x)\big) dx$$

Forma discreta

$$\sum -P(x) \log_2 P(x)$$

La teoría de Shannon es muy importante en las comunicaciones ya que mide la incertidumbre del siguiente estado del sistema que se analiza.

Por otra parte la complejidad algorítmica es la medida de la regularidad de una secuencia de símbolos.

Para calcular esta complejidad es necesario al principio emplear un esquema de codificación para convertir estos símbolos en una secuencia de elementos definiéndose como el número de secuencias que se observan en fracciones de la máxima secuencia posible. De esta forma la complejidad es equivalente al ratio de compresión conseguido por el algoritmo Lempel-Ziv-Welch.[1]

1.2 Algoritmo Lempel-Ziv-Welch

El Lempel-Ziv-Welch es un algoritmo de compresión sin pérdidas, posteriormente LZW, creado por Abraham Lempel, Jacob Ziv, y Terry Welch.

Este algoritmo presenta una ventaja clave frente a otros, ya que no necesita de un doble análisis de la señal, para primero crear el diccionario y luego, con este, hacer otra pasada para codificar el archivo. Con el sistema LZW el diccionario se va creando a la vez que se comprimen los datos con este, todo esto en una sola pasada sin necesidad de volver a analizar los datos entrantes.

Otra de las ventajas que encontramos es la nula necesidad de guardar o enviar el diccionario para la decodificación del archivo resultante con lo que permite un menor gasto de memoria, ya que esta puede ser liberada al acabar la codificación de los datos.

Para la codificación, se debe inicializar el diccionario con los 255 caracteres simples correspondientes al código ASCII, de esta forma las combinaciones de un solo carácter ya se encuentran dentro del diccionario de tal forma que la siguiente posición será ocupada por la combinación de dos caracteres. De esta forma se va desarrollando el diccionario a la vez que codificamos los datos a la salida, una vez leído el primer elemento se comprueba si se encuentra en el diccionario, este se encontrara por lo ya comentado con anterioridad de la inicialización, así que no se efectuara ninguna operación. Se leerá el siguiente elemento comprobando si la combinación del primer y segundo elementos se encuentra dentro del diccionario, si está presente se volverá a leer otro elemento más, de no estarlo se procederá a guardar la combinación en la siguiente posición libre del diccionario para su posterior utilización, se codificara el primer elemento de los dos obtenidos y con el segundo elemento se volverá a operar como ya se ha indicado, y así de forma recursiva hasta el fin del conjuntos de elementos, obteniendo a la salida la entrada codificada.

Input	Current String	Seen this Before?	Encoded Output	New Dictionary Entry/Index
b	b	yes	nothing	none
ba	ba	no	1	ba / 5
ban	an	no	1,0	an / 6
bana	na	no	1,0,3	na / 7
banan	an	yes	no change	none
banana	ana	no	1,0,3,6	ana / 8
banana_	a_	no	1,0,3,6,0	a_ / 9
banana_b	_b	no	1,0,3,6,0,4	_b / 10
banana_ba	ba	yes	no change	none
banana_ban	ban	no	1,0,3,6,0,4,5	ban / 11
banana_band	nd	no	1,0,3,6,0,4,5,3	nd / 12
banana_banda	da	no	1,0,3,6,0,4,5,3,2	da / 13
banana_bandan	an	yes	no change	none
banana_bandana	ana	yes	1,0,3,6,0,4,5,3,2,8	none

Fig. 1 LZA

En forma seudocódigo esto se entenderá mejor:

```
string s;
char ch;

while (Queden datos que leer)
{

    ch = lectura de Nuevo carácter;
    if (diccionario contiene s+ch)
    {

        s = s+ch;

    }
    else
    {

        Añadir s+ch al diccionario en la posición vacía siguiente;
        Codificamos s y enviamos a la salida;
        s = ch;

    }

}

Codificamos s y se envía a la salida;
```

La otra aproximación al algoritmo que se utiliza en este proyecto difiere de la ya comentada por lo que también se explicara a continuación.

Siendo $A=[s_1, s_2, s_3...s_i]$ una secuencia completa, *P* y *Q* *subsecuencias* de *A*, *PQ* es la concatenación de estas dos y PQ^- es *PQ* sin el ultimo carácter de la *subsecuencia Q*.

Con *c* como contador de nuevas secuencias inicializado a 1, $P = s_1$ and $Q = s_2$ así pues $PQ^- = s_1$ con i = 1 y j = 1 como posiciones de la *subsecuencia* se comprueba si *Q* pertenece a PQ^- de ser así se incrementa j = j+1 y $Q=Q(s_{i+j})$ repitiéndose la operación hasta que *Q* no se encuentre en PQ^- momento en el cual se incrementara $c=c+1$, $P = PQ$ y $Q=Q(s_{i+j+1})$ se actualizarán $i=i+j$ *y* $j=1$, para seguir de forma recursiva hasta llegar al final de la secuencia.[2]

Cuando termine se obtendrá en *c* el número de nuevas *subsecuencias* contenidas en *A.*

A partir de aquí para obtener la complejidad la cual es independiente del tamaño *A,* c debe ser normalizada.

Si el tamaño total de la secuencia es n y el número de símbolos distintos es α obtenemos que

$$Comp = \frac{c * \log n}{n * \log \alpha}$$

1.3 Hardware y Software

El hardware en el que se implementara el algoritmo es un dispositivo llamado RedPitaya desarrollado por la compañía del mismo nombre Spin-off de Instrumentation Technologies líder en el diseño y desarrollo de sistemas de instrumentación.

Cuenta con un sistema hardware y software ambos gratuitos, con aplicaciones de desarrollo propias como un Osciloscopio digital, un generador de funciones, un analizador de espectro..etc, además de contar con interfaz de conexión a Matlab y Labview, todo open source.

Las características técnicas son muy variadas contando con un ARM Cortex A9 de doble núcleo y una FPGA Zynq 7010 de Xilinx, módulo de conexión wifi y sistema operativo Linux.

Por la parte de las conexiones tenemos 4 conectores RF SMA, dos de entrada y dos de salida, además de mini USB y conector Ethernet.

Esta combinación de ARM mas FPGA permite una alta flexibilidad a la hora de trabajar con la placa, estas dos vertientes pueden funcionar de manera diferenciada pudiéndose apagar la parte de FPGA para un menos gasto de energía si fuera necesario.

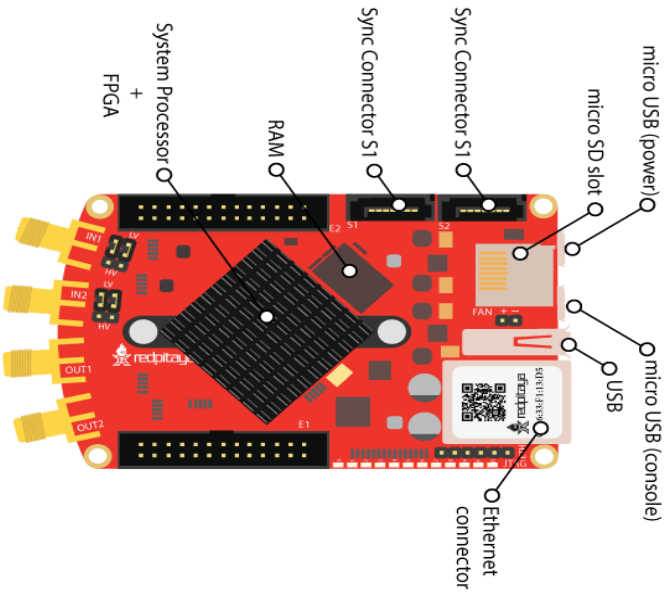

Fig. 2 RedPitaya

La inclusión de un procesador de aplicaciones permite la utilización de sistemas operativos de alto nivel como pude ser Linux u otros sistemas estándar utilizados con la familia Cortex-A9.

Por otra parte la FPGA (Field Programmable Gate Array) otorga la posibilidad de solucionar problemas que un procesador es incapaz de resolver o al menos no de manera óptima como puede ser el procesamiento en paralelo. El sistema de bloques de lógica de la FPGA trabaja

realmente en paralelo, no de manera secuencial como los procesadores, más rápido y a un coste energético menor.

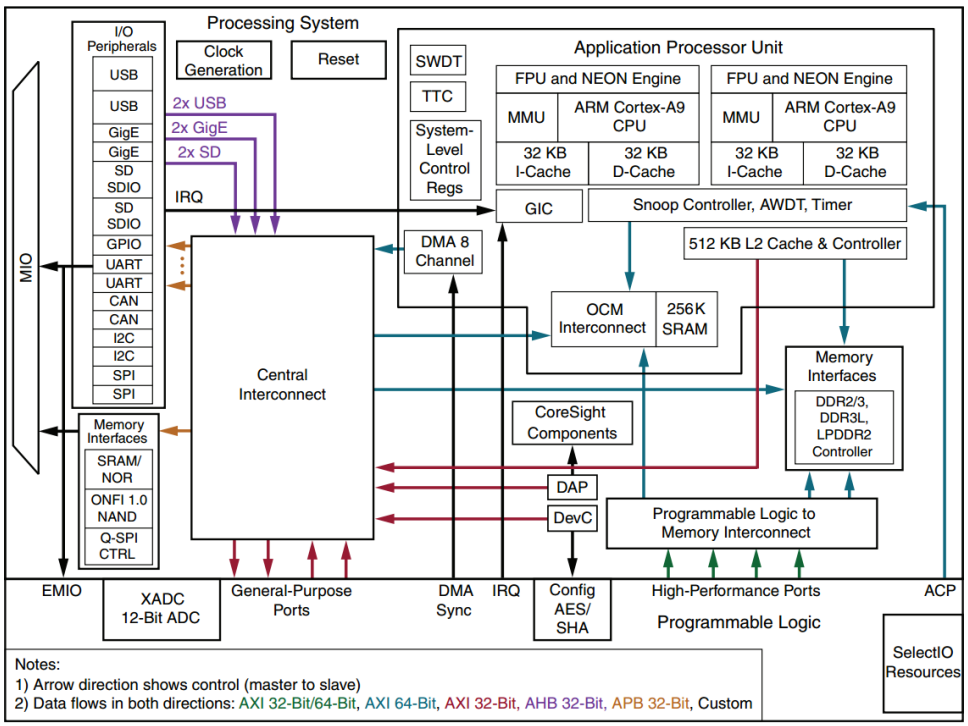

Fig. 3 Bloque funcional Zynq-7000 All Programmable SoC[5]

En cuanto al uso de software se ha visto necesaria la utilización de IDE (Integrated Development Environment) tales como NetBeans o Eclipse, para hacer el proceso de debug de los algoritmos implementados, ya que en todo momento se ha necesitado monitorizar la memoria que se usa al almacenar datos referentes al algoritmo, de forma que siempre se tenía constancia de los valores guardados en los diccionarios, para así comprobar que todo esto es correcto y va cumpliendo el algoritmo en tiempo de ejecución.

Por otro lado ha sido necesaria la utilización del Matlab para la creación de señales de test diversas y adaptación de otras para su posterior uso. Todas estas señales se utilizan para la comprobación del programa, así como el cálculo de la entropía de Shannon en el mismo Matlab, para así tener un valor de referencia con el que comprobar los resultados.

Se han usado las distintas herramientas de las proporcionadas por parte de la empresa de la RedPitaya tales como el "cross compiler" y la conexión a través de SSH con el instrumento.

Capítulo 2. Objetivos del TFG

El principal objetivo de este trabajo es la implementación del algoritmo LZW en el sistema de adquisición RedPitaya, para acelerar el procesamiento de los datos obtenidos sin necesidad de tener que sacar los datos de la plataforma, para poder procesarlos de forma independiente con programas como Matlab, ya que esto es un impedimento en la realización de las medidas de *Neurofeedback*.

Es esencial que el tratamiento de los datos se realice de forma rápida para saber, de esta forma, si los datos que se están obteniendo de la respuesta de los estímulos a pacientes, contienen información o no.

Esto se consigue con el cálculo de la complejidad ya explicado con anterioridad que indica la cantidad de información que contiene una secuencia dada.

Capítulo 3. Metodología

Este proyecto se puede dividir en tres grande bloques.

1. Documentación y estudio del algoritmo propuesto
2. Implementación del mismo con Matlab y C/C++
3. Pruebas sobre la plataforma RedPitaya

Tras tener claro tanto el funcionamiento del algoritmo como su teoría básica, se pasó a implementarse en C/C++ con la ayuda del Matlab, elemento crucial en algunas partes del proyecto por la facilidad para crear varias señales de prueba así como calcular con él la entropía de Shannon de dichas señales para poder comparar más tarde con el programa en C/C++.

A principios de julio se hizo un cambio en el enfoque del algoritmo ya que se pudo comprobar que el sistema de compresión funcionaba bien para archivos de un tamaño bastante superior al que se pretendía explorar en este proyecto pero no de forma tan idónea para este tipo de datos.

Con todo esto claro se pasó a implementar el nuevo sistema más acorde a las necesidades del proyecto.

Por último se probaron los programas sobre la plataforma para comprobar su funcionamiento en un entorno real.

3.1 Distribución de tareas

El proyecto ha sido dividido en varias:

1. Documentación sobre algoritmo. (3 semanas)
   Durante este periodo se buscó la información relevante sobre el algoritmo así como toda la teoría necesaria sobre entropía y complejidad necesaria para elaborar el proyecto.
2. Búsqueda y estudio sobre librerías disponibles en la arquitectura ARM. (2 semanas)
   Tarea necesaria para no utilizar librerías o funciones no soportadas por los ARM ya que el programa seria inservible en su función real.
3. Programación del algoritmo. (4 semanas)
   Aquí se programa el algoritmo tanto en su forma de compresor como en su segunda aproximación.
4. Pruebas sobre la RedPitaya. (1semana)

Pruebas sobre la RedPitaya con intención de comprobar que todo lo programa funciona sin problemas en la plataforma y con las señales necesarias para ello.

5. Conclusiones finales. (3dias)
   Argumental las conclusiones obtenidas de la programación e implementación del algoritmo.
6. Redacción del proyecto. (2 semanas)
   Preparación del documento que va a ser presentado.

3.2 Diagrama temporal

En este diagrama se desglosa el tiempo empleado en cada tarea de forma grafica

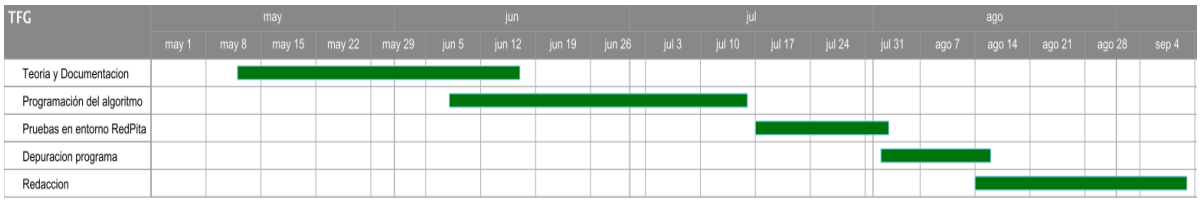

Fig. 4 Diagrama de Gannt

Capítulo 4. Algoritmo Compresión LZW

En este capítulo será desglosado el programa de compresión para una mayor claridad.

4.1 Programación

El primer programa tiene dos partes diferenciadas una primera la de compresión y otra segunda la de descompresión, que con la filosofía del algoritmo LZW, funcionan de manera similar creando las dos el diccionario de forma iterativa mientras se codifican o descodifican los datos entrantes.

Para la parte de compresión se utilizara un map para crear el diccionario.

```
#include <unordered_map>
```

El unordered_map es una función de C++11 que permite crear una matriz de rápido acceso y que puede relacionar de forma interna datos de diferente tipo. Esta función utiliza un hash para ubicar el dato en una posición de memoria. El tamaño del diccionario más óptimo es de 2^{16}, con este tamaño se evita el problema de tener que restablecer el diccionario de 0 cada vez que este se llena por completo.

```
#include <unordered_map>

std::unordered_map<std::string, int> dictionary(DICT_MAX);

    dictionary.clear();

    for ( int unsigned i = 0 ; i < 256 ; i++ ){

        dictionary[std::string(1,i)] = i;

    }
```

Aquí ya se inicializan las 255 primeras entradas del diccionario tal como dicta el algoritmo anteriormente expuesto.

A partir de este momento empieza el bucle compresor que funcionara hasta encontrar el EOF(end of file) del archivo.

```
ch=fgetc(in);

str=str+ch;
```

Con esta simple función se comprueba si el dato se encuentra dentro del diccionario o no, gracias a la función FIND de la clase unordered_map.

```
If(dictionary.find(str) ==dictionary.end())
```

Si el dato se encuentra se procederá a leer el siguiente dato y con *str=str+ch;* concatenarlo con el anterior y volver a comprobar si se encuentra dentro del diccionario.

Si el dato no se encuentra dentro del diccionario es añadido al diccionario como un nuevo dato en la siguiente posición vacía del mismo.

```
dictionary.insert(std::make_pair(str, posdic));

posdic++;
```

Justo después se procederá a buscar el str, sin el último dato leído, en el diccionario

```
str.erase(str.size()-1

std::unordered_map<std::string,int>::const_iterator encode =
dictionary.find (str);
```

Para a continuación sacar el dato con la codificación que le otorga el diccionario, ya que el fputc solo nos permite sacar datos de 8 bits y a partir del 256 los datos codificados superan los 8 bits primera se extraerán los 8 bits más altos y luego los 8 más bajos con estas operaciones.

```
dato=(encode->second);

fputc((dato >> 8), out);

fputc(dato , out);
```

Para el objetivo del trabajo la extracción de datos en un archivo comprimido no es del todo necesaria pero ha sido llevado a cabo para la posterior comprobación con el descompresor y para ver realmente que la compresión se efectuaba sin ningún tipo de problema.

Finalmente el último dato leído anteriormente pasa a ser el primer dato con el que continuar las iteraciones del algoritmo.

```
str = ch;
```

Una vez la codificación ha sido realizada la complejidad es calculada como el ratio de compresión conseguido con el algoritmo.

La parte de descompresión carece de interés en este trabajo ya que solo se ha realizado para comprobar que la compresión se está haciendo de forma correcta.

Capítulo 5. Diferente aproximación al algoritmo LZW

5.1 Introducción

Tras obtenerse buenos resultados por parte del compresor y realizar varias pruebas con diferentes datos y diferentes tamaños de datos, fue vista una clara dependencia entre la efectividad del algoritmo y el tamaño del archivo a ser tratado.

Esto es debido a que la compresión empieza a ser optima una vez el diccionario tiene suficientes datos para empezar a codificar los datos entrantes, provocando que para conjuntos de datos de tamaño reducido el ratio de compresión sea altamente dependiente de los datos y el orden de estos.

Por este motivo se planteó esta otra aproximación al algoritmo con la complejidad normalizada que hace que el cálculo sea menos dependiente del tamaño del archivo y por lo tanto más óptimo para el tipo de datos que se pretenden tratar.

5.2 Programación

En esta implementación se utiliza un sistema de administración de datos distinta al de la primera aproximación, debido a que necesitaremos recorrer todos los datos en busca de nuevas *subsecuencias.* Esta búsqueda se hace de forma recursiva, así que leeremos todos los datos y los guardaremos en un vector de *pairs* que nos permite asociar dos tipos de datos distintos y trabajar conjuntamente con ellos.

```
std::vector<std::pair < int,int > > myvector;
```

El vector se ira creando de forma dinámica con cada dato “o” y índice “i”.

```
myvector.push_back(std::pair<int,int> (o, i));
```

Una vez almacenados los datos junto con su índice, dentro del vector los datos son ordenados de mayor a menor con un *sort* para acotar de esta forma el número de símbolos disponibles en 100; numero al que se ha llegado con varias pruebas buscando el compromiso entre velocidad del algoritmo y precisión del cálculo.

```
std::sort (myvector.begin(), myvector.begin()+tam);
```

Una vez sustituidos los símbolos deshacemos el *sort* con el índice anteriormente creado en el vector de *pairs* volviendo a tener el vector en el orden inicial pero con los nuevos símbolos,

Para ello se hace uso de un vector auxiliar de tamaño de memoria ya reservado y que nos permite dejar de lado el vector de *pairs* que cuyo uso es más pesado y costoso para un procesador.

```
aux[myvector[i].second]=y[i]+1;
```

A partir de este punto empieza el bucle encargado de buscar subsecuencias dentro de secuencia total inicial, se inicializan las variables tal como se comenta en la introducción

```
int ns=1,nq=1,k=2,cont=0;
while (k<tam){
            if (issubstring(aux,np,nq)){
             /* Q is a substring of SQpi */
                  nq++;
            }
            else{
              /* Q is a new symbol sequence */
              cont++;
              np=np+nq;
              nq=1;
            }
            k++;
      }
```

Este bucle se encara de administrar las variables tal como indica el algoritmo aumentando el contador en 1 cada vez que se encuentra una nueva secuencia.

Para la búsqueda exhaustiva se llama a la función *issubstring(aux,np,nq)* que se encarga de buscar las nuevas secuencias pasándole un vector y el rango valores que ha de buscar siendo np y nq.

Este bucle *for* anidado se encargara de comparar cada símbolo de PQ^{-} con Q para ver si se encuentra dentro devolviendo un 1 si es parte o un 0 si es un nuevo símbolo.

```
for (i=0; i<=(ns-nq); i++){
            same=1;
            for (k=0; k<nq; k++){
                  if (aux[i+k]!=aux[np+k])
            same=0;
            }
            if (same){
             return(1);
            }
      }
      return(0);
```

Esta parte del programa es la más costosa computacionalmente hablando; conforme va aumentando el tamaño del archivo lo hace el número de veces que se han de recorrer las posiciones del vector, lastrando el cálculo y haciendo que sea poco viable.

5.3 Resultados

Después de las pruebas pertinentes en la RedPitaya se puede concluir que para el tamaño de datos utilizado en la aplicación de *neurofeedback* el tiempo de proceso es óptimo.

Para tamaños mucho mayores el proceso empieza a necesitar un mayor tiempo de procesado por parte del ARM llegando a ser inviable.

Capítulo 6. Conclusión y propuesta de trabajo futuro

En este proyecto se ha realizado la implementación del algoritmo LZW (Lempel-Ziv-Welch) para el cálculo de la complejidad de señales biomédicas con intención de determinar si estás tienen información relevante para la investigación o no.

Tras las diferentes pruebas efectuadas tanto con señales creadas con Matlab como señales reales se puede observar que el citado algoritmo es bastante útil a la hora de calcular la complejidad de bioseñales, siendo este bastante robusto al ruido como podemos ver en la Fig.5 manteniéndose estable independientemente de la potencia del ruido.

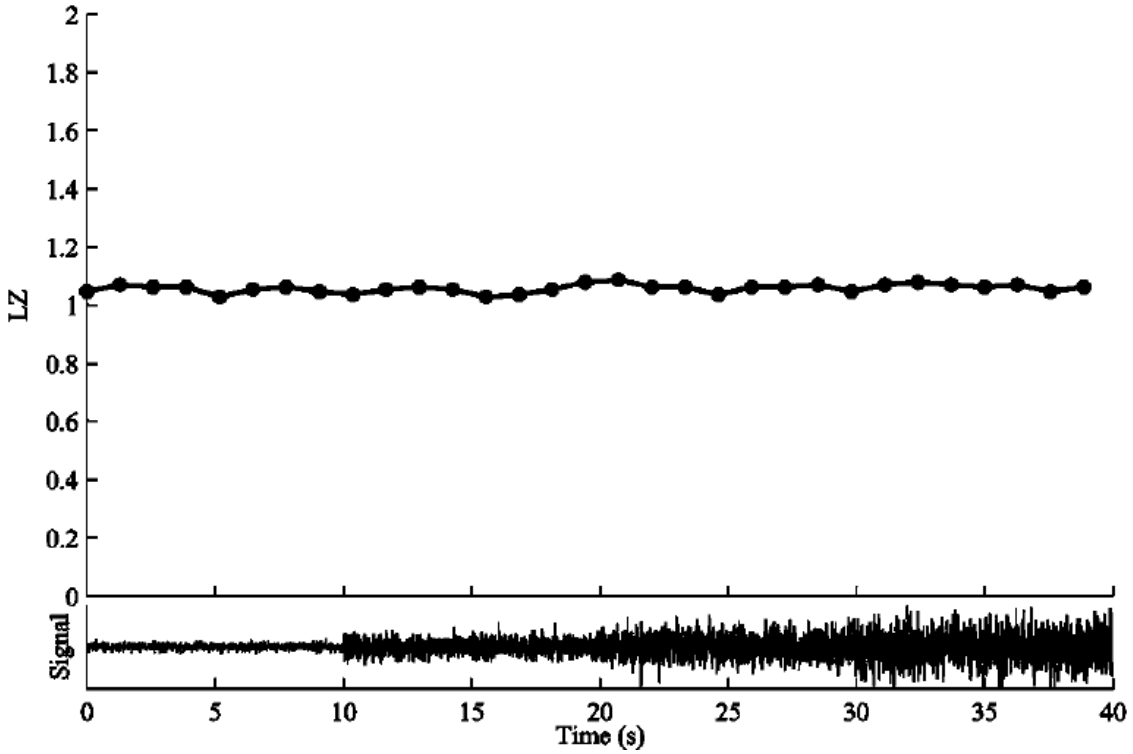

Fig. 5 LZ respecto potencia de ruido

También se observa una sensibilidad a los cambios de ancho de banda de las señales, esto puede ser de alta importancia en el análisis de bioseñales, ya que muchas aplicaciones están interesadas en estimar el ancho de banda de señales fisiológicas o residuales, estos cambios pueden observarse en la Fig.6.[3]

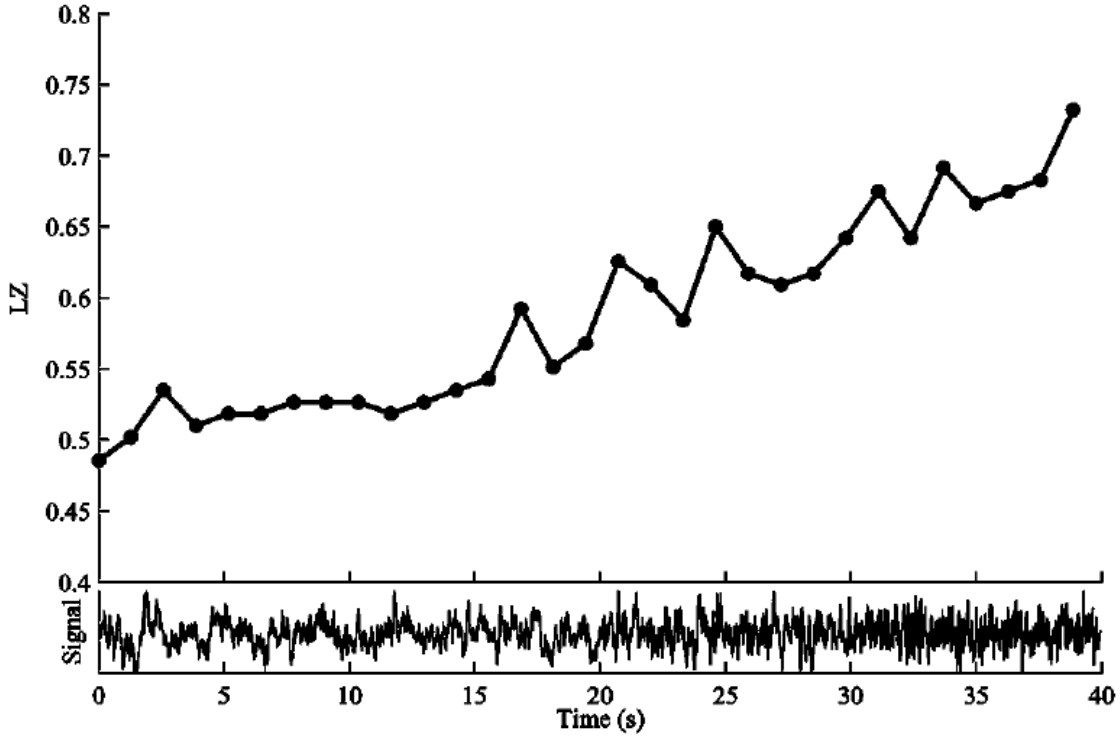

Fig. 6 LZ respecto variación ancho de banda

Finalmente se puede determinar que la utilización de este algoritmo con este tipo de señales es una propuesta acertada, además de lo suficientemente sencilla de implementar en un entorno como puede ser un ARM con el único impedimento ya comentado anteriormente del aumento sustancial del tamaño de las señales a analizar que lastraría el procesador.

6.1 Mejora con la ayuda de al FPGA

Una de las principales mejoras propuestas como trabajo futuro seria la implementación mixta entre el ARM y la FPGA que nos permite la RedPitaya.

La implementación del primer algoritmo explicado en este trabajo podría ser abordado de forma directa, otorgando al procesador la única misión de adaptar los datos para ser enviados a la FPGA mientras que en esta se dividirá la memoria en varios bloques para utilizar la versatilidad que otorga una FPGA para las operaciones en paralelo como la búsqueda en nuestro diccionario aumentando la rapidez con la que buscamos, a la salida de cada memoria se utilizara un comparador para comparar el dato de entrada con los datos de cada una de las memorias añadiendo el dato a la memoria de no encontrarse este ya dentro de alguna de ellas.[4]

El uso de la FPGA en la segunda aproximación al algoritmo se vuelve un poco más complicada ya que las operaciones matemáticas como la división se vuelven más costosas de desarrollar en una FPGA con lo cual se tendría menos margen a la hora de implementarse.

La única parte susceptible de ser implementada de forma directa en la FPGA sería la parte del algoritmo que se centra en la búsqueda de subsecuencias dentro de la señal a analizar ya que con las RAMS de doble puerto que nos proporciona Xilinx en su plataforma nos permitirían una búsqueda rápida aunque para aumentar la eficiencia de este sistema habría que hacer un estudio más amplio sobre las señales a tratar para averiguar que numero de memorias será optimo utilizar sin hacer un gasto excesivo e innecesario de memoria.

6.2 Adaptación y mejora del algoritmo para un tipo de datos concretos.

Otra propuesta de trabajo futuro seria el estudio de los datos a analizar para mejorar los algoritmos ya presentes para que funcionen de forma más adecuada a estos datos.

Como principal ejemplo se podría adaptar el diccionario de compresión para ser inicializado con datos más acordes a los que van a ser analizados para intentar mejorar de esta forma la dependencia que tiene el compresor con el tamaño de la señal que va a ser analizada, permitiendo la reducción de esta, siendo este uno de los mayores puntos flacos del compresor.

Para el segundo algoritmo se podría trabajar en una mejora de la adaptación de los datos lo que permitiría una mejor precisión en el cálculo de la complejidad.

Capítulo 7. Bibliografía

[1] Abraham Lempel, Jacob Ziv. “On the Complexity of Finite Sequences” IEEE Transactions on Information Theory, vol. it-22, no. 1, january 1976 pp 75-81

[2]Michael small, “Applied Nonlinear Time Series Analysis - Applications in Physics, Physiology and Finance”, World Scientific. Pp 54- 68

[3] M.Aboy, R.Hornero, D.Abasolo, D.Alvarez “Interpretation of the Lempel-Ziv Complexity Measure in the Context of Biomedical Signal Analysis” IEEE Transactions on Biomedical Engineering, vol. 53, no. 11, november 2006 pp 2282-2288

[4] Optimized RTL design and implementation of LZW algorithm for high bandwidth applications pp 279-285

[5] Zynq-7000 All Programmable SoC Overview, Xilinx, DS190 (v1.9) January 20, 2016

Capítulo 8. Anexos

Programas en C/C++

```
/*
 * File:   main.cpp
 * Author: Borja
 *
 * Created on 22 de junio de 2016, 21:22
 */

#include <cstdlib>
#include <iostream>
#include <sstream>
#include <fstream>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <string>
#include <bitset>
#include <unordered_map>
#include <utility>
#include <math.h>

#define        DICT_BITS        16
#define        DICT_MAX        (1 << DICT_BITS)

static void encode(FILE *in, FILE *out);
static void decode(FILE *in, FILE *out);

int main() {

    char ch, file_name1[25], file_name2[25];
    char codi;


    FILE *in;
    FILE *out;
    //Pido nombre de los archivos y si codifico o descodifico

    printf("Nombre del archivo:\n");
    gets(file_name1);
    in = fopen(file_name1, "rb");
    if (in == NULL) {
        fprintf(stderr, "Can't open %s.\n", file_name1);
        return 1;
    }
```

```
    printf("Nombre del segundo archivo:\n");
    gets(file_name2);
    out = fopen(file_name2, "wb");
    if (out == NULL) {
        fprintf(stderr, "Can't open %s.\n", file_name2);
        return 1;
    }

    printf("1-codificar 2-Decodificar:\n");
    scanf(" %c", &codi);

    if (codi == '1') {
        encode(in, out);

    } else {
        decode(in, out);
    }
    fclose(in);

    fclose(out);
}

static void encode(FILE *in, FILE *out) {

    //Inicializo diccionario con un map con los 256 ascii
    std::unordered_map<std::string, int> dictionary(DICT_MAX);
    dictionary.clear();
    for (int unsigned i = 0; i < 256; i++) {
        dictionary[std::string(1, i)] = i;
    }
    uint16_t dato = 0;
    long iter = 0;
    char datoc;
    std::string l;
    std::string ch;
    std::string str;
    int posdic = 256, lenght, bins = 1;
    std::string final;
    final = EOF;
    //Tamanyo del archivo
    fseek(in, 0, SEEK_END); // seek to end of file
    int length = ftell(in); // get current file pointer
    fseek(in, 0, SEEK_SET); // seek back to beginning of file

    //Bucle codificador
    for (;;) {

        ch = fgetc(in);
        str = str + ch;
        bins = (bins < str.size()) ? (str.size()) : bins;

        //Ultimo dato
        if (ch.compare(final) == 0) {
```

```
            if (dictionary.find(str) == dictionary.end()) {
                str.erase(str.size() - 1);
                dictionary.insert(std::make_pair(str, posdic));
                std::unordered_map<std::string,   int>::const_iterator
encode = dictionary.find(str);
                dato = (encode->second);

                fputc((dato >> 8), out);

                fputc(dato, out);
                //encode = dictionary.find (ch);
                //fputc((encode->second),out);
                break;
            }
            std::unordered_map<std::string,       int>::const_iterator
encode = dictionary.find(str);
            dato = (encode->second);

            fputc((dato >> 8), out);
            fputc(dato, out);
            break;
        }


        //Si el string no se encuentra en el dic lo meto en la
siguiente posicion
        if (dictionary.find(str) == dictionary.end()) {
            if (posdic < DICT_MAX) {
                dictionary.insert(std::make_pair(str, posdic));

                posdic++;
                str.erase(str.size()  -  1);  //borro  el  ultimo  char
busco la cadena resultante en el diccio y saco su codi.
                std::unordered_map<std::string,   int>::const_iterator
encode = dictionary.find(str);
                dato = (encode->second);




                fputc((dato >> 8), out);

                fputc(dato, out);
                str = ch;
            }
            else {
                str.erase(str.size() - 1);
                std::unordered_map<std::string,   int>::const_iterator
encode = dictionary.find(str);
                dato = (encode->second);
                fputc((dato >> 8), out);
                fputc(dato, out);
                dictionary.clear();
```

```
                posdic = 256;
                for (int unsigned i = 0; i < 256; i++) {
                    dictionary[std::string(1, i)] = i;
                }
                str = ch;
                //continue;
            }

        }

        iter++;

    }
    float cmp = (posdic * log(length)) / (length * log(bins));
    printf("%f \n", cmp);
    printf("%i \n", posdic);
}

static void decode(FILE *in, FILE *out) {

    int a, b, aux, flag = 0;
    char ch, ch2;
    std::string diccio[DICT_MAX];
    std::string str;
    std::string dato, datoc;
    int posdic = 256;
    char vector[256];


    //Tamanyo del archivo
    fseek(in, 0, SEEK_END); // seek to end of file
    int length = ftell(in); // get current file pointer
    fseek(in, 0, SEEK_SET); // seek back to beginning of file

    //Iniciamos diccionario
    for (int unsigned i = 0; i < 256; i++) {
        diccio[i] = i;
    }

    //Primera iteracion




    //Bucle decodificador.

    for (long i = 0; i < (length / 2) - 1; i++) {
        if (posdic == DICT_MAX) {
            //             for(int d = 256; d <4095; d++) {
            //            diccio[d].clear();
            //            }
            str.clear();
            posdic = 256;
            flag = 0;
```

```
    i++;

}
if (flag == 0) {
    a = fgetc(in);
    b = fgetc(in);

    a = (a << 8 | b);

    ch = a;

    if (a < posdic) {
        dato = diccio[a];
        int size = dato.size();
        for (int j = 0; j < size; j++) {
            fputc(dato[j], out);
        }
        str = str + ch;
        flag = 1;

    }
}
//Leo dos char y los uno en un int con el que opero
a = fgetc(in);
b = fgetc(in);
a = (a << 8 | b);

if (a < posdic) {

    datoc = diccio[a];
    str = str + datoc[0];



    diccio[posdic++] = str;
    int size = str.size();


    dato = diccio[a];

    size = dato.size();
    for (int j = 0; j < size; j++) {
        fputc(dato[j], out);
    }

    str = datoc;
    aux = a;
    //Si el dato leido no se encuentra aun en el diccionario
} else {
    datoc = diccio[aux] + datoc[0];
    int size = datoc.size();

    for (int j = 0; j < size; j++) {
        fputc(datoc[j], out);
```

```
            }
            str = datoc;
            if (posdic < DICT_MAX) {
                diccio[posdic++] = datoc;

            }
            aux = a;
        }

    }

}
```

Segundo programa en C/C++

```
#include <fstream>//Ifstream
#include <sstream>//sstream
#include <stdio.h>//printf
#include <stdint.h>
#include <string.h>//std::string
#include <stdlib.h>
#include <vector>//std::vector
#include <math.h>//log
#include <algorithm>//sort
#include <cmath>//floor
#include <time.h>

//conversion de string a int

template <typename T>
T StringToNumber ( const std::string &Text, T defValue = T() ){
    std::stringstream ss;
    for ( std::string::const_iterator i=Text.begin(); i!=Text.end();
++i )
        if ( isdigit(*i) || *i=='e' || *i=='-' || *i=='+' || *i=='.' )
            ss << *i;
    T result;
    return ss >> result ? result : defValue;
}

//comprobacion de substring
bool issubstring(std::vector<int> &aux, const int ns, const int nq){
     /* determine if Q is a substring of SQpi.*/

      int k,i,a,b;
      bool same;

      for (i=0; i<=(ns-nq); i++){
            same=1;

            for (k=0; k<nq; k++){
                  if (aux[i+k]!=aux[ns+k])
```

```
                        same=0;
            }
            if (same){

              return(1);
            }
      }

      return(0);

}

main(){

std::ifstream file ( "uint16.txt",std::ifstream::in );
std::vector<std::pair < int,int > > myvector;
std::string Text;
int T,a=0,c=100,bins=1,it;
float    res=0,o=0;
double i=0,tam=0,b=0;
clock_t t;

//Lectura de datos
while ( getline ( file, Text, '\n' ) ){




    o = StringToNumber(Text,0.0);
    myvector.push_back(std::pair<int,int> (o, i));
    i++;

}

    tam=i;
    b=i;
    std::sort (myvector.begin(), myvector.begin()+tam);

    std::vector<int> y(tam);
    std::vector<int> aux(tam);
//    for(int i=0;i<=4001;i++){
//
//        y[i]=floor(b--*16/(4001+1));
//
//    }

    for(int i=0;i<=tam;i++){
        y[i]=floor((b*c)/(tam+1));
        b--;
        aux[myvector[i].second]=y[i]+1;;
        bins= (bins<y[i])? (y[i]):bins;

}
```

```
   int ns=1,nq=1,k=2,cont=0;
  //Calculo de entropia

  while (k<tam){
        if (issubstring(aux,ns,nq)){
          /* Q is a substring of SQpi */
             nq++;
        }
        else{
          /* Q is a new symbol sequence */
          cont++;
          ns=ns+nq;
          nq=1;
        }
        k++;
            //printf("%i\n",nq);
    }

  res=(cont*log(tam))/(tam*log(bins));
  printf("%f\n",res);
 // printf("%i\n",cont);
  printf("Fin");
}
```