UNIVERSITAT DE VALÈNCIA

Facultad de Físicas
Departamento de Electrónica e Informática

COMPRESIÓN DE ECG EN TIEMPO REAL CON EL DSP TMS320C25

Grupo de Procesado Digital de Señales

Tesis de Licenciatura.
Valencia, Febrero 1997

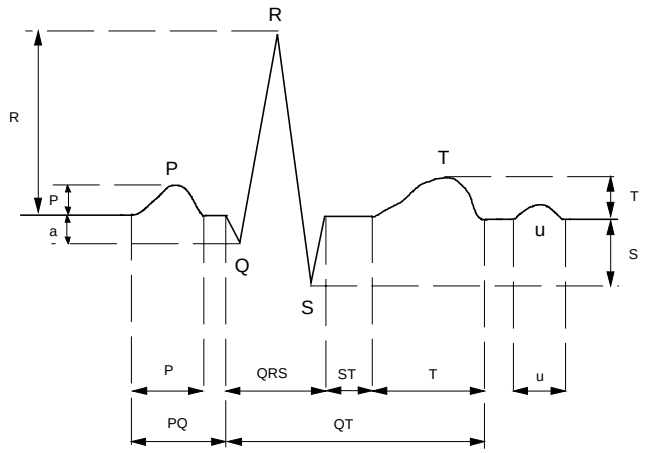
Autor: *Jorge Muñoz Marí*
Directores: *D. Juan F. Guerrero Martínez*
D. Javier Calpe Maravilla

INDICE

INTRODUCCIÓN

INTRODUCCIÓN.

La señal de electrocardiograma (ECG) es un registro de la actividad eléctrica del corazón. En la figura podemos ver su forma característica, así como las distintas partes que caracterizan la señal. A modo de ejemplo, se muestran también los valores típicos que suelen tomar estos registros, aunque hay variaciones significativas en el ECG según de que persona sea, o incluso en la misma persona según su estado.



	Amplitud
Onda P:	0.25 mV
Onda R:	1.60 mV
Onda Q:	25 % R
Onda T:	0.1 - 0.5 mV

	Duración
P-R:	0.12 - 0.20 s
Q-T:	0.35 - 0.44 s
S-T:	0.05 - 0.15 s
P:	0.11 s
QRS:	0.09 s

La señal de ECG aporta mucha información al cardiólogo acerca del estado de salud de una persona, y es una ayuda inestimable a la hora de detectar y corregir muchas de las patologías cardiacas actualmente conocidas.

Para recoger la señal de ECG se suelen emplear varios métodos, dependiendo del tipo de información que se esté buscando. Típicamente se suele muestrear a frecuencias comprendidas entre los 250Hz y los 1000Hz, llegando incluso a los 2000Hz en algunos estudios de investigación (ECG de alta frecuencia).

Un ejemplo es el estudio de ECG de 12 canales. En este tipo de estudio se toman 8 canales (los otros 4 son linealmente dependientes) a una frecuencia de 500Hz por canal. Los registros suelen ser de 5 minutos de duración, lo cual nos da un total de 1.200.000 muestras. Suponiendo que se utiliza un conversor de una resolución de 12 bits (4096 niveles) tenemos un total de 1.72 Mb por cada uno de estos registros.

Otro ejemplo típico son los registros en cintas magnéticas (registros Holter), que se toman durante 24 horas en dos canales con una frecuencia de muestreo efectiva de 250Hz por canal. Esto supone un total de 43.200.000 muestras. Suponiendo de nuevo un conversor de 12 bits esto resulta en un total de 61.8 Mb necesarios para poder almacenar dichos registros.

Debido a la gran cantidad de muestras generadas en estos registros se plantea la necesidad de comprimir la información, de cara a sacar más partido y abaratar los medios de almacenamiento y transmisión. La compresión realizada sobre estos registros ha de cumplir dos requisitos importantes:

- Ser en tiempo real. En muchas ocasiones los datos han de transmitirse inmediatamente para su análisis, bien debido a la urgencia del caso en cuestión, o bien a que no disponemos del suficiente espacio de almacenamiento en el aparato que recoge las muestras.

- Alcanzar una buena razón de compresión. Evidentemente, y tratando con la gran cantidad de muestras generadas en el caso que nos ocupa, es necesario reducir considerablemente el total de muestras almacenadas y/o transmitidas.

De estas dos propiedades, la más exigente es la que concierne a realizar la implementación de los algoritmos en tiempo real. Para poder llevar a cabo dicha implementación disponemos de una serie de plataformas más o menos adecuadas, como pueden ser micro procesadores, micro computadores, FPGA's, PLD's, DSP's, etc., y de entre las cuales nosotros vamos a utilizar en este estudio los DSP.

Los DSP presentan una serie de ventajas que los hacen muy buenos candidatos para este tipo de tareas. En primer lugar, son procesadores orientados específicamente al tratamiento digital de señales como su propio nombre indica, y el juego de instrucciones que poseen permiten realizar las operaciones típicas de procesado digital en pocos ciclos de reloj. En segundo, su programación por software nos aporta la flexibilidad necesaria para la implementación de varios algoritmos sobre un mismo procesador, todo ello además con la rapidez requerida para el caso planteado.

Los DSP han sufrido además una importante evolución en los últimos años, fruto de la mejoras tecnológicas en la implementación, que permiten realizar operaciones más complejas en tiempo real, y en coma flotante. Esto ha repercutido en un mayor uso de estos, así como del consiguiente abaratamiento de los mismos, haciendo de ellos una buena opción.

Este trabajo, cuyo objetivo fundamental es el estudio, implementación y evaluación de algoritmos de compresión de ECG en tiempo real, se ha organizado de la siguiente manera:

- Introducción teórica, donde se dan las bases teóricas de los algoritmos que se van a evaluar.
- Descripción de las herramientas hardware utilizadas.
- Implementación, donde se pasa a describir los pormenores de la implementación de los algoritmos sobre el DSP utilizado, el TMS320C25 de Texas Instruments, y en lenguaje C.
- Resultados, donde se presentan los resultados obtenidos para cada algoritmo.
- Conclusiones, aportaciones y proyección futura.
- Listados de los programas realizados.
- Referencias bibliográficas utilizadas.

CAPÍTULO 1

ALGORITMOS DE COMPRESIÓN PARA ECG

CAPÍTULO 1: ALGORITMOS DE COMPRESIÓN PARA ECG

En este capítulo se van a mostrar una serie de técnicas utilizadas habitualmente para la compresión de señales de ECG. Algunas de estas técnicas, como ya veremos, son también utilizadas para realizar la compresión de otros tipos de señales o datos.

En lo que a la compresión de señales médicas se refiere, y en concreto a la señal de ECG, lo que se busca mediante la compresión de datos es:

- Incrementar la capacidad de almacenamiento de las bases de datos de señales de ECG. Estas bases son utilizadas habitualmente para el estudio y clasificación de señales de ECG, y han de contener una gran cantidad de registros.

- Acelerar y abaratar la transmisión de datos que han sido obtenidos o lo están siendo en tiempo real a través de un canal de comunicación, que puede acabar siendo la línea telefónica normal, en cuyo caso la compresión se torna una herramienta casi imprescindible.

- Aumentar la funcionalidad de los monitores y sistemas de almacenamiento de los centros hospitalarios y ambulatorios.

El objetivo principal de la compresión de datos va a ser para nosotros obtener la mayor reducción de datos preservando las características útiles de la señal. Bajo este punto de vista, nos será posible en determinados casos eliminar muestras de la señal de ECG siempre que estas no aporten información de utilidad al cardiólogo.

Vamos a considerar los algoritmos de compresión en dos grupos. Algoritmos de Compresión Sin Pérdidas, los cuales almacenan todas las muestras que constituyen la señal y la reconstruyen exactamente tal como era, y Algoritmos de Compresión con Pérdidas, los cuales alcanzan una mayor compresión descartando partes de la señal que no aportan información de interés al cardiólogo.

Pasaremos a estudiar ahora el primero de estos grupos.

1.1. ALGORITMOS DE COMPRESIÓN SIN PÉRDIDAS (ACSP).

Si bien este tipo de algoritmos de compresión no son normalmente utilizados para realizar la compresión en tiempo real de ECG, debido a su mayor complejidad y gasto de computo que los Algoritmos de Compresión Con Pérdidas (ACCP), existen algunos de estos algoritmos cuyo estudio resulta de interés, pues son una referencia en ocasiones imprescindible en el tema de compresión de datos.

Este tipo de algoritmos son utilizados exhaustivamente para la compresión de datos en ordenadores (programas, imágenes, sonido, bases de datos, documentos, etc.). Los algoritmos que vamos a ver aquí son algoritmos de codificación, como la codificación de Huffman y la codificación de Shannon-Fano, y los algoritmos de sustitución, de los cuales se utilizan dos esquemas principalmente, el LZ77 y LZ78, propuestos en 1977 y 1978 por Jakob Ziv y Abraham Lempel. Ambos tipos de algoritmos son la base de los programas de compresión más conocidos como ARJ, LHA, RAR, ZIP, GZIP, y formatos gráficos como el GIF.

1.1.1 CODIFICACIÓN DE HUFFMAN.

La codificación de Huffman se basa en que los valores que toma una señal cuantizada no son equiprobables. La longitud (número de bits) del código asignado a cada valor depende, en este tipo de codificación, de la frecuencia con que este se de. Así, los valores que se dan con más frecuencia tienen un código de longitud más corto.

Supongamos que tenemos el siguiente conjunto de puntos:

{1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 4, 5, 5, 6, 6, 6, 7, 7, 7}

Para obtener el código de Huffman procedemos de la siguiente manera. En primer lugar calculamos la probabilidad de ocurrencia de cada uno de los elementos del conjunto. Una vez calculada, pasamos a la realización de un árbol binario que nos dará la codificación final. La creación de este árbol se realiza de la siguiente manera:

1. Se crea una lista ordenada de los elementos respecto a su probabilidad.
1. Procedemos a juntar los dos elementos de menor probabilidad, creando un nuevo elemento cuya probabilidad es la suma de ambos.
1. Reordenamos la lista con el nuevo conjunto de elementos.
1. Repetimos los pasos 2 y 3 hasta alcanzar un único nodo, llamado nodo raíz.

Este es el proceso de creación del árbol que permite asignar el código binario a cada elemento del conjunto. El proceso completo podemos observarlo en la figura 1.1. Se concluye en la realización de una tabla que asigna a cada elemento su codificación (figura 1.2.).

S_i	Lista de P_i						
1	.30	.30	.30	.30	.43	.57	1
3	.19	.19	.24	.27	.30	.43	
2	.15	.15	.19	.24	.27		
6	.12	.12	.15	.19			
7	.12	.12	.12				
5	.08	.12					
4	.04						

Fig. 1.1. - Construcción del árbol binario.

Símbolos, S_i	Código binario de 3 b	Probabilidad de Ocurrencia, P_i	Código Huffman
1	001	0.30	11
2	010	0.15	101
3	011	0.19	00
4	100	0.04	1000
5	101	0.08	1001
6	110	0.12	011
7	111	0.12	010

Fig. 1.2. - Tabla de Codificación de Huffman.

La razón de reducción de la codificación de Huffman depende de la distribución de los elementos fuente. En nuestro ejemplo teníamos un conjunto de 7 elementos, con lo que necesitamos tres bits para realizar su codificación. Utilizando la codificación de Huffman obtenemos una longitud media por elemento que podemos calcular como:

$$E[l] = \sum_{i=1}^{7} l_i P_i \tag{1.1}$$

donde l_i es la longitud del código Huffman para cada elemento, y P_i es la probabilidad de aparición de cada elemento. En nuestro ejemplo esta longitud media toma un valor de 2.63, con lo que la razón de compresión es de 3:2.63.

El proceso de reconstrucción del código se realiza simplemente recorriendo el árbol binario hasta llegar a un nodo terminal. Esto es así debido a que el código de Huffman tiene la propiedad de ser instantáneo, esto es, el código de los elementos con longitud de código menor nunca es precedente del código asociado a elementos con longitud de código mayor, o dicho de otra manera, todos los elementos del árbol binario que tienen asociado un código son nodos terminales. Es necesario transmitir o almacenar el árbol binario para poder descodificar los datos.

La codificación de Huffman no introduce pérdidas, sin embargo si durante la transmisión o almacenamiento se produce un error que afecte a un sólo bit, en la reconstrucción esto se traduce en más de un error. Es usual que las etapas posteriores a la codificación de Huffman utilicen algún tipo de protección para los datos, como pueden ser códigos de redundancia cíclica, que hacen crecer la cantidad de información a transmitir o almacenar. Es necesario pues encontrar

un punto medio entre estos sistemas de protección y la codificación de Huffman para asegurarnos de que efectivamente existe compresión.

La codificación de Huffman es óptima cuando se da el caso de que la probabilidad asociada a cada símbolo es potencia de ½. En el resto de ocasiones, la Codificación de Huffman será una buena aproximación a una codificación óptima.

Codificación de Shannon-Fano.

Relacionada directamente con la codificación de Huffman existe otra técnica de codificación, llamada *Shannon-Fano Coding,* que funciona de la siguiente manera:

1. Se divide el conjunto de símbolos en dos subconjuntos iguales o casi iguales basándonos en la probabilidad de ocurrencia de los caracteres en cada subconjunto. Al primer subconjunto se le asigna un cero, y al segundo un uno.
1. Se repite el paso 1 hasta que todos los subconjuntos constan de un único elemento.

Este algoritmo es peor que la codificación de Huffman, en el sentido de que en algunas ocasiones los códigos generados tienen una longitud mayor. Sin embargo es un algoritmo más sencillo de implementar con menor coste de cómputo.

La descompresión de los datos se realiza como en el caso de la codificación de Huffman recorriendo el árbol binario que se genera al crear el código.

1.1.2. ALGORITMOS DE SUSTITUCIÓN.

La familia LZ78 de algoritmos de compresión.

En general, los algoritmos basados en el esquema LZ78 funcionan introduciendo frases en un diccionario de manera que cuando se encuentra una ocurrencia repetida de una frase se saca un índice al diccionario en lugar de la frase en cuestión. Si bien existen gran variedad de algoritmos basados en este esquema, la diferencia fundamental entre todos ellos es la manera en que cada uno crea y gestiona el diccionario. El esquema más conocido de esta familia es el LZW, propuesto en 1984 por Terry Welch y pensado para la implementación hardware de controladores de discos de alto rendimiento.

El algoritmo LZW comienza con un diccionario de 4K, cuyas primeras 256 entradas (0-255) se refieren a cada byte, y las siguientes (256-4095) se refieren a cadenas de caracteres. Estas cadenas de caracteres se generan dinámicamente conforme se van leyendo los datos, de forma que una nueva cadena es generada añadiendo el carácter actual (K) a la cadena existente (w). El algoritmo LZW es el siguiente:

```
w = NIL
bucle
       leer un caracter K
       si wK existe en el diccionario
              w = wK
       sino
              sacar el código para w
              añadir wK a la tabla de cadenas
              w = K
finbucle
```

En la figura 1.3 tenemos un ejemplo del funcionamiento de este algoritmo, donde partiendo de una cadena de 15 caracteres a la entrada obtenemos 9 caracteres a la salida, 4 de ellos son parte del diccionario creado dinámicamente:

Cadena de entrada:/WED/WE/WEE/WEB			
Carácter de entrada: (K)	Carácter de salida:	Diccionario	Valor de w
/			/
W	/	256 =/W	W
E	W	257 = WE	E
D	E	258 = ED	D
/	D	259 = D/	/
W			/W
E	256	260 = /WE	E
/	E	261 = E/	/
W			/W
E			/WE
E	260	262 = /WEE	E
/			E/
W	261	263 = E/W	W
E			WE
D	257	264 = WED	D
<END>	<END>		

Fig. 1.3. - Ejemplo de compresión LZW.

La descompresión se realiza igual que la compresión, únicamente hace falta sustituir cada código de entrada en su cadena y sacarlo fuera. En la figura 1.4 tenemos el ejemplo de su funcionamiento:

Cadena de entrada: /WED<256>E<260><261><257><END>		
Carácter de entrada:	Carácter de salida:	Diccionario
/	/	
W	W	256 = /W
E	E	257 = WE
D	D	258 = ED
256	/W	259 = D/
E	E	260 = /WE
260	/WE	261 = E/
261	E/	262 = /WEE
257	WE	263 = E/W
<END>	<END>	264 = WEB

Fig. 1.4. - Ejemplo de descompresión LZW

Una de las características más relevantes de este tipo de compresión es que el diccionario de codificación se transmite implícitamente con el código, por lo que no será necesario reservar un espacio extra para el mismo.

Este algoritmo plantea problemas cuando el espacio para el diccionario se llena completamente, lo cual sucede con frecuencia en ficheros largos, puesto que el número de palabras en el diccionario crece muy rápidamente. Algunos algoritmos optan por comenzar con un nuevo diccionario desde cero, olvidándose del anterior, como es el caso del algoritmo utilizado para comprimir imágenes en formato GIF. Otros más elaborados, y en algunos casos más eficientes, utilizan algoritmos del tipo *menos recientemente usado* (least-recently-used) para descartar frases poco o nada utilizadas.

También ha de considerarse el hecho de que este algoritmo conlleva, en ocasiones, la aplicación de una codificación de Huffman (u otra que reduzca el número de bits de los caracteres de salida), pues el número de bits utilizados a la salida para la codificación de los símbolos es, necesariamente, mayor que a la entrada, si bien esto no se aplica en todos los casos como por ejemplo el algoritmo LZW utilizado en la compresión de imágenes en formato GIF.

Familia LZ77 de algoritmos de compresión

Los algoritmos basados en el esquema LZ77 funcionan llevando la cuenta de n bytes anteriores, de manera que si aparece una frase que ya apareció antes, se sacan un par de valores que indican la posición de la frase anterior y la longitud de la misma. Los algoritmos más usados son variaciones del esquema LZSS propuesto por James Storer y Thomas Szymanski en 1982. Como variantes aparecen principalmente el LZB, que aplica un codificación simple, el LZH, que aplica codificación de Huffman dinámica, y ZIP 1.x, que aplica codificación de Shannon-Fano.

La mayoría de compresores conocidos (ARJ, LHA, ZIP, ZOO, RAR) utilizan variantes del esquema LZ77.

1.2. ALGORITMOS DE COMPRESIÓN CON PÉRDIDAS (ACCP).

Podemos distinguir tres maneras de proceder para este tipo de algoritmos:

- compresión directa
- compresión mediante transformaciones ortogonales
- compresión por extracción de parámetros

En la compresión por extracción de parámetros nuestro sistema de compresión extrae de la señal de ECG algunos parámetros característicos de la misma como por ejemplo amplitudes y tiempos de duración de los segmentos estándar (ver figura 1 en introducción), y son estos los que son almacenados y / o transmitidos en lugar de la señal. La señal original por tanto se pierde. Por esta razón este tipo de compresión de ECG no goza de excesiva aceptación entre los cardiólogos, puesto que desde el punto de vista médico es preferible observar y diagnosticar sobre la señal original.

En la compresión mediante transformaciones ortogonales lo que se hace es aplicar una transformación sobre la señal de ECG para después quedarnos solamente con los términos que describen las partes de interés de la señal. Existen varias transformaciones que es posible aplicar, todas ellas casos particulares de la Transformada de Karhunen-Loeve (KLT). Entre estas podemos citar la Transformada de Fourier (FT), del Coseno (CT), de Walsh (WT), de Haar (HT). En este trabajo se estudian e implementan las transformadas de Fourier y del Coseno pues en ambas es posible implementar algoritmos que permiten su ejecución en tiempo real.

Por último en los algoritmos de compresión directa que pasaremos a ver a continuación se realiza la compresión tratando de eliminar muestras de la señal que entra al sistema compresor realizando un descarte selectivo mediante el uso de polinomios predictores o interpoladores. Dentro de este grupo también se contemplará el uso de DPCM (Diferential Pulse Code Modulation) en compresión de ECG.

1.2.1. TÉCNICAS DE COMPRESIÓN DIRECTA.

Antes de pasar a la descripción de los algoritmos propiamente dichos que se utilizan para este tipo de compresión se describirán a continuación lo que son los polinomios predictores e interpoladores, pues constituyen la base de esta familia de algoritmos.

1.2.1.1. POLINOMIOS PREDICTORES.

El polinomio predictor realiza una estimación de la muestra actual ($\hat{y}_n$) a partir de M muestras anteriores. Se basa en una técnica de diferencias finitas mediante la cual se obtiene un polinomio de enésimo orden con K+1 puntos. Matemáticamente esto se puede expresar como:

$$\hat{y}_n = y_{n-1} + \Delta y_{n-1} + \Delta^2 y_{n-1} + \cdots + \Delta^k y_{n-1} \quad (1.2)$$

donde:

$\hat{y}_n$ = muestra predicha en t_n
y_{n-1} = muestra tomada en t_{n-1}
$\Delta y_{n-1} = y_{n-1} - y_{n-2}$
$\Delta^k y_{n-1} = \Delta^{k-1} y_{n-1} - \Delta^{k-1} y_{n-2}$

- *POLINOMIO PREDICTOR DE ORDEN 0 (ZOP)*

Para k=0 tenemos $\hat{y}_n = y_{n-1}$. En este caso la predicción es sencillamente la muestra anterior. Los algoritmos basados en este tipo de predictor definen para cada muestra tomada una margen de tolerancia centrada alrededor de dicha muestra. Si la siguiente muestra entrante está dentro del margen de tolerancia se considera redundante y por tanto no se almacena, caso contrario se procede a almacenar el grupo de muestras que si entraban dentro del margen de tolerancia como el valor de la primera y la longitud (número de muestras del conjunto).

- *POLINOMIO PREDICTOR DE ORDEN 1 (FOP)*

Para el caso k=1 el polinomio queda $\hat{y}_n = 2y_{n-1} - y_{n-2}$. Es decir, la muestra predicha está en la recta descrita por la dos muestras anteriores. Análogamente a como ocurre con el ZOP los algoritmos basados en este predictor definen un margen de tolerancia para la muestra actual en función de las dos anteriores, considerándose redundante si cae dentro de este margen de tolerancia, o no en caso contrario.

1.2.1.2. POLINOMIOS INTERPOLADORES.

Los polinomios interpoladores utilizan muestras pasadas y futuras para determinar si la muestra actual es o no redundante.

- *POLINOMIO INTERPOLADOR DE ORDEN 0 (ZOI)*

Su funcionamiento en compresión de datos es análogo al ZOP. La diferencia es que en el ZOI el valor de la muestra a almacenar se determina cuando ya se dispone de un conjunto de muestras redundantes, en contraste con el ZOP donde se guardaba directamente la primera muestra del conjunto. Para el caso del ZOI se almacena el promedio del conjunto de muestras. En los algoritmos de compresión basados en polinomios normalmente se utiliza el ZOI y no el ZOP.

- *POLINOMIO INTERPOLADOR DE ORDEN 1 (FOI)*

Como en el caso del FOP se trata de determinar la redundancia de las muestras utilizando para ello una línea recta entre dos muestras y definiendo un margen de tolerancia. Habitualmente se utilizan el *Polinomio Interpolador de Primer Orden con dos grados de libertad* (FOI-2DF), por ser el que mejores resultados obtiene.

El FOI-2DF funciona trazando una recta entre la muestra actual y última muestra almacenada. Como siempre se define un margen de tolerancia, de manera que si el resultado de interpolar las muestras que quedan entre las dos muestras, mediante las cuales se trazó la recta, está dentro de dicho margen, la muestra actual se considera redundante y se pasa a la siguiente muestra para trazar una nueva recta; en caso contrario la muestra actual se almacena y se considera como punto inicial para el siguiente grupo de muestras.

1.2.1.3. ALGORITMO TURNING POINT.

Si bien este algoritmo no está basado en el uso de polinomios, si entra directamente en la familia de compresores por compresión directa. Su finalidad original es reducir la frecuencia de muestreo de los ECG tomados a 200Hz a la mitad, basándose en realizar una selección adecuada de las muestras que componen la señal de manera que se conserven los *puntos de giro*, o turning-points, que son aquellos donde la señal muestra un inflexión.

El algoritmo procesa tres puntos a la vez, X0, X1 y X2. Se guarda y se toma como referencia el punto X0 para almacenar uno de los otros dos que quedan. Para ello se utiliza la operación de signo definida como:

$$sign(x) = \begin{cases} 0 & x = 0 \\ +1 & x > 0 \\ -1 & x < 0 \end{cases} \qquad (1.3)$$

Los operandos s1=sign (X2-X1) y s2 = sign (X1-X0) indican si las rectas definidas por los pares (X2, X1) y (X1, X0) son ascendentes o descendentes. Habrá un *punto de giro* si hay un cambio de signo, y no en caso contrario. Para escoger el punto a almacenar, X1 ó X2, se utiliza el siguiente criterio: si la condición

$$\{ \text{NOT}(s1) \text{ OR } (s1 + s2) \} \qquad (1.4)$$

es falsa se almacena X1, y si es cierta X2. Esta condición es fácil de implementar pues está basada en operaciones lógicas sencillas NOT y OR.

En la figura 1.5 tenemos un representación de todas las posibles configuraciones (a) y que punto se escoge en cada una de ellas (b). En la figura 1.6 disponemos del código en C que implementa este algoritmo.

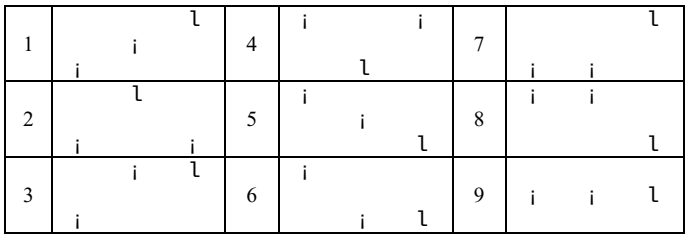


Fig. 1.5a. - Posibles configuraciones de entrada en el Turning Point.

Entrada	s1=sign (X1-X0)	s2=sign (X2-X1)	NOT(s1) OR (s1+s2)	Muestra guardada
1	+1	+1	1	X2
2	+1	-1	0	X1
3	+1	0	1	X2
4	-1	+1	0	X1
5	-1	-1	1	X2
6	-1	0	1	X2
7	0	+1	1	X2
8	0	-1	1	X2
9	0	0	1	X2

Fig 1.5b. - Criterio matemático utilizado para determinar el punto a almacenar.

El algoritmo TP es rápido y simple, y produce una razón de compresión fija de 2:1. La reconstrucción se realiza interpolando entre los pares de datos guardados. La desventaja de este algoritmo es que introduce una pequeña distorsión en el tiempo, pero sin embargo esta distorsión no es visible en monitores y plotters.

```
#define sign(x) ( (x) ? (x>0) ? 1 : -1 ) : 0 )
short *org, *tp;          /* muestra original y muestra tp */
short x0, x1, x2;         /* datos */
short s1, s2;       /* signos */

x0 = *tp++ = *org++;      /* guarda la primera muestra */
while (haya_muestras) {
       x1 = *org++;
       x2 = *org++;
       s1 = sign (x1 - x0);
       s2 = sign (x2 - x0);
       *tp++ = x0 = (!s1 || (s1 + s2) ) ? x2 : x1;
}
```

Figura 1.6. - Implementación en C del algoritmo Turning-Point

1.2.1.4. ALGORITMO AZTEC

El algoritmo Aztec (Amplitude Zone Time Epoch Coding) data de 1.968 y sus bases fueron publicadas en IEEE por J.R. Cox, F.M. Nolle. H.A. Fozzard y G.C. Oliver [29]. Inicialmente propuesto como algoritmo de preprocesado del ECG para medida del ritmo cardiaco, AZTEC utiliza por una parte el polinomio interpolador de orden 0 (ZOI) para comprimir las regiones isoeléctricas del ECG las cuales quedan aproximadas por rectas horizontales, o *plateaus* como las

denominan sus autores, y realiza una aproximación por rectas (*rampas*) de las zonas de alta frecuencia del ECG, principalmente del complejo QRS.

El AZTEC alcanza compresiones de 10:1 con ECG muestreados a 500Hz con 12b de resolución. Sin embargo la señal reconstruida presenta notables discontinuidades y distorsión, principalmente en las ondas P y T debido a su lenta variación. Es habitual tratar la señal proveniente de AZTEC con un filtro suavizador de Savitzky-Golay[30]. Estos filtros son utilizados sobre señales de ECG por presentar un buen índice de reducción del ruido en la señal (NRR) en comparación con filtros FIR promediadores, sin producir distorsión de fase. Además, y a diferencia de otro tipo de filtrado, estos preservan la anchura de los picos de las señales a las que se aplican.

El AZTEC tiene dos modos de funcionamiento, el modo *plateau* que se basa en el ZOI para generar líneas horizontales, y el modo *rampa*. Al comenzar a funcionar se encuentra en modo *plateau*, de manera que, según el esquema del ZOI, se toma una apertura ε sobre la primera muestra y sólo cuando la muestra actual se sale fuera del margen establecido por dicha apertura se almacena el valor promedio de las muestras anteriores y su longitud, comenzado un nuevo grupo de muestras. El algoritmo permanece en este modo hasta que el conjunto de muestras a almacenar es menor que 3, en cuyo caso pasa a modo *rampa*, permaneciendo en este hasta que es posible volver a generar rectas horizontales (*plateaus*) de 3 o más muestras. Los valores almacenados para las rampas son la duración y el punto final de la misma. La descompresión de los datos generados por el algoritmo AZTEC se realiza expandiendo en una secuencia discreta de puntos las plateaus y rampas.

Las figuras 1.7 y 1.8 contienen el diagrama de flujo del funcionamiento del algoritmo AZTEC. En la figura 1.7 tenemos el funcionamiento en modo *plateau*. Las variables V_{mx} y V_{mn} dan la diferencia entre el valor máximo y mínimo de las muestras tomadas. Si la diferencia (V_{mxi} - V_{mni}) es mayor que un determinado valor de tolerancia V_{th} fijado a priori se sale del bucle de toma de muestras (también llamado de *detección de línea*) para pasar a almacenar el conjunto tomado, o a modo *rampa (slope)*, en función de si el número de muestras, indicado por *LineLen*, es mayor o menor que 3 respectivamente. Como puede observarse, también se sale del bloque de detección de línea si se toma un conjunto de 50 muestras. Esto se hace como medida para evitar la generación de plateaus excesivamente largas que pueden distorsionar en exceso la señal de ECG original.

La figura 1.8 muestra la parte del algoritmo (*procesado de línea*) que decide cuando hay que almacenar el conjunto de muestras tomadas o bien pasar a modo *slope*. La variable que indica en que modo estamos es *LineMode*. Las variables V_{si} y T_{si} son las encargadas de almacenar los valores del último punto de la rampa y su duración respectivamente cuando nos encontramos en modo *slope*. T_{si} es siempre negativo, indicando de esta forma que se trata de una rampa y no de una línea horizontal.

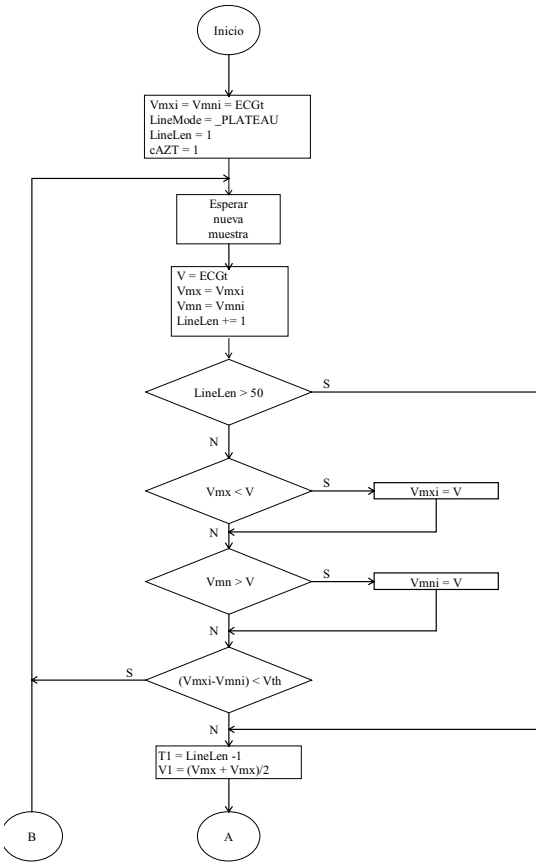


Fig. 1.7. - Diagrama de flujo de la operación de detección de línea del algoritmo AZTEC.

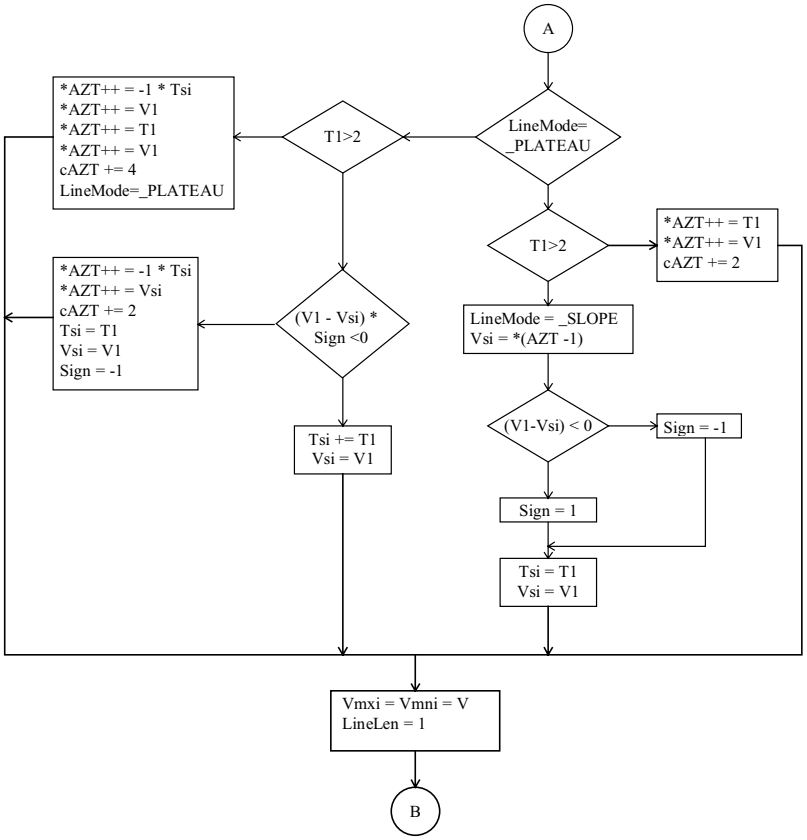


Fig. 1.8. - Diagrama de flujo de la operación de procesado de línea del algoritmo AZTEC.

Las variables que nos permiten ajustar el funcionamiento del algoritmo son la tolerancia V_{th} y el valor máximo permitido para *LineLen*. Ambas han de ajustarse teniendo en cuenta parámetros como son la frecuencia de muestreo del ECG, la resolución del conversor analógico digital utilizado, etc. En función de estas variables, aunque principalmente de la tolerancia V_{th}, el algoritmo AZTEC alcanzará cuotas de compresión más o menos altas perdiendo más o menos calidad en la señal reconstruida. Así, para valores altos de V_{th} tendremos más compresión con más perdidas, mientras que para valores de V_{th} bajos la compresión será menor pero se preservará mejor la señal. La elección del valor óptimo para V_{th} dependerá en última instancia de la aplicación en curso.

1.2.1.5. CORTES.

El algoritmo CORTES (Coordinate Reduction Time Encoding System) utiliza los algoritmos T.P. (*Turning Point*) y AZTEC para realizar la compresión de la señal de ECG. Estos dos algoritmos funcionan en paralelo, y CORTES determina cuando se guardan los datos comprimidos por uno u otro. Generalmente, para el complejo QRS y otros complejos de más alta frecuencia se utiliza el algoritmo T.P., mientras que el resto (región isoeléctrica) se comprime utilizando el AZTEC. Por tanto, el algoritmo AZTEC solamente genera líneas horizontales, pues en las rampas se utiliza el T.P.

La reconstrucción se realiza expandiendo las líneas AZTEC e interpolando los puntos T.P. Tras esto, se aplica una filtro suavizador parabólico a las regiones AZTEC.

1.2.1.6. FAN

El algoritmo FAN [1][11] (Bohs and Barr, 1988), está basado en el polinomio interpolador de primer orden con dos grados de libertad (FOI-2FD), y es uno de los que mejores resultados ofrece en compresión de ECG. Su funcionamiento es el siguiente:

Se empieza aceptando y almacenando la primera muestra X0. Tras recoger la segunda muestra, X1, se generan dos rectas, U1 y L1. La primera recta pasa por los puntos X0 y X1+ ε, donde ε representa la tolerancia del algoritmo, y fija la relación cantidad/calidad de compresión obtenida. La segunda recta pasa por X0 y X1 - ε. Si la tercera muestra X2 cae en el área cubierta por estas dos rectas, se pasa a generar dos nuevas rectas U2 y L2 que pasarán por los puntos X0 y X2 + ε, y X0 y X2 - ε, respectivamente. Tras esto, de las cuatro rectas obtenidas {U1, L1, U2, L2} se escogerán las dos que dan lugar al área más restrictiva. A partir de aquí se toma el valor de X2 como X1 y se recoge una nueva muestra, repitiéndose el algoritmo hasta que la nueva muestra tomada se sale fuera del área cubierta por dichas rectas. Cuando esto ocurre se procede a guardar la longitud de la línea (T) y su amplitud final, X1. La muestra X2 que queda fuera

del área se toma como X0 y el algoritmo comienza desde el principio. En la figura 1.9. tenemos una ilustración del funcionamiento del FAN. De una forma gráfica vemos, en la figura 1.9(a), como se obtienen las rectas U y L a partir de las muestras tomadas en t_1 y t_2, y que muestras son eliminadas y cuales almacenadas. La figura 1.9(b) muestra la extrapolación de X_{U2} y X_{L2} a partir de X_{U1}, X_{L1} y X_0.

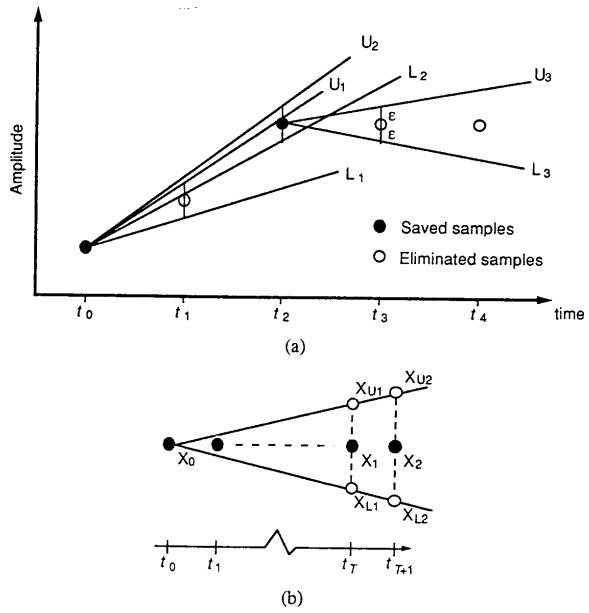


Fig. 1.9. - Funcionamiento del algoritmo FAN.

El diagrama de flujo del algoritmo FAN se muestra en la fig. 1.10. La implementación de este mismo algoritmo en C se muestra en la fig. 1.11. Las variables XU1, XU2, XL1 y XL2 nos servirán para determinar si la muestra X2 se encuentra o no en el área descrita por las rectas U1 y L1.

El algoritmo FAN asegura que el error al juntar las líneas entre los puntos almacenados al reconstruir la señal es menor o igual que el valor escogido para ε. Comparado con los algoritmos T.P. y AZTEC, el algoritmo FAN produce una reconstrucción más fiel de la señal de ECG para las mismas o mayores razones de compresión.

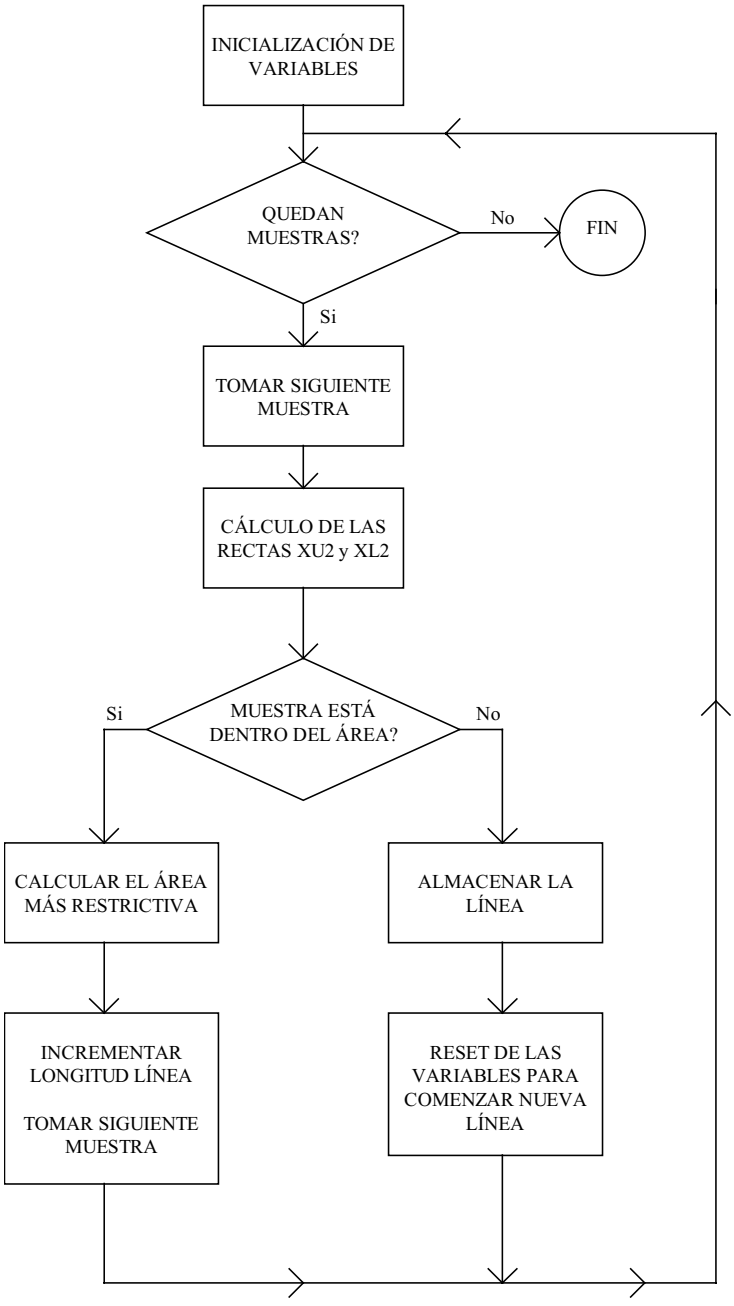


Fig. 1.10. - Diagrama de flujo del algoritmo FAN.

```
short X0, X1, X2;                /* muestras */
short XU2, XL2, XU1, XL1;        /* variables para determinar las áreas */
short Epsilon;
short *org, *fan;                /* datos original y fan */
short T;                         /* longitud de la línea */
short V;                         /* muestra */

/* Inicialización de variables */
X0 = *org++;              /* punto original */
X1 = *org++;              /* siguiente punto */
T = 1;
XU1 = X1 + Epsilon;
XL1 = X1 - Epsilon;
*fan++ = X0;              /* se guarda la primera muestra */

while (haya_datos) {
      V2 = *org++;        /* se obtiene la siguiente muestra */
      XU2 = (XU1 - X0)/T + XU1;
      XL2 = (XL2 - X0)/T + XL2;

      if (X2 <= XU2 && X2 >= XL2) {          /* cae dentro */
            /* se obtiene el área más restrictiva */
            XU2 = (XU2 < X2 + Epsilon) ? XU2 : X2 + Epsilon;
            XL2 = (XL2 > X2 - Epsilon) ? XL2 : X2 - Epsilon;
            T++                /* se incrementa la longitud de la línea */
            X1 = X2;
      }
      else {                   /* cae fuera, se guarda la línea */
            *fan++ = T;        /* se guarda la longitud de la línea */
            *fan++ = X1;       /* y la amplitud final */

            /* reinicialización de las variables */
            X0 = X1;
            X1 = X2;
            T = 1;
            XU1 = X1 + Epsilon;
            XL1= X1 - Epsilon;
      }
}
```

Fig. 1.11. - Implementación en C del algoritmo FAN.

1.2.1.7. ALGORITMOS SAPA

Los algoritmos SAPA (Scan - Along Approximation) son tres algoritmos presentados por M. Ishijima, S.B. Shin, G.H. Hostetter y J. Sklansky en 1983 [30]. De los tres, el que mejores resultados ofrece es el SAPA-2, cuyo funcionamiento es prácticamente el mismo que el del algoritmo FAN. Difiere de este en que se cambia la manera de decidir si la muestra actual (y_n) es redundante o no. En SAPA-2, además de las dos rectas calculadas en FAN, se calcula una tercera recta, llamada recta central, que pasa por la muestra inicial y la muestra actual. Si esta tercera recta está dentro del área encerrada por las otras dos, se considera que la muestra es redundante y se pasa a analizar la siguiente. Caso contrario se almacena el valor de la muestra y se inicia una nueva línea.

1.2.1.8. COMPRESIÓN POR DPCM.

Esta técnica de compresión, basada en la *modulación delta*, trata de cuantificar y almacenar o transmitir la diferencia entre la muestra actual y una estimación de la misma ($e_n = y_n - \hat{y}_n$). Para reconstruir la señal a partir de e_n se utiliza la misma estimación, de manera que la señal se recupera tal y como era en origen siempre y cuando los valores de e_n estén dentro del rango del cuantificador escogido. La estimación se obtiene utilizando bien algún tipo de polinomio predictor de orden cero u orden uno como los que se muestran en el apartado *1.2.1.1.*, o bien el *predictor lineal,* que para este caso es óptimo en el sentido de que minimiza el error cuadrático medio. El predictor lineal corresponde a la expresión:

$$\hat{y}(t) = \sum_{j=1}^{M} \beta_j y(t_{n-j}) \tag{1.5}$$

Los pesos β_j se calculan de manera que se minimice el error cuadrático medio, que viene dado por la siguiente expresión:

$$\frac{1}{N}\sum_{k=1}^{N}\left\{ y(t_{n-k}) - \sum_{j=1}^{M} \beta_j (M,N) y(t_{n-k-j}) \right\}^2 \tag{1.6}$$

donde M es el número de muestras precedentes a la actual y N es la longitud de la ventana.

Si los coeficientes del predictor lineal en lugar de estar precalculados los calculamos dinámicamente en función de las variaciones en amplitud de la señal de entrada estaremos hablando de un caso particular de la ADPCM, o *Adaptative Pulse Code Modulation,*. técnica que obtiene mejores resultados pues es posible comprimir más la señal introduciendo un error menor. El caso más general de la ADPCM es aquel en el que tanto el predictor como el cuantificador varían en función de la señal de entrada.

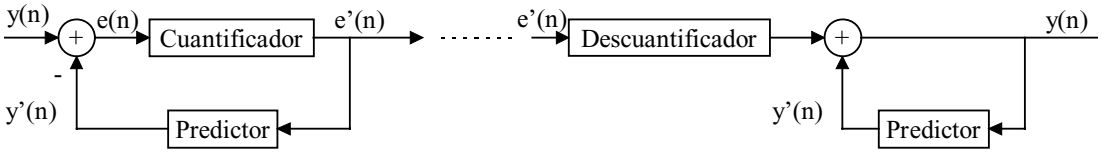


Fig. 1.12 - Diagrama de bloques de la DPCM. Modulación y demodulación.

A veces se realiza una codificación de Huffmann a salida de la DPCM, de manera que se obtiene una mayor compresión y se optimiza la salida del cuantificador, en el sentido de que podemos utilizar un cuantificador sobredimensionado de manera que se eviten, o al menos se reduzcan, los errores a la salida.

Para simplificar, en este trabajo se realizará la implementación de la DPCM, no de la ADPCM. Se hará a partir de un polinomio predictor de orden 1 como caso particular del predictor lineal con coeficientes $\beta_1=2$ y $\beta_2=-1$.

La figura 1.12 muestra el diagrama de bloques que describe el funcionamiento de la DPCM, las partes de modulación y de demodulación. El bloque predictor es el mismo en ambas partes.

1.2.2. TÉCNICAS DE COMPRESIÓN POR TRANSFORMACIÓN.

Como se dijo en la introducción del capítulo, este tipo de técnicas se basan en realizar una transformación ortogonal sobre la señal para después eliminar de esta las muestras que no aportan información de interés. Para recuperar la señal original se procede a aplicar la transformación inversa.

Las transformaciones ortogonales son todas casos particulares de la Transformada de Karhunen-Loeve (KLT), que es la que mejores resultados ofrece en compresión pues es capaz de representar la señal con un menor número de términos. Sin embargo su cómputo es complejo y costoso, pues sus vectores base se determinan a partir de los valores y vectores propios de la matriz de covarianza de la señal original. Por este motivo se utilizan en su lugar otras transformaciones más sencillas cuya propiedad principal es que sus vectores base, a diferencia de los de la KLT, son independientes de la señal de entrada, y pueden calcularse *a priori*, de manera que al realizar la transformación estos se encuentran ya calculados y almacenados, y el cómputo se realiza mucho más rápidamente.

Nosotros vamos a centrarnos en dos de estas transformadas, que son, por otra parte, las más conocidas y utilizadas: La Transformada Discreta de Fourier (DFT), cuyos vectores base son senos y cosenos, y la Transformada Discreta del Coseno (DCT), cuyos vectores base son cosenos.

1.2.2.1. TRANSFORMADA DISCRETA DE FOURIER.

La Transformada Discreta de Fourier se define como:

$$X(k) = \sum_{n=o}^{N-1} x(n) \cdot e^{-j\frac{2\pi nk}{N}} \tag{1.7}$$

donde:

X(k): DFT.
x(n): Señal original.
N: Número de elementos de la secuencia.

La Transformada Inversa se define como:

$$x(n) = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X(k) \cdot e^{j\frac{2\pi nk}{N}} \tag{1.8}$$

A continuación se muestran algunas propiedades de las DFT que ayudarán a establecer las bases para poder realizar un algoritmo de bajo coste de cómputo para su cálculo en tiempo real:

- **Periodicidad:**

$$X(k+N) = X(k) \qquad \forall \text{ k} \tag{1.9}$$

- **Linealidad:**

$$\begin{gathered} x_1(n) \Leftrightarrow X_1(k) \\ x_2(n) \Leftrightarrow X_2(k) \\ a_1 x_1(n) + a_2 x_2(n) \Leftrightarrow a_1 X_1(k) + a_2 X_2(k) \end{gathered} \tag{1.10}$$

- **Complejo conjugado:**

$$\begin{gathered} x(n) \Leftrightarrow X(k) \\ x^*(n) \Leftrightarrow X^*(N-n) \end{gathered} \tag{1.11}$$

Por comodidad, se define el factor W como:

$$W_N^{nk} = e^{j\frac{2\pi nk}{N}} \tag{1.12}$$

de manera que podemos escribir la expresiones (1.7) y (1.8) como:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) \cdot W_N^{-nk} \\ x(n) &= \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot W_N^{nk} \end{aligned} \tag{1.13}$$

Algunas propiedades de los factores W son:

- **Simetría:**

$$W_N^{k+N/2} = -W_N^k \tag{1.14}$$

- **Periodicidad:**

$$W_N^{k+N} = W_N^k \tag{1.15}$$

1.2.2.2. ALGORITMO FFT (Fast Fourier Transformer)

Si se realiza la implementación de la DFT directamente puede observarse que para calcular cada término de la DFT se necesita según la expresión (1.7) los N términos de la secuencia original *x(n)*, y que por tanto se obtiene un algoritmo cuyo coste es de orden cuadrático $\Theta(N^2)$.

El algoritmo FFT surge para mejorar ese coste cuadrático y está basado en aplicar la técnica de divide y vencerás aprovechando las propiedades de la DFT y los factores W que vimos anteriormente. El algoritmo FFT se basa en calcular la transformada de N elementos a partir de las transformadas de las secuencias pares e impares de N / 2 obtenidas a partir de la secuencia original.

Podemos escribir la expresión (1.7) como:

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{nk} \\ &= \sum_{m=0}^{N/2-1} x(2m+1) W_N^{(2m+1)k} + \sum_{m=0}^{N/2-1} x(2m) W_N^{2mk} \end{aligned} \tag{1.16}$$

sin más que dividir la secuencia *x(n)* en sus secuencias par *x(2n)* e impar *x(2n+1)*. Dado que $W_N^2 = W_{N/2}$, tenemos que:

$$\begin{aligned} X(k) &= \sum_{m=0}^{N/2-1} x(2m) W_{N/2}^{mk} + W_N^k \sum_{m=0}^{N/2-1} x(2m+1) W_{N/2}^{mk} \\ &= F_1(k) + W_N^k F_2(k) \end{aligned} \tag{1.17}$$

Donde $F_1(k)$ y $F_2(k)$ son las transformadas de Fourier para las secuencias par e impar respectivamente.

De esta manera vemos como es posible realizar el cómputo de la DCT a partir de las transformadas triviales para un elemento hasta el número N que queramos. La ventaja principal que tiene esta manera de proceder es que el coste de cómputo de un algoritmo basado en esta técnica es $\Theta(n^* \log n)$, que frente a $\Theta(n^2)$ supone una substancial mejora tal y como se puede apreciar en la figura 1.13.

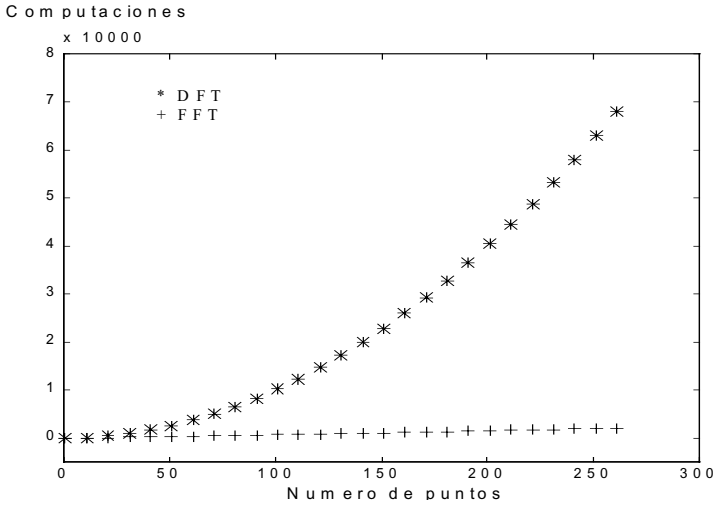


Fig. 1.13 - Comparación frente a N de la DFT y la FFT

Sin embargo, el uso de la FFT para calcular la DFT plantea dos inconvenientes a tener en cuenta. El primero de ellos es que la longitud de la secuencia ha de ser potencia de 2 para que sea posible su división en secuencias de la mitad de elementos hasta llegar a la unidad. En principio esto no supone mayor problema puesto que si no se dispone de una secuencia de longitud potencia de 2 es posible alcanzar la siguiente potencia de 2 por arriba añadiendo ceros sin

que esto suponga introducir errores en el cálculo de la FFT, pero aparecerán lógicamente muestras añadidas en la DFT que quedan interpoladas entre sus términos.

El segundo, y más problemático, es el que se puede apreciar en la figura 1.14. Como consecuencia de la división de cada secuencia en sus términos pares e impares la secuencia final queda desordenada, o mejor dicho ordenada en lo que se denomina *orden bit inverso*. Este orden corresponde al orden normal si se coge cada elemento de la secuencia bit inversa y se les da la vuelta a los bits de manera que el bit menos significativo (LSB) pase a ser el más significativo (MSB) y viceversa, es decir, si se invierte el orden de los bits. Por ejemplo, en una secuencia de 8 elementos como la que se muestra en la figura 1.14 se precisan de tres bits para su codificación. Para el elemento 1 (001) en la secuencia normal su correspondiente en la secuencia bit inversa sería el 4 (100), resultado de invertir el orden de los bits.

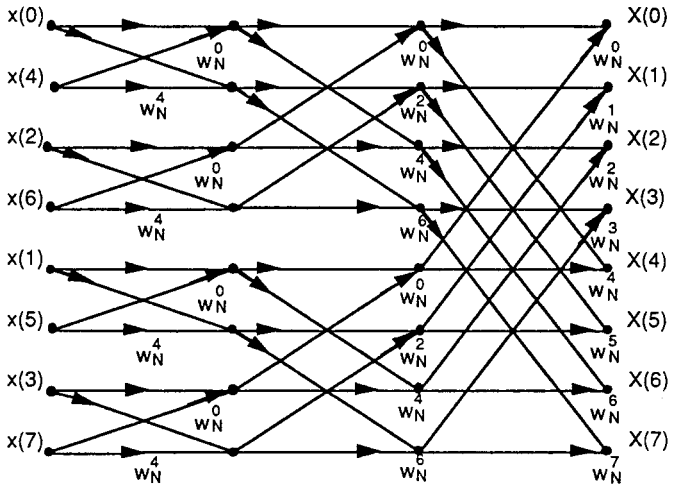


Fig. 1.14 - Ordenación bit inversa para N=8.

La consecuencia de este reordenamiento de los términos de la DFT es tener que implementar una función que organice de nuevo los términos en su orden natural, invirtiendo un mayor tiempo de cálculo. En el caso de el DSP TMS320C25 que nosotros vamos a utilizar para realizar la implementación de la FFT este problema queda elegantemente solucionado, pues el propio hardware del DSP ofrece la posibilidad de realizar la ordenación de forma transparente si que sea necesario añadir código adicional. Para otras plataformas, sin embargo, será necesaria la implementación de dicha función de reordenación.

En la figura 1.15 podemos ver el diagrama de flujo del algoritmo FFT. La operación central del algoritmo, denominada *Operación Butterfly* por la manera en que se combinan los términos pares e impares de la secuencia para obtener el resultado final, es el corazón de dicho algoritmo. Es una operación del tipo A=A+B*C para la cual los DSP están optimizados para realizarla en un solo ciclo de reloj.

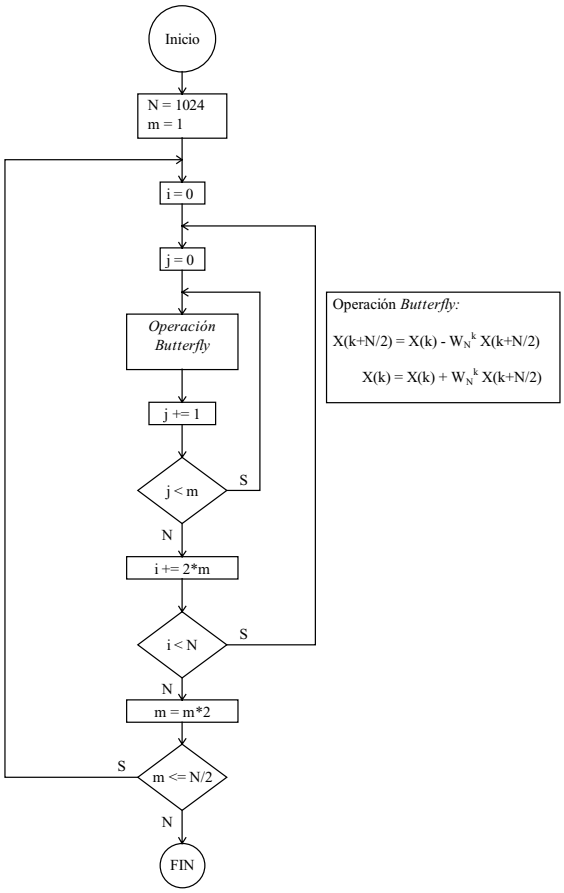


Fig. 1.15. - Diagrama de flujo del algoritmo FFT.

1.2.2.3. COMPRESIÓN MEDIANTE LA FFT.

La compresión de datos de ECG mediante la FFT se realiza normalmente sobre señales tomadas a frecuencias de 500Hz o más, eliminado los términos a partir de los cuales se considera que no hay información de interés para el cardiólogo. En principio es suficiente con aplicar la FFT a la señal de ECG tal y como entra a nuestro sistema para después truncar y quedarnos con los términos frecuenciales de interés. Sin embargo, truncar de esta manera supone que en la reconstrucción de la señal aparezcan discontinuidades en los extremos, discontinuidades que es posible suavizar aplicando algún tipo de promediado en dichos puntos. Si se dispone de sistemas con la suficiente capacidad es posible tratar de evitar este problema aplicando un tipo adecuado de enventanado.

En algunos sistemas, dependiendo de la aplicación, también se procede a un reconocimiento previo de la señal de ECG (detección del QRS, etc.) para realizar el cálculo de la FFT siempre a partir del mismo punto del ECG.

1.2.2.4. TRANSFORMADA DISCRETA DEL COSENO (DCT).

La definición matemática de la Transformada del Coseno es:

$$C_x(k) = \sum_{n=0}^{N-1} 2x(n)\cos\left(\frac{\pi}{2N}k(2n+1)\right), \qquad 0 \le k \le N-1 \tag{1.18}$$

Así como la transformada de Fourier es una representación de la señal sobre la base e^{jwn}, la transformada del coseno es una representación sobre términos de tipo coseno. La ventaja que esto supone en algunas ocasiones es que esta base es mejor para la representación de la señal, quedando esta en su mayoría en los primeros términos de dicha transformada. En el tema que nos ocupa, esto supone un menor número de términos para representar la señal, y por tanto una mayor compresión.

La transformada del coseno se utiliza actualmente en compresión de audio y vídeo. Nosotros vamos a comprobar en este trabajo los resultados de su utilización en la compresión de ECG en tiempo real.

1.2.2.5. ALGORITMO DCT.

A continuación se presenta el desarrollo teórico que muestra como es posible realizar el cálculo de la DCT a partir del cálculo de la FFT. En nuestro caso, esto es especialmente adecuado dado que partimos de un algoritmo FFT que funciona en tiempo real que ya tenemos implementado.

Para calcular la DCT vamos a seguir los siguientes pasos:

$$x(n) \leftrightarrow y(n) \xleftrightarrow{FFT} Y(k) \leftrightarrow C_x(k)$$

es decir, partiendo de la secuencia *x(n)* de *N* puntos llegaremos a la secuencia *y(n)* de *2N* puntos como ahora indicaremos, de la cual calcularemos la Transformada de Fourier para obtener *Y(k)* de *2N* puntos, que nos llevará a la Transformada del Coseno $C_x(k)$ final de *N* puntos.

La secuencia *y(n)* se obtiene a partir de *x(n)* como:

$$y(n) = \begin{cases} x(n), & 0 \le n \le N-1 \\ x(2N-1-n), & N \le n \le 2N-1 \end{cases} \tag{1.19}$$

la secuencia *y(n)* es pues simétrica con respecto a la muestra intermedia, *n=N*. Calculamos ahora *Y(k) = DFT (y(n))* como:

$$Y(k) = \sum_{n=0}^{2N-1} y(n)W_{2N}^{kn}, \quad 0 \le k \le 2N-1 \tag{1.20}$$

Que podemos escribir teniendo en cuenta (1.19) como:

$$Y(k) = \sum_{n=0}^{N-1} x(n) W_{2N}^{kn} + \sum_{n=N}^{2N-1} x(2N-1-n) W_{2N}^{kn}, \quad 0 \le k \le 2N-1 \tag{1.21}$$

Realizando un cambio de variables y efectuando algunas operaciones, obtenemos:

$$Y(k) = W_{2N}^{-k/2} \sum_{n=0}^{N-1} 2x(n) \cos\frac{\pi}{2N} k(2n+1), \quad 0 \le k \le 2N-1 \tag{1.22}$$

que nos conduce finalmente a la expresión de la transformada del coseno para *x(n)*:

$$C_x(k) = \begin{cases} W_{2N}^{k/2} Y(k), & 0 \le k \le N-1 \\ 0, & \text{en otro caso} \end{cases} \tag{1.23}$$

que es justamente:

$$C_x(k) = \sum_{n=0}^{N-1} 2x(n) \cos\left(\frac{\pi}{2N} k(2n+1)\right), \quad 0 \le k \le N-1 \tag{1.24}$$

la expresión (1.18) de la Transformada del Coseno para *x(n)*.

Con esto queda demostrado que es posible calcular la Transformada del Coseno de *N* elementos a partir de la Transformada de Fourier de *2N* elementos. Vamos a ver ahora que, para cuando el número *N* de elementos es par, es posible calcular la Transformada del Coseno de *N* elementos a partir de la Transformada de Fourier de *N* elementos. Para ello dividiremos la secuencia *y(n)* obtenida en (1.19) en dos subsecuencias:

$$\begin{aligned} v(n) &= y(2n), \quad 0 \le n \le N-1 \\ w(n) &= y(2n+1), \quad 0 \le n \le N-1 \end{aligned} \tag{1.25}$$

subsecuencias que cumplen la siguiente propiedad:

$$w(n) = v(N-1-n), \quad 0 \le n \le N-1 \tag{1.26}$$

De esta manera, podemos volver a escribir la expresión (1.20) como:

$$Y(k) = \sum_{n=0}^{N-1} v(n) W_N^{kn} + W_{2N}^{k} \sum_{n=0}^{N-1} w(n) W_N^{kn}, \quad 0 \le k \le 2N-1 \tag{1.27}$$

teniendo en cuenta la propiedad que muestra la expresión (1.26), la expresión (1.27) queda como:

$$Y(k) = W_N^{-k} \sum_{n=0}^{N-1} w(n) W_N^{-nk} + W_{2N}^{k} \sum_{n=0}^{N-1} w(n) W_N^{nk}, \quad 0 \le k \le 2N-1 \tag{1.28}$$

A partir de esta expresión, realizando el álgebra correspondiente, llegamos finalmente a:

$$C_x(k) = W_{2N}^{k/2} \sum_{n=0}^{N-1} w(n) W_N^{nk} \quad 0 \le k \le N-1 \tag{1.29}$$

Expresión que relaciona la Transformada del Coseno de *x(n)* con la Transformada de Fourier de la secuencia *w(n),* que se obtiene a partir de *x(n),* y que no es más que una reordenación de la misma en la que en primer lugar se colocan los elementos pares de la secuencia, y después los impares en orden inverso. La expresión (1.29) muestra como es posible realizar la Transformada del Coseno de *N* elementos a partir de la Transformada de Fourier de *N* elementos.

1.2.2.6 COMPRESIÓN MEDIANTE LA DCT.

Tal y como ocurre con todos o casi todos los algoritmos de compresión basados en transformaciones ortogonales, todo lo que hay que hacer tras realizar la transformada es descartar aquellos términos de la misma que no contienen información de interés de la señal.

Como en el caso de la FFT, es posible tratar la señal de entrada directamente sin necesidad de realizar un reconocimiento previo, y truncar la DCT a partir del término a partir del cual se considere que no hay información de interés, pero en muchos casos nos quedarán discontinuidades en los extremos de la señal reconstruida. Se pueden evitar estas discontinuidades aplicando un enventanado previo adecuado, aunque para ello necesitaremos más velocidad y potencia de cálculo.

En algunos casos puede resultar conveniente realizar un reconocimiento previo de la señal de ECG antes de proceder a calcular la DCT, de manera que esta se aplique siempre a partir de un mismo punto reconocido de la señal de ECG.

CAPÍTULO 2

DESCRIPCIÓN DEL HARDWARE

CAPÍTULO 2: DESCRIPCIÓN DEL HARDWARE UTILIZADO.

Los DSP son chips orientados al procesado digital de señales, como su propio nombre indica (Digital Signal Processor). Su arquitectura interna y juego de instrucciones están pensados para facilitar la implementación de funciones relacionadas con el tratamiento digital de señales, como lo son filtros, transformaciones ortogonales, convoluciones, etc.

La gran parte de sus instrucciones de cálculo se realizan en un sólo ciclo de reloj. Particularmente están optimizadas las instrucciones para realizar suma y producto en un solo ciclo de reloj.

Además, su arquitectura interna contiene buses separados para acceder a la memoria de datos y a la memoria de programa , en vistas a acelerar los accesos a memoria. Este tipo de arquitectura se llama Arquitectura Harvard.

Estas y otras características que veremos más adelante, hacen de los DSP candidatos idóneos para implementar algoritmos de compresión en tiempo real para ECG. Para el cálculo de algoritmos de transformación se han introducido en las últimas generaciones de estos chips importantes mejoras, como son direccionamiento bit inverso, operaciones en coma flotante, nuevas instrucciones, etc.

Para la implementación de nuestros algoritmos se ha utilizado el DSP de Texas Instruments TMS320C25. Como veremos a continuación, este DSP reúne las características necesarias para el trabajo propuesto en cuanto a prestaciones de velocidad y cálculo.

El desarrollo de la programación del TMS320C25 se ha realizado a través de la placa deAriel DTK - C25+, cuyas especificaciones y aspectos más importantes se presentan más adelante en este mismo capítulo. A continuación se muestran las características más relevantes del TMS320C25.

2.1. TMS320C25.

CARACTERÍSTICAS GENERALES.

- Ciclo de instrucción: de 80ns a 120ns (Operación a 50Mhz o 40Mhz respectivamente).
- 544 palabras (16 bits) de memoria de datos (RAM) interna.
- 4K palabras de memoria de programa (ROM) interna.
- 128K palabras de memoria de datos / programa en total.
- 32 bits ALU / Acumulador.
- 16 x 16 bits multiplicador paralelo para productos de 32 bits.
- Instrucciones producto / acumulación en un solo ciclo.
- Instrucciones de repetición para un uso eficiente de la memoria de programa y más rápida ejecución.
- Posibilidad de mover bloques de memoria de datos a programa y viceversa.
- Timer interno para operaciones de control.
- 8 registros auxiliares con una ALU dedicada.
- 8 niveles de pila en hardware.
- 16 canales de entrada y 16 canales de salida.
- Desplazador paralelo de 16 bits.
- Estados de espera para comunicaciones.
- Puerto serie para codec interface directo.
- Entrada de sincronización para configuraciones con multiprocesador.
- Interface de memoria global.
- Instrucciones para filtrado adaptativo, FFT, y aritmética de precisión extendida.
- Modo de direccionamiento bit inverso para FFT de raíz 2.
- Generador de señal de reloj interno.

ARQUITECTURA.

La arquitectura del C25 pone especial énfasis en la velocidad, comunicación y flexibilidad en la configuración del procesador. Las características más importantes de esta arquitectura son:

Arquitectura Harvard.

Este tipo de arquitectura mantiene separados los buses de programa y datos para mejorar el tiempo de ejecución. Externamente es posible multiplexar estos buses para formar uno solo y de esta manera aumentar el rango de direcciones de memoria accesibles.

Memoria Interna.

Internamente se dispone de dos bloques de memoria RAM que forman un total de 544 palabras de 16 bits (fig. 2.1). Uno de estos bloques es, además, configurable como memoria de programa o datos.

Externamente es posible acceder hasta 64K palabras directamente, lo cual facilita la implementación de algoritmos para procesado digital de señales.

Se dispone también de una ROM interna de 4K de rápida ejecución que puede reducir el costo del sistema. Si se utiliza una EPROM es posible evaluar y modificar el sistema rápidamente. Esta EPROM está equipada además con un sistema de protección de la información para evitar copias del código.

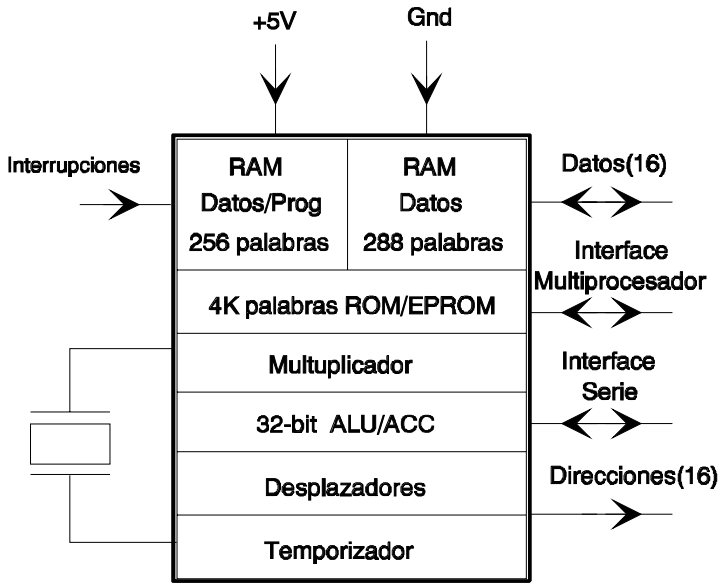


fig. 2.1 - Diagrama de Bloques Simplificado.

Unidad Aritmético Lógica.

La ALU puede trabajar con números de 16 bits tomados directamente de la memoria o instrucción en curso, o bien con números de 32 bits provenientes del multiplicador. Además de los cálculos habituales, la ALU puede realizar cálculos booleanos y de manipulación de bits.

El acumulador recoge la salida de la ALU, y este es a su vez reentrada de la misma. El acumulador es de 32 bits y está preparado para trabajar independientemente con su parte alta y / o baja, de 16 bits cada una.

Multiplicador.

El multiplicador es capaz de realizar productos de dos números de 16 bits, con un resultado de 32 bits, en un solo ciclo de reloj. Consta de tres elementos: el

registro T (de 16 bits), el registro P (de 32 bits) y un array. El registro T contiene el multiplicando y el registro P guarda el resultado del producto. La rapidez con la que el multiplicador realiza su función permite implementar eficientemente las operaciones DSP básicas, como son la convolución, correlación y filtrado.

Se dispone de un desplazador a la salida de -6, 0, 1 ó 4 bits, y otro de 16 bits proveniente del bus de datos, ambos con salidas de 32 bits, que mediante un multiplexor se conectan a la ALU (fig. 2.2).

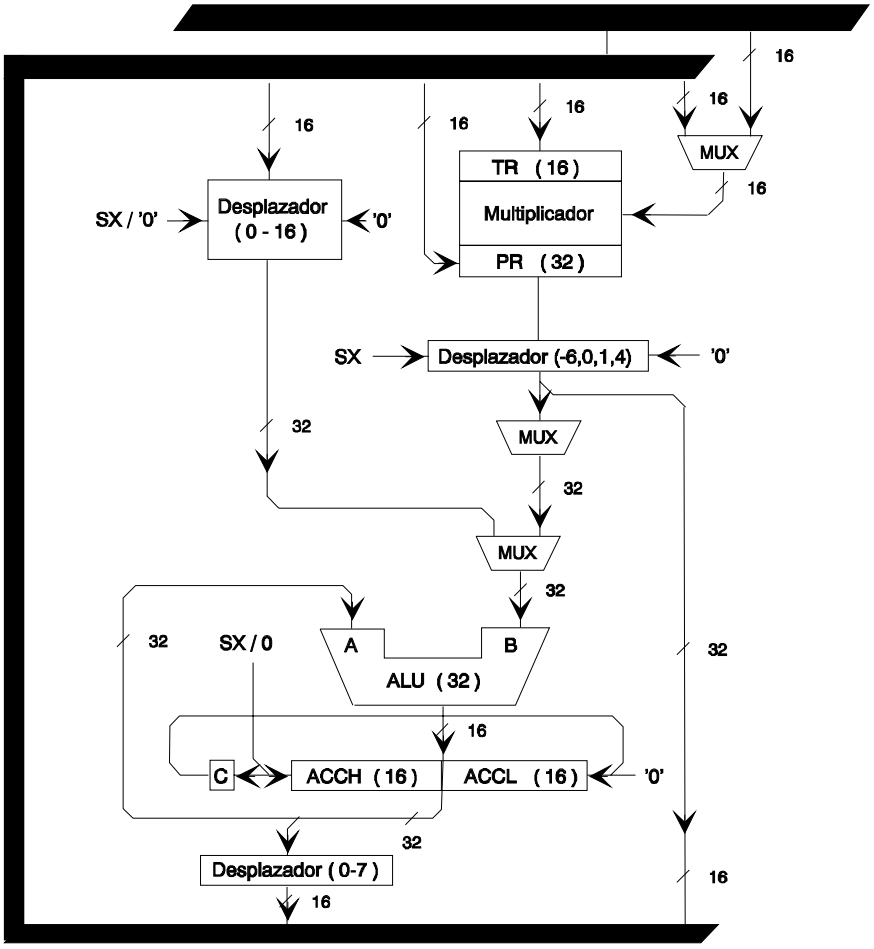


fig. 2.2 - Diagrama de bloques de la CALU.

Interface de memoria.

Consta de un bus de datos paralelo de 16 bits, un bus de direcciones de 16 bits, tres pines para memoria de datos/programa o selección del espacio I/O, y varias señales de control.

Usando la RAM interna, la ROM/EPROM o memoria externa de alta velocidad el TMS320C25 funciona a máximo rendimiento.

Puerto serie.

Se dispone de un puerto serie interno full-duplex para comunicaciones externas con codecs, conversores A / D, etc.

Aplicaciones Multiproceso.

El TMS320C25 puede gestionar memoria global y comunicarse con ella vía las señales de control / BR (bus request) y READY.

Acceso Directo a Memoria (DMA).

Se puede acceder directamente a la memoria externa utilizando las señales / HOLD y / HOLDA. Dispone de dos modos de operación: en uno de ellos el C25 queda suspendido, y en el otro puede seguir funcionando concurrentemente con la operación DMA utilizando la memoria interna.

ORGANIZACIÓN DE LA MEMORIA.

Tenemos memoria de datos y memoria de programa.

En la memoria de datos hay 544 palabras de 16 bits divididas en tres bloques (B0, B1 y B2, fig. 2.3) de las cuales 288 pertenecen a la memoria de datos, y 256 (bloque B0) pueden configurarse como memoria de datos o programa.

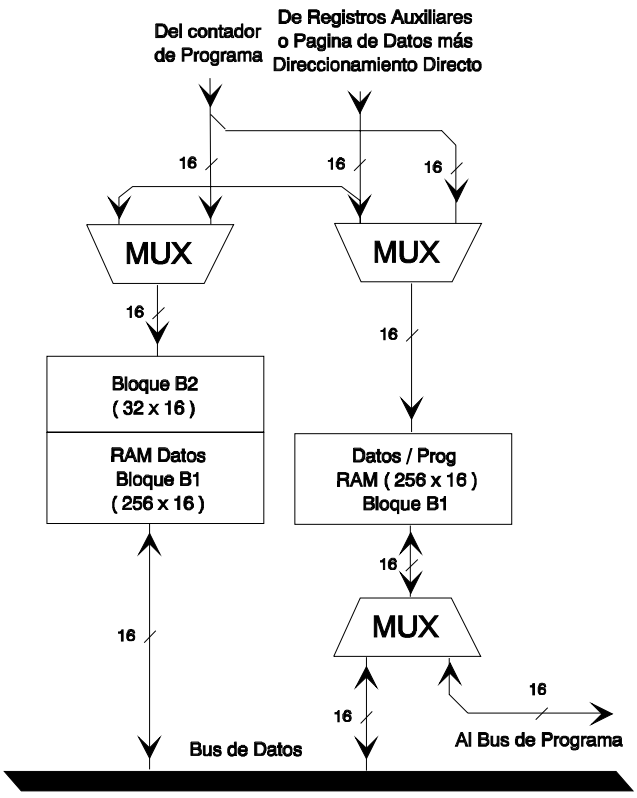


fig. 2.3 - Organización de la memoria interna.

En cuanto a la memoria de programa, puede ser RAM, ROM/EPROM o externa de alta velocidad para que no se produzcan retrasos. El espacio accesible es de 64K palabras.

Mapa de memoria.

El mapa de memoria se halla representado en las fig.2.4a y 2.4b. El bloque B0, de 256K palabras, reside en las páginas 4 y 5 (direcciones 200h - 2FFh) de la memoria de datos, si está configurado como tal (instrucción CNFD), o bien en las direcciones 0FF00h - 0FFFFh si reside en memoria de programa (instrucción CNFP). El bloque B1 reside en las páginas 6 y 7 (direcciones 300h - 3FFh) y el B2 se encuentra en la página 0 (direcciones 60h - 7Fh). La página 0 contiene también los registros mapeados en memoria (ver registros mapeados) y algunas posiciones internas reservadas. La páginas 1-3 (direcciones 80h - 1FFh) están también reservadas.

La memoria ROM/EPROM interna se mapea por debajo de las primeras 4K palabras de la memoria de programa. La memoria de programa contiene en sus primeras 1Fh palabras las zonas para las interrupciones y memoria reservada.

Registros mapeados.

Son 6 y residen en las primeras 6 palabras de la memoria de datos. Se muestran en la tabla 2.1.

Nombre	**Localización**	**Definición**
DRR (15-0)	0	Datos recibidos puerto serie.
DXR (15-0)	1	Datos transmitidos puerto serie
TIM (15-0)	2	Timer
PRD (15-0)	3	Periodo
IMR (5-0)	4	Registro de Máscara de Interrupciones
GREG (7-0)	5	Localización de memoria global.

Tabla 2.1 - Registros Mapeados.

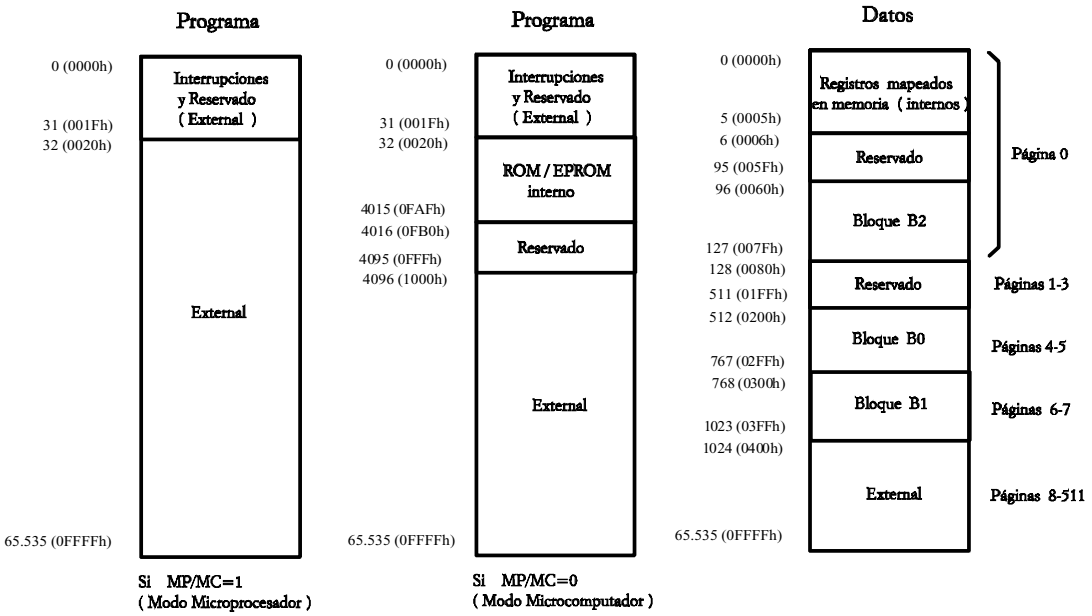


fig. 2.4a - Mapa de Memoria tras la instrucción CNFD.

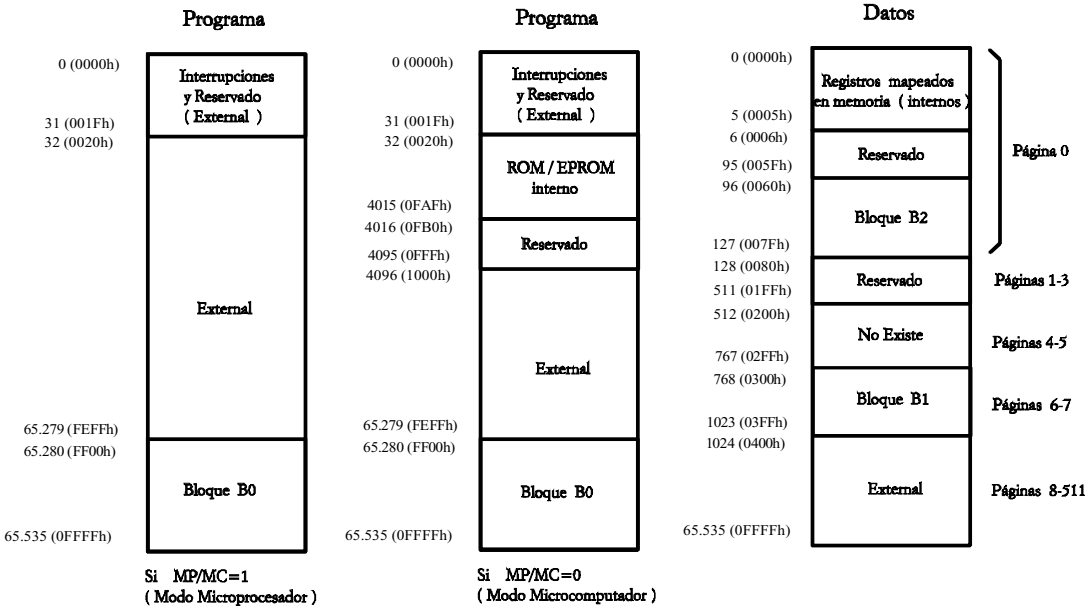


fig. 2.4b - Mapa de Memoria tras la instrucción CNFP.

PIPELINE.

Una de las características más interesantes del hardware del C25 es la existencia de una pipeline. El método de utilizar pipeline consiste en dividir cada instrucción a ejecutar en distintas fases. Estas fases son, para el C25, prebúsqueda, decodificación y ejecución. Si bien cada instrucción ha de pasar por estas fases, nada impide comenzar cualquiera de estas fases en la siguiente instrucción a ejecutar cuando la que está en curso ya ha terminado con ella.

El número de instrucciones que se ejecutan simultáneamente depende del número de niveles de la pipeline. El C25 dispone de una pipeline de tres niveles, que se reducen a dos cuando se opera con memoria interna RAM, ya que en este caso las operaciones de prebúsqueda y decodificación pueden realizarse en un mismo ciclo. En la figura 2.5 se muestra la operación para esta pipeline.

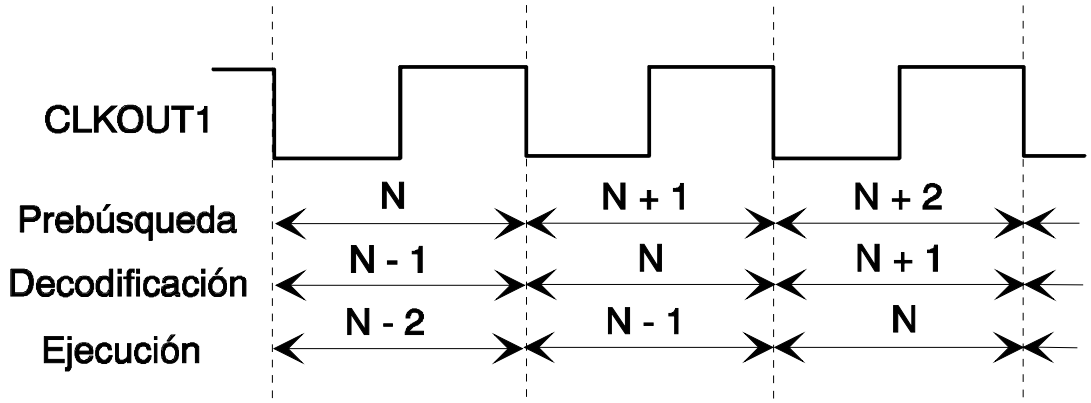


Fig. 2.5 - Operación *pipeline* de tres niveles.

El uso de la pipeline es transparente al programador, con lo cual este no tiene porque preocuparse de ella. Sin embargo, es interesante conocer su funcionamiento, dado que la pipeline funciona a pleno rendimiento cuando la ejecución del programa es de tipo secuencial, debido a que las instrucciones de salto provocan que se vacíe . Convendrá pues, para obtener un funcionamiento óptimo, evitar en la medida de lo posible estas instrucciones de salto que rompen el funcionamiento de la pipeline. En bucles, para aprovechar la pipeline, el C25 ofrece instrucciones de repetición que permiten su uso sin que se produzca el vaciado.

REGISTROS DE ESTADO.

Para conocer el estado interno del DSP y los resultados de las operaciones realizadas, el TMS320C25 dispone de dos registros de estado, ST0 y ST1. Estos registros pueden guardarse en memoria de datos y también cargarse desde ella, de cara a poder realizar adecuadamente el tratamiento de las interrupciones. La organización de estos registros es la siguiente:

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST0	ARP			OV	OVM	1	INTM	DP								

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ST1	ARB			CNF	TC	SXM	C	1	1	HM	FSM	XF	FO	TXM	PM	

Detallada información acerca de los registros auxiliares y sus bits de estado podemos encontrar en [9].

REGISTROS AUXILIARES.

El TMS320C25 dispone de 8 registros auxiliares, numerados como AR0 .. AR7, que se utilizan principalmente para operaciones de control de bucles, direccionamiento y almacenamiento temporal. Mediante los registros auxiliares es posible además realizar direccionamientos complejos como el que se utiliza en la implementación de la FFT, donde se requiere obtener direcciones invirtiendo el orden los bits.

La organización de los registros auxiliares se puede observar el la figura 2.6.

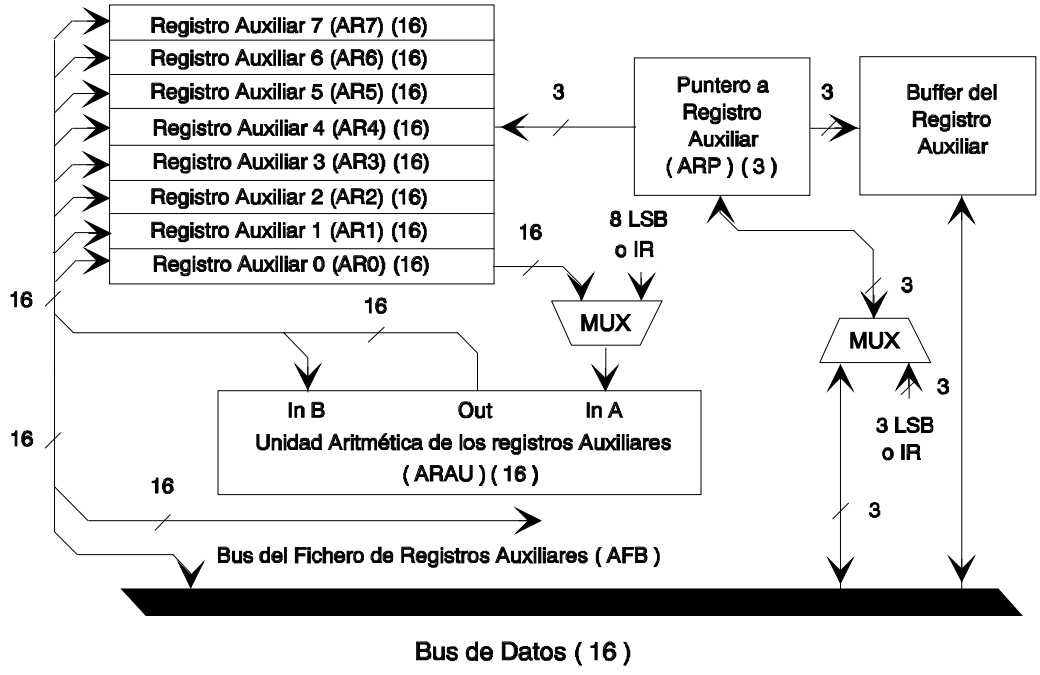


Fig. 2.6 - Registros Auxiliares. Organización interna.

Los registros auxiliares se usan para el direccionamiento indirecto. Estos registros se hayan apuntados por un registro de tres bits, el ARP (Auxiliary Register Pointer), que indica cual de ellos,.AR0 .. AR7, está activo. Los registros pueden ser cargados con valores residentes en memoria, o bien por una constante que se indica en la instrucción correspondiente. También es posible almacenar el valor de estos registros en memoria.

Los registros auxiliares están conectados a una unidad aritmética propia, la ARAU (Auxiliary Register Aritmetic Unit), como muestra la figura 2.6. Esta unidad se encarga de calcular la dirección a la que apuntan los registros. Puede efectuar incrementos o decrementos en una unidad, o bien sumar y restar a cada registro el contenido del registro auxiliar AR0. El resultado de tener una unidad aritmética propia es que la actualización de estos registros no requiere la utilización de la unidad aritmética central (CALU), incrementando significativamente la velocidad del sistema.

Como se ve en la figura 2.6, a una de las entradas de la ARAU puede ir conectado el registro AR0 o los 8 bits inferiores del registro de instrucción. A la otra entrada se conecta el registro auxiliar apuntado por ARP. En la tabla 2.2 se resumen las operaciones que puede realizar la ARAU. AR(ARP) se refiere al contenido del registro auxiliar apuntado por ARP.

AR(ARP) + AR0 => AR(ARP)	Suma al registro auxiliar apuntado por ARP el contenido del registro AR0
AR(ARP) - AR0 => AR(ARP)	Resta al registro auxiliar apuntado por ARP el contenido del registro AR0
AR(ARP) + 1 => AR(ARP)	Incrementa en 1 el contenido del registro apuntado por ARP
AR(ARP) - 1 => AR(ARP)	Decrementa en 1 el contenido del registro apuntado por ARP
AR(ARP) => AR(ARP)	Se mantiene el valor de AR(ARP)
AR(ARP) + IR (7-0) => AR(ARP) *	Suma un valor inmediato de 8 bits al contenido de AR
AR(ARP) - IR (7-0) => AR(ARP) *	Resta un valor inmediato de 8 bits al contenido de AR
AR(ARP) + rcAR0 => AR(ARP) *	Función de direccionamiento bit - inversa. Suma al contenido de AR con propagación de acarreo inversa (rc) de AR0
AR(ARP) - rcAR0 => AR(ARP) *	Resta al contenido de AR con propagación de acarreo inversa de AR0

(*) A partir del TMS320C25

Tabla 2.2 - Operaciones de los registros auxiliares.

Además de como unidad para el direccionamiento, la ARAU puede usarse también como unidad aritmética de propósito general, pues los registros auxiliares están directamente conectados con la memoria de datos. La ARAU implementa una aritmética sin signo de 16 bits.

Hay instrucciones de salto que actúan según el resultado de la comparación entre AR0 y el registro apuntado por ARP. Concretamente, la instrucción BANZ permite utilizar los registros auxiliares como contadores en bucles.

El registro ARB que se muestra en la figura 2.6 sirve de almacenamiento para ARP cuando se producen llamadas a subrutinas o interrupciones.

MODOS DE DIRECCIONAMIENTO.

Para el TMS320C25 se disponen de tres modos de direccionamiento:

- *Direccionamiento directo:* Para este modo de direccionamiento se utiliza el DRB (direct address bus).

- *Direccionamiento indirecto:* Se utilizan los registros auxiliares a través del AFB (auxiliary register file bus). Este tipo de direccionamiento incluye la posibilidad de acceder a las posiciones de memoria en orden bit inverso.

- *Direccionamiento inmediato:* Se accede a los operandos por el contenido del contador de programa.

2.2. PLACA DE DESARROLLO ARIEL - DTK-C25.

Para comprobar el funcionamiento de los algoritmos implementados se ha utilizado la placa de desarrollo Ariel DTK-C25. Esta es una placa que se inserta en cualquier IBM PC o compatible que disponga de un *slot* de 8 ó 16 bits. La DTK-C25 actúa como placa coprocesadora, pues dispone de una CPU propia (el DSP TMS320C25 de Texas Instruments) que puede realizar tareas concurrentemente a la operación normal del PC. Los componentes principales de esta placa son:

- CPU:

Constituida por el TMS320C25 a 50 Mhz (capaz de realizar 12.5 MIPS) y su circuiteria de descodificación de reloj y bus.

- Sección Interface con el Host (PC):

Se compone de la memoria, la circuiteria de entrada/salida buffers del bus de memoria, lógica de interrupciones y un puerto de entrada/salida de 16 bits.

La comunicación del PC con el C25 puede realizarse a través de la memoria o a través del puerto serie. Las transferencias de memoria entre la placa DTK y el PC obligan a detenerse al C25. Es posible realizar transferencias utilizando DMA.

Las transferencias vía puerto serie requieren de un protocolo a establecer entre el host (PC) y el C25, y en ellas el C25 puede continuar con su funcionamiento normal.

- Memoria:

Compuesta por 64K palabras RAM de 0 estados de espera para memoria de programa, y otros tantos para memoria de datos, que completan el mapa de memoria del C25 en su totalidad.

El PC tiene acceso a las 64K palabras que constituyen la memoria de programa utilizando dos páginas de 64 K bytes. Este acceso permite la carga de datos y programas en el C25.

Es posible también acceder a la memoria de datos del C25 utilizando el registro global de memoria del mismo, GREG, como se muestra en la siguiente tabla.

GREG	RAM programa	RAM datos
0	0 - FFFFh	nada
80h	0 - 7FFFh	8000h - FFFFh
C0h	0 - 3FFFh	C000h - FFFFh
E0h	0 - 1FFFh	E000h - FFFFh
F0h	0 - 2FFFh	F000h - FFFFh

- Entrada salida (I / O) serie:

Se trata de una puerta serie bidireccional que puede ser programada para transferencias de 8 ó 16 bits. Esta puerta se utiliza para la comunicación entre el C25 y el conversor AIC TLC32044 de Texas Instruments.

Este conversor está especialmente diseñado para ser conectado al C25. Añade al sistema la posibilidad de conversiones A / D y D / A de 14 bits. Además posee filtros de reconstrucción y *antialiasing*, así como ganancia y frecuencia de muestreo programables por software.

- Puerto paralelo de 50 pines, que puede ser configurado para funcionar en dos modos de operación, simple o múltiple. En el modo simple este puerto actúa como un puerto normal de 16 bits. El modo múltiple nos da acceso a los 8 puertos de entrada / salida del mapa del C25.

CAPÍTULO 3

IMPLEMENTACIÓN DE LOS ALGORITMOS

CAPÍTULO 3: IMPLEMENTACIÓN DE LOS ALGORITMOS.

En este apartado estudiaremos los programas que se han implementado en el presente trabajo.

Para cada uno de los tres algoritmos sometidos a estudio se realizaron tres implementaciones. Dos de ellas en ensamblador del TMS320C25, y la otra en lenguaje C.

La primera versión del programa en ensamblador se realiza para su funcionamiento en el simulador de C25. Esto permite testear y mejorar rápida y adecuadamente los algoritmos, así como comprobar y verificar los tiempos cálculo de cada uno de ellos. El objetivo final es obtener rutinas optimizadas y preparadas para funcionar sobre el C25, pero creando los programas en un entorno mucho más fácil y rápido de manejar.

La versión en ensamblador que funciona sobre el propio C25 en la placa Ariel DTK-C25+ es exactamente igual que la primera, pero se introduce nuevo código para la implementación de las rutinas de tratamiento de interrupciones y comunicación con el PC.

Por último, la versión en C del algoritmo permite probar su funcionamiento sobre distintas máquinas PC, a fin de poder realizar la comparación entre las dos implementaciones (sobre C25 y PC) y obtener conclusiones sobre las ventajas e inconvenientes que una y otra aportan. Dada la portabilidad de los programas escritos en C, esta implementación ofrece además la posibilidad de poder trabajar en el futuro sobre distintas plataformas, incluidos otros DSP provistos de compilador de C, realizando mínimos cambios.

Los programas en código máquina requieren ser cargados en la memoria del C25, alojado en la placa de Ariel DTK-C25+. Si bien esta placa viene acompañada de software adicional para realizar estas tareas, su uso no se ajustaba a las necesidades requeridas, con lo que se creó este software partiendo de cero, utilizando comunicación a bajo nivel con dicha placa.

Los listados fuentes de los programas que a continuación se pasan a comentar se encuentran en el Apéndice A. Estos Apéndices están organizados de la siguiente manera:

Sección A1: Listado de programas ensamblador y C del algoritmo AZTEC.
Sección A2: Listado de programas ensamblador y C del algoritmo FAN.
Sección A3: Listado de programas ensamblador y C de la DPCM.
Sección A4: Listado de programas ensamblador y C de la FFT.
Sección A5: Listado de programas ensamblador y C de la DCT.
Sección A6: Listado de programas ensamblador y C de comunicación del PC con el C25, control y carga de código.

3.1. ALGORITMO AZTEC.

La versión ensamblador, presentada en la sección A1 del apéndice A, sigue el diagrama de bloques de las figuras 1.7 y 1.8 del capítulo 1. Difiere de estos en que la versión implementada admite un número finito de muestras, fijado al principio de la rutina sobre el registro auxiliar AR2. Esto se hace para poder obtener los resultados de tiempo de cálculo, razón de compresión y calidad de la señal comprimida.

La versión que funciona sobre la placa de Ariel DTK C25 es igual. Únicamente se introduce en ella un nuevo segmento de código para la comunicación con el PC.

Se presenta también la versión C del algoritmo, que sigue también el diagrama de bloques presentado en el capítulo 1.

Por último, se presenta también el código fuente en C de la rutina Un-Aztec que descomprime los ficheros comprimidos por este método.

El código necesario para la comunicación entre el host y el C25 se presenta en la sección A6 del apéndice A, y es el mismo para los tres algoritmos.

3.2. ALGORITMO FAN.

La implementación del algoritmo FAN para ensamblador del C25 es una versión optimizada para lenguaje máquina del diagrama de bloques de la figura 1.10 presentado en el capítulo 1. Cabe destacar la implementación de una pequeña subrutina que se encarga de realizar la división entera de dos números, necesaria para el cálculo de las rectas que definen las áreas en este algoritmo. Dicha rutina se encuentra al final del listado ensamblador incluido en el sección A2 del apéndice A.

En cuanto a la versión ensamblador preparada para funcionar sobre el C25, es exactamente igual a esta, en la que como siempre se añade nuevo código para la comunicación con el host.

Se presenta también la versión C, que se utilizó para realizar la comparativa de tiempos entre el C25 y diversas plataformas PC, y el código C Un-Fan que des comprime los ficheros comprimidos con este compresor.

3.3 DPCM (Differential Pulse Code Modulation).

Para la implementación de un algoritmo compresor basado en DPCM se ha seguido el diagrama de bloques de la figura 1.12 del capítulo 1.

Se realizaron pruebas con el polinomio interpolador de orden cero (ZOI), $\hat{y}_n = y_{n-1}$, y el polinomio predictor de orden 1 (FOI), que responde a la ecuación

$\hat{y}_n = 2y_{n-1} - y_{n-2}$. Como era de esperar en principio el FOI obtiene mejores resultados.

En la mayoría de estudios realizados [11], entre otros [31][32], se concluye que el orden óptimo, para el predictor lineal o polinomio interpolador a utilizar, es 1,y que al aumentar el orden no se obtiene una mejora significativa en los resultados. Por otra parte, en [31] se evalúa la utilización del predictor lineal y el polinomio interpolador (FOI), obteniéndose que los resultados obtenidos por ambos son bastante similares, si bien el predictor lineal da resultados un poco mejores. En ambos casos, tanto si aumentamos el orden del predictor escogido, como si utilizamos el predictor lineal, el algoritmo se complica respecto a la utilización del FOI sin que se obtenga una substancial mejora.

Por estos motivos se prefirió realizar la implementación del predictor de la DPCM utilizando el FOI, pues da buenos resultados, su implementación es sencilla, y es muy rápido dado que solamente requiere de rotaciones de bits y sumas.

Los ECG de entrada están tomados a una frecuencia de 250Hz con 12 bits de resolución, y el cuantizador a la salida es de 8 bits. Esto conlleva una ventaja importante, y es que la salida del algoritmo son bytes, con los cuales es más sencillo operar que en el caso de un número de bits no múltiplo de 8. Puesto que en el C25 las celdas de memoria almacenan palabras de 16 bits la implementación del algoritmo se ha realizado de tal forma que a la salida se agrupan dos muestras.

En el sección A3 del apéndice A está el listado ensamblador correspondiente a la implementación del algoritmo DPCM. En el mismo listado está también resuelta el algoritmo que realiza la demodulación.

Se presenta también el algoritmo implementado en C, cuyo funcionamiento es análogo al funcionamiento del programa ensamblador, de cara a realizar las comparaciones pertinentes.

3.4. ALGORITMO FFT.

La Transformada de Fourier de 1024 puntos necesita 2048 posiciones de memoria para trabajar con los datos de la secuencia, pues se trata de una secuencia de números complejos. Para poder realizar el cálculo en tiempo real es necesario también de un buffer de entrada que se vaya llenando al tiempo que se realiza el cálculo de la FFT, buffer que tendrá que albergar otras 1024 muestras. Se requieren también 2048 posiciones más para albergar los 1024 factores W necesarios, en esta implementación, para su cómputo. Estos factores W son previamente calculados e insertados como una tabla en el código del programa. Por último, hay que contar con el propio código del programa, que para la implementación desarrollada ocupa 104 posiciones de memoria. El código del programa se alojará en el bloque B0 del C25, a fin de que este esté en memoria interna y la velocidad ejecución sea óptima.

La base del algoritmo FFT la constituye la operación *butterfly*, que se encarga de realizar la operación que combina las secuencias par e impar:

$$X(k) = X(k) \pm W_N^k \, X(k+N/2)$$

Esta es la operación que se encuentra en el núcleo del algoritmo de la FFT, y por tanto es la que más veces se repite. Además, al tratarse de operaciones con números complejos, el número de productos y sumas a realizar aumenta. Por esta razón esta operación ha de estar optimizada al máximo, independientemente de la implementación que se escoja, que solamente se diferenciará, en líneas generales, en la forma de tratar los bucles y almacenar los datos en memoria.

La implementación propuesta para esta operación es, en ensamblador del TMS320C25, la siguiente:

```
; Operacion BUTTERFLY:
; Parte real del producto:
        lt      *+,AR2  ; T=Re[X(k+N/2)]
        mpy     *+,AR1  ; P=(1/2) Re[X(k+N/2)]*Re[W]
        ltp     *-,AR2  ; T=Im[X(k+N/2)], ACC=(1/2) Re[X(k+N/2)]*Re[W]
        mpy     *-      ; P=(1/2) Im[X(k+N/2)]*Im[W]
        spac            ; ACC=(1/2) (Re[X(k+N/2)]*Re[W] - Im[X(k+N/2)]*Im[W])
        mpy     *+,AR1  ; P=(1/2) Im[X(k+N/2)]*Re[W]
        lt      *       ; T=Re[X(k+N/2)]
        sach    *+,AR2  ; Re[X(k+N/2)]=(1/2) (Re[x(k+N/2)*Re[W] - Im[X(k+N/2)]+Im[W])
; Parte imaginaria del producto:
        pac             ; ACC=(1/2) Im[X(k+N/2)]*Re[W]
        mpy     *+,AR1  ; P=(1/2) Re[X(k+N/2)]*Im[W]     ; AR2 -> Re[W_N^k+1]
        apac            ; ACC=(1/2) (Im[X(k+N/2)]*Re[W] + Re[X(k+N/2)]*Im[W])
        sach    *-      ; Im[X(k+N/2)]=ACC     ; AR1 -> Re[X(k+N/2)]
; Sumar y guardar datos:
; Parte real:
        lac     *0-,15  ; ACC=(1/4) (Re[X(k+N/2)]*Re[W] - Im[X(k+N/2)]*Im[W])
        add     *,14    ; ACC=(1/4) (Re[X(k)] + (Re[X(k+N/2)]*Re[W]-Im[X(k+N/2)]*Im[W]))
        sach    *0+,1   ; Re[X(k)]=(1/2) ACC
        subh    *       ; ACC=(1/4) (Re[X(k)] - (Re[X(k+N/2)]*Re[W]-Im[X(k+N/2)]*Im[W]))
        sach    *+,1    ; Re[X(k+N/2)]=(1/2) ACC     ; AR1 -> Im[X(k+N/2)]
; Parte imaginaria:
        lac     *0-,15  ; ACC=(1/4) (Im[X(k+N/2)]*Re[W] + Re[X(k+N/2)*Im[W])
        add     *,14    ; ACC=(1/4) (Im[X(k)] + (Im[X(k+N/2)]*Re[W]+Re[X(k+N/2)*Im[W]))
        sach    *0+,1   ; Im[X(k)]=(1/2) ACC
        subh    *       ; ACC=(1/4) (Im[X(k)] - (Im[X(k+N/2)]*Re[W]+Re[X(k+N/2)*Im[W]))
        sach    *+,1,AR3; Im[X(k+N/2)]=(1/2) ACC   ; AR1 -> Re[X(k+1+N/2)]
```

Fig. 3.1 - Implementación de la operación *butterfly*

A destacar de esta implementación que el producto, la suma y el almacenamiento de los resultados se realiza en un número mínimo de operaciones, todas ellas además de ejecución en un sólo ciclo de reloj. En total se usan 22 instrucciones, que se ejecutan en un tiempo de 22 * 80 ns = 1,76 us.

El número de variables utilizadas también es mínimo, recurriendo a guardar el resultado del producto temporalmente en las posiciones que ocupan los elementos de la secuencia impar, Re[X(k+N/2)] e Im[X(k+N/2)], con lo que no se requieren variables adicionales a este efecto.

En algunas de las implementaciones de la FFT sobre la arquitectura TMS320Cx se opta por definir la operación *butterfly* que hemos descrito como una macro que se repite tantas veces como haga falta para el cómputo total de la FFT. Esto supone un gasto mayor de memoria, pero se gana en rapidez pues se eliminan bucles, condiciones y saltos que rompen la dinámica de la pipeline y consumen más ciclos de reloj. Para implementaciones de la FFT de no muchos puntos esta es la mejor manera de proceder. Sin embargo, para nuestro caso tratando con 1024 puntos se decidió realizar una implementación con bucles cuyo funcionamiento está descrito en el diagrama de flujo de la figura 1.15. Con esta implementación se pierde velocidad, pero la obtenida es más que suficiente para la frecuencia de muestreo a la que se trabaja habitualmente, 1kHz, incluso con varios canales. Por contra se ahorra memoria, con lo cual es posible alojar todo el código en el bloque B0 y trabajar desde memoria interna.

Otro de los pormenores de esta implementación lo constituyó la tabla de factores W para alojar en memoria. Tras configurar el bloque B0 como bloque de memoria de programa para alojar allí el código, sólo quedan 2Fh (47d) palabras en el bloque B2 que se utilizan para las variables de control de bucles, etc., y 100h (256d) palabras en el bloque B1. Como ya se dijo se precisan 2048 posiciones para los datos más 2048 para el buffer de entrada que permite trabajar en tiempo real, con lo cual operar con los datos de la secuencia de entrada en memoria interna queda descartado. Por otra parte, como mínimo se necesitan 512 factores W para el cómputo de la FFT de 1024 puntos, y puesto que se trata de números complejos necesitaríamos de 1024 posiciones de memoria, frente a las 256 de las que disponemos en el bloque B1.

Una posibilidad podría ser alojar los factores W en memoria interna los que cupiesen, para el caso 128, y reemplazarlos según fuese necesario. Esta solución, sin embargo, conlleva tener que realizar los pertinentes bucles para el traslado de los coeficientes de memoria externa a interna, resultando en aumento del costo de cómputo de la FFT que termina por hacer inútil el esfuerzo.

Así pues, se decidió trabajar con los factores W en memoria externa. La penalización que supone esto es que, en general, pero no siempre, las instrucciones que acceden a memoria externa utilizan dos ciclos de reloj en lugar de uno como ocurre cuando se accede a memoria interna, a lo que hay que añadir los estados de espera de la memoria en uso. En nuestro caso, la placa Ariel DTK-C25+ dispone de memoria externa de 0 estados de espera.

Trabajando con memoria externa la implementación mostrada en la figura 3.1 de la operación *butterfly* requiere un total de 35 ciclos de reloj, es decir, 2,8us para un tiempo de ciclo de 80ns, lo que supone un incremento del 37% respecto a la velocidad que se obtiene trabajando con memoria interna.

No obstante, trabajar con memoria externa reporta dos ventajas muy a tener en cuenta. En primer lugar, supone que es posible realizar el cómputo de FFTs de un número distinto a 1024 sin más que cambiar la tabla factores W y la variable N que controla en el algoritmo la longitud de la secuencia. Esto es un punto a

tener en cuanto a la hora de la portabilidad de dicho algoritmo a distintas aplicaciones.
La segunda ventaja es que en memoria externa disponemos de espacio suficiente como para organizar los factores W de manera que podamos realizar el cómputo más rápidamente. Esto es, si en lugar de los 512 factores W necesarios utilizamos otros 512 puntos más y una ordenación adecuada es posible realizar el recorrido del array de factores W secuencialmente, evitándonos de esta manera el cálculo necesario para acceder a cada factor en función de la longitud de la secuencia en proceso, y ganando por tanto, tiempo. La ordenación a la que nos referimos se obtiene sin más que disponer en memoria de manera secuencial los factores W necesarios para el cálculo de la FFT para N=2, N=4, N=8, etc., obteniendo de esta manera una secuencia de 1024 factores W.

El listado completo en ensamblador del programa que calcula la FFT en tiempo real sobre la placa Ariel DTK-C25+ se presenta en la sección A4 del apéndice A. Como es lógico, a la propia implementación de la FFT se han añadido las partes de código dedicadas al control de rutinas de inicialización, tratamiento de interrupciones y comunicación con el host (PC) (secciones A4 y A6, apéndice A).

Por último, se presenta la versión en C del algoritmo FFT. Como se indicó en el capítulo 1, para esta implementación se requiere además de una rutina que se encargue de ordenar los datos de entrada según el orden bit inverso, función que realiza el procedimiento *ordena()*. Con el fin de acelerar la rutina se recurre a realizar las divisiones por 2 o por potencias de dos mediante rotaciones de bits, y se utilizan variables intermedias para evitar cálculos reiterativos.

Entre los listados presentados, también se incluye el de la rutina que calcula la tabla de factores W precalculados que han de incluirse en los programas para el C25 y PC.

3.5. DCT (Transformada Discreta del Coseno)

El cálculo de la DCT se realiza, como se vio en el capítulo 1, apartado 1.2.2.5, sobre el cálculo de la FFT. La implementación por tanto sólo requiere añadir el código necesario sobre la implementación de la FFT para obtener la DCT.

Hay un nuevo gasto de memoria, proveniente de el alojo en memoria de la tabla de cosenos (ver expresión 1.29) que lógicamente está precalculada, tabla que consta de 1024 elementos y que por tanto ocupará 2048 posiciones de memoria.

Por otra parte se requiere realizar el producto de estos cosenos sobre la FFT. La implementación de dicho producto es mostrada en la figura 3.2.

Como se puede observar dicha implementación es análoga a la implementación de la operación *butterfly* de la FFT, y presenta las mismas propiedades de

ahorro de memoria y tiempo. Este producto tarda 12 ciclos de reloj si se realiza operando en memoria interna, y 19 ciclos en memoria externa, que con un ciclo de 80ns, dan unos tiempos de 0,96us y 1,52us respectivamente.

```
        LT      *+,AR2          ; T=Re[x]
        MPY     *+,AR1          ; P=(1/2) Re[x]*Re[ww]
        LTP     *-,AR2          ; T=Im[x], Acc=(1/2) Re[x]*Re[ww]
        MPY     *-              ; P=(1/2) Im[x]*Im[ww]
        SPAC                    ; Acc=(1/2) (Re[x]*Re[ww] - Im[x]*Im[ww])
        MPY     *+,AR1          ; P=(1/2) Im[x]*Re[ww]
        LT      *               ; T=Re[x]
        SACH    *+,0,AR2        ; Re[x]=Acc
; parte imaginaria:
        PAC                     ; Acc=(1/2) Im[x]*Re[ww]
        MPY     *+,AR1          ; P=(1/2) Re[x]*Im[ww]  , AR2->siguiente
        APAC                    ; Acc=(1/2) (Re[x]*Im[ww] + Im[x]*Re[ww])
        SACH    *+,0,AR3        ; Im[x]=Acc             , AR1->siguiente
```

Fig. 3.2 - Implementación producto DCT.

Se ha retocado la implementación de la FFT de manera que ahora se precisan 1024 posiciones de memoria menos, debido a que trabajamos con un array de factores W de 512 muestras en lugar de 1024. Es necesario código extra en el bucle principal de la FFT (operación *butterfly*) para poder acceder correctamente a cada factor W en función de la longitud de la FFT que estemos calculando. Se han de añadir 3 instrucciones más, que hacen que la operación *butterfly* se lleve a cabo en 2us en memoria interna, y 3.2us en memoria externa.

La Rutina de Tratamiento de Interrupción que se encarga de tomar muestras de forma concurrente al cálculo de la DCT también ha sido modificada de manera que además de tomar las muestras, lo hace de forma que se obtiene directamente la secuencia w(n) (véanse las expresiónes 1.25 y 1.26 en el capítulo 1), sobre la cual se aplica directamente la FFT. Esto supone un incremento sobre la rutina que usábamos para la FFT de 0,76us, e implica que la frecuencia de muestreo máxima y/o el número de canales pueden verse decrementados.

El listado completo del programa que funciona sobre la placa Ariel DTK-C25+ está en la sección A5 del apéndice A.

En cuanto a la implementación en C se utilizan 1024 bytes como memoria temporal para poder realizar rápidamente el cálculo de la secuencia *w(n)* de la expresión 1.26 a partir de la secuencia *x(n)* original. También se ha modificado la rutina que calcula la FFT de manera que ahora sólo se necesitan 512 factores W, además de utilizar una variable temporal para acelerar el cálculo y de optimizar para el caso de secuencias de longitud N=1.

En la sección A5 del apéndice A se encuentran los listados en C que implementan la DCT, divididos en los siguientes archivos: FFT.C, que contiene las rutinas de cálculo de la FFT, FDCT.C, que contiene las rutinas para el cálculo de la secuencia *w(n)* a partir de *x(n)*, y el producto por los cosenos; y OMEGAS.C que realiza el cálculo *a priori* de las tablas de factores W de la FFT y cosenos de la DCT.

3.6. RUTINAS DE CONTROL.

Es necesario establecer una comunicación entre el PC y el C25 para conocer cuando este último dispone de los datos. A tal efecto se implementaron las rutinas que gestionan esta comunicación en ensamblador y en C, que se encuentran en la sección A6 del apéndice A.

Estas rutinas establecen un protocolo de comunicación muy sencillo. Cuando el C25 quiere comunicar algo al PC, el PC previamente tiene que estar a la escucha. El PC testea el puerto de comunicaciones con el C25 hasta recibir la señal de que hay datos, recoge los mismos ya sea a través de ese mismo puerto de comunicaciones o a través de la memoria, y manda una señal al C25 para que continúe con su trabajo.

Como se comentó en la introducción de este capítulo, para realizar la carga de los programas código máquina en la memoria del C25 fue necesario crear rutinas propias que se adaptaran a los requerimientos exigidos. Asimismo, también fueron implementadas rutinas que permiten visualizar el estado de la memoria del C25, y órdenes de control para la placa Ariel DTK. Todo este código se halla presente en la sección A6.

CAPÍTULO 4

RESULTADOS

CAPÍTULO 4: RESULTADOS.

En el presente capítulo se muestran los resultados obtenidos en la pruebas experimentales realizadas. Para llevarlas a cabo se han utilizado como registros de ECG algunos de los que se encuentran en la base de datos MIT-BIH. En esta base de datos existen varios tipos de registros, desde los más normales hasta algunos más atípicos, de los que un conjunto han sido escogidos por los creadores de esta base para realizar pruebas para algoritmos de compresión de ECG. Nosotros hemos escogido a su vez 10 de estos registros, de 5120 muestras cada uno y muestreados originalmente a una frecuencia de 250Hz.

Para realizar la evaluación de los algoritmos se miden el tiempo y requisitos de memoria de cada uno de los algoritmos. Los algoritmos han sido probados en el C25 y sobre varias plataformas PC, de manera que se tiene una referencia conocida para los resultados obtenidos.

También se ha estudiado la razón de compresión y la calidad de la misma. Para medir la calidad se utiliza comúnmente, para compresión de registros de ECG, el PRD (Percent Root-Mean-Square Difference), cuya definición viene dada por la expresión:

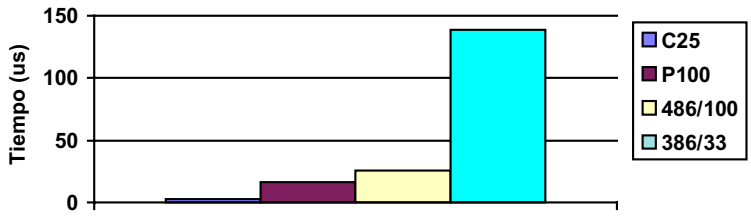
$$PRD = 100 \cdot \sqrt{\frac{\sum_{i=0}^{N} [X_{org}(i) - X_{rec}(i)]^2}{\sum_{i=0}^{N} X_{org}^2(i)}}$$

Se muestran a continuación los resultados de tiempo y memoria obtenidos en las pruebas:

4.1 TIEMPOS Y MEMORIA REQUERIDA.

4.1.1 AZTEC

	C. P. U.			
	C25	**386/33 MHz**	**486/100 Mhz**	**Pentium 100 Mhz**
Tiempo (µs)	2,9	139,9	26,3	15,9
Espacio requerido (bytes)	166	1150		

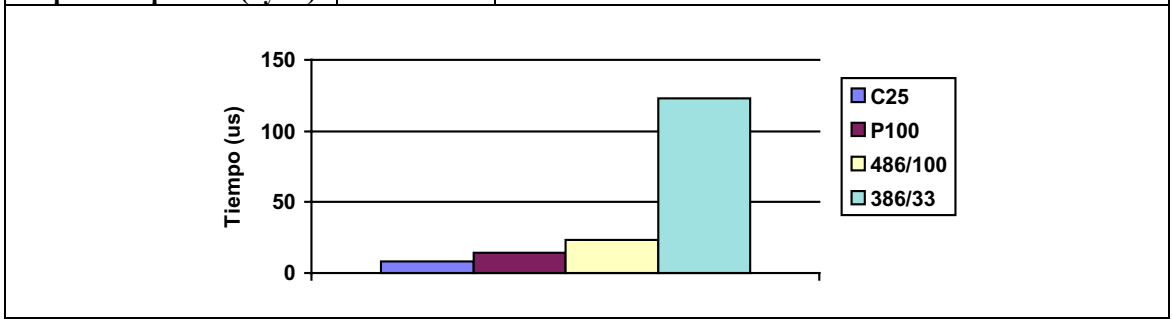


Los resultados obtenidos en la implementación para el C25 del algoritmo AZTEC nos muestran como es posible su utilización en tiempo real. Sería posible llegar a tomar la señal con frecuencias de muestreo de más de 300kHz, aunque esto no tiene mucho sentido para señales de ECG. Si lo tiene, sin embargo, la implementación de varios canales en un mismo sistema. Para el caso las frecuencias típicas de muestreo son de 250Hz, para la cual sería posible implementar aproximadamente 1380 canales. Aumentado la frecuencia de muestreo a 1kHz podríamos tener hasta 345 canales. No se requieren hoy en día sistemas con una cantidad tan grande de canales, pero queda demostrada la potencia del DSP C25 para la aplicación del algoritmo AZTEC. Su rapidez de cálculo, y escasos requisitos de memoria de tan solo 166 bytes, permite utilizar el algoritmo AZTEC como un paso más de un sistema más complejo de procesado de la señal, que podría incluir preprocesado de la señal de ECG, compresión, postprocesado y transmisión.

En cuanto a la comparativa de velocidad frente al PC, el C25 es más rápido que incluso un Pentium a 100Mhz de última generación, mostrándose muy superior a sistemas más antiguos como el 80386 y 80486.

4.1.2. FAN.

	C. P. U.			
	C25	**386/33 MHz**	**486/100 Mhz**	**Pentium 100 Mhz**
Tiempo (μs)	7,6	123,4	23,2	14,2
Espacio requerido (bytes)	109	822		



Como ocurría con el algoritmo AZTEC, la rapidez con la que se ejecuta el algoritmo FAN sobre el C25 permite su utilización en tiempo real con frecuencias de muestreo muy altas, un gran número de canales y/o como parte de un sistema más complejo de tratamiento y compresión de la señal de ECG. Los requisitos de espacio en memoria son asimismo mínimos, menores incluso que en el caso del AZTEC.

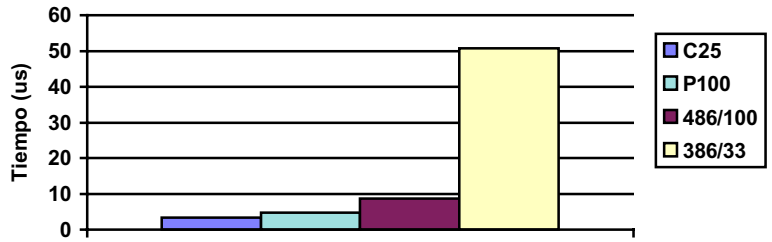
Frente al AZTEC, y en lo que se refiere al C25, FAN es un algoritmo más costoso que requiere más tiempo para tratar las muestras de la señal de ECG. Su coste superior se debe a la realización de la división entera para obtener los límites de las rectas que determinan si la muestra entrante es o no redundante. Como se puede observar comparando los resultados obtenidos para ambos algoritmos, esta operación de división no afecta sin embargo a las plataformas

x86 que obtienen mejores resultados de tiempo de cálculo para el algoritmo FAN que para el AZTEC.

Como en el caso del AZTEC, nuevamente el C25 se muestra superior en cuanto a velocidad de cálculo frente al PC, aunque en esta ocasión el sistema Pentium 100Mhz obtiene tiempos muy cercanos a los del C25, sin llegar a alcanzarlo.

4.1.3. DPCM.

	C. P. U.			
	C25	**386/33 MHz**	**486/100 Mhz**	**Pentium 100 Mhz**
Tiempo (µs)	3,44	50,70	8,81	4,79
Espacio requerido (bytes)	168	975		

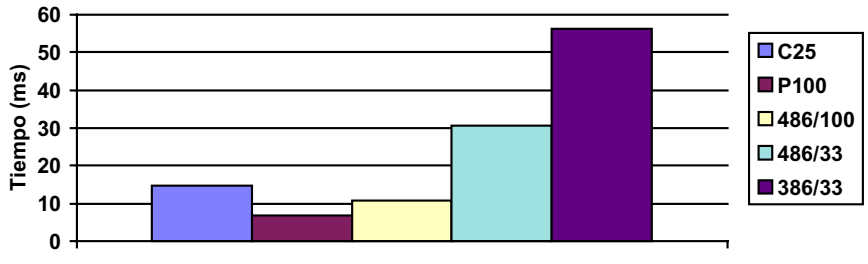


Los tiempos obtenidos para la DPCM son muy bajos, como era de esperar dada la sencillez del algoritmo. La DPCM se presta a ser aplicable como última etapa de compresión después de la aplicación de otro tipo de algoritmo/s, pues no introduce pérdidas, o estas caso de existir son mínimas, aunque esto depende de la frecuencia de muestreo, resolución de la señal de entrada y resolución del cuantificador escogido a la salida.

Como se puede observar, frente al PC el C25 muestra tiempos mucho menores, excepto para el caso del Pentium 100Mhz al que todavía logra superar. Es más del doble de rápido que un 486/100Mhz y casi 15 veces más rápido que un 386/33Mhz.

4.1.3. FFT.

	C. P. U.				
	C25	**386/33 MHz**	**486/33 MHz**	**486/100 MHz**	**P100 MHz**
Tiempo 1024 muestras (ms)	14,70	56,44	30,50	10,84	6,83
Espacio requerido (bytes)	(8.192 + 212)*	(8.192 + 1.545)*			



(*) [memoria datos] + [memoria de programa]

Observando los resultados obtenidos vemos que es posible la realización de la FFT en tiempo real utilizando el C25. Es posible, como ocurría con los algoritmos de compresión directa llegar a frecuencias de muestreo altas (hasta casi 60kHz teóricamente) y disponer de un elevado número de canales (más de 60 muestreando a una frecuencia de 1kHz).

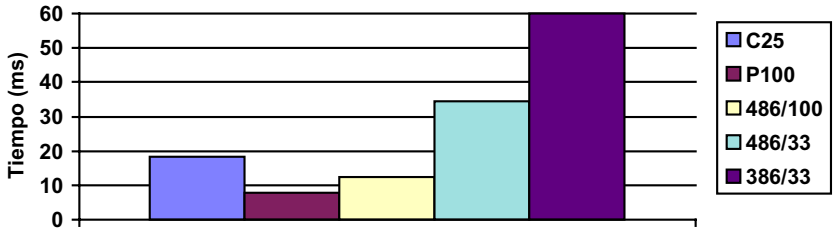
Hacer notar que en esta ocasión los tiempos se obtienen para grupos de 1.024 muestras, pues no tiene sentido hablar de tiempo para cada muestra puesto que el algoritmo FFT trabaja sobre grupos de muestras y, como ya vimos, siguiendo una función del tipo $n \cdot \log_2 (n)$. Además no se tiene en cuenta el retardo producido por la toma de muestras de forma concurrente a la ejecución del algoritmo. En ese caso, suponiendo una frecuencia de muestreo típica de 1kHz, y teniendo en cuenta además que la rutina de servicio de interrupción que se encarga de tomar las muestras invierte un tiempo aproximado de 1,6us / muestra, habría que sumar al total para 1024 muestras un incremento de tiempo de aproximadamente 24us, que como se observa, no afecta prácticamente el tiempo total de cómputo de la FFT. El tiempo de la rutina servicio de interrupción afectará más a medida que se aumente la frecuencia de muestreo, pues se le llamará más veces, y además se dispondrá de menor tiempo global para realizar el cómputo de la FFT.

En contraste con los algoritmos de compresión directa que tenían unos requisitos de memoria mínimos, la FFT requiere de 8.192 bytes (4.096 palabras, posiciones de memoria del C25) para albergar los datos correspondientes a los factores W y la secuencia $x(n)$ original. A esto hay que añadir 4.096 bytes más para el buffer de entrada de datos cuando se trabaja en tiempo real, tanto con el C25 como con el PC. Sin embargo, el código del programa ocupa tan solo 212 bytes.

En cuanto a la comparativa de tiempos frente al PC, podemos ver en la tabla como en esta ocasión el C25 logra superar en velocidad al 386 / 33Mhz y al 486 / 33Mhz, pero sin embargo el 486 / 100Mhz consigue ya rebajar el tiempo de cálculo del C25 en casi 5ms, mientras que el Pentium 100Mhz llega a alcanzar tiempos para 1.024 muestras de tan sólo 6,83ms. No obstante, la FFT de 1.024 puntos obliga por sus requisitos de memoria a trabajar al C25 con memoria de datos externa, lo cual implica que la mayoría de instrucciones que trabajando en memoria de datos interna se ejecutarían en un sólo ciclo de reloj ahora lo hacen en dos ciclos, penalizando notablemente la rapidez del algoritmo en dicho sistema.

4.1.4. DCT

	C. P. U.				
	C25	**386/33 MHz**	**486/33 MHz**	**486/100 MHz**	**P100 MHz**
Tiempo 1024 muestras (ms)	18,20	59,92	34,62	12,28	7,80
Espacio requerido (bytes)	(10.240 + 412)*	(12.288 + 1.905)*			



(*) [memoria datos] + [memoria programa]

El análisis de resultados para la DCT es análogo al realizado para la FFT, y, tal y como ocurría con esta, tenemos que medir el tiempo que tarda en realizar la secuencia completa de 1.024 muestras. En este caso, para el C25, se aprecia un aumento en el tiempo de cálculo de 3,5ms que junto con el aumento del tiempo requerido por la rutina de tratamiento de interrupción, que recoge las muestras cada milisegundo, de 0,76us van a hacer que para la DCT la frecuencia de muestreo máxima sea aproximadamente 48kHz. Si fijamos la frecuencia de muestreo en 1kHz podemos disponer de 54 canales como máximo. Siguen siendo valores muy altos en ambos casos.

Los requisitos de memoria para la DCT lógicamente aumentan respecto a los de la FFT debido al almacenamiento de la tabla de 1024 cosenos, que ocupa un total de 2.048 palabras. En el caso de la implementación para el PC se utilizan 2.048 bytes más que se utilizan para acelerar el cálculo de la secuencia a introducir al bloque FFT, *w(n)*, a partir de la secuencia original, *x(n)*. Cuando se trabaja en tiempo real es necesario además de un buffer de entrada para 1024 datos que ocupa un total de 4.096 bytes.

Como ocurría con la FFT, en la comparativa de tiempos frente al PC, el C25 es más rápido que el 386/33Mhz y el 486/33Mhz, pero más lento que el 486/100Mhz y el Pentium 100Mhz, invirtiendo este último menos de la mitad de tiempo en el cómputo de la DCT.

4.2 RAZÓN DE COMPRESIÓN Y CALIDAD DE COMPRESIÓN (PRD).

4.2.1 AZTEC

FICHERO	TAMAÑO ORIGINAL (BYTES)	TAMAÑO COMPRIMIDO (BYTES)	RAZÓN	PRD (%)	PRD (%) (SMOOTH)
PAC111 *	30.720	13.922	2,2	8,89	11.35
PAC112	"	13.092	2,3	8,68	9.96
PAC121	"	11.556	2,7	12,83	13.56
PAC122	"	12.576	2,4	8,34	8.99
PAC211 *	30.720	14.904	2,1	12,07	17,24
PAC212 *	"	12.396	2,5	31,78	35,62
PAC311	30.720	13.116	2,3	17,32	24,86
PAC312 *	"	9.660	3,1	25,07	30,69
PAC321	"	14.676	2,1	19,01	25,86
PAC322	"	11.184	2,7	23,87	30,35

Para la realización de la tabla de valores presentada se ha utilizado la implementación del AZTEC del C25, aunque la implementación en C es equivalente y se obtienen los mismos resultados. Los ficheros marcados con un asterisco en la tabla se encuentran representados al final de este capítulo.

Como se vio en el capítulo 1, apartado 1.2.1.4, dos son los parámetros que permiten ajustar el funcionamiento del algoritmo AZTEC, *linelen*, que indica la máxima longitud que pueden tener las líneas horizontales generadas, y *Vth*, que especifica la apertura utilizada en el interpolador de orden cero. Para el primer parámetro, *linelen*, se ha tomado un valor típico para la frecuencia de muestreo a la que estamos trabajando de 50 muestras. Para el segundo, *Vth*, después de realizar un conjunto de pruebas se escogió un valor de aproximadamente el 0,3% de la amplitud de la señal, valor para el cual se obtienen buenas razones de compresión y PRD aceptables.

La razón de compresión media alcanzada por el algoritmo AZTEC es de 2.4:1, es decir, del 58% aproximadamente. Es una buena razón de compresión que se ve sin embargo contrarrestada por valores del PRD demasiado altos en algunas ocasiones, como ocurre con los ejemplos PAC212 y PAC312.

AZTEC no funciona bien, en general, cuando la señal de entrada presenta variaciones de amplitud bruscas, como ocurre en el caso de las señales citadas. Por el contrario, si la señal varia lentamente en amplitud, AZTEC se muestra como un buen algoritmo de compresión para ECG tanto por la razón de compresión alcanzada como por la calidad de la misma.

Observando la gráficas junto con los resultados de la tabla, vemos que si bien la aplicación del filtro suavizador cumple su función de suavizar la señal, introduce una distorsión significativa en la misma que hace aumentar notablemente el PRD. En general la aplicación de un filtro suavizador a la salida del AZTEC se hace para su presentación al cardiólogo en monitores o impreso.

De cualquier forma, el AZTEC fue pensado originalmente para poder tomar medidas del ritmo cardiaco, lo cual es posible hacer perfectamente tanto trabajando directamente con la señal descomprimida como haciéndolo con la señal pasada por el filtro suavizador.

4.2.2. FAN

FICHERO	TAMAÑO ORIGINAL (BYTES)	TAMAÑO COMPRIMIDO (BYTES)	RAZÓN	PRD (%)
PAC111	30.720	9.546	3,2	12,00
PAC112 *	"	8.226	3,7	11,59
PAC121 *	"	13.938	2,2	5,65
PAC122	"	10.806	2,8	8,85
PAC211	30.720	12234	2,5	12,54
PAC212	"	11514	2,7	6,70
PAC311 *	30.720	11.130	2,8	10,95
PAC312	"	9.510	3,2	15,46
PAC321	"	14.658	2,1	10,62
PAC322 *	"	11.502	2,7	9,80

Se ajusto el parámetro de tolerancia (*Vth*) del algoritmo FAN tras realizar las oportunas pruebas donde se contrastaban las razones de compresión frente al PRD obtenidas. Finalmente se tomó un valor de 0,3% la amplitud máxima de la señal de entrada.

En esta ocasión se obtuvo una razón de compresión media de 2.8:1, ó 64%. Es una razón de compresión mayor que la obtenida en el algoritmo AZTEC que además se ve reforzada por la obtención de PRD menores en general, y sobretodo menos dependientes de la forma de la señal de entrada como ocurría con ACTEC.

FAN es, de los algoritmos de compresión directa, el que mejores resultados obtiene en cuanto a la razón de compresión y PRD, siendo posible además, como ya vimos en el apartado 4.1., su funcionamiento en tiempo real como parte de un sistema de procesado completo y con varios canales.

En la tabla se marcan con un asterisco las señales que han sido representadas al final del capítulo.

4.2.3. DPCM

FICHERO	RAZÓN DE COMPRESIÓN	ERRORES COMETIDOS	PRD (%)
PAC111	1,5	0	0,00
PAC112	"	0	0,00
PAC121	"	0	0,00
PAC122	"	0	0,00
PAC211	1,5	0	0,00
PAC212	"	4	0,55
PAC311	1,5	0	0,00
PAC312	"	0	0,00
PAC321	"	0	0,00
PAC322	"	0	0,00

FICHERO	RAZÓN DE COMPRESIÓN	ERRORES COMETIDOS	PRD (%)
PAC111	2	29	8,08
PAC112	"	8	0,19
PAC121	"	42	6,56
PAC122	"	2	0,09
PAC211	2	37	1,37
PAC212	"	1.372	182,01
PAC311	2	669	114,76
PAC312	"	643	135,91
PAC321	"	925	131,24
PAC322	"	883	146,29

Se presentan dos tablas de datos, la primera de ellas obtenida utilizando un cuantificador de 8 bits a la salida, y la segunda mediante un cuantificador de 6 bits. En las tablas la columna de *errores cometidos* se refiere a las veces en las que el cuantificador es sobrepasado por la diferencia entre el valor predicho y la muestra actual.

En cuanto a los valores del PRD, como vemos en la primera tabla la señal reconstruida es la señal original, no hay pérdidas para el caso excepto para la señal PAC212, y estas son mínimas. La señal PAC212 presenta variaciones bruscas en amplitud, con lo que en algunas ocasiones supera el rango del cuantificador de 8 bits. La señal PAC212 se encuentra representada en las gráficas para el algoritmo AZTEC, al final del capítulo.

En la segunda tabla se ha reducido el número de bits a 6. Con ello se alcanza mayor compresión, pero los resultados obtenidos no son satisfactorios en la mayoría de los casos, llegándose a superar el 100% para el valor del PRD en el 50% de las pruebas. A la vista de las gráficas se observa que, como era de esperar, cuando se disminuye la resolución del cuantificador la DPCM falla en las señales donde tenemos variaciones más acusadas de la amplitud de la señal.

4.2.4. FFT

Resultados obtenidos con el C25, coma fija formato Q15			
FICHERO	**PRD para RC = 5:1**	**PRD para RC = 2,5:1**	**PRD para RC = 1:1**
PAC111	13,59	3,50	0,57
PAC112	10,52	3,02	0,66
PAC121 *	6,26	1,85	0,59
PAC122 *	9,03	4,94	0,63
PAC211 *	23,13	3,87	0,40
PAC212	25,98	2,12	0,72
PAC311	29,38	4,11	0,68
PAC312	29,92	3,67	0,55
PAC321 *	22,11	3,56	0,63
PAC322	26,52	3,71	0,58

Resultados obtenidos en coma flotante			
FICHERO	**PRD para RC = 5:1**	**PRD para RC = 2,5:1**	**PRD para RC = 1:1**
PAC111	13,54	3,44	0,00
PAC112	10,51	2,97	0,00
PAC121	6,27	1,81	0,00
PAC122	7,70	2,43	0,00
PAC211	23,10	3,86	0,00
PAC212	25,96	1,99	0,00
PAC311	29,69	4,05	0,00
PAC312	29,91	3,63	0,00
PAC321	22,04	3,48	0,00
PAC322	26,47	3,66	0,00

Se presentan dos tablas de datos para la FFT, la primera obtenida con el C25 trabajando en coma fija, formato Q15, y la segunda con *Matlab* trabajando en coma flotante. Esto permite tener una idea de cuales son las pérdidas por el hecho de trabajar en coma fija para el cálculo de un número de sumas y productos muy alta como resulta de la implementación de la FFT.

La razón de compresión es ahora igual para todos las señales a las que se aplica este método de compresión, cambiando solamente el PRD obtenido para cada una de ellas. Como se observa en las tablas de datos, la forma de la señal de entrada influye notablemente en los resultados obtenidos, lo cual dificulta la elección del número de muestras a eliminar en términos generales.

Para una razón de compresión de 5:1 (80%), donde nos quedamos con los primeros 100 armónicos del espectro de 1.024, se obtienen PRD aceptables en todos los casos, si bien son algo más elevados que los obtenidos con el algoritmo FAN. Como se aprecia comparando las tablas de resultados en coma fija y coma flotante, el error introducido por tener coma fija es prácticamente despreciable.

Con una razón de compresión de 2,5:1 (60%), donde nos quedamos con los primeros 200 armónicos del espectro de 1.024, obtenemos valores del PRD muy buenos. A la vista de las gráficas del final del capítulo, donde se muestran las señales originales y reconstruidas, observamos que no hay una diferencia

apreciable. Como ocurría antes, el error introducido por trabajar en coma fija es también mínimo.

Por último se presentan en las tablas el PRD obtenido cuando no se realiza compresión alguna, es decir, el espectro se almacena o transmite en su totalidad. En este caso podemos ver que el hecho de trabajar en coma fija introduce un error que es detectado por el PRD, muy pequeño, siempre menor del 1%, que podemos considerar a efectos prácticos despreciable. Trabajando en coma flotante el error cometido no es apreciable.

Al final del capítulo están representadas las señales originales y reconstruidas de los resultados trabajando en coma fija, marcadas en la tabla de datos con un asterisco. Se puede apreciar como para las señales comprimidas con una razón de 5:1 las componentes frecuenciales de alta frecuencia desaparecen. Con una razón de compresión menor de 2,5:1 la señal reconstruida es prácticamente idéntica a la señal original. En las representaciones de las señales PAC122 y PAC211 se pueden ver las discontinuidades que aparecen en las fronteras de los bloques de 1.024 muestras, motivadas por el truncamiento de la transformada de Fourier al realizar la compresión y por la diferencia en el valor medio de los grupos de muestras.

4.2.5. DCT

Resultados obtenidos con el C25, coma fija formato Q15			
FICHERO	**PRD para RC = 5:1**	**PRD para RC = 2,5:1**	**PRD para RC = 1:1**
PAC111	22,84	21,89	21,84
PAC112	17,36	16,39	16,44
PAC121	17,36	17,07	17,13
PAC122	15,85	15,63	15,72
PAC211	33,85	29,71	29,73
PAC212	43,13	37,93	37,93
PAC311	55,02	50,04	49,81
PAC312	50,27	45,01	44,85
PAC321	49,78	46,38	46,30
PAC322	51,32	47,44	47,45

Resultados obtenidos en coma flotante			
FICHERO	**PRD para RC = 5:1**	**PRD para RC = 2,5:1**	**PRD para RC = 1:1**
PAC111	13,56	3,41	0,00
PAC112	10,41	2,92	0,00
PAC121	6,14	1,75	0,00
PAC122	7,71	2,38	0,00
PAC211	23,09	3,85	0,00
PAC212	26,09	1,98	0,00
PAC311	28,75	4,02	0,00
PAC312	29,73	3,61	0,00
PAC321	22,15	3,46	0,00
PAC322	26,41	3,63	0,00

Como en el caso de la FFT también se presentan dos tablas de datos en coma fija y coma flotante, construidas a partir de los resultados obtenidos a partir del C25 trabajando en formato Q15 y el programa *Matlab* respectivamente. En esta ocasión si que vamos a poder apreciar una diferencia importante entre los resultados obtenidos entre las pruebas de coma fija y las de coma flotante.

Para una compresión de 5:1 ó 80%, donde nos quedamos con los primeros 200 términos de la DCT de 1.024 puntos, el PRD obtenido en coma fija es, en la mayoría de casos, excesivamente alto. En comparación con la FFT para la misma razón de compresión los resultados de la DCT son peores. Por el contrario, en coma flotante los resultados del PRD son aceptables, y prácticamente iguales a los obtenidos con la FFT en coma fija y en coma flotante. Vemos como el gran número de operaciones de suma, producto y división a realizar en el cómputo de la DCT en coma fija introduce un error muy importante.

En el caso de obtener una razón de compresión en coma fija de 2,5:1 ó 60%, quedándonos con los primeros 400 términos de la DCT de 1.024 puntos, los PRD obtenidos siguen siendo demasiado altos, aunque se nota una pequeña mejoría respecto al caso anterior de 5:1. Para la misma compresión trabajando en coma flotante se aprecia un notable descenso en el PRD, equiparándose de nuevo a los resultados obtenidos con la FFT.

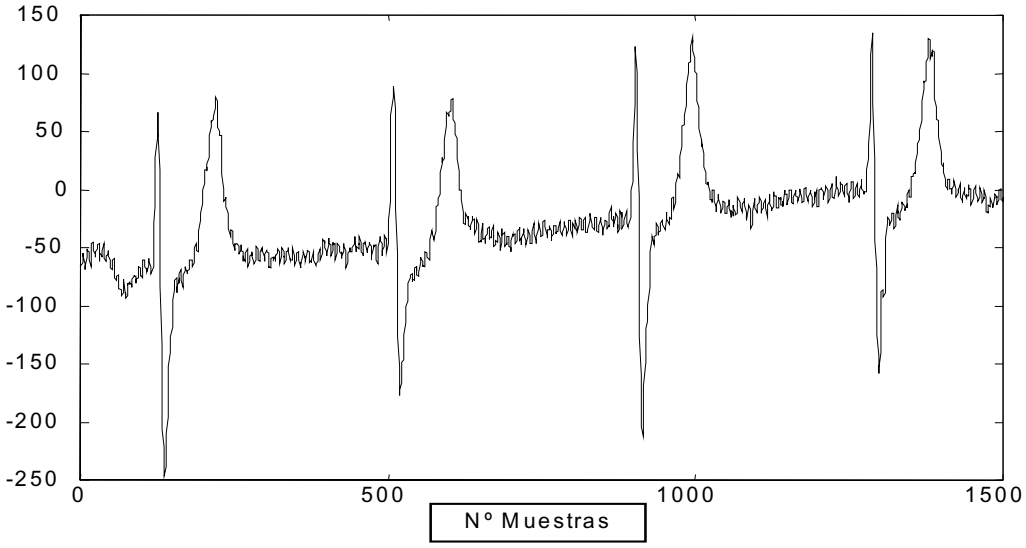
Por último, si no se realiza compresión y se transmite o almacena toda la DCT, vemos como en coma fija el simple hecho de realizar la transformada y después la transformada inversa introduce un error de precisión en los cálculos que da lugar a un PRD prácticamente igual al obtenido para el caso de comprimir a una razón de 2,5:1, incluso superior en algunas ocasiones. Se nota un ligero descenso del PRD respecto al caso de comprimir con una razón de 5:1.

Si trabajamos en coma fija sin realizar compresión no existe error apreciable en la recuperación de la señal.

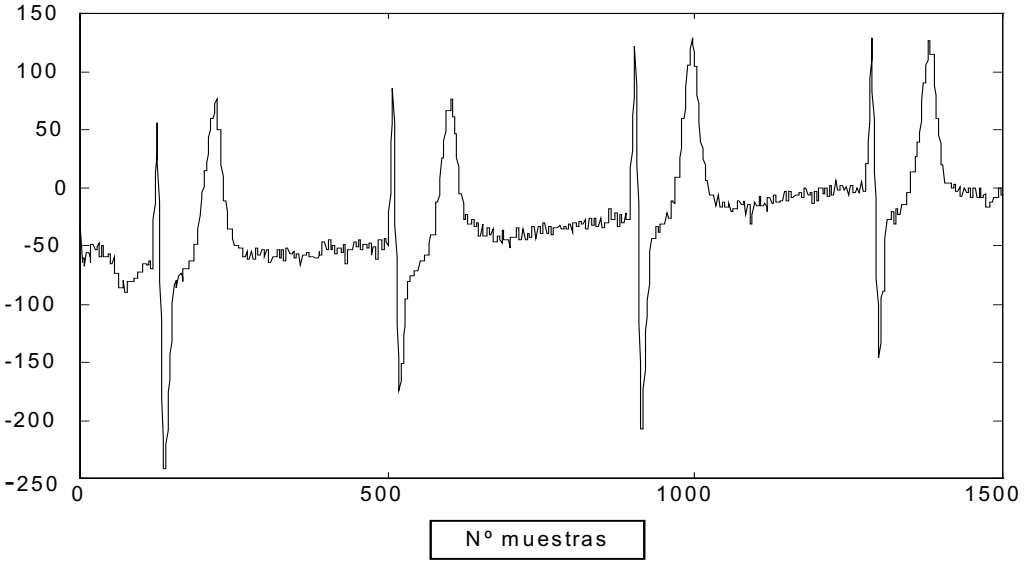
Al final del capítulo se hallan representadas gráficamente las señales marcadas con un asterisco en la tabla de resultados de coma fija. Se observa como efectivamente las señales reconstruidas a partir de la DCT y las señales originales presentan una mayor diferencia entre ellas que en el caso de la FFT. También es posible apreciar que la diferencia entre la representación de las señales comprimidas a 5:1 y las comprimidas a 2,5:1 es notable, pese a que los PRD obtenidos en los casos de compresión de 5:1 y 2,5:1 están muy próximos. En los ficheros PAC112 y PAC212 es donde mejor se puede observar las discontinuidades en las fronteras de los bloques de 1024 elementos, como consecuencia de las diferencias de valor medio entre dichos bloques, y de truncar la DCT para realizar la compresión.

REPRESENTACIONES GRÁFICAS

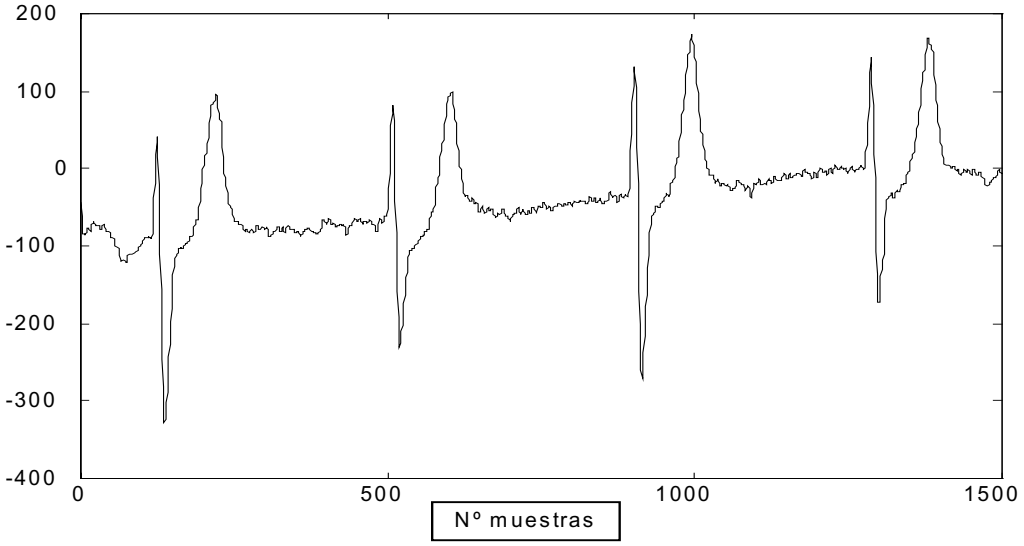
PAC111, Señal original, Fs = 250Hz. 1500 de 5120 muestras representadas.



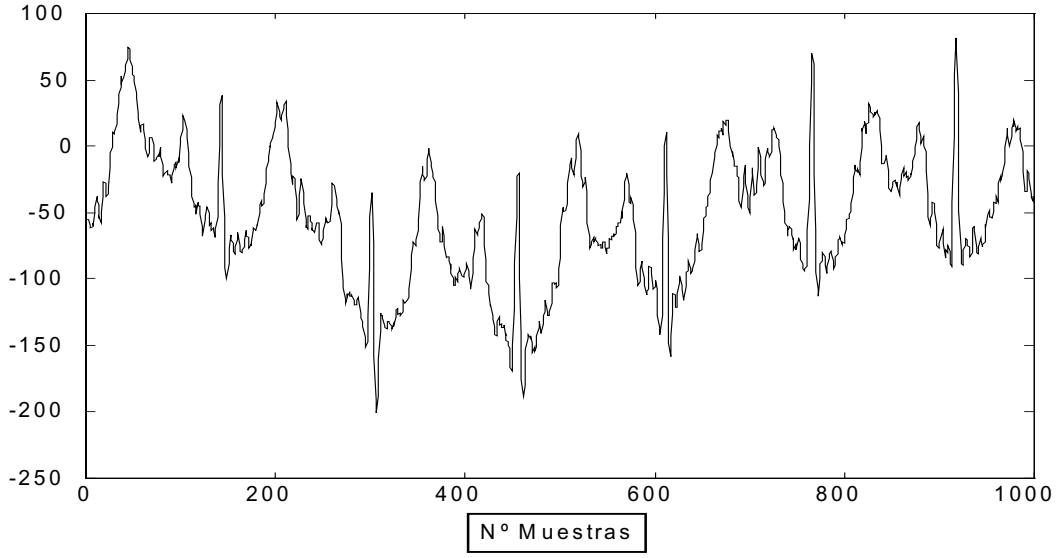
Compresión AZTEC, CR = 2,2:1, PRD = 8,7%. Señal reconstruida.



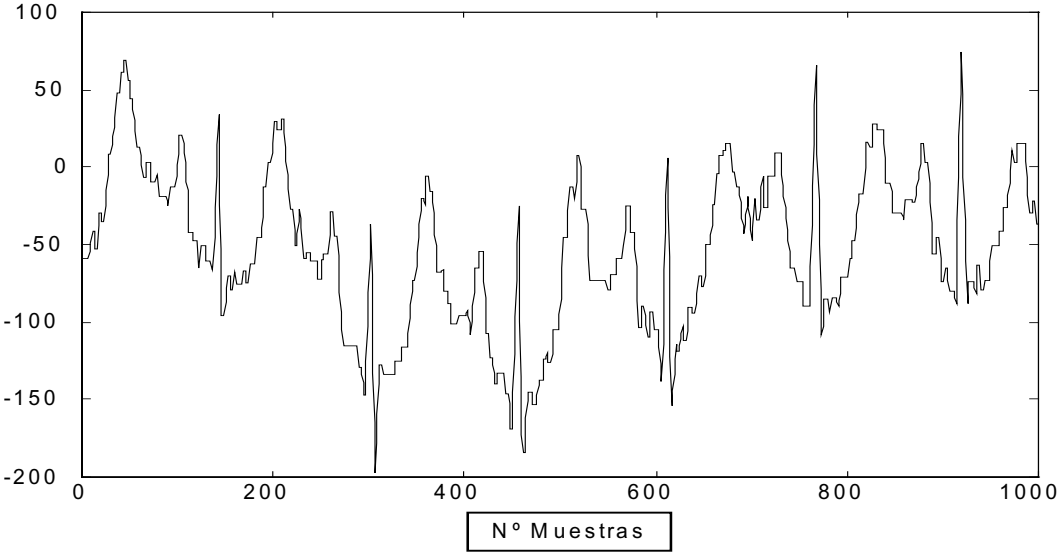
Compresión AZTEC, CR = 2,2:1, PRD = 37,4%, Señal reconstruida y filtrada.



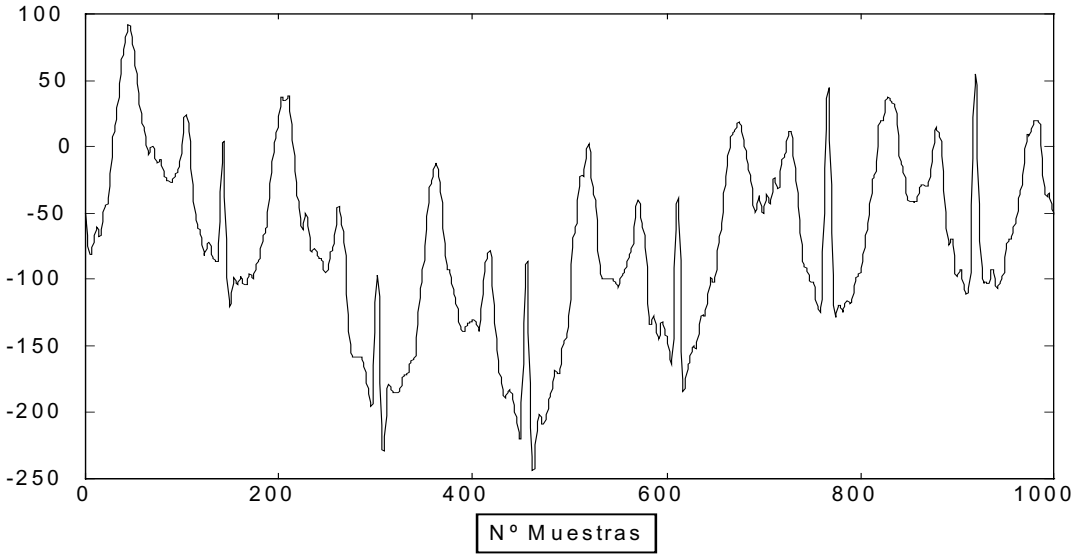
PAC211, Señal original, Fs = 250Hz. 1000 de 5120 muestras representadas.



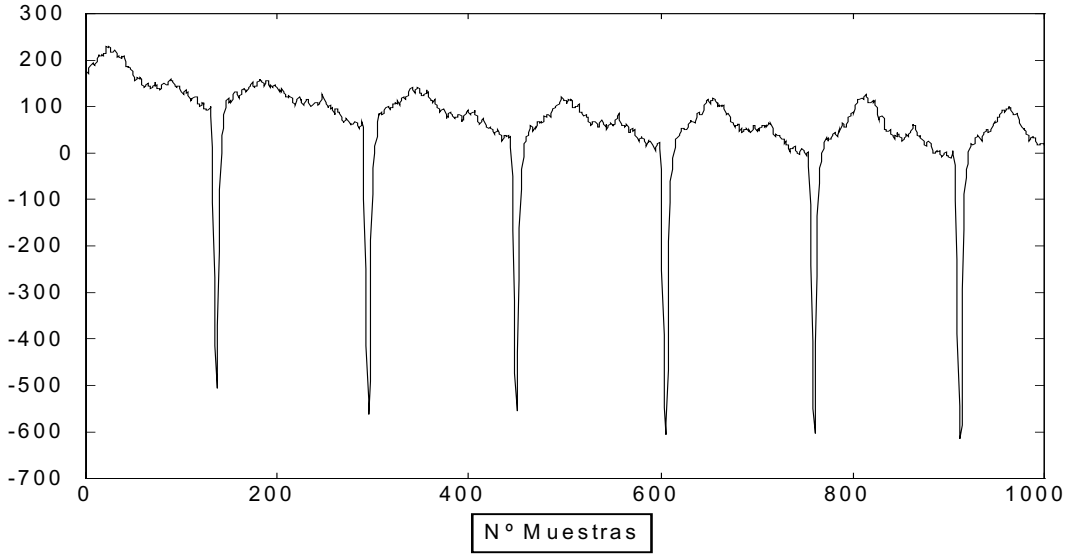
Compresión AZTEC, CR = 2,1:1, PRD = 8,26%. Señal reconstruida



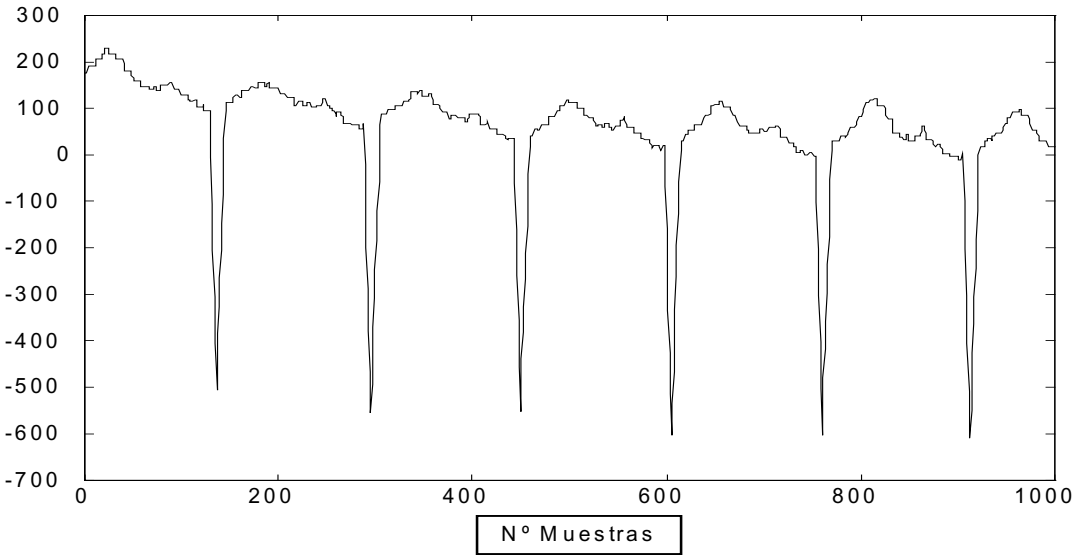
Compresión AZTEC, CR = 2,1:1, PRD = 37%. Señal reconstruida y filtrada.



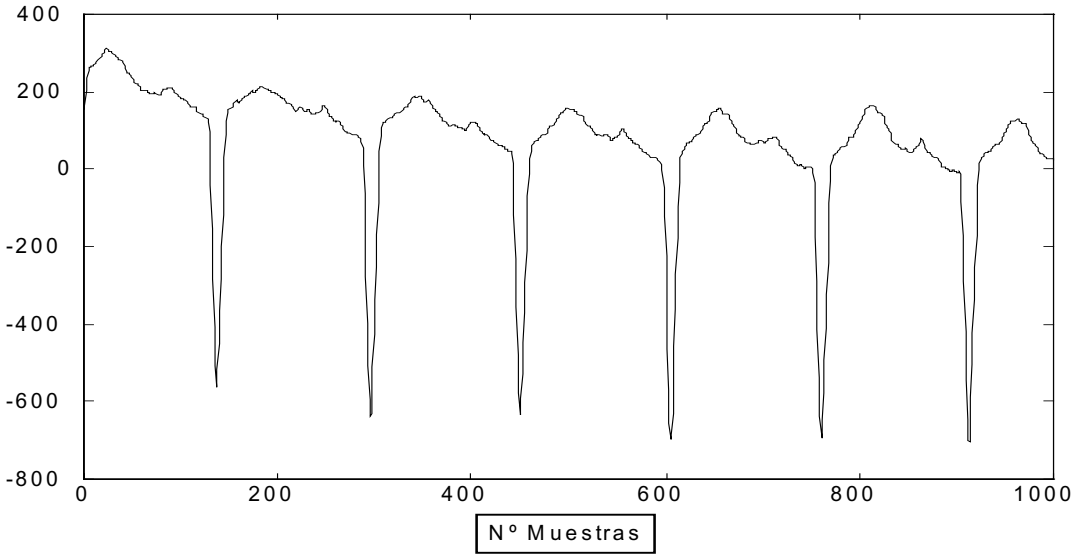
PAC212, Señal original, Fs = 250Hz. 1000 de 5120 muestras representadas.



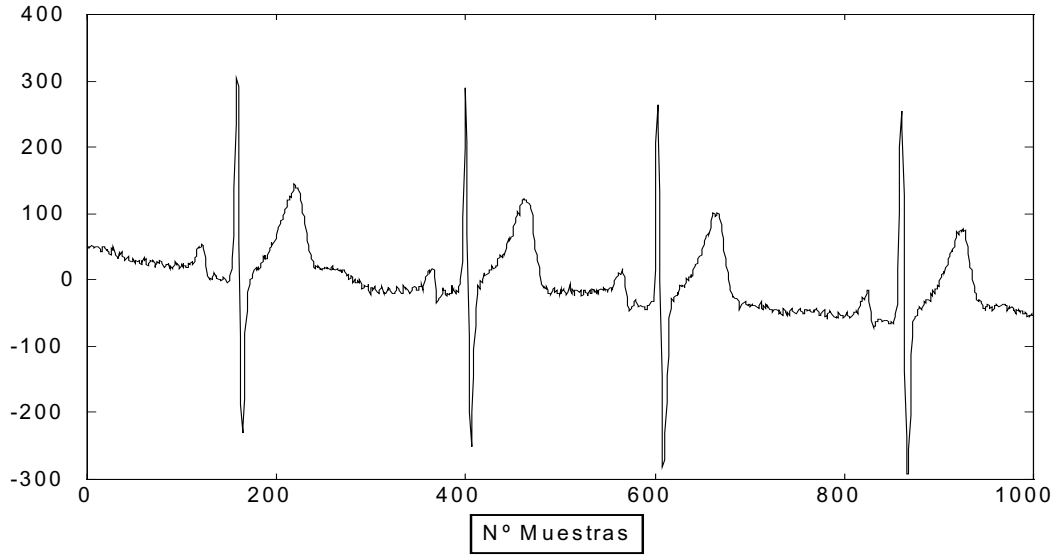
Compresión AZTEC, CR = 2,5:1, PRD =36,8%. Señal reconstruida



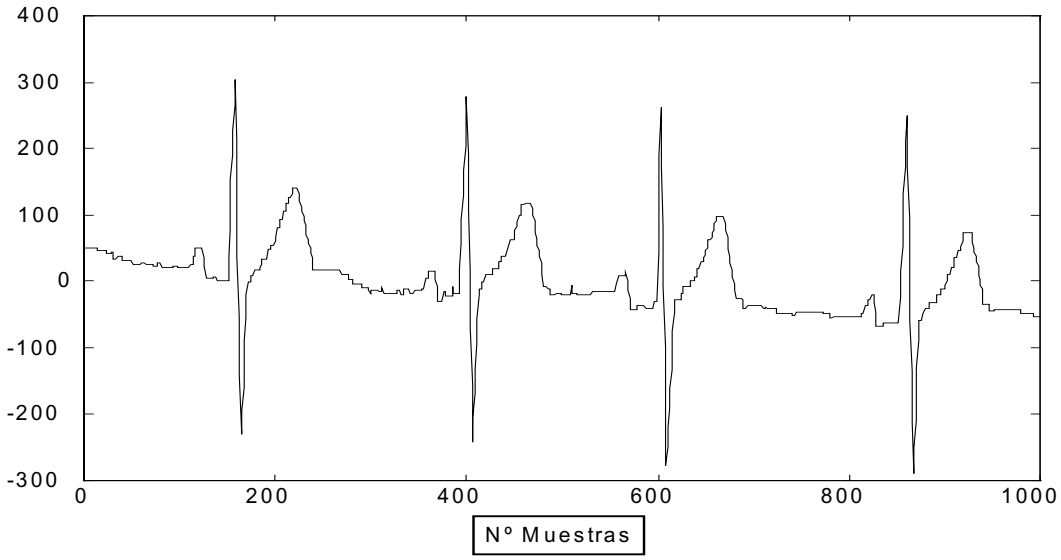
Compresión AZTEC, CR = 2,5:1, PRD = 67,3%. Señal reconstruida y filtrada.



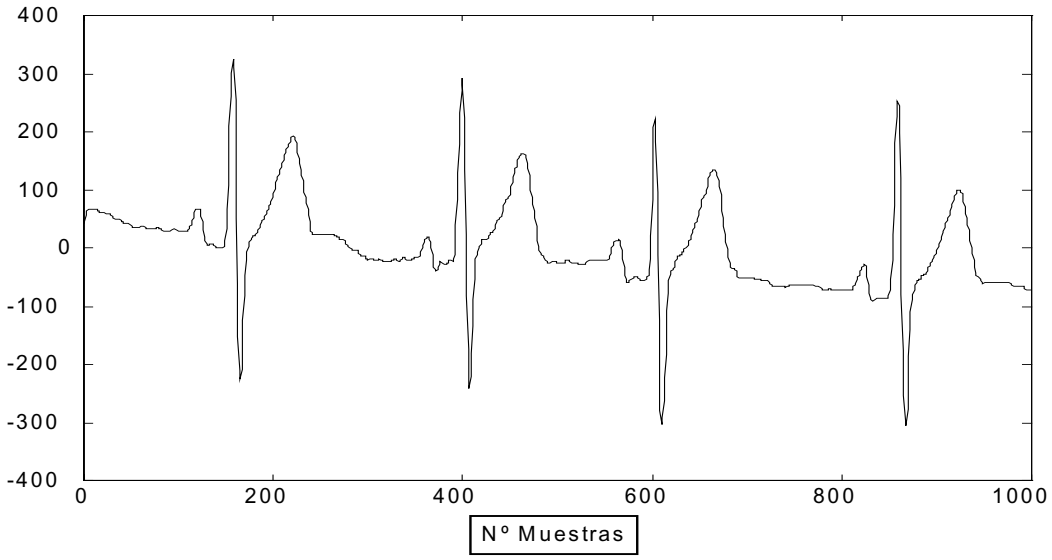
PAC312, Señal original, Fs = 250Hz. 1000 de 5120 muestras representadas.



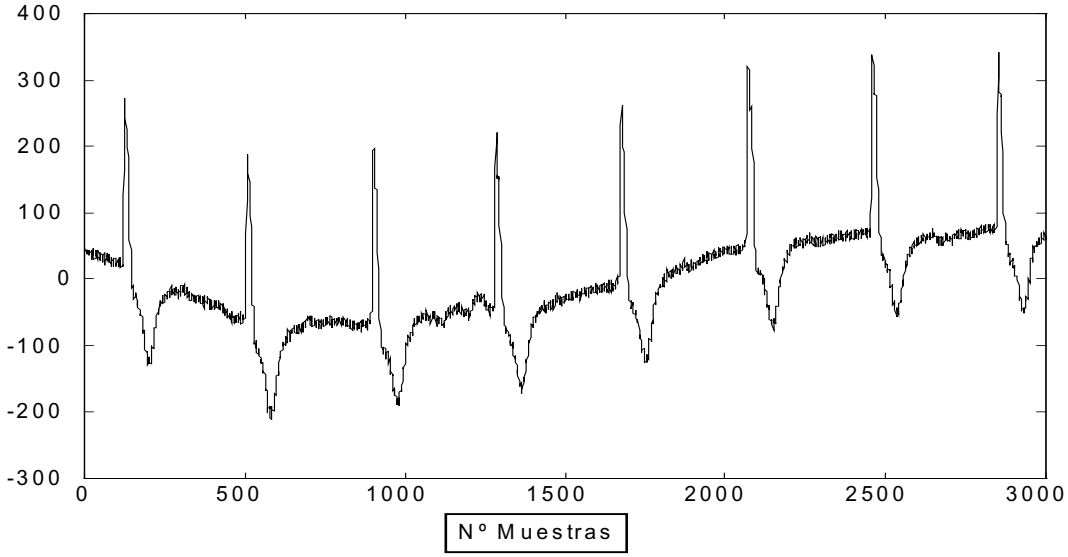
Compresión AZTEC, CR = 3,1:1, PRD =23,9%. Señal reconstruida



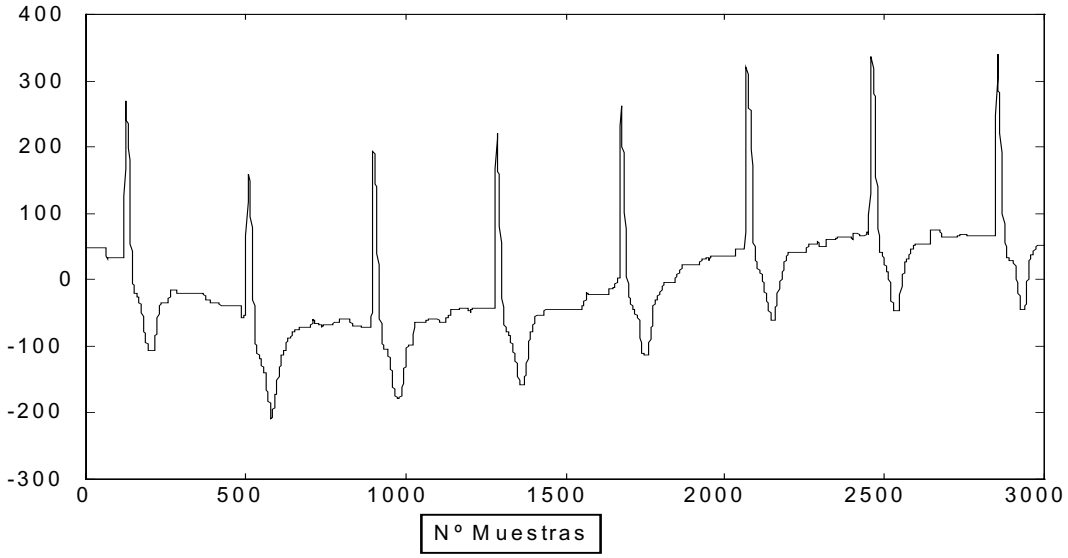
Compresión AZTEC, CR = 3,1:1, PRD = 41,9%. Señal reconstruida y filtrada.



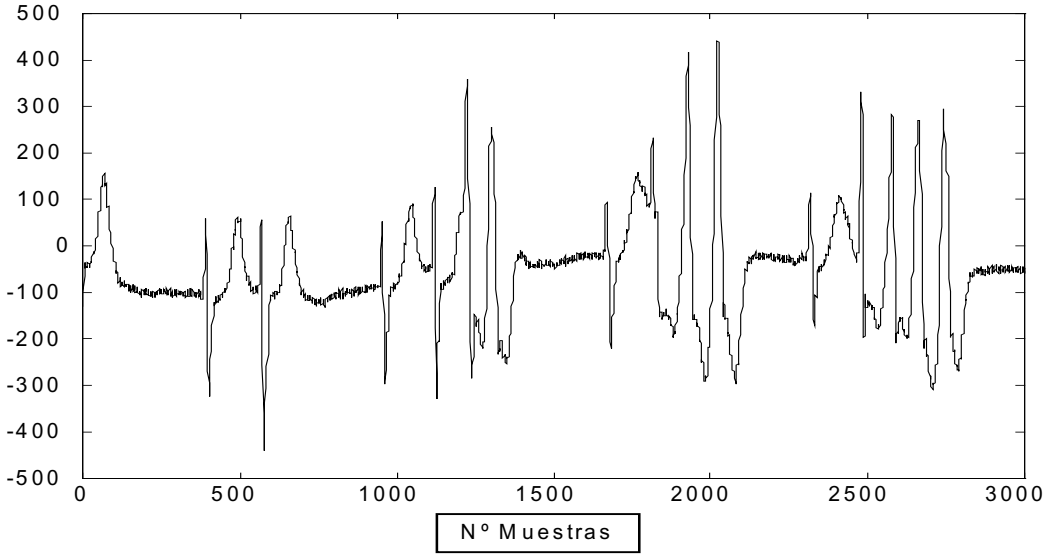
PAC112, Señal original, Fs = 250Hz. 3000 de 5120 muestras representadas.



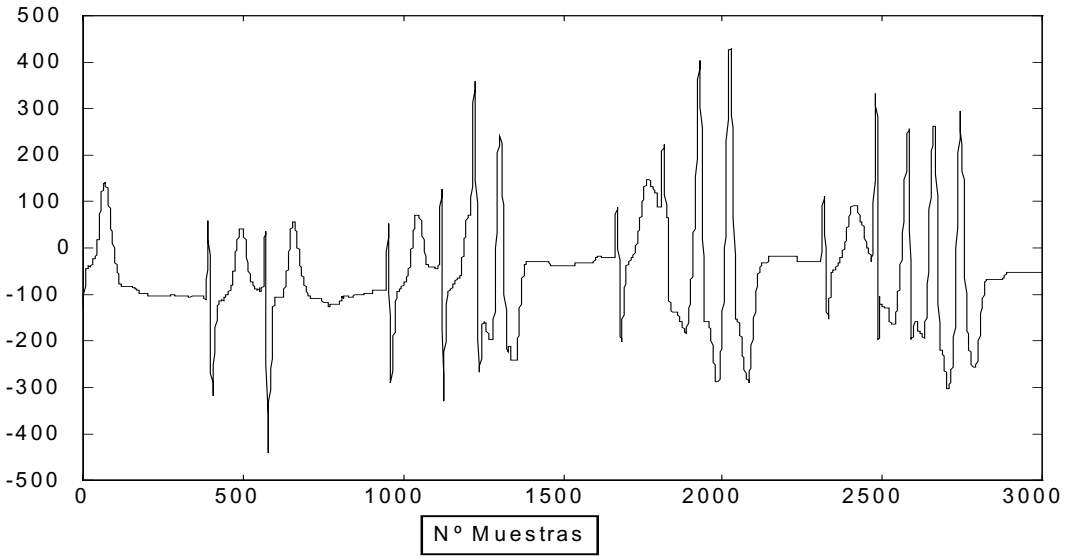
Compresión FAN, CR = 3,7:1, PRD =11,6%. Señal reconstruida



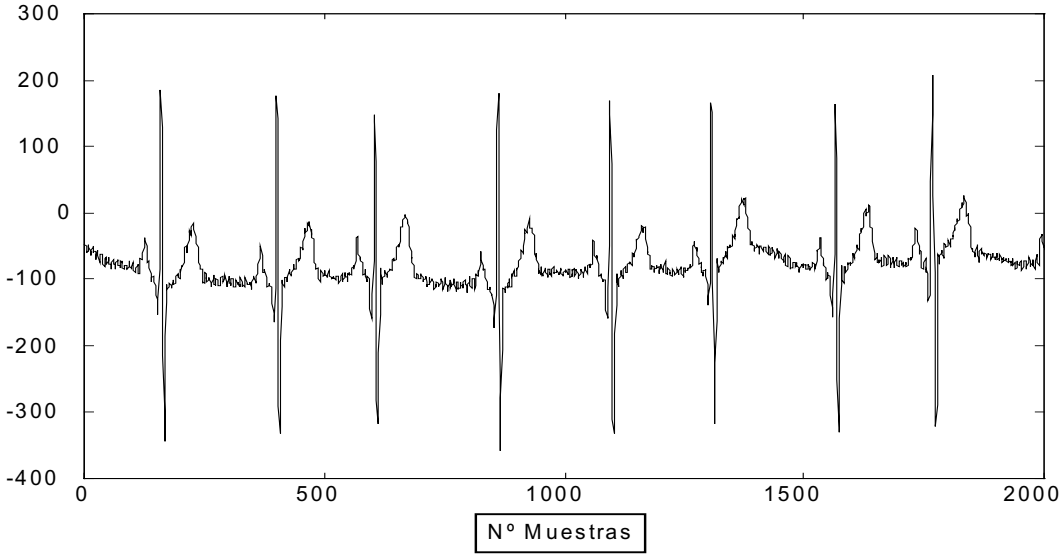
PAC121 Señal original, Fs = 250Hz. 3000 de 5120 muestras representadas.



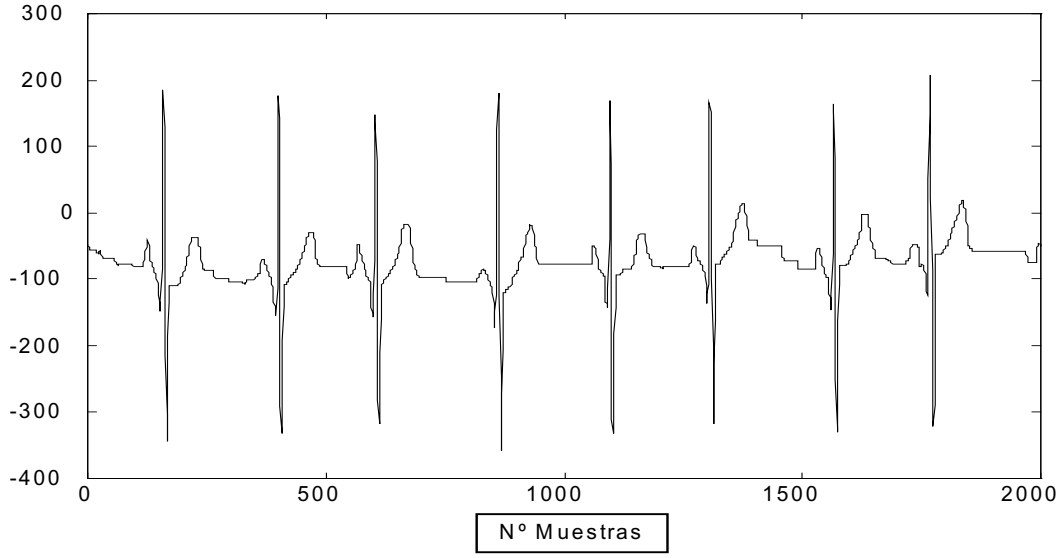
Compresión FAN, CR = 2,2 PRD =5,7%. Señal reconstruida



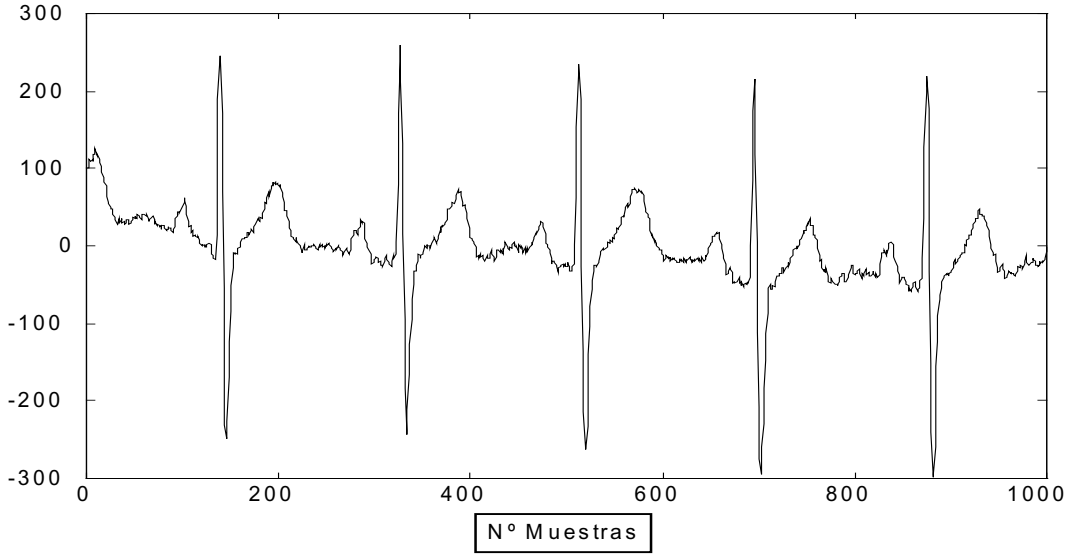
PAC311 Señal original, Fs = 250Hz. 2000 de 5120 muestras representadas.



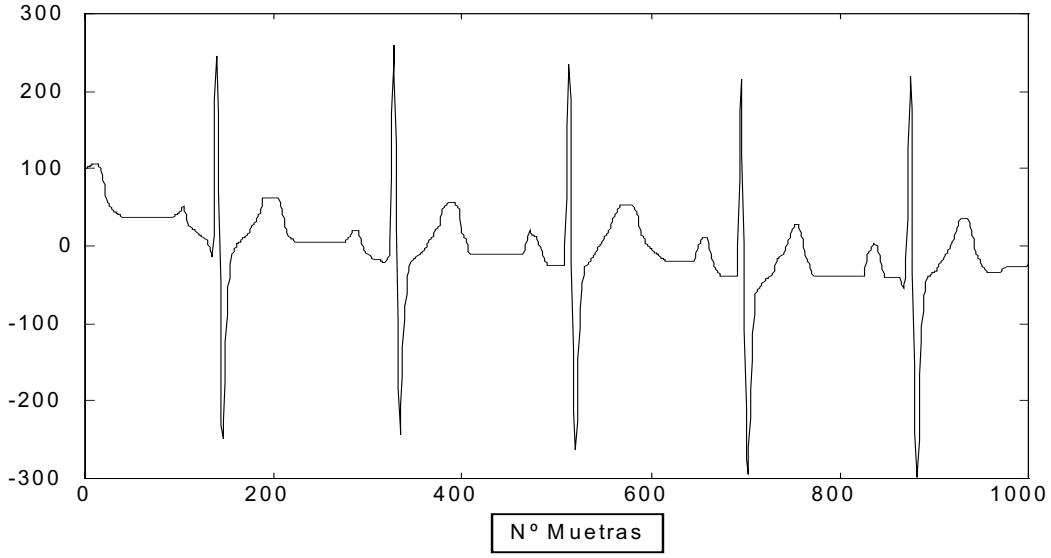
Compresión FAN, CR = 2,8:1 PRD = 11%. Señal reconstruida



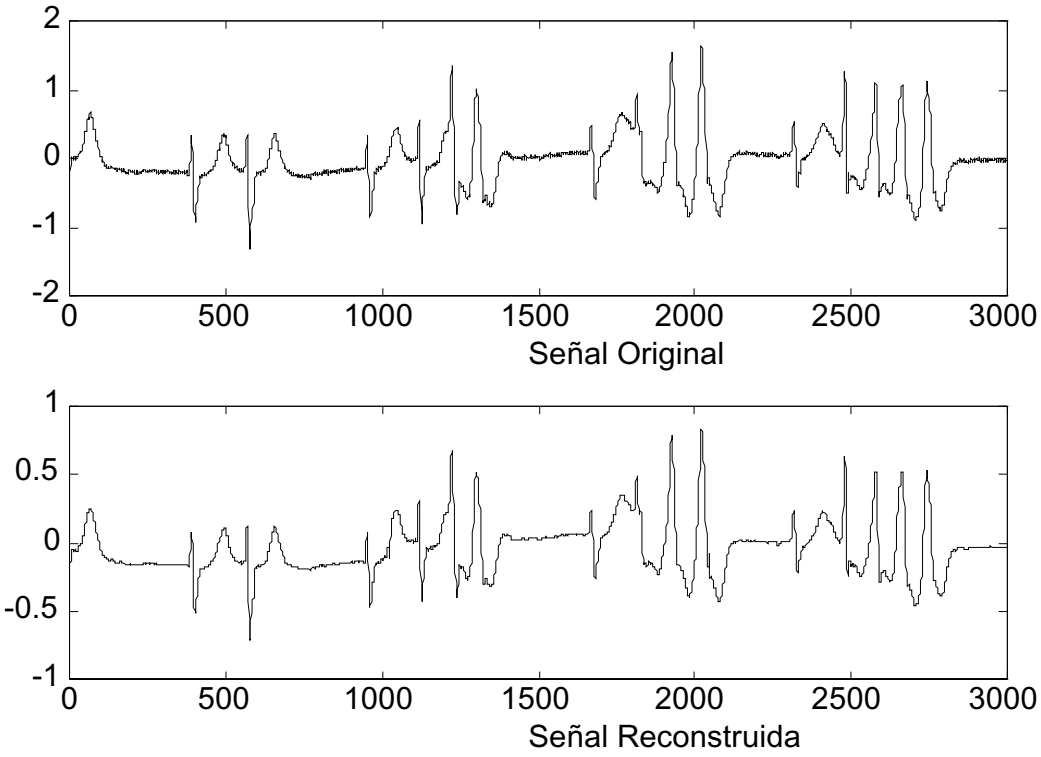
PAC322 Señal original, Fs = 250Hz. 1000 de 5120 muestras representadas.



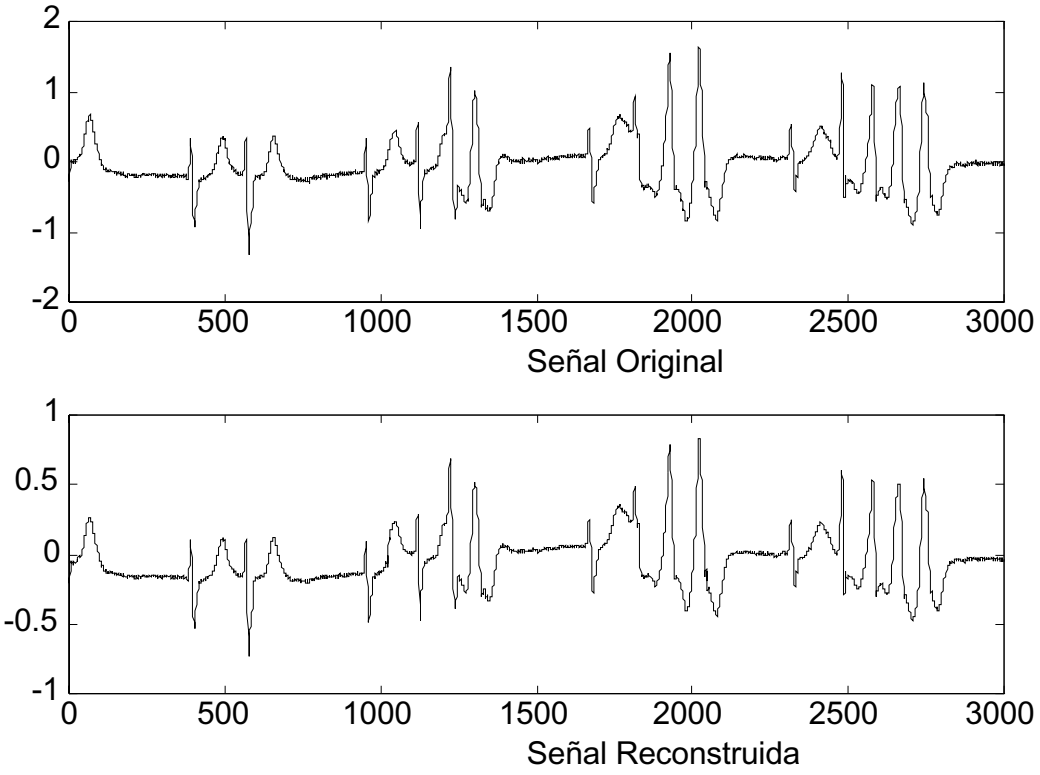
Compresión FAN, CR = 2,7:1 PRD = 9,8%. Señal reconstruida



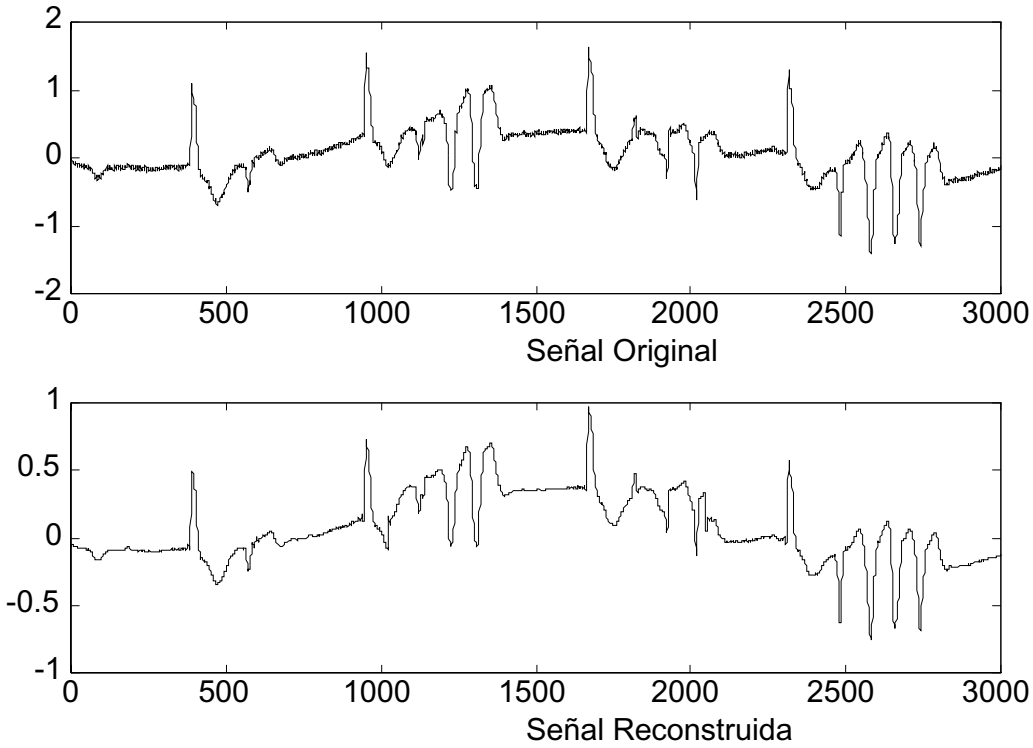
Compresión FFT 5:1. PAC121: 5.120 muestras, Fs = 250Hz.



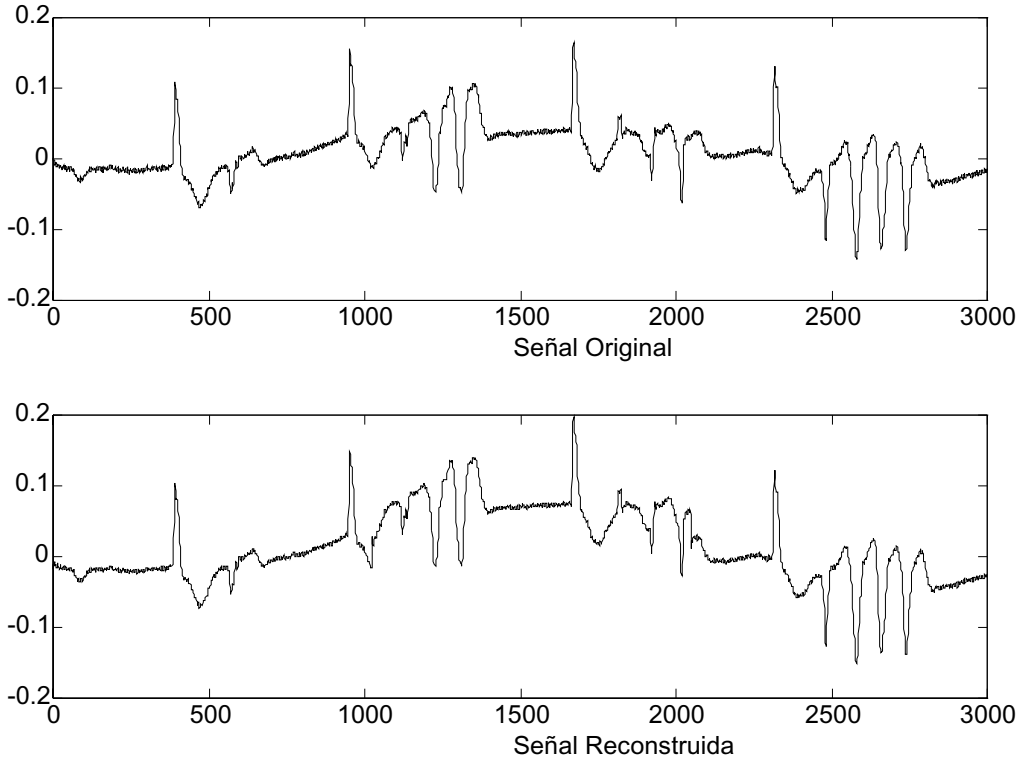
Compresión FFT 2,5:1 PAC121: 5.120 muestras, Fs = 250Hz.



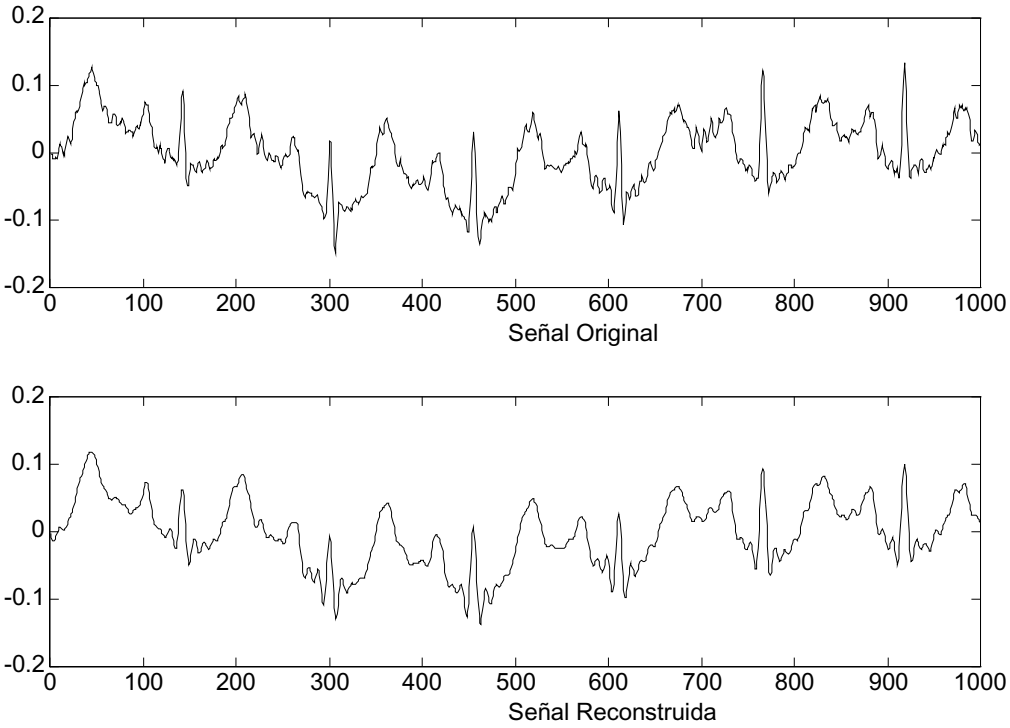
Compresión FFT 5:1 PAC122: 5.120 muestras, Fs = 250Hz.



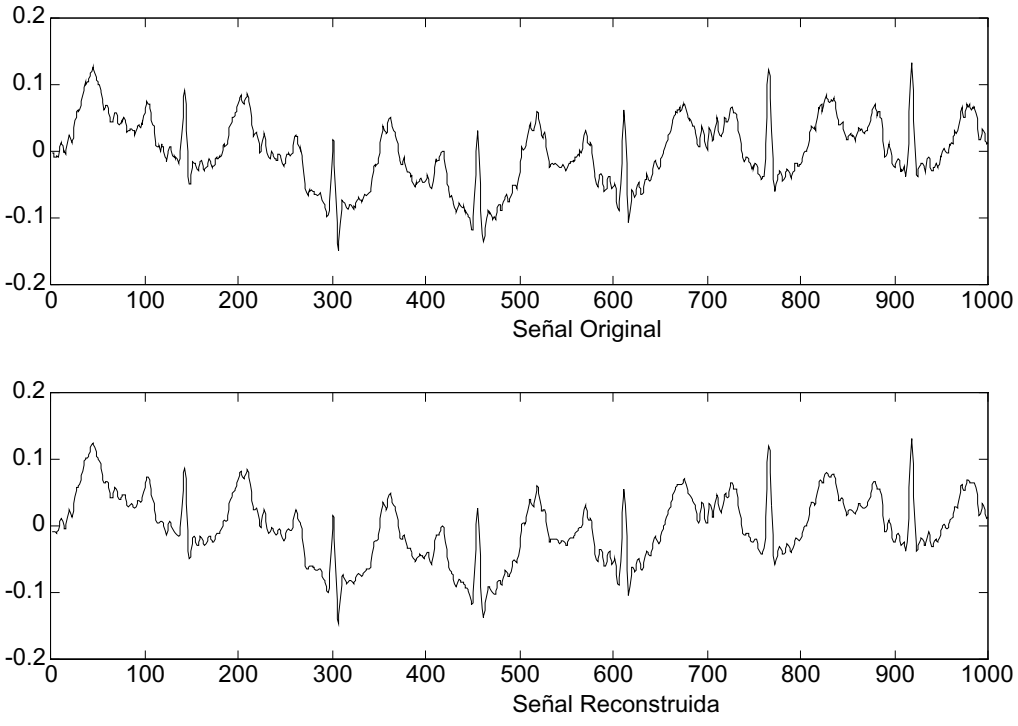
Compresión FFT 2,5:1 PAC122: 5.120 muestras, Fs = 250Hz.



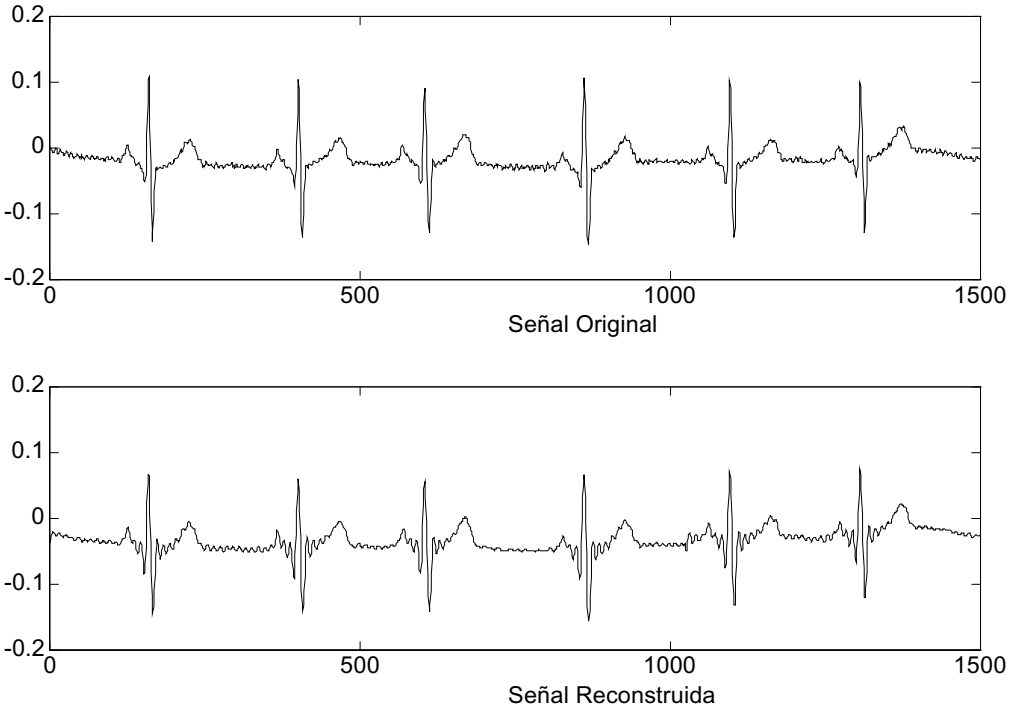
Compresión FFT 5:1 PAC211: 5.120 muestras, Fs = 250Hz.



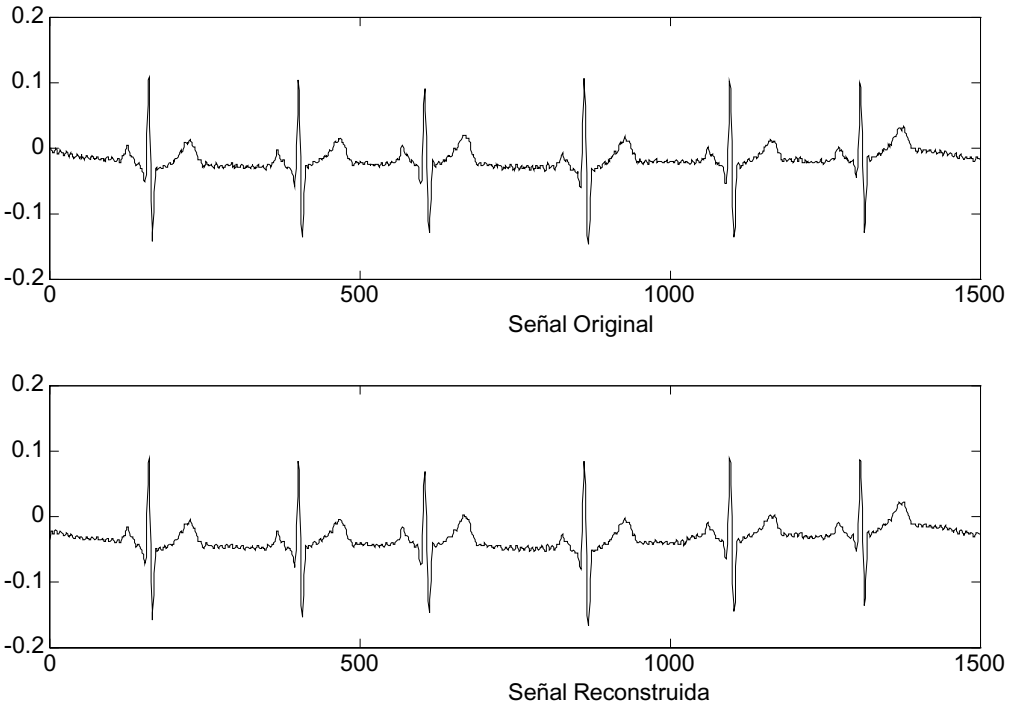
Compresión FFT 2,5:1 PAC211: 5.120 muestras, Fs = 250Hz.



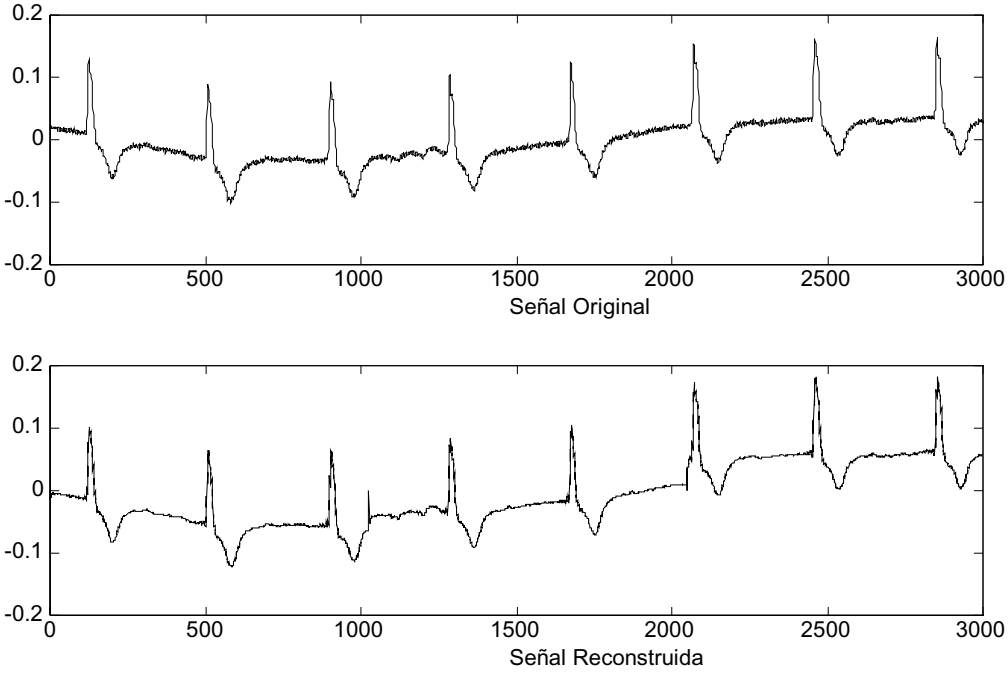
Compresión FFT 5:1 PAC311: 5.120 muestras, Fs = 250Hz.



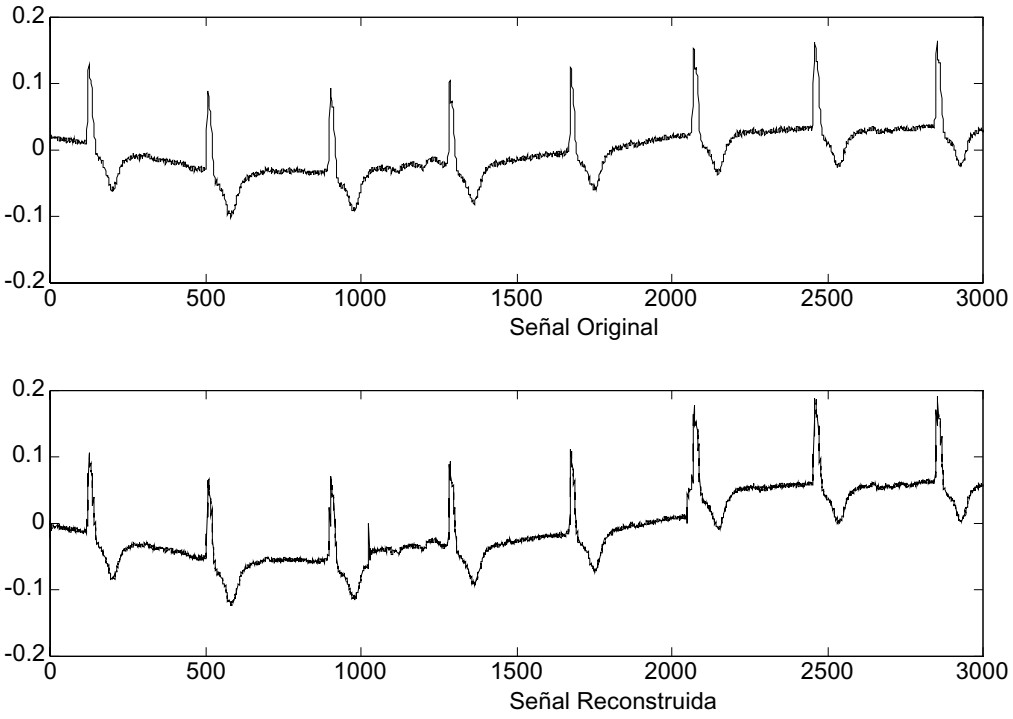
Compresión FFT 2,5:1 PAC311: 5.120 muestras, Fs = 250Hz.



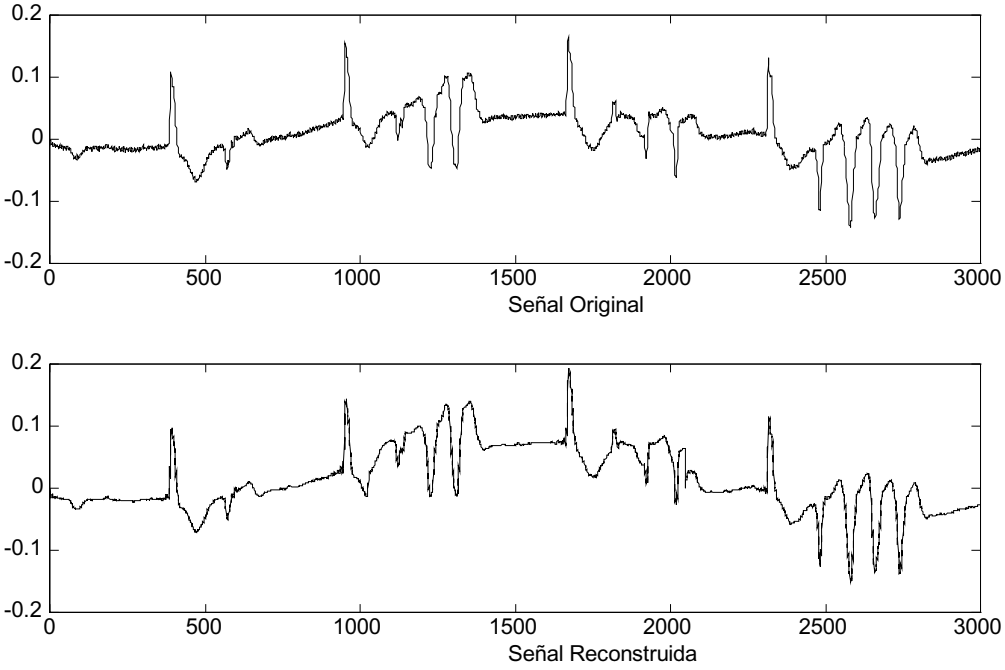
Compresión DCT 5:1 PAC112: 5.120 muestras, Fs = 250Hz.



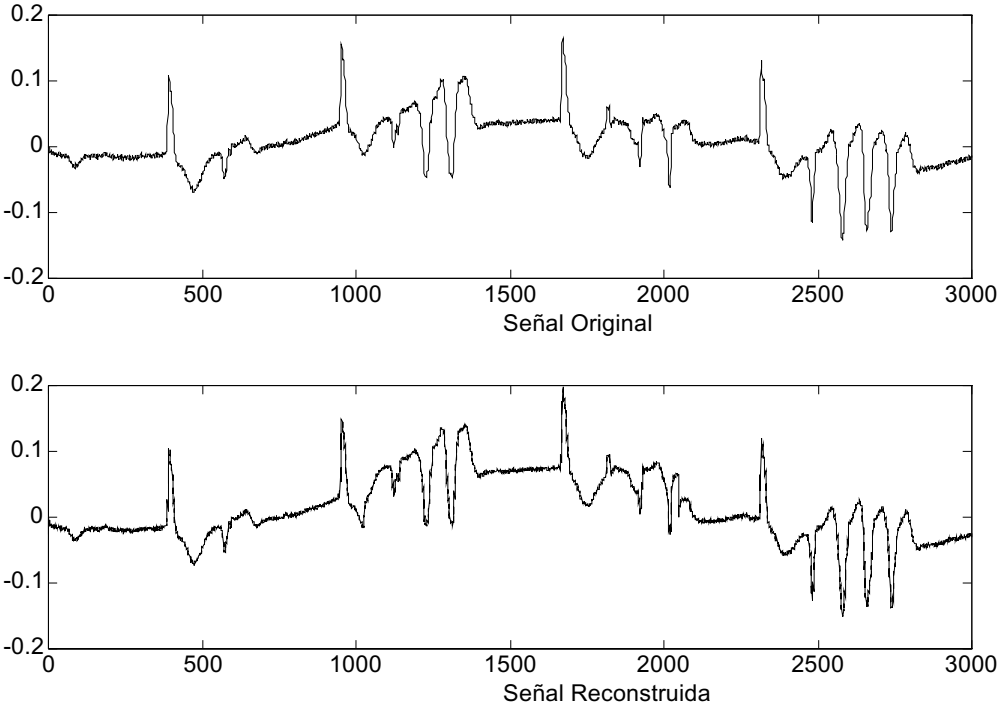
Compresión DCT 2,5:1 PAC112: 5.120 muestras, Fs = 250Hz.



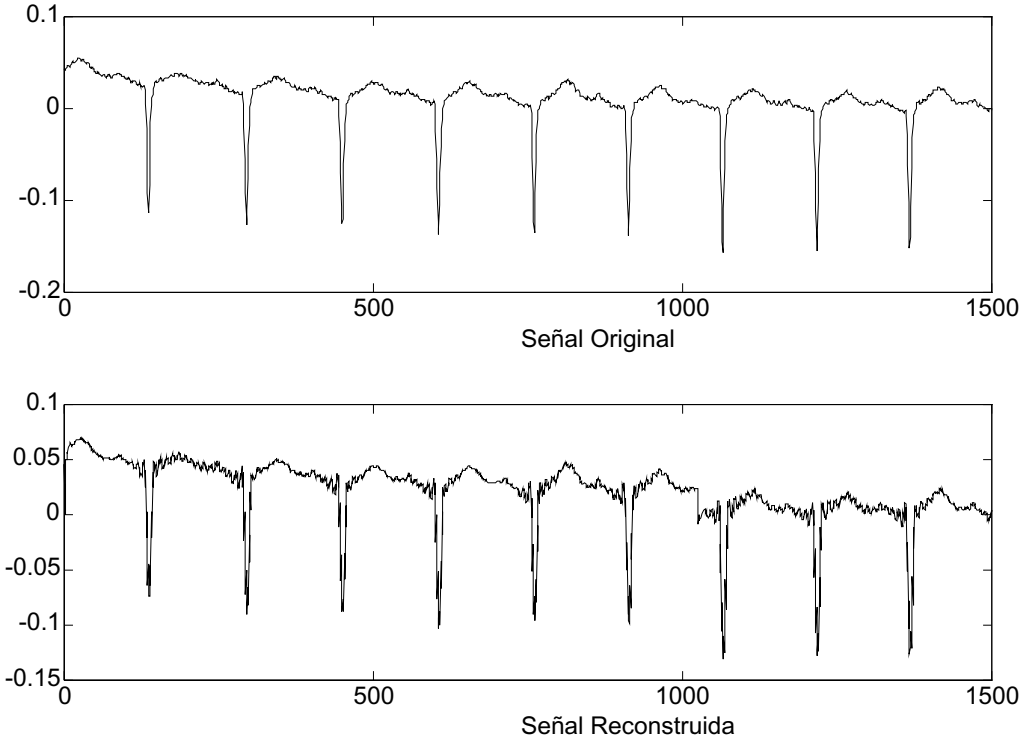
Compresión DCT 5:1 PAC122: 5.120 muestras, Fs = 250Hz.



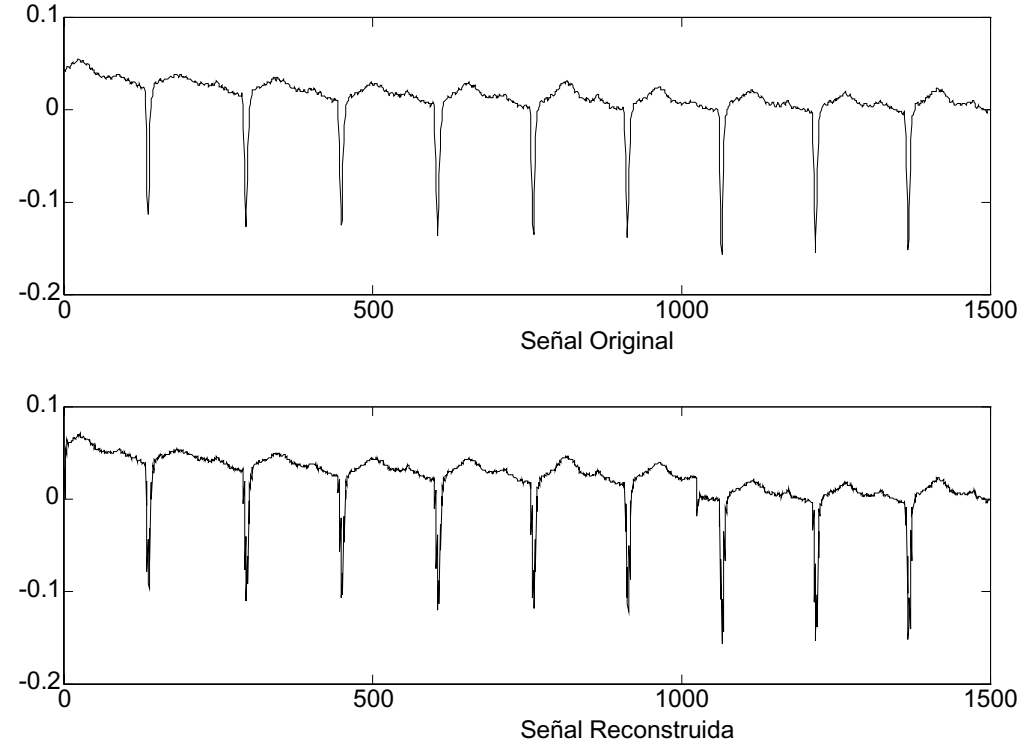
Compresión DCT 2,5:1 PAC122: 5.120 muestras, Fs = 250Hz.



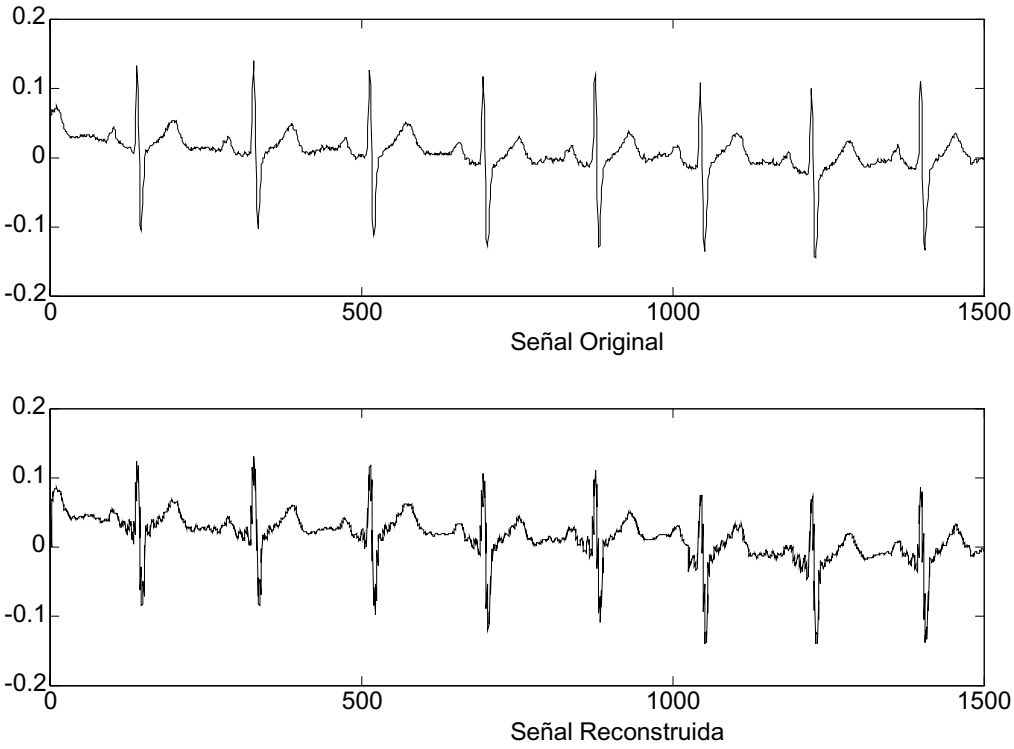
Compresión DCT 5:1 PAC212: 5.120 muestras, Fs = 250Hz.



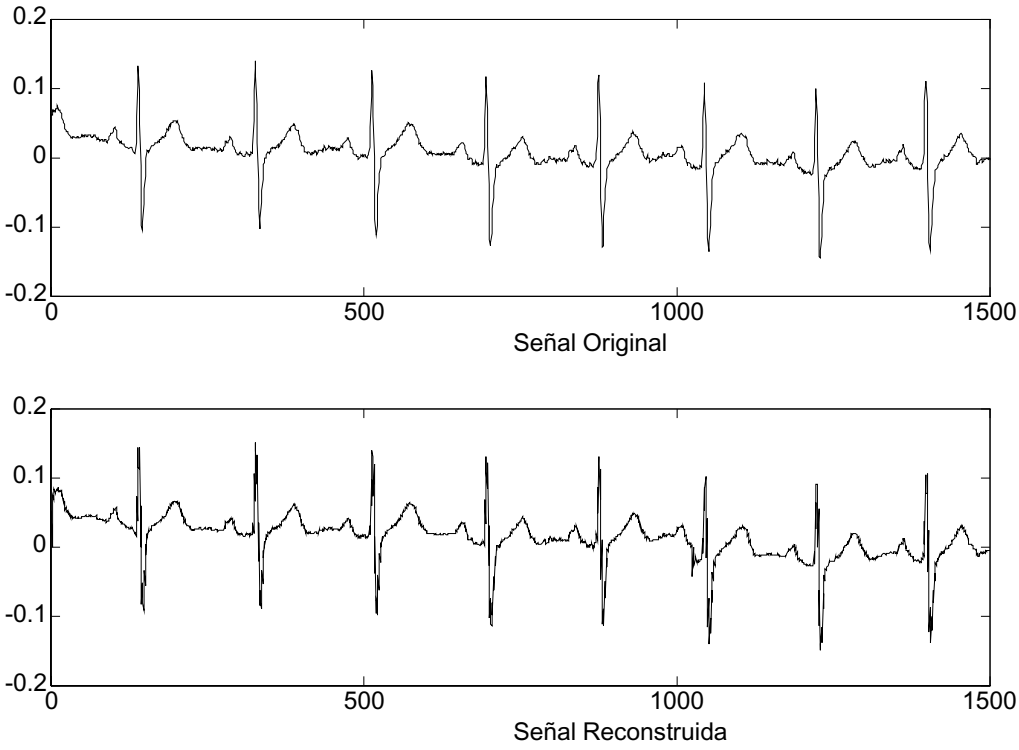
Compresión DCT 2,5:1 PAC212: 5.120 muestras, Fs = 250Hz.



Compresión DCT 5:1 PAC322: 5.120 muestras, Fs = 250Hz.



Compresión DCT 2,5:1 PAC322: 5.120 muestras, Fs = 250Hz.



CAPÍTULO 5

CONCLUSIONES, APORTACIONES Y PROYECCIÓN FUTURA

CAPÍTULO 5: CONCLUSIONES, APORTACIONES Y PROYECCIÓN FUTURA.

5.1. CONCLUSIONES.

En este trabajo han sido estudiados, implementados y evaluados dos grandes grupos de algoritmos de compresión que funcionan en tiempo real: los algoritmos de compresión directa y los algoritmos de compresión por transformación.

Del primero de estos grupos se estudiaron el AZTEC, el FAN y la DPCM. De estos tres, el algoritmo FAN es el que mejores resultados obtiene, considerando la razón de compresión alcanzada frente a la bondad de la señal reconstruida. Los tres algoritmos tienen un tiempo de tratamiento por muestra del orden de microsegundos, lo cual permite su uso en tiempo real con una frecuencia de muestreo grande y/o con varios canales de adquisición al unísono. De los tres algoritmos el más rápido es el AZTEC, el cual es capaz de alcanzar cuotas de compresión altas, pero distorsionando en gran medida la señal original. FAN puede alcanzar cuotas de compresión iguales o mayores sin dañar tanto la señal original, aunque requiere, para el caso del TMS320C25, más tiempo de cómputo, y la DPCM puede conseguir incluso almacenar la señal comprimida sin que existan pérdidas o estas sean inapreciables, pero a costa de una menor compresión que los otros dos.

En cuanto al grupo de algoritmos de compresión basados en transformaciones ortogonales, se estudiaron la FFT y la DCT. A la vista de los resultados se observa que la FFT es un método completamente válido para trabajar en tiempo real, que además obtiene una buena compresión de la señal sin perder información de interés de la misma. Por el contrario, la DCT no obtiene resultados aceptables, debido principalmente a el hecho de tener que trabajar en coma fija.

Si comparamos los métodos de compresión directa frente a los de compresión por transformación, vemos por un lado que la FFT obtiene una mayor compresión preservando mejor la señal original que los algoritmos de compresión directa evaluados, aun cuando el método de la FFT se aplica normalmente a registros muestrados a 500Hz o más, aunque como hemos visto también es posible su uso en registros muestreados a 250Hz con 12 bits de resolución. Por otro lado se observa que los requisitos de memoria de la FFT, y la DCT, son notablemente superiores a los de los algoritmos de compresión directa. Esto supone un problema a tener en cuenta cuando se trabaja con sistemas como el TMS320C25 donde la memoria interna del chip es escasa, y trabajar con memoria externa, además de encarecer la aplicación, retrasa apreciablemente la ejecución del algoritmo.

En cuanto a la comparativa del TMS320C25 frente al PC se observa que siempre el TMS320C25 obtiene tiempos de cálculo del mismo orden que el PC, y tiempos

mucho menores en sistemas x86, como el 80386, que datan de la misma época que el TMS320C25. Teniendo en cuenta que el TMS320C25 apareció en 1.985, y que en la actualidad existen DSP mucho más rápidos capaces además de trabajar en coma flotante, nos da una idea de la potencia de cálculo de estos chips y de la capacidad de los mismos para realizar este tipo de tareas. En las figuras 5.1 y 5.2 están representados los tiempos de cómputo para una FFT de 1024 puntos de distintos DSP de coma fija y coma flotante. Podemos ver como las nuevas generaciones de DSP consiguen, sobre todo los de coma flotante, tiempos mucho menores que los del TMS320C25, y superan también los tiempos obtenidos en este trabajo utilizando un Pentium 100Mhz.

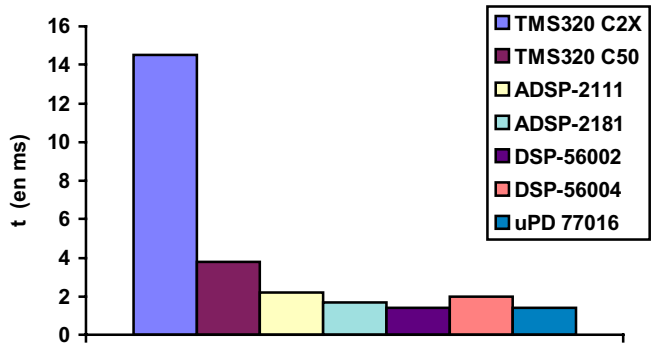


Fig. 5.1 - Tiempo de cálculo de la FFT de 1024 puntos en DSP de coma fija.

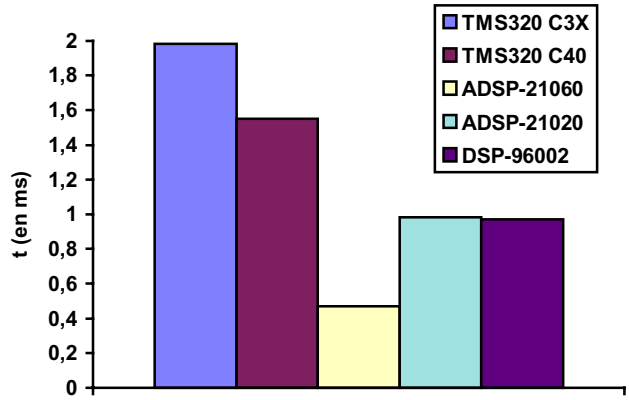


Fig. 5.2. - Tiempo de cálculo de la FFT de 1024 puntos en DSP de coma flotante.

Trabajar con el TMS320C25 en una placa coprocesadora tiene además la ventaja de que se liberan recursos del procesador principal del *host*, siendo posible realizar los cálculos de manera concurrente, y se potencia el sistema.

5.2. APORTACIONES.

Se han implementado los algoritmos AZTEC, FAN, DPCM, FFT y DCT (de 1024 muestras estos dos últimos), para el TMS320C25 funcionando sobre la placa ARIEL DTK-C25+. Asimismo se ha desarrollado el software, de carácter general, necesario para la carga, ejecución, comunicación y control de programas en dicha placa.

Los algoritmos citados han sido también implementados en C con vistas a su uso y evaluación sobre otras plataformas, y su posible implementación en el futuro sobre sistemas DSP más potentes.

Se ha estudiado el comportamiento de los algoritmos en tiempo real, así como los resultados de compresión y bondad de la señal reconstruida a partir del cálculo del PRD (*Percent Root-Mean-Square Difference)*.

5.3. PROYECCIÓN FUTURA.

En la actualidad, el CEN TC 251 (Comité Europeo de Normalización, Comité Técnico 251) se encarga de estudiar métodos y formas de realizar la transmisión de señales de ECG de forma eficiente. En este apartado, la compresión de datos en tiempo real juega un papel fundamental ofreciendo rapidez y calidad, y reduciendo costes.

En el futuro se proyecta la implementación de los algoritmos aquí estudiados sobre DSP de mayor potencia, capaces de trabajar mucho más rápidamente y realizando las operaciones en coma flotante, como es el caso del TMS320C30, lo cual permitirá el uso de algoritmos, como la DCT, cuyo rendimiento no era satisfactorio por el hecho de trabajar en coma fija. Además se podrán realizar sistemas que trabajen tanto de forma independiente como concurrentemente con la operación de otros en los que se hallen integrados, ofreciendo la calidad y velocidad necesarias.

También se podrán realizar nuevas implementaciones a partir de la mejora de los que ya se tienen, como por ejemplo la ADPCM, y, mediante el uso de hardware con más potencia de cálculo, nuevas implementaciones de algoritmos más costosos computacionalmente, pero con mejores resultados para la compresión.

Se proyecta también el uso de la FFT y la DCT sobre otro tipo de sistemas encaminados no sólo a la compresión de señales, sino también con vistas a realizar análisis de la señal de ECG, o preprocesado de la misma, para poder aplicar después otro tipo de tratamiento como pueden ser filtros, redes neuronales u otros. También se podrán aplicar estas transformadas sobre otros tipos de señales digitales, como por ejemplo las de audio y vídeo.

APÉNDICE A

LISTADOS DE LOS PROGRAMAS

LISTADO ENSAMBLADOR DEL ALGORITMO AZTEC.

```
; Algoritmo AZTEC en tiempo real

        .asect  "reset",0
        b       aztec

; Variables:

        .bss    Vmx,1
        .bss    Vmn,1
        .bss    Vmxi,1
        .bss    Vmni,1          ; Variables para determinar el rango de la
                                ; entrada
        .bss    Vth,1           ; Thresold
        .bss    V,1             ; Entrada muestra t
        .bss    AZT,1           ; Salida
        .bss    LineMode,1      ; Modo PLATEAU=0, SLOPE=1
        .bss    T1,1
        .bss    V1,1            ; Para guardar, T1=nº muestras, V1 valor medio
        .bss    Tsi,1
        .bss    Vsi,1           ; Para funcionamiento SLOPE
        .bss    Sign,1          ; Ver signo de la rampa (+ hacia arriba,
                                ; - hacia abajo)
        .bss    aux,1           ; Variable auxiliar para realizar productos


; Programa:

        .text

; Inizialización del DSP:

aztec   ldpk    6               ; DP <- 6, pág. nº 6, donde se encuentra
                                ; el bloque B1, dirección 300h (se especifica
                                ; en el archivo de comandos)
        ssxm                    ; Modo de extensión de signo
        spm     0               ; Salida de P a ACC -> Sin desplazamiento
        rhm                     ; Reset Hold Mode
        rovm                    ; Reset Overflow Mode


; Inicialización de variables:

        zac
        sach    AZT
        sach    LineMode        ; AZT=0, LineMode=0 --> Plateau
        lack    10
        sacl    Vth             ; Inicializo Vth

        in      Vmxi,PA0
        lac     Vmxi
        sacl    Vmni            ; Vmni=Vmxi=ECGt
        lrlk    AR2,512         ; Contador de nº de muestras, 512
        lark    AR0,50          ;
        lark    AR1,0           ; Contador LineLen
        larp    AR1


; Bucle de toma de muestras:

bucle1  in      V,PA0
        lac     Vmxi
        sacl    Vmx             ; Vmx=Vmxi
        lac     Vmni
        sacl    Vmn             ; Vmn=Vmni
        adrk    1               ; AR1+=1, (->LineLen+=1)
        cmpr    1               ; Comparo AR1 con AR0=50
        bbz     mediac
        lac     V
        sub     Vmx
        blz     sig1            ; Salta si V<Vmx -> (V-Vmx)<0
        lac     V
        sacl    Vmxi            ; Si no, Vmxi=V
sig1    lac     V
```

```
        sub     Vmn
        bgz     sig2            ; Salta si V>Vmn -> (V-Vmn)>0
        lac     V
        sacl    Vmni            ; Si no, Vmni=V
sig2    lac     Vmxi
        sub     Vmni
        sub     Vth             ; (Vmxi-Vmni)-Vth
        bgez    mediac          ; Si es >=0 hay que tratarlo
        larp    AR2             ; Comprueba si se han acabado las muestras
        banz    bucle1,*-,AR1
        larp    AR2             ; Hay que cambiar de nuevo a AR2, pues "banz"
                                ; cambió a AR1
        adrk    1               ; Añade 1 para que la comprobación final
                                ; "otravez" sea correcta
        larp    AR1             ; Paso a AR1 para tratar lo que falta


mediac  lac     Vmx
        add     Vmn
        ror                     ; (Vmx+Vmn)/2
        sacl    V1              ; Guardo en V1

        sar     AR1,T1          ; Guardo T1
        lac     LineMode
        bnz     slopm           ; Comprueba si está en mode PLATEAU o SLOPE


; Modo Plateau:

        lac     T1
        subk    2               ; T1-2 <= 0 ? --> T1 <= 2 ?
        blez    mog1            ; Salta si ACC <= 0
        out     T1,PA0          ; Saca T1
        lac     V1
        sacl    AZT
        out     AZT,PA0         ; Saca V1, y queda este almacenado en AZT
        b       otravez

; T1 <=2, hay que pasar a modo SLOPE:

mog1    lack    1               ; ACC=1
        sacl    LineMode        ; Ahora está en modo SLOPE, pues LineMode=1
        lac     V1
        sub     AZT             ; (V1-Vsi), Vsi=AZT (=valor anteriormente
                                ; guardado)
        blz     menor           ; Salta si (V1-Vsi)<0
        lack    1               ; Si es >=0 Sign=1
        b       guarda
menor   lack    -1              ; Si es <0 Sign=-1
guarda  sacl    Sign
        lac     T1              ; Se carga con T1 para no perder la longitud inicial
        sacl    Tsi
        lac     V1
        sacl    Vsi             ; Vsi=V1
        b       otravez


; Modo SLOPE:

slopm   lac     T1
        sub     2               ; Comprueba si T1 es o no > 2
                                ; T1 <= 2 --> T1-2 <= 0
        blez    mog2            ; Salta si ACC <= 0
        lac     Tsi             ; T1 > 2
        neg
        sacl    Tsi             ; Invierto Tsi
        out     Tsi,PA0         ; Lo saco
        out     Vsi,PA0         ; Saco Vsi,
        out     T1,PA0          ; Saco T1
        out     V1,PA0          ; Y por último V1
        lac     V1
        sacl    AZT             ; Por último copio V1 en AZT, para saber el
                                ; último valor sacado
        zac
        sacl    LineMode        ; Pasa a modo PLATEAU, LineMode=0
        b       otravez
```

```
; T1 <=2, hay que comprobar si ha habido cambio de signo en la rampa:

mog2    lt      Sign
        lac     V1
        sub     Vsi
        sacl    aux
        mpy     aux


        pac                     ; ACC=(V1-Vsi)*Sign
        blz     mog3            ; Si ACC<0
        lac     Tsi
        add     T1
        sacl    Tsi             ; Tsi+=T1
        lac     V1
        sacl    Vsi             ; Vsi=V1
        b       otravez

; Ha habido cambio de signo. Se guarda la rampa y empieza otra:

mog3    lac     Tsi
        neg
        sacl    Tsi
        out     Tsi,PA0         ; Saca -Tsi
        lac     Vsi
        sacl    AZT
        out     AZT,PA0         ; Saca Vsi, y queda almacenado en AZT
lac     T1              ; Lo mismo que antes, Tsi=T1 para no perder
                        ; la longitud inicial de la rampa
        sacl    Tsi
        lac     V1
        sacl    Vsi             ; Vsi=V1
        lac     Sign
        neg
        sacl    Sign            ; Sign*=-1


; Reinizialicación y comprobación de final de muestras:

otravez lac     V
        sacl    Vmxi
        sacl    Vmni            ; Vmxi=Vmni=V

        lark    AR1,0
        larp    AR2             ; Comprueba si quedan más muestras
        banz    bucle1,*-,AR1

fin     ret

        end
```

LISTADO C DEL ALGORITMO AZTEC.

```
#include <stdio.h>
#include <stdlib.h>

FILE *f1,*f2;
int Vth=10,s=0,d=0;

void aztec (int Vth) {   /* Entrada, thresold */
    enum {PLATEAU,SLOPE};
    long Vmx,Vmn,Vmxi,Vmni;
    int V,
        LineMode=PLATEAU,   /* Indica modo Plateau o Slope */
        LineLen=1,
        V1,
        T1,
        Vsi,
        Tsi,
        Sign,
        AZT;
    int count=0,np=0,ns=0,sp=0,ss=0;

    fscanf (f1,"%x\n",&V);
    count++;
    Vmni=Vmxi=V;
    AZT=V;
    LineMode=PLATEAU;
    while (!feof (f1)) {
        while (LineLen<51) {
            LineLen++;
            if (feof (f1)) break;
            fscanf (f1,"%x\n",&V);
            count++;
            Vmx=Vmxi;
            Vmn=Vmni;
            if (Vmx<V) Vmxi=V;
            if (Vmn>V) Vmni=V;
            if ((Vmxi-Vmni)>=Vth) break;
        }
        T1=LineLen-1;
        V1=(Vmx+Vmn)>>1;
        if (LineMode==PLATEAU) {
            if (T1>2) {
                fprintf (f2,"%04x\n",T1);
                fprintf (f2,"%04x\n",V1);
                sp+=T1; np++;
                AZT=V1;
            }
            else {
                LineMode=SLOPE;
                if ((V1-AZT)<0) Sign=-1;
                else Sign=1;
                Tsi=T1;
                Vsi=V1;
            }
        }
        else {
            if (T1>2) {
                fprintf (f2,"%04x\n",-Tsi);
                fprintf (f2,"%04x\n",Vsi);
                fprintf (f2,"%04x\n",T1);
                fprintf (f2,"%04x\n",V1);
                sp+=T1; np++;
                ss+=Tsi; ns++;
                AZT=V1;
                LineMode=PLATEAU;
            }
            else {
                if (((V1-Vsi)*Sign)<0) {
                    fprintf (f2,"%04x\n",-Tsi);
                    fprintf (f2,"%04x\n",Vsi);
                    ss+=Tsi; ns++;
                    AZT=Vsi;
                    Tsi=T1;
                    Vsi=V1;
```

```
                    Sign*=-1;
                }
                else {
                    Tsi+=T1;
                    Vsi=V1;
                }
            }
        }
        Vmxi=Vmni=V;
        LineLen=1;
    }
    printf ("\nNúmero de muestras tratadas: %d",count);
    printf ("\nNúmero de líneas generadas : %5d\tsuma: %d",np,sp);
    printf ("\nNúmero de rampas generadas : %5d\tsuma: %d",ns,ss);
}
```

LISTADO C DEL PROGRAMA DESCOMPRESOR UN-AZTEC,

```
void unaztec () {
    FILE *f1;
    register i;
    int m,s,d,old=0;

    while (!feof (f1)) {
        fscanf (f1,"%x\n%x\n",&s,&d);
        if (s>0) {
            for (i=0;i<s;i++) fprintf (f2,"%04X\n",d);
            old=d;
        }
        else {
            s=-s;
            m=(d-old)/s;
            for (i=1;i<=s;i++) {
                d=m*i+old;
                fprintf (f2,"%04X\n",d);
            }
            old=d;
        }
    }
}
```

LISTADO ENSAMBLADOR DEL ALGORITMO FAN.

```
; FAN en tiempo real

        .sect   "init"
        b       inicio

; Variables:

        .bss    X0,1
        .bss    X1,1
        .bss    X2,1    ; Muestras
        .bss    XU2,1
        .bss    XL2,1
        .bss    XU1,1
        .bss    XL1,1   ; Variables para determinar áreas
        .bss    Eps,1   ; Epsilon
        .bss    T,1     ; Longitud de la línea
                        ; Usar, AR2 para contarla

; Programa:
        .text

; Inicialización DSP
inicio  ssxm            ; Modo de extensión de signo
        rovm            ; Reset Overflow Mode
        dint            ; Esto es una prueba
        ldpk    6       ; Bloque B1

; Inicialización variables
        in      X0,0
        in      X1,0
        lack    10
        sacl    Eps     ; Inicializo Epsilon
        larp    AR2
        lark    AR2,1   ; T=1
        lark    AR1,4997 ; Para repetir 5000-2 veces
        lac     X1
        add     Eps
        sacl    XU1     ; XU1=X1+Eps
        sub     Eps,1   ; Resto 2*Eps
        sacl    XL1     ; XL1=X1-Eps
        out     X0,0    ; El primer dato se guarda sin más

loop    in      X2,0

; Cálculo de XU2 y XL2
;        rsxm            ; Pongo SXM a 0 para utilizar la instrucción SUBC
        sar     AR2,T
        lac     XU1
        sub     X0
        call    divide  ; Divide (XU1-X0) por T
        add     XU1
        sacl    XU2     ; XU2 calculado
        lac     XL1
        sub     X0
        call    divide
        add     XL1
        sacl    XL2     ; XL2 calculado

; Comprobaci¢n de si X2 está dentro o fuera del área
        lac     X2
        sub     XU2     ; (X2-XU2)>0 <--> X2>XU2 --> Fuera del área
        bgz     fuera
        lac     X2
        sub     XL2     ; (X2-XL2)<0 <--> X2<XL2 --> Fuera del área
        blz     fuera

; Dentro del área --> Obtener el área más restrictiva
        lac     X2
        add     Eps     ; X2+Eps
        sub     XU2     ; (X2+Eps)-XU2 > 0 ? <--> (X2+Eps) > XU2 ?
        bgz     sig     ; Si es > 0 no hay que cambiarlo
        add     XU2     ; Le vuelvo a sumar XU2 --> Quedar  X2+Eps
        sacl    XU2     ; Lo guardo en XU2
```

```
sig     lac     X2
        sub     Eps     ; X2-Eps
        sub     XL2     ; (X2-Eps)-XL2 < 0 ? <--> (X2-Eps)<XL2
        blz     sig2    ; Si es < 0 no ha que cambiarlo
        add     XL2     ; Queda X2-Eps
        sacl    XL2     ; Lo guardo

sig2    mar     *+      ; T+=1
        lac     X2
        sacl    X1      ; X1=X2
        larp    AR1
        banz    loop,*-,AR2
        out     T,0
        out     X1,0    ; Saco los últimos datos
        b       fin

;        b       loop    ; Busco el siguiente

; Fuera del área --> Guardar datos y reinicializar
fuera   out     T,0
        out     X1,0

        lac     X1
        sacl    X0      ; X0=X1
        lac     X2
        sacl    X1      ; X1=X2
        add     Eps
        sacl    XU1     ; XU1=X1+Eps
        sub     Eps,1   ; Resto con desplazamiento, así resto 2*Eps
        sacl    XL1     ; XL1=X1-Eps
        lark    AR2,1   ; T=1

        larp    AR1
        banz    loop,*-,AR2

fin     ret
        ret             ; Para distinguirlo

;        b       loop

divide  bgz     divpos
        neg
        rptk    15
        subc    T
        andk    0ffffh  ; Se elimina el resto
        neg
        ret
divpos  rptk    15
        subc    T
        andk    0ffffh  ; Se elimina el resto

findiv  ret

        end
```

LISTADO C DEL ALGORITMO FAN.

```
#include <stdio.h>
#include <string.h>

FILE *f1,*f2;

void fan (int Epsilon) {
    int X0,X1,X2;
    int XU2,XL2,XU1,XL1;
    int T;

    fscanf (f1,"%x\n",&X0);
    fscanf (f1,"%x\n",&X1);
    T=1;
    XU1=X1+Epsilon;
    XL1=X1-Epsilon;
    fprintf (f2,"%04X\n",X0);

    while (!feof (f1)) {
        fscanf (f1,"%x\n",&X2);
        XU2=(XU1-X0)/T+XU1;
        XL2=(XL1-X0)/T+XL1;

       if (X2<=XU2 && X2>=XL2) {        /* La muestra está dentro de los límites */
           /* Se toma el área más restrictiva */
           XU2=(XU2<X2+Epsilon) ? XU2:X2+Epsilon;
           XL2=(XL2>X2-Epsilon) ? XL2:X2-Epsilon;
           T++;        /* Incrementa la longitud de la línea */
           X1=X2;  /* X2 pasa a X1 y volvemos */
       }
       else {        /* Muestra fuera de los límites, hay que guardar */
           fprintf (f2,"%04X\n",T);
           fprintf (f2,"%04X\n",X1);

           /* Reset de la variables */
           X0=X1;
           X1=X2;
           T=1;
           XU1=X1+Epsilon;
           XL1=X1-Epsilon;
       }
    }
    /* Guarda los últimos datos */
    fprintf (f2,"%04X\n",T);
    fprintf (f2,"%04X\n",X1);
}
```

LISTADO C DEL ALGORITMO DESCOMPRESOR UN-FAN.

```
void unfan () {
    register i,m;
    int s,d,old;

    fscanf (f1,"%x\n",&d);
    old=d;
    fprintf (f2,"%04X\n",d);
    while (!feof (f1)) {
        fscanf (f1,"%x\n%x\n",&s,&d);
        m=(d-old)/s;
        for (i=1;i<=s;i++) {
            d=m*i+old;
            fprintf (f2,"%04X\n",d);
        }
        old=d;
    }
}
```

LISTADO ENSAMBLADOR DPCM.

```
; DPCM v0.1 para C25

        .asect "reset", 0
        b       dpcm

; ************************************************************************
; Variables
; ************************************************************************

        .bss    y0,1
        .bss    y1,1
        .bss    y2,1
        .bss    z,1
        .bss    z2,1

; ************************************************************************
; Programa compresor
; ************************************************************************

        .text

dpcm:   ldpk    6         ; bloque B1
        lrlk    ar1,2558 ; numero de muestras a tratar ((5120-2)/2-1)
                          ; 5210 - 2 (las 2 primeras)
                          ; dividido por 2 porque en cada pasada del bucle
                          ; tratamos 2 muestras
                          ; -1 por que el bucle es con un BANZ
        larp    ar1

        in      y0,0
        out     y0,0
        in      y1,0
        out     y1,0    ; los dos primeros términos salen tal cual

bucle:  in      y2,0
         lac     y1,1
         sub     y0       ; ACC=2*y1-y0
         sacl    y0       ; y0=ACC
         sub     y2       ; ACC=2*y1-y0-y2
         neg              ; ACC=y2-2*y1+y0
         sacl    z        ; z=ACC

         ; Implementación modo saturación para 8 bits
         subk    127
         blez    next00  ; si z<=127 esta ok
         lack    127
         sacl    z        ; si z>127 entoces z=127
         b       next01
next00:  lac     z
         addk    128
         bgez    next01  ; si z>=-128 esta ok
                          ; si z<0 => z+128 >= 0 sii z>=-128
         lalk    -128
         sacl    z        ; si z<-128 => z=-128
next01:
         ; Fin implem. modo saturación

         lac     y0
         sacl    y2       ; y2=y0, ó y2 = estimacion de y2 = E(y2)
         lac     y1
         sacl    y0       ; y0=y1
         lac     z
         add     y2       ; ACC=z+E(y2)
         sacl    y1       ; y1=ACC

         ; Siguiente muestra (es lo mismo que la anterior pero en z2)
         in      y2,0
         lac     y1,1
         sub     y0       ; ACC=2*y1-y0
         sacl    y0
         sub     y2
         neg              ; ACC=y2-2*y1+y0
         sacl    z2
```

```
        ; Saturación 8 bits:
        subk    127
        blez    next02  ; si z2<=127 esta ok
        lack    127
        sacl    z2      ; si z2>127 entoces z2=127
        b       next03
next02: lac     z2
        addk    128
        bgez    next03  ; si z2>=-128 esta ok
                        ; si z2<0 => z2+128 >= 0 sii z2>=-128
        lalk    -128
        sacl    z2      ; si z2<-128 => z2=-128
next03:
        ; Fin saturación 8 bits

        lac     y0
        sacl    y2      ; y2=E(y2)
        lac     y1
        sacl    y0      ; y0=y1
        lac     z2
        add     y2
        sacl    y1      ; y1=z2+E(y2)

        ; Ahora se mezclan los dos resultados en uno
        lac     z2
        andk    00ffh   ; fuera la parte alta
        sacl    z2
        lac     z,7
        sfl             ; desplazo 1 a la izqda
        or      z2
        sacl    z
        out     z,0     ; al puerto

        banz    bucle,*-

        end


; IDPCM para C25

       .asect "reset", 0
       b      idpcm

; ************************************************************************
; Variables
; ************************************************************************
       .bss   y0,1
       .bss   y1,1
       .bss   y2,1
       .bss   z,1

; ************************************************************************
; Programa descompresor DPCM
; ************************************************************************
        .text

idpcm:  ldpk   6       ; Bloque B1
       lrlk   ar1,2558 ; Muestras a tratar -1
       larp   ar1

       in     y0,0
       out    y0,0
       in     y1,0
       out    y1,0    ; Las dos primeras salen tal cual

ibucle: in     z,0
        lac     z,4
       sach   y2,4    ; desplazo 8 bits
       lac    y2
       add    y1,1
       sub    y0      ; ACC=z+2*y1-y0
       sacl   y2
       out    y2,0    ; saco el dato descomprimido
       lac    y1
       sacl   y0      ; y0=y1
       lac    y2
       sacl   y1      ; y1=y2
```

```
        lac     z
        andk    00FFh   ; borro bits altos
        sacl    y2
        subk    80h     ; si y2>=80h (128 decimal) es negativo
        blz     positiv ; si es positivo salta
        lac     y2      ; si en negativo se anyade el signo
        ork     0FF00h  ; poniendo los 8 bits altos a 1
        sacl    y2
positiv lac      y2
        add     y1,1
        sub     y0      ; ACC=z+2*y1-y0
        sacl    y2
        out     y2,0    ; saco dato
        lac     y1
        sacl    y0      ; y0=y1
        lac     y2
        sacl    y1      ; y1=y2
        banz    ibucle,*-

        end
```

LISTADO EN C PARA DPCM

```
/* DPCM, GNU C, v0.1
 * De entrada se le pasan los registros de la base de datos del MIT, muestrados a 250Hz
 * y 12b, en ficheros ASCII con los números en formato hexadecimal tal y como los coje
 * el simulador del C25. De 12b se pasa a 8b; como el resultado se almacena en palabras
 * en cada una hay dos muestras de 8 bits.
 */

#include <stdio.h>
#include <string.h>
#include "iobj.h"

void error (char tipo, char *msg) {
    if (!tipo) {
        puts ("DPCM 0.0, DPCM [-d] <entrada> [salida]");
        puts ("          -d: descomprimir");
        puts ("              si no se especifica [salida] se toma 'out'");
    }
    else puts (msg);
    exit (1);
}
void main (int argc, char **argv) {
    char desc=0;
    short int y[3], z1, z2, errores=0;
    FILE *fi, *fo;
    if (argc<2) error (0,"");
    if (!strcmp (argv[1],"-d")) {
        desc=1;
        if (argc<3) error (0,"");
        else fi=xopen (argv[2],"rt");
        if (argc>3) fo=xopen (argv[3],"wt");
        else fo=xopen ("out","wt");
    }
    else {
        fi=xopen (argv[1],"rt");
        if (argc>2) fo=xopen (argv[2],"wt");
        else fo=xopen ("out","wt");
    }
    if (!desc) { /* Comprimir */
        fscanf (fi,"%hx\n%hx\n",y,y+1);
        fprintf (fo,"%04hX\n%04hX\n",y[0],y[1]); /* las dos primeras tal cual */
        while (!feof (fi)) {
            fscanf (fi,"%hx\n",y+2);
            y[0]=2*y[1]-y[0]; // Almaceno esta operación en y[0] para no tener que
                              // volver a repetirla.
            z1=y[2]-y[0];
            if (z1>127) { z1=127; errores++; }
            else if (z1<-128) { z1=-128; errores++; }
            y[2]=y[0]; // Utilizo y[2] como variable temporal para y[0]
            y[0]=y[1];
            y[1]=z1 + y[2]; // Aqui y[2] = y[0] = 2*y[1]-y[0]
            fscanf (fi,"%hx\n",y+2);
            y[0]=2*y[1]-y[0];
            z2=y[2]-y[0];
```

```
            if (z2>127) { z2=127; errores++; }
            else if (z2<-128) { z2=-128; errores++; }
            y[2]=y[0]; // Uso y[2] como variable temporal para y[0]
            y[0]=y[1];
            y[1]=z2 + y[2]; // y[2] = y[0] = 2*y[1]-y[0]
            z1=(z1<<8) | (z2 & 0x00FF); // Formo la palabra a partir de los bytes
            fprintf (fo, "%04hX\n", z1);
        }
        printf ("Errores: %hd\n", errores);
    }
    else { /* Descomprimir */
        fscanf (fi,"%hx\n%hx\n",y,y+1);
        fprintf (fo,"%04hX\n%04hX\n",y[0],y[1]); /* las dos primeras */
        while (!feof (fi)) {
            fscanf (fi,"%hx\n",&z1);
            y[2]=(z1>>8)+2*y[1]-y[0];
            fprintf (fo,"%04hX\n",y[2]);
            y[0]=y[1];
            y[1]=y[2];
            z1&=0xFF;
            if (z1>=0x80) z1|=0xFF00; /* si es negativo (en 8 bits eso es
                                       * equivalente a ser >=80h) extiende el signo */
            y[2]=z1+2*y[1]-y[0];
            fprintf (fo,"%04hX\n",y[2]);
            y[0]=y[1];
            y[1]=y[2];
        }
    }
    fclose (fi);
    fclose (fo);
}
```

LISTADO ENSAMBLADOR DEL ALGORITMO FFT.

```
FFT
        .asect  "start",0
        b       begin

        .include "omegas.asm"

datos   .usect  "datos",2048
ws      .usect  "ws_ram",2048

        .bss    i,1
        .bss    n1,1            ; n1 indica el tamanyo de la tranf. que se
                                ; esta realizando en cada paso del bucle.
                                ; Comienza en 2 y aumenta (*2 cada vez) hasta
                                ; llegar a n
        .bss    mn1,1           ; m=n1/2, e indica la separacion entre la
                                ; secuencia par y la impar

n       .set    1024            ; N£mero de datos

;       AR1 para datos
;       AR2 para ws
;       AR0 para marcar el offset de m=n1/2

        .text
begin
        ldpk    0
; Copio w's en RAM:
        larp    AR1
        lrlk    AR1,ws
        rptk    255
        blkp    wsrom,*+
        rptk    255
        blkp    wsrom+256,*+
        rptk    255
        blkp    wsrom+512,*+
        rptk    255
        blkp    wsrom+768,*+
        rptk    255
        blkp    wsrom+1024,*+
        rptk    255
        blkp    wsrom+1280,*+
        rptk    255
        blkp    wsrom+1536,*+
        rptk    255
        blkp    wsrom+1792,*+

; Cargo datos:
        lrlk    AR1,datos
        lrlk    AR0,1024        ; N§ puntos FFT
        lrlk    AR2,1023
sgld
        in      *+,PA0          ; Cargo parte real
        in      *-,PA0          ; Cargo parte imag
mar     *BR0+,AR2               ; Me muevo al siguiente
        banz    sgld,*-,AR1

; Comienzo rutina FFT
        lack    1
        sacl    n1,1            ; n1=2
        subk    1
        sacl    mn1             ; n1/2 - 1 = m - 1
                                ; Guardo m-1 porque m sirve para controlar
                                ; un bucle con 'banz'. Como se ejecuta m veces,
                                ; hay que cargar AR3 (que es quien lo controla)
                                ; con m-1
mainlp
        lrlk    AR1,datos
        lrlk    AR2,ws-2
        lar     AR0,n1          ; Cargo AR0 con n1, osea, con m*2. AR0 sirve
                                ; para marcar la separacion de la secuencia par
                                ; con la impar. Esta separacion es de n1/2, pero
                                ; realmente como son num. complejos es el doble,
                                ; osease n1
```

```
        lar     AR3,mn1

        larp    AR2
        mar     *0+,AR4         ; AR2=AR2+(m-1), realmente +(m-2), pues son
                                ; numeros complejos
        lrlk    AR4,n
        mar     *0-,AR1         ; AR4=n-n1
        mar     *0+             ; AR1=datos+m

loop1
; Producto:
; Parte real:
        lt      *+,AR2  ; T=RXijm
        mpy     *+,AR1  ; P=(1/2) RXijm*RWm
        ltp     *-,AR2  ; T=IXijm, ACC=(1/2) RXijm*RWm
        mpy     *-      ; P=(1/2) IXijm*IWm
        spac            ; ACC=(1/2) (RXijm*RWm - IXijm*IWm)
        mpy     *+,AR1  ; P=(1/2) IXijm*RWm
        lt      *       ; T=RXijm
        sach    *+,AR2  ; RXijm=(1/2) (RXijm*RWm - IXijm+IWm)
; Parte imaginaria:
        pac             ; ACC=(1/2) IXijm*RWm
        mpy     *+,AR1  ; P=(1/2) RXijm*IWm     ; AR2 -> RW(m+1)
        apac            ; ACC=(1/2) (IXijm*Rwm + RXijm*IWm)
        sach    *-      ; IXijm=ACC     ; AR1 -> RXijm
; Guardar datos:
; Parte real:
        lac     *0-,15  ; ACC=(1/4) (RXijm*Rwm - IXijm*IWm)
        add     *,14    ; ACC=(1/4) (RXij + (RXijm*Rwm - IXijm*IWm))
        sach    *0+,1   ; RXij=(1/2) ACC
        subh    *       ; ACC=(1/4) (RXij - (RXijm*Rwm - IXijm*IWm))
        sach    *+,1    ; RXijm=(1/2) ACC     ; AR1 -> IXijm
; Parte imaginaria
        lac     *0-,15  ; ACC=(1/4) (IXijm*Rwm + RXijm*IWm)
        add     *,14    ; ACC=(1/4) (IXij + (IXijm*Rwm + RXijm*IWm))
        sach    *0+,1   ; IXij=(1/2) ACC
        subh    *       ; ACC=(1/4) (IXij - (IXijm*Rwm + RXijm*IWm))
        sach    *+,1,AR3; IXijm=(1/2) ACC   ; AR1 -> RXij(m+1)

; Final bucle interior:
        banz    loop1,*-,AR1

        mar     *0+,AR2         ; AR1+=n1, deja AR1 en su sitio
        mar     *0-,AR4         ; AR2-=n1, deja AR2 en su sitio
        lar     AR3,mn1         ; AR3=m-1

; Final segundo bucle:
        banz    loop1,*0-,AR1

fuera
        lac     n1
        subk    n
        bz      saca            ; n1<=n ? (Realmente: n1=n ? )

        lac     n1
        sacl    n1,1            ; n1*=2
        subk    1
        sacl    mn1             ; n1/2 - 1
        b       mainlp

; Saco datos fuera:
saca    larp    AR1
        lrlk    AR1,datos
        lark    AR2,7           ; Se repite 8 veces, hay que sacar 2048 datos,
                                ; 8*256
sloop   rptk    255
        out     *+,PA0
        larp    AR2
        banz    sloop,*-,AR1

fin     ret
        end
```

LISTADOS ENSAMBLADOR DE LA RUTINAS DE INICIALIZACIÓN E INTERRUPCIONES.

```
; INICIALIZACION DEL C25

BEGIN   DINT

        SSXM            ; modo extension de signo
        SPM     0       ; salida de P
        ROVM            ; sin modo overflow
        CNFD            ; bloque es de datos

; Copia del programa en el bloque B0 de memoria interna:
        LRLK    AR1,200h ; comienzo bloque B0 en memoria de datos
        LARP    AR1
        RPTK    (FIN-CZO)
        BLKP    CZO,*+  ; copio el programa en B0
        CNFP            ; lo configuro como bloque de programa
        B       C_B0

; Inicialización de registros de interrupción y memoria:
        ORG     C_B0
CZO     LDPK    0       ; punt. pag. a 0
        LACK    80h
        SACL    GREG    ; configuro la memoria tal que asi:
                        ;       prog: 0000 - 7FFFh
                        ;       data: 8000 - FFFFh
                        ; las W's pasan directamente a memoria de datos
        LALK    12499   ; (50 MHz / 4 = 12.5 MHz) / 1 kHz = 12500
        SACL    PRD     ; ajusto el timer para 1 kHz -> Frecuencia de muestreo.
        LACK    0008h   ; 0000 0000 0000 1000 -> int. timer solamente
        SACL    IMR

; Programacion del conversor A/D:
        LALK    0FDE3h  ; 1111 1101 1110 0011 -> programacion AIC
                        ; elimina el filtro pasa-alta a la entrada y el filtro
                        ; sinc(x) a la salida
        SACL    DXR
        LRLK    AR1,500
ESPERA  BANZ    ESPERA,*- ; espera para la programacion

START => Comienzo de la rutina FFT.
```

```
; R.T.I. (RUTINA TRATAMIENTO INTERRUPCION)
; Esta rutina recoge la llamada de TIM y carga un dato en memoria cada ms
; Ar6 lleva la direccion, y Ar7 el numero de datos

RTI     SST     PSST    ; guardo ST0
        SST1    PSST1   ; guardo ST1
        SACL    PACCL
        SACH    PACCH   ; guardo Acc

        LARP    AR7
        BANZ    CARGA,*
        ZALH    PACCH
        OR      PACCL   ; recupero Acc
        LST1    PSST1
        LST     PSST    ; recupero estado
        RET

CARGA
        MAR     *-,AR6
        LAC     DRR,14
        SACH    *+,0
        ZALH    PACCH
        OR      PACCL   ; recupero Acc
        LST1    PSST1
        LST     PSST    ; recupero ST0 y ST1
        EINT
FIN     RET             ; equivale a IRET
```

LISTADO EN C DEL ALGORITMO FFT.

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <dos.h>
#include "iobj.h"
#define NE 200
#define PDW 1

/* VARIABLES GLOBALES */
typedef unsigned int uint;
typedef struct {
    float real,imag;
} complejo;
uint n=1024;
complejo *x,*w;
FILE *f1,*f2;
int d;
long l;

/* CALCULA LA T.F. ITERATIVO SOBRE UN SOLO ARRAY */
void FFT (void) {
    complejo p;
    register uint j,i,n1=2,m;

    while (n1<=n) {
        i=0;
        m=n1>>1;           /* m=n1 / 2 */
        while (i<n) {
            for (j=0;j<m;j++) {
                p.real=(x+i+j+m)->real*(w+m+j)->real - (x+i+j+m)->imag*(w+m+j)->imag;
                p.imag=(x+i+j+m)->imag*(w+m+j)->real + (x+i+j+m)->real*(w+m+j)->imag;
               (x+i+j+m)->real=(x+i+j)->real - p.real;
               (x+i+j+m)->imag=(x+i+j)->imag - p.imag;
               (x+i+j)->real=(x+i+j)->real + p.real;
               (x+i+j)->imag=(x+i+j)->imag + p.imag;
            }
            i+=n1;                  /* i=i+n1 */
        }
        n1<<=1;                  /* n1=n1*2 */
    }
}

/* ORDENA LA SECUENCIA DE ENTRADA */
void ordena () {
    uint i,j,pos,k;
    complejo aux;

    for (i=1;i<n-1;i++) {
        pos=0;
        k=i;
        for (j=1;j<n;j<<=1) {
            pos=(pos<<1) | (k & 1);
            k>>=1;
        }
        /* Se filtra el último (k & 1) y luego se pasa a 'pos'. */
        if (pos>i) {        /* Solamente hay que intercambiar elementos */
            aux=*(x+pos);
            *(x+pos)=*(x+i);
            *(x+i)=aux;
        }
    }
}

/* CALCULA EL ARRAY DE W's */
void cw (int s) {
    FILE *f;
    register unsigned k,i;
    for (i=1;i<n;i*=2) {
        for (k=0;k<i;k++) {
            (w+i+k)->real=PDW*cos (M_PI*k/i);
            (w+i+k)->imag=PDW*s*sin (M_PI*k/i);
        }
    }
}
```

LISTADO ENSAMBLADOR DEL ALGORITMO DCT PARA C25.

```
; DCT v0.1

WS      EQU     8000h   ; direccion W
WW      EQU     8400h   ; direccion WW
DATOS   EQU     9000h   ; direccion datos
BUFFER  EQU     0A000h  ; direccion buffer de entrada
N       EQU     1024    ; numero de datos (400h)
N1      EQU     60h     ; long. transf. que se ejecuta en cada bucle
MN1     EQU     61h     ; M-1
PSST    EQU     62h     ; para guardar SST
PSST1   EQU     63h     ; para guardar SST1
PACCL   EQU     64h     ; para guardar ACC bajo
PACCH   EQU     65h     ; para guardar ACC alto
CW      EQU     66h     ; salto para la W
ARX     EQU     67h     ; control carga datos

C_B0    EQU     0FF00h  ; aqui empieza B0 como bloque de programa

        NCLUD   OMEGAS.ASM   ; para la FFT
        NCLUD   OMEGAW.ASM   ; para la DCT

        ORG     0
        B       BEGIN   ; reset

        ORG     18h
        B       RTI     ; interrupcion del timer

        ORG     1Ch
        EINT
        RET             ; interrupcion XINT, simplemente retorna

        ORG     1024

BEGIN   DINT

        SSXM            ; modo extension de signo
        SPM     0       ; salida de P
        ROVM            ; sin modo overflow
        CNFD            ; bloque es de datos

        LRLK    AR1,200h ; comienzo bloque B0 en memoria de datos
        LARP    AR1
        RPTK    (FIN-CZO)
        BLKP    CZO,*+  ; copio el programa en B0
        CNFP            ; lo configuro como bloque de programa
        B       C_B0

        ORG     C_B0
CZO
        LDPK    0       ; punt. pag. a 0
        LACK    80h
        SACL    GREG    ; configuro la memoria tal que asi:
                        ;       prog: 0000 - 7FFFh
                        ;       data: 8000 - FFFFh
                        ; las W's pasan directamente a memoria de datos

; Programacion Frecuencia Muestreo (Timer Interno):

        LALK    12499   ; (50 MHz / 4 = 12.5 MHz) / 1 kHz = 12500
;        LALK    1249    ; frec. de 10k
        SACL    PRD     ; ajusto el timer para 1 kHz

; Programacion AIC:

        LACK    20h
        SACL    IMR     ; 0010 | 0000 -> XINT enabled
        EINT
        LALK    0FDA3h  ; 1111 1101 1010 0011
        IDLE            ; antes de mandar algo al DXR se hace un IDLE para
                        ; asegurarnos de que la ultima transmision ha terminado.
                        ; Por supuesto, XINT debe estar conectada, exclusivam.
        SACL    DXR,2   ; manda algo con LSB=00
                        ; esto me asegura que lo que mando a continuacion
                        ; no es una programacion erronea
```

```
        ORK     6,15     ; LSB ACCH = 11
        IDLE
        SACH    DXR     ; modo programacion
        IDLE
        SACL    DXR     ; programacion
        IDLE
        SACL    DXR,2   ; cierra comando LSB=00
        IDLE            ; estas dos ultimas tambien por seguridad

        DINT
        LACK    8       ; 0001 0000
        SACL    IMR     ; TINT enabled
        LALK    BUFFER
        SACL    ARX
        LRLK    AR7,1023
        EINT
PFD
        SAR     AR7,N1  ; primera carga
        LAC     N1
        BNZ     PFD     ; espera a que AR7 sea 0 (finaliza carga datos)

        B       INIDATA ; inicializa datos y comienza

START
        LALK    BUFFER
        SACL    ARX        ; zona de datos
        LRLK    AR7,1023   ; num. de datos -1
        EINT

; Comienzo rutina FFT:

;        B       SALTA

FFT
        LACK    1
        SACL    N1,1      ; n1=2
        SUBK    1
        SACL    MN1       ; n1/2 - 1 = m - 1
                          ; Guardo m-1 porque m sirve para controlar
                          ; un bucle con 'banz'. Como se ejecuta m veces,
                          ; hay que cargar AR3 (que es quien lo controla)
                          ; con m-1
        LALK    N
        SACL    CW        ; primer salto 512, 1024 al ser complejos
        LRLK    AR2,WS

MAINLP
        LRLK    AR1,DATOS
        LAR     AR0,N1    ; Separacion entre las secuencias par e impar
        LAR     AR3,MN1
        LARP    AR4
        LRLK    AR4,N
        MAR     *0-,AR1   ; AR4=n-n1
        MAR     *0+       ; AR1=datos+n1

LOOP1
; Producto:
; Parte real:
        LT      *+,AR2    ; T=Re[x']
        MPY     *+,AR1    ; P=(1/2) Re[x']*Re[w]
        LTP     *-,AR2    ; T=Im[x'], ACC=(1/2) Re[x']*Re[w]
        MPY     *-        ; P=(1/2) Im[x']*Im[w]
        SPAC              ; ACC=(1/2) (Re[x']*Re[w] - Im[x']*Im[w])
        MPY     *+,AR1    ; P=(1/2) Im[x']*Re[w]
        LT      *         ; T=Re[x']
        SACH    *+,1,AR2  ; RXijm=(1/2) (RXijm*RWm - IXijm+IWm)
; Parte imaginaria:
        PAC               ; ACC=(1/2) Im[x']*Re[w]
        MPY     *-        ; P=(1/2) Re[x']*Im[w]     ; AR2 -> Re[w]
        LAR     AR0,CW
        MAR     *0+,AR1   ; AR2-> Re[w+cw]
        APAC              ; ACC=(1/2) (Im[x']*Re[w] + Re[x']*Im[w])
        SACH    *-,1      ; Im[x']=ACC     ; AR1 -> Re[x']
; Guardar datos:
; Parte real:
        LAR     AR0,N1
        LAC     *0-,15    ; ACC   =(1/4) (Re[x']*Re[w] - Im[x']*Im[w])
```

```
        ADD     *,15      ; ACC   =(1/4) Re[x] + ACC
        SACH    *0+,0     ; Re[x] =(1/2) ACC
        SUBH    *         ; ACC   =(1/4) Re[x] - ACC
        SACH    *+,0      ; Re[x']=(1/2) ACC     ; AR1 -> Im[x']
; Parte imaginaria:
        LAC     *0-,15    ; ACC   =(1/4) (Im[x']*Re[w] + Re[x']*Im[w])
        ADD     *,15      ; ACC   =(1/4) Im[x] + ACC
        SACH    *0+,0     ; Im[x] =(1/2) ACC
        SUBH    *         ; ACC   =(1/4) Im[x] - ACC
        SACH    *+,0,AR3  ; Im[x']=(1/2) ACC   ; AR1 -> Re[x'+1]
; Final bucle interior:
        BANZ    LOOP1,*-,AR1
        LRLK    AR2,WS          ; deja AR2 en su sitio
        MAR     *0+,AR4         ; AR1+=n1, deja AR1 en su sitio
        LAR     AR3,MN1         ; AR3=m-1
; Final segundo bucle:
        BANZ    LOOP1,*0-,AR1
        LAC     N1
        SBLK    N
        BZ      PRODUCT         ; n1<=n ? (Realmente: n1=n ? )
        LAC     N1
        SACL    N1,1            ; n1*=2
        SUBK    1
        SACL    MN1             ; n1/2 - 1
        LAC     CW,15
        SACH    CW              ; cw=cw/2
        B       MAINLP

; Producto DCT:
product
        LRLK    AR3,1023
        LRLK    AR1,DATOS
        LRLK    AR2,WW
blprod  LT      *+,AR2          ; T=Re[x]
        MPY     *+,AR1          ; P=(1/2) Re[x]*Re[ww]
        LTP     *-,AR2          ; T=Im[x], Acc=(1/2) Re[x]*Re[ww]
        MPY     *-              ; P=(1/2) Im[x]*Im[ww]
        SPAC                    ; Acc=(1/2) (Re[x]*Re[ww] - Im[x]*Im[ww])
        MPY     *+,AR1          ; P=(1/2) Im[x]*Re[ww]
        LT      *               ; T=Re[x]
        SACH    *+,0,AR2        ; Re[x]=Acc
; parte imaginaria:
        PAC                     ; Acc=(1/2) Im[x]*Re[ww]
        MPY     *+,AR1          ; P=(1/2) Re[x]*Im[ww]  , AR2->siguiente
        APAC                    ; Acc=(1/2) (Re[x]*Im[ww] + Im[x]*Re[ww])
        SACH    *+,0,AR3        ; Im[x]=Acc             , AR1->siguiente
        BANZ    blprod,*-,AR1

; Saco datos fuera:
SALTA
        LARP    AR1
PFDL
        SAR     AR7,N1
        LAC     N1
        BNZ     PFDL    ; espera a que AR7 sea 0 (finaliza carga datos)
        DINT

; Senyal al Host para que los recoja:
        OUT     GREG,PA15 ; senyal para el Host
WAIT0   IN      N1,PA14
        LAC     N1
        ANDK    1
        BZ      WAIT0     ; espera senyal para continuar
        IN      N1,10     ; lee dato para resetear el bit de estado

; Inicializa datos:
INIDATA
        LRLK    AR1,DATOS+1
        LARK    AR0,2
        ZAC
        RPTK    0FFh
        SACL    *0+,0
        RPTK    0FFh
        SACL    *0+,0
        RPTK    0FFh
        SACL    *0+,0
        RPTK    0FFh
```

```
        SACL    *0+,0    ; parte imaginaria a 0
        LRLK    AR1,DATOS
        LRLK    AR0,1024
        RPTK    0FFh
        BLKD    BUFFER,*BR0+
        RPTK    0FFh
        BLKD    BUFFER+100h,*BR0+
        RPTK    0FFh
        BLKD    BUFFER+200h,*BR0+
        RPTK    0FFh
        BLKD    BUFFER+300h,*BR0+ ; parte real
        B       START

; R.T.I. (RUTINA TRATAMIENTO INTERRUPCION)
; Esta rutina recoge la llamada de TIM y carga un dato en memoria cada ms
; Ar6 lleva la direccion, y Ar7 el numero de datos
; Ademas los datos se introducen de forma 'ordenada' para la DCT

RTI     SST     PSST    ; guardo ST0
        SST1    PSST1   ; guardo ST1
        SACL    PACCL
        SACH    PACCH   ; guardo Acc
        LARP    AR7
        BANZ    CARGA,* ; si AR7 = 0 retorna y se desconecta (no hace EINT)
        ZALH    PACCH
        OR      PACCL   ; recupero Acc
        LST1    PSST1
        LST     PSST    ; recupero estado
        RET
CARGA
        MAR     *-,AR6  ; decrementa num. datos
        LAR     AR6,ARX
        LAC     DRR,14
        SACH    *,0     ; carga dato
        OUT     *,PA15
        LAC     *,2
        SACL    DXR
        LAC     ARX
        XORK    03FFh
        SACL    ARX
        BIT     ARX,6   ; bit 9
        BBNZ    NOZERO  ; si el bit es 1 no hay que incrementar
        LAC     ARX
        ADDK    1
        SACL    ARX
NOZERO
        ZALH    PACCH
        OR      PACCL   ; recupero Acc
        LST1    PSST1
        LST     PSST    ; recupero ST0 y ST1
        EINT
FIN     RET             ; es IRET
```

PROGRAMAS C PARA EL ALGORITMO DCT.

- **COMPLEJO.H**

```
#ifndef __COMPLEJO_H__
#define __COMPLEJO_H__

typedef struct complejo   { short real,imag; } complejo;
typedef struct complejo32 { int   real,imag; } complejo32;

#endif /* __COMPLEJO_H__ */
```

- **FFT.C**

```
/* GNU C */

#include "complejo.h"

extern short n;

void ordena (complejo *x) {          /* Ordena la secuencia de entrada */
       unsigned int i,j,pos,k;
```

```
        complejo aux;
        for (i=1;i<n-1;i++) {
                pos=0;
                k=i;
                for (j=1;j<n;j<<=1) {
                        pos=(pos<<1) | (k & 1);
                        k>>=1;
                }
                if (pos>i) {          /* Solamente hay que intercambiar elementos */
                        aux=*(x+pos);
                        *(x+pos)=*(x+i);
                        *(x+i)=aux;
                }
        }
}

void fft (complejo *x, complejo *w,char iv) {
        register unsigned lsec, i, subs;
        int locw=256;
        complejo waux;
        complejo32 p;
        /* notacion: 'lsec'    es la longitud de la secuencia
                             'locw'    para localizar las W's
                             'waux'    variable intermedia para acelerar
                             'subs'    es la subsecuencia que se esta realizando
                             'i'       variable de control de bucles
        */
        /* Se intenta emular el funcionamiento del C25. Por ello las variables
           de almacenamiento son de 16b, mientras que la del producto es de 32b.
           El formato de los numeros de entrada es Q15, con lo que -1 = 32768, y
           1 es 32767. A 'grosso modo' multiplicar por (+,-) 1 consistira en
           realizar un desplazamiento de 15b, con lo que si nos quedamos con la
           parte alta del ACC el resultado 'efectivo' sera el de dividir el
           resultado por 2.
                [0000000000000000] [xxxxxxxxxxxxxxxx] * (+,-) 1 =
                [0xxxxxxxxxxxxxxx] [x000000000000000]
        */

    (iv) ? (iv=0) : (iv=1);  /* invierte i */

        /* Para lsec=1 lo mejor es hacerlo 'a pelo' */
        for (i=0;i<n;i+=2) {
                p.real=(x+i+1)->real;
                p.imag=(x+i+1)->imag;
         (x+i+1)->real=((x+i)->real - (x+i+1)->real)>>(iv&1);
         (x+i+1)->imag=((x+i)->imag - (x+i+1)->imag)>>(iv&1);
         (x+i)->real  =((x+i)->real + p.real)>>(iv&1);
         (x+i)->imag  =((x+i)->imag + p.imag)>>(iv&1);
        }

        /* Comienzo ahora para lsec=2 */
        for (lsec=2; lsec<n; lsec<<=1) {
                for (subs=0; subs<n; subs+=(lsec<<1)) {
                        for (i=0;i<lsec;i++) {
                                waux=*(w+i*locw);
                                p.real=(int) (x+i+subs+lsec)->real*waux.real -
                                          (int) (x+i+subs+lsec)->imag*waux.imag;
                                p.imag=(int) (x+i+subs+lsec)->real*waux.imag +
                                          (int) (x+i+subs+lsec)->imag*waux.real;
                                p.real>>=15; /* Me quedo con la parte alta -1 bit */
                                p.imag>>=15;
                  (x+i+subs+lsec)->real=((int) (x+i+subs)->real-p.real)>>(iv&1);
                  (x+i+subs+lsec)->imag=((int) (x+i+subs)->imag-p.imag)>>(iv&1);
                  (x+i+subs)->real     =((int) (x+i+subs)->real+p.real)>>(iv&1);
                  (x+i+subs)->imag     =((int) (x+i+subs)->imag+p.imag)>>(iv&1);
                        }
                }
                locw>>=1;
        }
}
```

- **FDCT.C**

```
/* GNU C */

#include "complejo.h"
```

```
extern short n;

void ordenaDCT (complejo *x) {
    register i;
    complejo xt[512];

    for (i=0;i<n/2;i++) *(xt+i)=*(x+2*i+1);  /* Guardo elementos impares  */
    for (i=0;i<n/2;i++) *(x+i)=*(x+2*i);     /* 1 parte: elementos pares  */
    for (i=0;i<n/2;i++) *(x+n-1-i)=*(xt+i);  /* 2 parte: elementos impares
                                                al reves         */
}

void unordenaDCT (complejo *x) {
    register i;
    complejo xt[1024];
    int t=n/2;

    for (i=0;i<t;i++) *(xt+2*i)  =*(x+i);    /* Elementos pares   */
    for (i=0;i<t;i++) *(xt+2*i+1)=*(x+n-i);  /* Elementos impares */
    for (i=0;i<n;i++) *(x+i)     =*(xt+i);   /* Copia en x */
}

void productoDCT (complejo *x, complejo *ww) {
    register i;
       complejo32 p;

       for (i=0;i<n;i++) {
              p.real=(int) (x+i)->real*(int) (ww+i)->real -
                        (int) (x+i)->imag*(int) (ww+i)->imag;
              p.imag=(int) (x+i)->real*(int) (ww+i)->imag +
                        (int) (x+i)->imag*(int) (ww+i)->real;
        (x+i)->real=p.real>>15;
        (x+i)->imag=p.imag>>15;
       }
}
```

- **OMEGAS.C**

```
void cw (short s) {    /* Calcula el array de W's, 512. FFT. */
       register i;
       float k=2*M_PI/n;
       float temp;
       #define K  32768
       #define K1 32767

       for (i=0;i<(n>>1);i++) {
              temp=cos (i*k);
              if (temp>0) (w+i)->real=K1*temp;
              else (w+i)->real=K*temp;             /* Conversion a Q15 */
              temp=s*sin (i*k);
              if (temp>0) (w+i)->imag=K1*temp;
              else (w+i)->imag=K*temp;
       }
}

void cww (short s) {   /* COSENOS PARA LA DCT */
       register i;
       float k=M_PI/(2*n);
       float temp;
    #define K2  32768  /* 16384 */
    #define K21 32767  /* 16383 */


       for (i=0;i<n;i++) {
              temp=cos (i*k);
              if (temp>0) (ww+i)->real=K21*temp;
              else (ww+i)->imag=K2*temp;             /* Conversion a Q15 */
        temp=s*sin (i*k);
              if (temp>0) (ww+i)->imag=K21*temp;
              else (ww+i)->imag=K2*temp;
       }
}
```

A) COMANDOS DE CONTROL PARA LA TARJETA DTK - C25+.

```
#define RESETC25 outportb (BASE+2,0x81)
                                    /* 1000 0001: bit 7: memoria abierta
                                                  bit 3: c25 reset
                                                  bit 1: banco de mem. 2 */
#define C25ON    outportb (BASE+2,0x48)
                                    /* 0100 1000: bit 6, quito hold mode
                                                  bit 3, puede funcionar */
#define DATAC25  (inportb (BASE+2) & 8)
                                    /* Devuelve 1 si el C25 tiene datos
                                       listos, 0 si no */
#define ACTPAG0  outportb (BASE+2,0x88)
                                    /* 1000 1000: act. pag. 0 */
#define ACTPAG1  outportb (BASE+2,0x89)
```

B) PROGRAMAS DE COMUNICACIÓN.

- **LISTADO ENSAMBLADOR (C25).**

```
OUTPUT  MACRO
        OUT     $0,PA15         ; saco el dato
L=      IN      AUX,PA14
        LAC     AUX
        ANDK    1
        BZ      L=              ; espero senyal del host para continuar
        IN      AUX,PA10        ; reseteo bit de estado
        MACND
```

- **LISTADO C (PC).**

```
/* RUTINA QUE ESPERA Y RECOGE EL DATO DEL PUERTO DEL C25 */

    do {
        while (!DATAC25); /* ESPERO DATO */
        d=inport (BASE);  /* LEO DATO */
        fprintf (f1,"%04d\n",d); /* GUARDO EL DATO */
        outport (BASE,0); /* MANDO SENYAL AL C25 PARA QUE CONTINUE */
} while (d!=9999);  /* CODIGO DE SALIDA */


/* RUTINA QUE ESPERA SEÑAL DEL C25 PARA LEER DATOS DE LA MEMORIA (512 EN ESTE CASO) */

    while (!DATAC25);     /* ESPERO DATOS */
        d=inport (BASE);  /* RESETO BIT -> EL DATO MANDADO NO ES RELEVANTE */
        ACTPAG1;          /* PARO AL C25 Y ACTIVO PAGINA 1 */
        for (i=0;i<512;i++) *(pdr+i)=*(p+0x1000+2*i+1); /* COPIO MEMORIA */
        C25ON;            /* C25 A FUNCIONAR OTRA VEZ */
        outport (BASE,0); /* SEÑAL PARA QUE CONTINUE */
    }
```

C) PROGRAMAS DE CARGA Y VISUALIZACIÓN DE MEMORIA.

```
#include <stdio.h>
#include <dos.h>        /* para inportb/outportb, MK_FP */
#include <io.h>         /* para filelength */
#include <alloc.h>      /* para farmalloc */

#define BASE 0x338
#define MEM  0xD000
#define INT  2

/* VARIABLES GLOBALES */
unsigned far *p=MK_FP (MEM,0);   /* puntero a memoria del C25 */
unsigned pdr[512];               /* puntero a datos a representar */
FILE *f;
char pagact=0;

/* VISUALIZACIÓN DE LA MEMORIA DEL C25 */

void dispmem (unsigned ini,unsigned fin) {
```

```
    register unsigned i,j;
    char pag=0;

    if (ini<0x8000) ACTPAG0; /* activa pag. 0 */
    else {
       ACTPAG1;              /* activa pag. 1 */
       ini-=0x8000;
       fin-=0x8000;
       pag=1;
    }
    for (i=ini;i<=fin;i+=8) {
        if (pag) printf ("\n%04X: ",i+0x8000);
        else printf ("\n%04X: ",i);
        for (j=0;j<8;j++) printf ("%04X ",*(p+i+j));
    }
}

unsigned leenum (char numbyt) {
    register i;
    unsigned ret=0,it=1;
    unsigned char c;
    numbyt*=2;
    for (i=0;i<numbyt-1;i++) it*=16;
    for (i=0;i<numbyt;i++) {
        fscanf (f,"%c",&c);
        if (c>='A') { (c>='a') ? (c-=('a'-10)) : (c-=('A'-10)); }
        else c-='0';
        ret+=c*it;
        it/=16;
    }
    return (ret);
}

/* CARGA CODIGO EN MEMORIA DEL C25 */

void cargacod (char *nfich) { /* cargacod (fichero) */
    register unsigned i;
    unsigned char nd,pag=0;
    unsigned addr,d;

    f=fopen (nfich,"rt");
    if (f==NULL) {
        printf ("\nError abriendo fichero %s",nfich);
        exit (1);
    }
    fscanf (f,":");
    nd=leenum (1);
    ACTPAG0;
    while (nd) {
        addr=leenum (2);
        if (addr<0x8000) {
            if (pag) {
                ACTPAG0;
                pag=!pag;
            }
        }
        else {
            if (!pag) {
                ACTPAG1;
                pag=!pag;
            }
            addr-=0x8000;
        }
        leenum (1); /* byte de record -> inutil aqui */
        nd/=2;
        for (i=0;i<nd;i++) {
            d=leenum (2);
            *(p+addr+i)=d;
        }
        leenum (1); /* descarto el cogido de control */
        fscanf (f,"\n:");
        nd=leenum (1);
    }
    fclose (f);
}
```

BIBLIOGRAFÍA

BIBLIOGRAFÍA.

[1] "Biomedical Digital Signal Processing". W. J. Tompkins Ed. Prentice Hall. (1992).

[2] "Introduction to Digital Signal Processing". J. Proakis, D. Manolakis. MacMillan Publishing Company. (1988)

[3] "Introductory Digital Signal Processing with Computer Applications". P. Lynn, W. Fuerst. Prentice-Hall. (1989)

[4] “Digital Signal Processing with the TMS320C25”. Rulph Chassaing. Wiley-Interscience, 1992.

[5] “Digital Signal Processing in VLSI”. R. Higgins. Prentice Hall, 1990.

[6] “Second Generation TMS320”: User's Guide. Texas Instruments, 1989.

[7] “Digital Signal Processing Applications with the TMS320 Family”, vol. I. Texas Instruments, 1988.

[8] “Digital Signal Processing Applications with the TMS320 Family”, vol. III. Texas Instruments, 1990.

[9] “TMS320C2x User’s Guide”. Texas Instruments, 1993.

[10] “TMS320C1x / TMS320C2x Assembly Language Tools”. Texas Instruments, 1989.

[11] “ECG Data Compression Techniques - A Unified Approach”. Sateh M.S. Jalaleddine, Chiswell G. Hutchens, Rober D. Strattan, Willian A. Coberly. IEEE Transactions on Biomedical Engineering, vol. 37, no 4, abril 1990.

[12] “System Overview and DSP Developer’s Toolkit for the Texas Instruments TMS320C25-based PC-C25 Attached Processor Board”. Ariel Corporation, 1992.

[13] “Recursive algorithm for the discrete cosine transform with general lengths”. L. P. Chau and W. C. Siu. Electronic Letters, febrero 1994, vol. 3.

[14] “Text compression using rule based encoder”. H.U. Khan and H.A. Fatmi. Electronic Letters, febrero 1994, vol. 30, no. 3.

[15] “ECG Data Compression by sub-hand coding”. M.C. Aydin, A. E. Çetin, H. Köymen. Electronic Letters, febrero 1991, vol. 27, no.4.

[16] “Data Compression of ECG’s by High-Degree Polynomial Approximation”. Wilfried Philips and Geert De Jonghe. IEEE Transactions on Biomedical Engineering, vol. 39, no. 4, abril 1992.

[17] “ECG Data Compression with Time-Warped Polynomials”. Wilfried Philips. IEEE Transactions on Biomedical Engineering, vol. 40, no. 11, noviembre 1993.

[18] “ECG Data Compression by Modelling”. Budagavi Madhukar and I.S.N. Murthy. Computer and Biomedical Research 26, 310-317 (1993).

[19] “Analysis of ECG from Pole-Zero Models”. I.S.N. Murthy and G. S. S. Durga Prasad. IEEE Transactions on Biomedical Engineering, vol. 39, no. 7, julio 1992.

[20] “ECG Data Compression Using Fourier Descriptors”. B. R. Shankara Reddy and I. S. N. Murthy. IEEE Transactions on Biomedical Engineering, vol. BME-33, no.4, abril 1986.

[21] “Compression of the ECG by Prediction or Interpolation and Entropy Encoding”. URS E. Ruttimann and Hubert V. Pipberger. IEEE Transactions on Biomedical Engineering, vol. BME-26, no. 11, noviembre 1979.

[22] “Análisis automático de datos en un sistema Holter basado en PC”. Juan F. Guerrero Martínez , Juan J. Martinez Durá, José Espí López y Manuel Bataller Mompeán. Mundo Electrónico, marzo 1991.

[23] “Sistema para la obtención de ECG de alta resolución”. Juan F. Guerrero Martínez, José Espí Lópes, Manuel Bataller Monpeán, Vicente Cavero Millán, Salvador Bayarri Romar. Mundo Electrónico, abril 1993.

[24] "Instrumentación y medidas biomédicas". L. Cromwell, F. Weibell, E. Pfeiffer, L. Usselman. Marcombo. (1980).

[25] "Principles of Bioinstrumentation". R. Norman. Wiley, (1988)

[26] "The Art of Electronics". P. Horowitz, W. Hill. Cambridge University Press. (1989).

[27] “Computer Organization and Architecture”. W. Stallings. Macmillan Inc., 1987.

[28] “The C Programming Language”. B. Kernighan, D. Ritchie. Prentice-Hall, 1978.

[29] “AZTEC, a pre-processing program for real-time ECG rhythm analysis”. J.R. Cox, F.M. Nolle, H.A. Fozzard, G. C. Oliver. IEEE Transactions on Biomedical Engineering, vol. BME 15, 1968.

[30] “Scan along polygon approximation for data compression of electrocardiograms”. M. Ishijima, S.B. Shin, G.H. Hostetter and J. Sklansky. IEEE Trans. Biomed. Eng. Vol. BME-30, pp. 723-729, Nov. 1983.

[31] “Comments on Compression of the ECG by prediction or interpolation and entropy encoding”. P. Borjesson, G. Einarsson, O. Pahlm. IEEE Trans. Biomed. Eng. Vol. BME-27, Nov. 1980.

[32] “ECG data compression by linear prediction”. U.E. Ruttiman, A.S. Berson and H.V. Pipherger. Computers Cardiol.. MO 1976.