UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

BANCO DE FILTROS PARA CODIFICADOR DE AUDIO

T E S I S

QUE PARA OBTENER EL TITULO DE:

I N G E N I E R O E N T E L E C O M U N I C A C I O N E S

P R E S E N T A N:

JAIME GONZALEZ MENDEZ

ALBERTO BARRERA LOZANO

DIRECTOR DE TESIS:

DR. BOHUMIL PSENICKA

Cd. Universitaria, México D.F. Junio 2009

A todas aquellas personas que saben son importantes para mí.

Jaime González Méndez

A mi familia.

Alberto Barrera Lozano

Agradecimientos

A mi padre por ser mi ejemplo de superación.
A mi madre, por apoyarme, motivarme siempre, por su infinito amor y compresión.
A mi hermana, por su apoyo y su voto de confianza.
A Jaime, compañero de vida y amigo sincero que siempre estuvo ahí para apoyarme.
A mi profesor Bohumil, por su orientacin para el desarrollo de la presente.
A los profesores, amigos y colegas de la universidad, por los momentos inolvidables que hemos vivido.
A la UNAM, por ayudar a mi formacin personal y profesional.
Finalmente, a todas las personas que se cruzaron en este camino y que me dieron palabras de aliento y apoyo.

Alberto Barrera Lozano

Es difícil poder agradecer en unas líneas a todos aquellos que han contribuido en mi formación profesional y en este trabajo, fruto de la primera. Sin embargo, con la intención de no omitir a nadie, a continuación agradezco de la manera más sincera:

A mis padres, por ser ejemplo de vida, por demostrarme que luchando se pueden lograr muchas cosas, por supuesto, bien hechas.
A mis hermanos, por hacerme ver la vida de diferentes perspectivas y no sólo de la mía.
A mi tía Mónica y a su familia, por su apoyo prácticamente incondicional desde mi arribo a esta ciudad y durante estos últimos cinco años.
Al resto de mi familia, por eso, una familia, de las que apesar de los malos momentos, siempre están ahi apoyándome y apoyándonos entre nosotros.
A mis amigos, por sus calidez, amistad y sobre todo por su confianza.
A Alberto, por ser compañero y amigo en esta carrera.
A Oscar, por todos aquellos buenos momentos que pasamos juntos.
A mis profesores, por su enseñaza, dedicación y por sus consejos y confianza.
Al Dr. Bohumil Psenicka, por la oportunidad brindada, y por su disponibilidad para que este trabajo llegara a buen fin.
A la Universidad Nacional Autónoma de México, mi Alma Mater, por formarme como ingeniero y como persona y por darme la oportunidad de ser parte de ella, de ser orgullosamente universitario y ser puma de corazón.

Por todo eso, y mucho más, ¡Muchas Gracias!

Jaime González Méndez

Índice general

Capítulo 1

Introducción

En los últimos 20 años, se han presentado grandes avances en el campo de los bancos de filtros y los sistemas multitasa. Estos sistemas ofrecen nuevas y eficaces formas de representar señales para tener un fácil manejo, procesamiento y compresión. Los bancos de filtros encuentran aplicaciones en prácticamente todo el campo de procesamiento de señales. Por lo tanto, es de extrema importancia la capacidad de diseñar un banco de filtros ya que se pueden aprovechar plenamente las propiedades y la naturaleza de una clase particular de señales o aplicaciones.

En muchas aplicaciones, la señal digital obtenida por un convertidor analógico digital (ADC) de una señal analógica tiene que ser transmitida por un canal con banda limitada o almacenada en un medio con capacidad limitada. Debido a lo anterior, los métodos de compresión de señales han estado siendo utilizados de manera creciente para incrementar la eficiencia de transmisión o almacenamiento. Uno de los esquemas más populares para la compresión de señales es la codificación de subbanda, la cual emplea banco de filtros y codifica eficientemente la señal explotando la distribución uniforme de la energía de la señal en el dominio de la frecuencia.

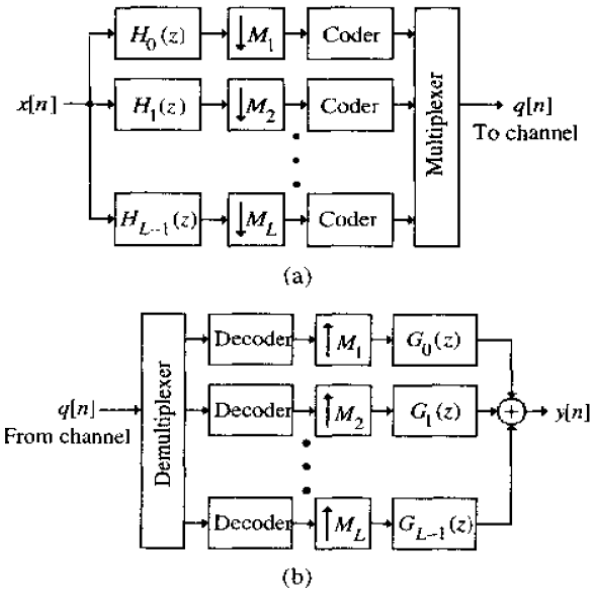


Figura 1.1: Estructura de codificación subbanda

Una representación simplificada del esquema de codificación de subbanda se muestra en la figura 1.1 como se indica ahi, la señal de entrada $x[n]$ es descompuesta en un conjunto de señales de banda reducida que ocupan bandas de frecuencia continuas. Estas señales de banda reducida son después submuestreadas reduciendo entonces las señales subbanda. Cada señal subbanda es después comprimida por un codificador y luego todas las señales subbanda comprimidas son multiplexadas y enviadas sobre el canal de comunicaciones al receptor. En el receptor, la señal compuesta es primero demultiplexada en un conjunto de señales subbanda. Cada señal subbanda, en este caso es decomprimida por un decodificador, sobremuestreada y introducida al banco de síntesis. Las salidas de banco de síntesis son combinadas para producir una señal $y[n]$ que es una réplica aceptable de la señal de entrada original $x[n]$.

La codificación en subbanda ofrece una mejor compresión que la compresión directa de la señal original $x[n]$, ya que asigna diferente número de bits a cada señal de subbanda, tomando así, ventaja de sus características espectrales, lo que resulta en una baja tasa de bits por muestra.

La codificación en subbanda para audio basada en banco de filros ha sido estudiada por algunos investigadores entre ellos E.B. Richardson y N.S. Jayant. En este trabajo se diseña e implementa un banco de filtros de 4 canales con reconstrucción perfecta que pueda ser utilizada para audio.

1.1. Objetivo de la tesis

El objetivo del presente trabajo es diseñar e implementar con el DSP de *Texas Instruments* TMS320C6713 un banco de filtros de 4 canales para la codificación en subbanda de audio en tiempo real. Dicho objetivo solo se restringe al diseño de implementación del banco, no a la codificación propia de las señales de subbanda, lo cual está fuera de los límites de este trabajo.

1.2. Estructura de la tesis

A lo largo de esta tesis se explicarán las bases teóricas y el procedimiento utilizado para obtener un banco de filtros para ser empleado en una aplicación en tiempo real para un codificador de audio. Esta tesis está dividida en 9 capitulos y dos apéndice. El capitulo 1 es propiamente la introducción del presente trabajo.

Las funciones de transferencia del banco de filtros son obtenidas de acuerdo al procedimiento descrito en el capitulo 2. En este capitulo se pueden encontrar las diferentes maneras de calcular las funciones de transferencia de filtros digitales a partir de funciones de transferencia de filtros analógicos.

Así mismo en los capitulos 3 y 4 se describen los dos tipos de funciones de transferencia de los filtros que se pueden utilizar y sus características, lo cual resulta de crucial importancia revisar previamente para determinar cuales son las más apropiadas para obtener resultados satisfactorios durante la aplicación. Debido a que se necesita tener una aplicación en tiempo real, se utilizará ayuda de sistemas computacionales, es importante verificar cual de los tipos de filtros es más conveniente utilizar.

Cuando se habla de decimación e interpolación se introducen conceptos muy importantes en el campo de los bancos de filtros pues estos conceptos representan una de las principales ventajas al trabajar con bancos de filtros, en el capitulo 5 se explica todo lo relacionado con los sistemas multitasa, su funcionamiento y la forma en que ayudaran en la transmisión de las señales.

En la etapa de síntesis, las subbandas son combinadas por medio de un conjunto de sobremuestreadores y M filtros de síntesis $F_i(z)$ para así obtener una señal reconstruida $x'[n]$. Asumiendo que no hay pérdida de información en la etapa de procesamiento, los bancos de filtros producen una salida $x'[n]$ como una versión retrasada de la señal de entrada $x[n]$, entonces $x'[n] = x[n - n_0]$, esto es lo que llamamos reconstrucción perfecta. La propiedad de reconstrucción perfecta es altamente deseable ya que provee una representación sin pérdidas de la señal lo que simplifica el error de análisis significativamente.

En numerosas aplicaciones, especialmente en el procesamiento de voz, imágenes y video, es crucial que los filtros de análisis y síntesis tengan una fase lineal. Además de la eliminación de la distorsión de fase, los sistemas de fase lineal nos permiten utilizar métodos de extensión de simetría simple para manejar con precisión los límites de las señales de longitud finita. Así mismo, la propiedad de linealidad de fase puede ser explotada para la implementación más rápida y eficaz de los bancos de filtros.

En el capitulo 7 se explica más a fondo el concepto de banco de filtros y sus diferentes representaciones, se comienzá con bancos de filtros de 2 canales que es lo más básico que se utiliza y servirá para comprender aun más su funcionamiento, de esta forma se desarrolla el procedimento basado en diferentes teoremas para construir bancos de filtros de L canales y las condiciones que deben cumplir las funciones de transferencias para obtener una reconstrucción perfecta de la señal.

El capitulo más importante es el número 8, debido a que en él se presenta propiamente el trabajo de investigación, objetivo de este trabajo. Es descrito con gran detalle el proceso de diseño simulación e implementación de banco de filtros para codificador de audio.

Finalmente en el capitulo 9 se presentan las conclusiones a las que se llegó en base a los resultados obtenidos en esta tesis y la utilización que ésta puede tener en el futuro.

Capítulo 2

Transformación bilineal

La trasformación bilineal es un método algebraico entre las variables s y z, que mapea todo el eje $j\Omega$ del plano s analógico a una revolución sobre el perímetro del círculo unitario en el plano z digital. La conversión mapea los polos y ceros analógicos en polos y ceros digitales, donde un punto del plano s es mapeado a un único punto del plano z. Desde $-\infty \leq \Omega \leq \infty$ mapea sobre $-\pi \leq \omega \leq \pi$ [7]. Esta transformación entre las variables de tiempo continuo y tiempo discreto frecuentemente será no lineal. Por tanto el uso de esta técnica se restringue donde la correspondiente deformación del eje de la frecuencia es aceptable.

2.1. La Transformada Bilineal

$H_c(s)$ denota la función de transferencia del sistema en tiempo continuo y $H(z)$ la función del sistema en tiempo discreto, la transformación bilineal corresponde a cambiar s por

$$s = \frac{2}{T_d}\left(\frac{1-z^{-1}}{1+z^{-1}}\right), \tag{2.1}$$

esto es

$$H(z) = H_c\left[\frac{2}{T_d}\left(\frac{1-z^{-1}}{1+z^{-1}}\right)\right]. \tag{2.2}$$

Como resultado de la invarianza al impulso un parámetro de muestreo T_d es incluído en la definición de la transformación bilineal. En el diseño de filtros, el uso de la transformación bilineal está basado en las propiedades de la transformación algebraica de la ecuación (2.1). El parámetro T_d no tiene consecuencia en el procedimiento de diseño, ya que se asume que el problema de diseño siempre comienza con las especificaciones del filtro en tiempo discreto $H(e^{jw})$. Donde estas especificaciones son mapeadas a las especificaciones de tiempo continuo, entonces el filtro de tiempo continuo es mapeado de nuevo a un filtro de tiempo discreto, de esta forma el efecto de T_d se cancelará. Por lo tanto el valor del parámetro T_d en problemas específicos y ejemplos puede ser escogido según convenga.

Para el desarrollo de las propiedades de la transformación algebraica de la ecuación (2.1), se resuelve para z y así se obtiene

$$z = \frac{1+(T_d/2)s}{1-(T_d/2)s} \tag{2.3}$$

y si se sustituye $s = \sigma + j\Omega$ en la ecuación(2.3),

$$z = \frac{1 + \sigma T_d/2 + j\Omega T_d/2}{1 - \sigma T_d/2 - j\Omega T_d/2} \tag{2.4}$$

Si $\sigma < 0$, entonces de la ecuación(2.4) se obtiene que para cualquier valor de Ω se tendrá que $\mid z \mid < 1$. Similarmente, si $\sigma > 0$, entonces para todo Ω se tendrá que $\mid z \mid > 1$. Es decir si un polo de $H_c(s)$ está en la mitad izquierda del plano s su imagen en el plano z estará dentro del circulo unitario. Esta condición es necesaria para preservar las propiedades de estabilidad de los filtros analógicos. En otras palabras, se necesita un procedimiento de mapeo que transforme los filtros analógicos estables en filtros digitales estables.

Ahora para mostrar que el eje $j\Omega$ del plano s mapea dentro sobre en circulo unitario, se sustituye $s = j\Omega$ en la ecuación(2.3)

$$z = \frac{1 + j\Omega T_d/2}{1 - j\Omega T_d/2} \tag{2.5}$$

En la ecuación(2.5) es claro que $\mid z \mid = 1$ para todos los valores de s en el eje $j\Omega$. Es decir, el eje $j\Omega$ mapea sobre el circulo unitario, por lo que la ecuación(2.5) toma la siguiente forma

$$e^{j\omega} = \frac{1 + j\Omega T_d/2}{1 - j\Omega T_d/2} \tag{2.6}$$

Para obtener una relación entre las variable ω y Ω, en la ecuación (2.1) se sustituye $z = e^{j\omega}$:

$$s = \frac{2}{T_d}\left(\frac{1 - e^{-j\omega}}{1 + e^{-j\omega}}\right), \tag{2.7}$$

o equivalentemente,

$$s = \sigma + j\Omega = \frac{2}{T_d}\left[\frac{2e^{-j\omega/2}(jsin\omega/2)}{2e^{-j\omega/2}(cos\omega/2)}\right] = \frac{2j}{T_d}tan(\omega/2). \tag{2.8}$$

Comparando la parte real e imaginaria de ambos lados de la ecuación (2.8) se obtienen las siguientes relaciones para $\sigma = 0$

$$\Omega = \frac{2}{T_d}tan(\omega/2), \tag{2.9}$$

$$\omega = 2arctan(\Omega T_d/2). \tag{2.10}$$

Las propiedades de la transformación bilineal como un mapeo del plano s sobre el plano z son resumidas en las figuras 2.1 y 2.2. De la ecuación (2.10) y la figura 2.2, se observa que el rango de frecuencias $0 \leq \Omega \leq \infty$ mapea sobre $0 \leq \omega \leq \pi$ mientras que el rango $-\infty \leq \Omega \leq 0$ mapea sobre $-\pi \leq \omega \leq 0$.

La transformación bilineal evita el problema del traslape encontrado a partir del uso de la invarianza al impulso, porque mapea todo el eje imaginario del plano s sobre el circulo unitario en el plano z. El precio a pagar por esto es la compresión no lineal del eje de frecuencias representado en la figura

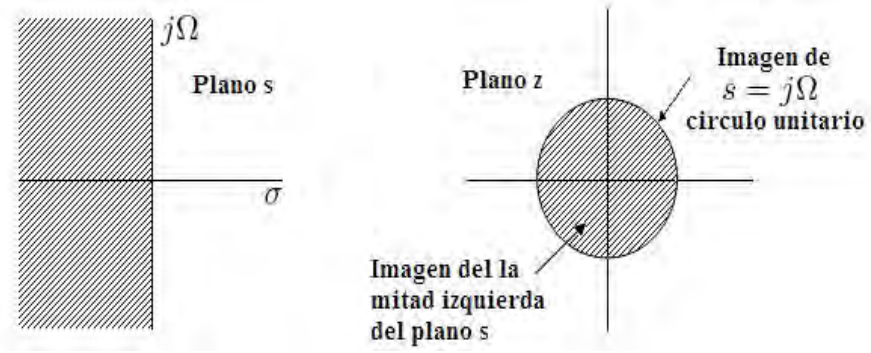


Figura 2.1: Mapeo del plano s sobre el plano z usando la transformación bilineal.

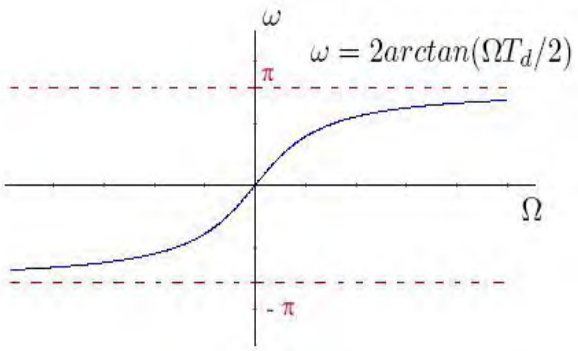


Figura 2.2: Mapeo del eje de frecuencia en tiempo continuo sobre el circulo unitario por transformación bilineal.

2.2. Consecuentemente el diseño de filtros en tiempo discreto usando la transformación bilineal es útil sólo cuando la compresión puede ser tolerada o compensada, como en el caso de filtros que tienen una magnitud constante aproximada a la ideal.

Por ejemplo, para el diseño de un filtro paso bajas, se requiere de una aproximación de las características del filtro paso bajas ideal mostradas en la figura 2.3. Si ha diseñado un filtro ideal paso bajas en tiempo continuo con una frecuencia de corte $\Omega_c = (2/T_d)tan(\omega_c/2)$ y mapeado al plano z, lo cual implica usar la transformada bilineal, la respuesta en frecuencia ideal resultaría en lo mostrado en la figura 2.3.

Se sabe que tanto para el caso de tiempo continuo y el caso de tiempo discreto es imposible realizar un filtro con estas características. En general se tiene una respuesta en frecuencia aproximada, permitiendo una pequeña desviación de la unidad en la banda de paso y una pequeña desviación del cero en la banda supresora, con una banda de transición de un ancho diferente de cero. La figura 2.4 representa el mapeo de la respuesta en frecuencia en tiempo continuo y el régimen de tolerancia correspondiente de la respuesta en frecuencia en tiempo discreto [7] .

Si las frecuencias críticas del filtro en tiempo continuo son predeformadas de acuerdo a la ecuación (2.11)

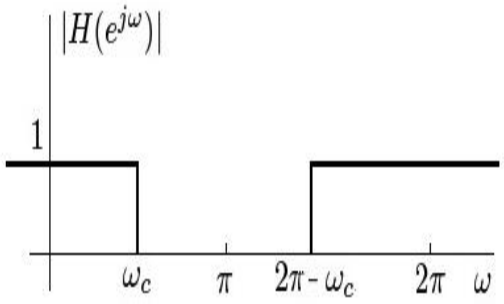


Figura 2.3: Respuesta en frecuencia del un filtro pasobajas ideal.

$$\Omega = \frac{2}{T_d} tan(\omega/2). \tag{2.11}$$

entonces cuando el filtro en tiempo continuo es transformado a un filtro de tiempo discreto usando la ecuación (2.2), el filtro en tiempo discreto tendrá las especificaciones deseadas.

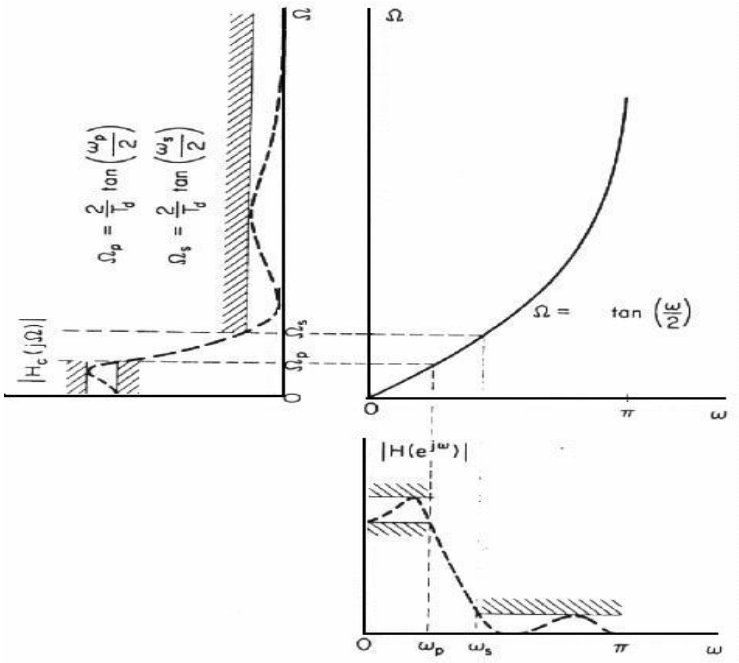


Figura 2.4: Mapeo de la frecuencia analógica con respecto a la frecuencia digital.

La transformación bilineal es uno de los métodos más simples para el diseño de los filtros IIR, el método consiste en:

⋆ Especificar los requerimientos en términos de la frecuencia analógica en Hz, valores de atenuación y la frecuencia de muestreo.

⋆ Transformar las especificaciones a un filtro analógico equivalente.

⋆ Diseñar el filtro analógico $H_c(s)$.

⋆ Utilizar la transformación bilineal para obtener el filtro digital $H(z)$.

La figura 2.1, muestra las etapas en el diseño de filtros IIR usando la transformación bilineal.

Figura 2.5: Diagrama de bloques del proceso de diseño de un filtro IIR con el uso de la transformación bilineal.

2.2. Uso de la Matriz de Pascal

Un gran número de procedimientos están disponibles para el diseño de filtros digitales, muchas de esas transformaciones ayudan a convertir filtros analógicos a filtros digitales equivalentes. La función de transferencia de un filtro analógico está representada por la siguiente ecuación.

$$H(s) = \frac{B_0 + B_1 s + B_2 s^2 + \cdots + B_m s^m}{A_0 + A_1 s + A_2 s^2 + \cdots + A_m s^m} \tag{2.12}$$

De la ecuación (2.12) se puede obtener los vectores

$$\begin{aligned} B &= (B_0, B_1, B_2, \cdots, B_m) \\ A &= (A_0, A_1, A_2, \cdots, A_m) \end{aligned} \tag{2.13}$$

Donde B_i y A_i son coeficientes reales.

Por otro lado, la función de transferencia que caracteriza a un filtro digital en el dominio z es:

$$H(z) = \frac{b_o + b_1 z + b_2 z^2 + \cdots + b_n z^n}{a_o + a_1 z + a_2 z^2 + \cdots + a_n z^n} \tag{2.14}$$

De igual forma se puede obtener los vectores:

$$\begin{aligned} b &= (b_0, b_1, b_2, \cdots, b_n) \\ a &= (a_0, a_1, a_2, \cdots, a_n) \end{aligned} \tag{2.15}$$

Con coeficientes reales a_i y b_i

2.2.1. Transformación paso bajas a paso bajas

Para filtros paso bajas la función de transferencia digital H(z) puede ser obtenida del prototipo en tiempo real (2.12) usando la transformada bilineal.

$$s = C\frac{1 - z^{-1}}{1 + z^{-1}} \tag{2.16}$$

Donde

$$C = cot\left(\frac{\pi f_1}{f_m}\right) \tag{2.17}$$

Y las constantes f_1 y f_m representan la frecuencia de corte y la frecuencia de muestreo respectivamente

Si se considera $m = 2$ y $n = 2$, se puede obtener una relación matricial entre los ventores (2.13) y (2.15) aplicando la transformación (2.16) en (2.12) para generar (2.14). La relación encontrada es la siguiente:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & -2 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_0 \\ A_1C \\ A_2C^2 \end{bmatrix} \tag{2.18}$$

De la misma manera se puede obtener una matriz similar para los coeficientes de b_i

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & -2 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1C \\ B_2C^2 \end{bmatrix} \tag{2.19}$$

En este caso de las ecuaciones matriciales (2.18) y (2.19) se obtiene lo siguiente:

$$a = P_{lP}^{n} A' \tag{2.20}$$

Donde P_{LP}^{n} es la matriz de Pascal paso bajas y los vectores A' y B' están representados por:

$$\begin{aligned} B' &= (B_0, B_1C, B_2C^2, \cdots, B_mC^m) \\ A' &= (A_0, A_1C, A_2C^2, \cdots, A_mC^m) \end{aligned} \tag{2.21}$$

Como ha demostrado [8] el cálculo de la matriz P_{lP}^{n} se puede realizar de una forma sistemática. Para ello se considera el triángulo de Pascal.

Observe que los coeficientes del renglón $n = 2$ crean la última columna de la matriz de Pascal paso bajas (2.18) y (2.19) con excepción de que el renglón par es negativo. Así de esta forma se puede generar un procedimiento para encontrar los valores de la matriz de Pascal.

1. En el primer renglón de la matriz de Pascal, todos los elementos deben ser igual a uno.

Si n = 2

$$P_{lp}^{n} = \begin{bmatrix} 1 & 1 & 1 \\ & & \\ & & \end{bmatrix} \tag{2.22}$$

$$
\begin{array}{ccccccccc}
 & & & 1 & & & & & n=0 \\
 & & 1 & & 1 & & & & n=1 \\
 & 1 & & 2 & & 1 & & & n=2 \\
1 & & 3 & & 3 & & 1 & & n=3 \\
 & 1 & & 4 & & 6 & & 4 & 1 \quad n=4 \\
1 & & 5 & & 10 & & 10 & & 5 \quad 1 \quad n=5
\end{array}
$$

Figura 2.6: Triángulo de Pascal

2. Los elementos de la última columna pueden ser calculados de la siguiente forma:

$$p_{i,m+1} = (-1)^{i-1} \frac{n!}{(n-i+1)!(i-1)!}, \tag{2.23}$$

donde $i = 1, 2, \cdots, n+1$

$$P_{lp}^{n} = \begin{bmatrix} 1 & 1 & 1 \\ & & -2 \\ & & 1 \end{bmatrix} \tag{2.24}$$

3. Los elementos restantes, $p_{i,j}$ se pueden obtener usando la siguiente ecuación

$$p_{i,j} = p_{i-1,j} + p_{i-1,j+1} + p_{i,j+1}, \tag{2.25}$$

donde $i = 2, \cdots, n+1$; $j = n, n-1, n-2, \cdots, 2, 1$

$$\overline{P_{lp}^{n}} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & -2 \\ 1 & -1 & 1 \end{bmatrix}$$

2.2.2. Transformación paso bajas a paso altas

En este segundo caso, para transformar la función de transferencia paso bajas a una función de transferencia digital paso altas H(z), se sustituye la variable s por $1/s$, de esta forma se obtiene la transformación bilineal de la siguiente forma:

$$s = K \frac{1 + z^{-1}}{1 - z^{-1}} \tag{2.26}$$

con

$$K = tan\left(\frac{\pi f_c}{f_m}\right) \tag{2.27}$$

Donde f_c representa la frecuencia de corte paso altas y f_m representa la frecuencia de muestreo. Siguiendo el mismo procedimiento para n=2 se obtiene:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & 0 & 2 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_0 \\ A_1K \\ A_2K^2 \end{bmatrix} \tag{2.28}$$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & 0 & 2 \\ 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} B_0 \\ B_1K \\ B_2K^2 \end{bmatrix} \tag{2.29}$$

Por lo que se pueden escribir las ecuaciones (2.28) y (2.29) como

$$\begin{aligned} a &= P^n_{HP} x A'' \\ b &= P^n_{HP} x B'' \end{aligned} \tag{2.30}$$

Donde P^n_{HP} es la variante de la matriz de Pascal que corresponde al filtro paso altas. El procedimiento para obtener la matriz de Pascal paso altas es el siguiente:

1. Todos los elementos del primer renglón son unos.

Si n=2

$$P^n_{HP} = \begin{bmatrix} 1 & 1 & 1 \\ & & \\ & & \end{bmatrix} \tag{2.31}$$

2. Los elementos de la primera columna pueden ser calculados como lo indica la ecuación(2.23)

$$P^n_{HP} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & & \\ 1 & & \end{bmatrix} \tag{2.32}$$

3. Los elementos restantes, $p_{i,j}$ se pueden obtener usando la siguiente ecuación

$$p_{i,j} = p_{i,j-1} + p_{i-1,j-1} + p_{i-1,j}, \tag{2.33}$$

donde $i = 2, \cdots, n+1$; $j = 2, \cdots, n+1$

$$\overline{P^n_{HP}} = \begin{bmatrix} 1 & 1 & 1 \\ -2 & 0 & 2 \\ 1 & -1 & 1 \end{bmatrix}$$

2.2.3. Transformación paso bajas a paso banda

Basados en [9], el filtro paso banda se puede obtener con la superposición de un filtro paso bajas y un filtro paso altas, así la transformación s a z queda de la siguiente forma:

$$s = \left[C\frac{1-z^{-1}}{1+z^{-1}} + K\frac{1+z^{-1}}{1-z^{-1}} \right] m \tag{2.34}$$

donde

$$C = cot\left(\frac{\pi f_1}{f_m}\right), K = tan\left(\frac{\pi f_{-1}}{f_m}\right), m = \frac{cot\frac{\pi(f_1 - f_{-1})}{fm}}{k+c} \tag{2.35}$$

Donde f_1 y f_{-1} representa la frecuencia de corte superior y la frecuencia de corte inferior, respectivamente y f_m representa la frecuencia de muestreo. De la misma forma que en los procedimientos pasados se pueden encontrar los vectores a y b a partir de los vectores de coeficientes de la función de transferencia analógica A y B . Por ejemplo para $m = 2$ (analógico) se obtendrá un $n = 4$ (digital) de la siguiente forma:

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -4 & -2 & 0 & 2 & 4 & 0 \\ 6 & 0 & -2 & 0 & 6 & -2 \\ -4 & 2 & 0 & -2 & 4 & 0 \\ 1 & -1 & 1 & -1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_2C^2m^2 \\ A_1Cm \\ A_0 \\ A_1Km \\ A_2K^2m^2 \\ 2A_2CKm^2 \end{bmatrix} \tag{2.36}$$

La matriz P^n_{BP} no es cuadrada, entonces no es posible calcular matriz inversa y hacer la transformación z^{-s}. Por lo que la ecuación (2.36) se puede escribir como

$$a = P^n_{lP} \cdot A'' \tag{2.37}$$

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -4 & -2 & 0 & 2 & 4 \\ 6 & 0 & -2 & 0 & 6 \\ -4 & 2 & 0 & -2 & 4 \\ 1 & -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_2C^2m^2 \\ A_1Cm \\ A_0 + 2A_2CKm^2 \\ A_1Km \\ A_2K^2m^2 \end{bmatrix} \tag{2.38}$$

La ventaja de esta representación es que la matriz es cuadrada y se puede calcular la matriz inversa

$$(P^n_{lp})^{-1} = \frac{1}{n}P^n_{lp}$$

De igual manera P^n_{BP} se puede obtener con la concatenación de dos matrices como se muestra la siguiente ecuación:

$$[P^n_{BP}] = [S^n_{BP} \mid R^n_{BP}] \tag{2.39}$$

Donde S^n_{BP} es una matriz cuadrada y es calculada exactamente de la misma forma que en la transformación de paso bajas a paso altas (2.31).

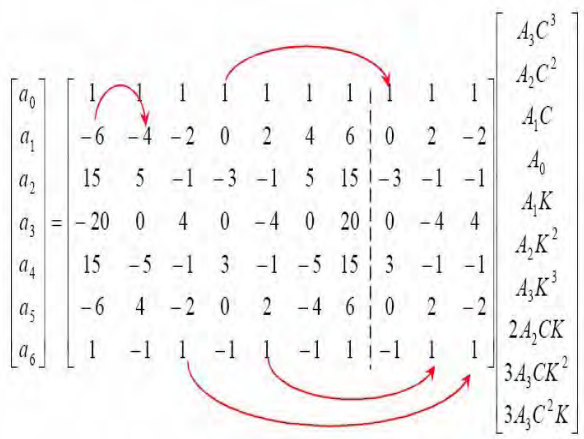
Por otro lado la matriz R^n_{BP} en (2.39) es una matriz rectangular con $n+1$ renglones. A priori el número de columnas han sido calculados por medio de contar el número de elementos diferentes de

M	N	Col
2	4	1
3	6	3
4	8	6

Cuadro 2.1: Número de columnas de la matriz R^n_{BP}

1, incluido en el triángulo superior de Pascal. Por ejemplo, la siguiente tabla muestra el número de columnas de la matriz R^n_{BP} para diferentes valores de m y n

Por ejemplo, para $m = 2$ y $n = 4$ se tiene que R^n_{BP} tiene una columna, que es precisamente la columna central de S^n_{BP} . Otro ejemplo es para $m = 3$ y $n = 6$, los números diferentes de uno en el triángulo de Pascal para $n = 6$ son tres, por lo que el número de columnas en R^n_{BP} es 3, que son la columna central (pivote) de S^n_{BP} , la siguiente es la columna contigua que se encuentra a la derecha del pivote y finalmente la última columna es la que se encuentra contigua al pivote pero del lado izquierdo. Por ejemplo para obtener los coeficientes a del filtro digital paso banda con $m = 3$ y $n = 6$, se obtiene la siguiente ecuación matricial.



Para el caso del vector b, el procedimiento se realiza de igual manera.

La matriz anterior se puede escribir de la siguiente manera

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -6 & -4 & -2 & 6 & 2 & 4 & 6 \\ 15 & 5 & -1 & -3 & -1 & 5 & 15 \\ -20 & 0 & 4 & 0 & -4 & 0 & 20 \\ 15 & -5 & -1 & 3 & -1 & -5 & 15 \\ -6 & 4 & -2 & 0 & 2 & -4 & 6 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} A_3C^3m^3 \\ A_2C^2m^2 \\ A_1Cm + 3A_3C^2Km^3 \\ A_0 + 2A_2CKm^2 \\ A_1Km + 3A_3CK^2m^3 \\ A_2K^2m^2 \\ A_3K^3m^3 \end{bmatrix} \tag{2.40}$$

Donde

$$a = P^n_{lP} \cdot A''$$

$$A'' = (P^n_{lp})^{-1} a = \frac{1}{n} P^n_{lp} A'' \tag{2.41}$$

De esta forma se calcula la transformación inversa z^{-s}. Si se conoce $H(z)$, mediante la ecuación(2.41) se calcula $H(s)$.

Capítulo 3

Filtros con respuesta finita al impulso unitario

Como su nombre lo indica, los filtros FIR presentan una respuesta finita al impulso, es decir, se encuentran limitados por el número de términos de salida al aplicarse un impulso unitario en su entrada.
La salida de un filtro FIR depende únicamente del presente y pasado de las entradas. Esta característica es de vital importancia para el diseño e implementación de esta clase de filtros.
La función de transferencia de los filtros con respuesta finita al impulso (FIR) tiene la siguiente forma

$$H(z) = \frac{Y(z)}{X(z)} = b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} \cdots + b_k z^{-k} = \sum_{i=0}^{k} b_i z^{-i} \tag{3.1}$$

Si se aplica la transformada z inversa a la ecuación (3.1), se obtiene la ecuación en diferencias que describe a los filtros FIR

$$y(n) = \sum_{i=0}^{k} b_i x(n-i) \tag{3.2}$$

3.1. Filtro FIR en la forma Transversal

A partir de esta función, ecuación (3.2), se puede dibujar la estructura directa de los filtros FIR que se muestra en la figura 3.1. A esta estructura se le denomina Estructura Transveral. En la figura 3.2 se presenta el filtro FIR en su primera estructura canónica

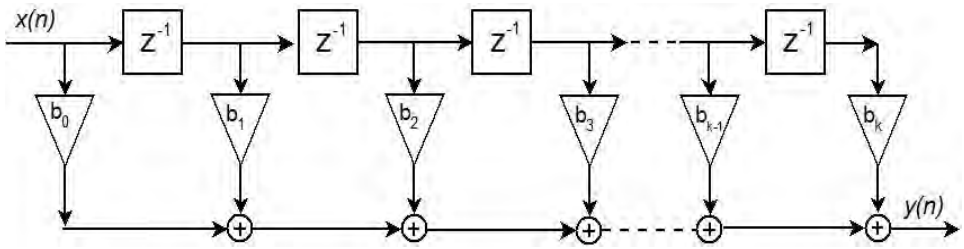


Figura 3.1: Estructura directa del filtro FIR

De la ecuación (3.2) se observa que los coeficientes de la ecuación son al mismo tiempo las muestras de la respuesta al impulso.

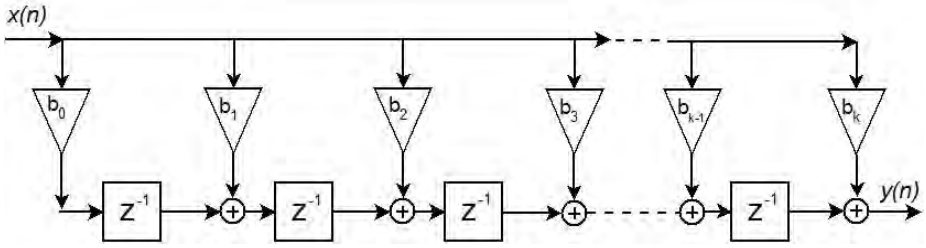


Figura 3.2: Primera estructura canónica de filtro FIR

3.2. Filtro FIR en la forma de Cruz

Las estructuras de cruz fueron usadas por primera vez en modelos de señales autorregresivas. Al utilizarlas con banco de filtros, su más frecuente aplicación es en la realización de robustos sistemas con reconstrucción perfecta.[1]

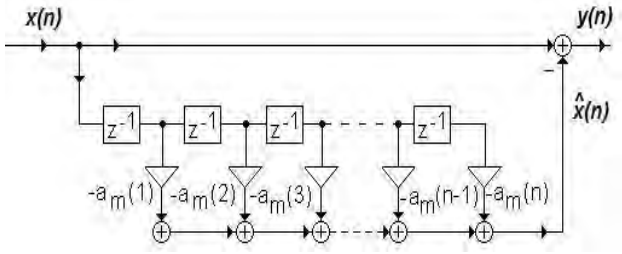


Figura 3.3: Filtro predictor de errores

La estructura del filtro FIR en la figura 3.1 puede dibujarse también como se muestra en la figura 3.3. En la salida del filtro se tiene la siguiente señal:

$$y(n) = x(n) - \hat{x}(n)$$

donde

$$\hat{x}(n) = -\sum_{k=1}^{m} a_m(k)x(n-k)$$

es una predicción del valor $x(n)$, esta estructura se llama Filtro de predicción de errores. Si se supone que hay un filtro FIR de primer orden, en la salida se obtiene

$$y(n) = x(n) + a_1(1)x(n-1) \tag{3.3}$$

La misma respuesta se obtiene mediante la estructura de cruz que se muestra en la figura 3.4.
Si se alimentan ambas entradas de la estructura de cruz con la señal $x(n)$ se obtiene en las salidas

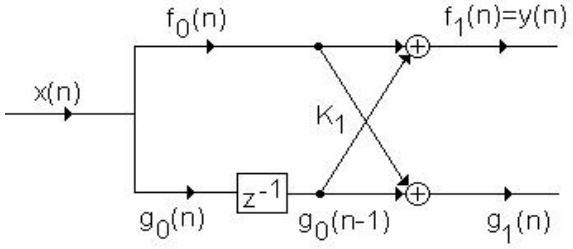


Figura 3.4: Filtro en la forma de cruz. Primer orden

$$f_0(n) = g_0(n) = y(n) \tag{3.4}$$

$$f_1(n) = x(n) - K_1 x(n-1) \tag{3.5}$$

$$g_1(n) = K_1 x(n) + x(n-1) \tag{3.6}$$

3.3. Filtro FIR con estructura Polifase

Otra muy interesante realización de un filtro FIR se basa en la descomposición polifase de su función de transferencia que resulta en una estructura en paralelo. Para mostrar dicha estructura, considere por simplicidad un filtro FIR con función de transferencia $H(z)$ de longitud 9:

$$\begin{aligned} H(z) = h[0] + h[1]z^{-1} + h[2]z^{-2} + h[3]z^{-3} + h[4]z^{-4} \\ +h[5]z^{-5} + h[6]z^{-6} + h[7]z^{-7} + h[8]z^{-8} \end{aligned} \tag{3.7}$$

La función de arriba puede ser expresada como una suma de dos términos, uno conteniendo los coeficientes con índices pares y otro conteniendo los coeficientes con índices impares, como se muestre a continuación:

$$\begin{aligned} H(z) = &\ (h[0] + h[2]z^{-2} + h[4]z^{-4} + h[6]z^{-6} + h[8]z^{-8}) \\ + &\ (h[1]z^{-1} + h[3]z^{-3} + h[5]z^{-5} + h[7]z^{-7}) \\ = &\ (h[0] + h[2]z^{-2} + h[4]z^{-4} + h[6]z^{-6} + h[8]z^{-8}) \\ + &\ z^{-1}(h[1] + h[3]z^{-2} + h[5]z^{-4} + h[7]z^{-6}) \end{aligned} \tag{3.8}$$

Si se usa la notaciíon

$$E_0(z) = h[0] + h[2]z^{-1} + h[4]z^{-2} + h[6]z^{-3} + h[8]z^{-4}$$
$$E_1(z) = h[1] + h[3]z^{-1} + h[5]z^{-2} + h[7]z^{-3}$$

se puede reescribir la ecuación (3.8) como

$$H(z) = E_0(z^2) + E_1(z^2) + E_2(z^2) \tag{3.9}$$

De forma similar, si se agrupa los términos de la ecuación(3.7) de forma diferente, ésta pues ser expresada en la forma

$$H(z) = E_0(z^3) + E_1(z^3) + E_2(z^3) + E_3(z^3) \tag{3.10}$$

donde ahora

$$E_0(z) = h[0] + h[3]z^{-1} + h[6]z^{-2}$$

$$E_1(z) = h[1] + h[4]z^{-1} + h[7]z^{-2}$$

$$E_2(z) = h[2] + h[5]z^{-1} + h[8]z^{-2}$$

Las descomposiciónes de $H(z)$ en la forma de las ecuaciones (3.9) y (3.10) son usualmente conocidas como *Descomposición Polifase*. De forma general, una descomposición en L ramas de la función de tranferencia de la ecuación (3.1) de orden N es de la forma

$$H(z) = \sum_{m=0}^{L-1} z^{-m} E_m(z^L), \tag{3.11}$$

donde

$$E_m(z) = \textstyle\sum_{n=0}^{[(k+1)/L]} h[Ln+m]z^{-n}, \quad 0 \leq m \leq L-1$$

con $h[n] = 0$ para $n > N$ y siendo $[(k+1)/L]$ la parte entera del número entre corchetes. Una realización de $H(z)$ basada en la descomposición de la ecuación (3.11) es denominada *Realización Polifase*. La figura 3.5 muestra la realización polifase de una función de transferencia en dos, tres y cuatros ramas. Es necesario mencionar que la expresión para la funcion de transferencia $E_m(z)$ es diferente segn el número de ramas en cada estructura realizada.
Los subfiltros $E_m(z^L)$ en la estructura polifase de un filtro FIR son también filtros FIR y pueden ser realizados usando cualquiera de los métodos explicados anteriormente. Sin embargo, para obtener una estructura canónica de la estructura completa, los retardos en todos los subfiltros deben ser compartidos.
Las estructuras polifases tiene amplia aplicación en procesamiento multitasa de señales digitales ya que proporcionan una eficiencia computacional muy buena.

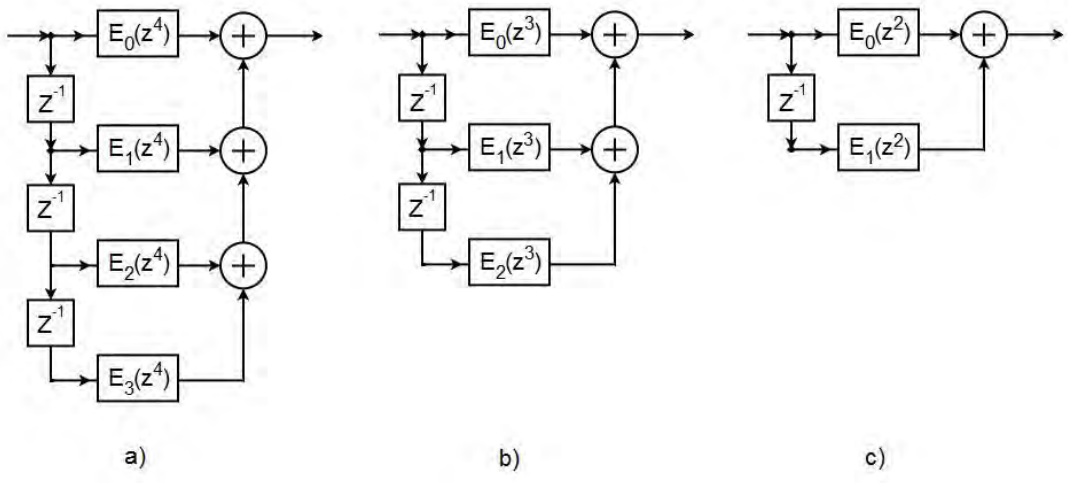


Figura 3.5: Diversas realizaciones polifase de la función de transferencisa de un filtro FIR

3.4. Tipos de respuesta al impulso del filtro FIR

Debido a que el número de muestras de la respuesta al impulso del filtro puede ser par o impar y a que existen dos formas de simetrías de la respuesta al impulso, dicha respuesta al impulso del filtro FIR puede ser clasificada en 4 tipos como se describe a continuación. [3]

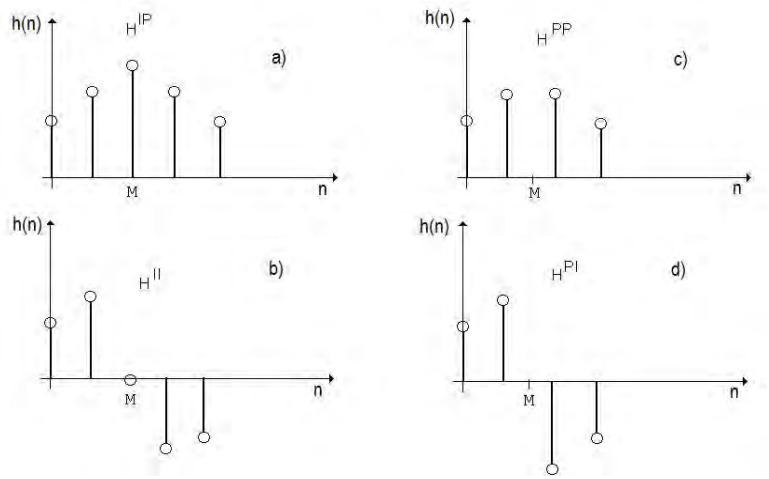


Figura 3.6: Los 4 tipos de respuesta al impulso de un filtro FIR

3.4.1. FIR Tipo 1

En el caso a) de la figura 3.6 el número de impulsos es impar, y la simetría con respecto al punto M es par, y debido a eso se puede expresar la función de transferencia como en la ecuación (3.12)

$$H(w)^{IP} = h(\frac{n}{2}) + \sum_{i=0}^{\frac{n}{2}-1} 2h(i)cos(\frac{n}{2} - i)w \tag{3.12}$$

3.4.2. FIR Tipo 2

En la figura 3.6b el número de los impulsos es par y la simetría con respecto al punto M también es par. La función de transferencia se calcula mediante la ecuación (3.13)

$$H(w)^{PP} = \sum_{i=0}^{\frac{n-1}{2}} 2h(i)cos(n - 2i)\frac{w}{2} \tag{3.13}$$

3.4.3. FIR Tipo 3

En el caso c) de la figura 3.6 el número de impulsos es impar, y la simetría con respecto al punto M también es impar. La función de transferencia se calcula mediante la ecuación (3.14)

$$H(w)^{II} = \sum_{i=0}^{\frac{n-1}{2}} 2h(i)sin\frac{(n - 2i)}{2}w \tag{3.14}$$

3.4.4. FIR Tipo 4

En la figura 3.6d el número de impulsos es par, y la simetría con respecto al punto M es impar, y debido a eso se puede expresar la función de transferencia como en la ecuación (3.15)

$$H(w)^{PI} = \sum_{i=0}^{\frac{n}{2}-1} 2h(i)sin(\frac{n}{2} - i)w \tag{3.15}$$

3.5. Localización de Ceros

Considere la funcion de transferencia de un filtro FIR de orden 6 que se presenta a continuación

$$H_6(z) = -1 + 2z^{-1} - 3z^{-2} + 6z^{-3} - 3z^{-4} + 2z^{-5} - 1z^{-6} \tag{3.16}$$

modificando la función de transferencia para encontrar los polos, se observa que todos se encuentran en el origen y debido a eso los filtros FIR son estables.

$$H_6(z) = \frac{-1z^6 + 2z^5 - 3z^4 + 6z^3 - 3z^2 + 2z - 1}{z^6}$$

La simetría o la antisimetría de la respuesta al impulso de una función de tranferencia FIR impone restricciones que tienen suma importancia en la localización de ceros de la función de transferencia, lo que a su vez, determina el tipo de filtro que esa función de transferencia puede implementar [3].

Una función de transferencia que satisface la ecuación (3.17) es llamada *Polinomial refleja-imagen* (MIP), un ejemplo de una función de transferencia polonomial refleja-imagen es el de la ecuación (3.16).

$$H(z) = z^{-N}H(z^{-1}) \tag{3.17}$$

donde N denota el orden del fltro FIR.

$$H(z) = -z^{-N}H(z^{-1}) \tag{3.18}$$

Si una función de transferencia satisface la ecuación (3.18) es denominada *Polinomial antirefleja-imagen* (AIP), un ejemplo de este caso es el que se muestra a continuación:

$$H(z) = 1 - 2z^{-1} + 3z^{-2} - 3z^{-4} + 2z^{-5} - z^{-6}$$

Para un filtro FIR con función de transferencia MIP o AIP se tiene que si $z = \epsilon_0$ es un cero de $H(z)$ entonces, tambien $z = \frac{1}{\epsilon_0}$ lo es. Además para un filtro FIR con respuesta real al impulso, los ceros ocurren en pares complejos conjugados. Por consiguiente un cero en $z = \epsilon_0$ está asociado con un cero en $z = \epsilon_0^*$.

Un cero que no está localizado en el círculo unitario, está asociado con un grupo de 4 ceros dados por

$$z = re^{\pm j\phi}, \quad z = \frac{1}{r}e^{\pm j\phi}$$

Para un cero en el círculo unitario, su recíproco es también su complejo conjugado. Por lo tanto los ceros aparecen como un par complejo conjugado.

$$z = e^{\pm j\phi}$$

Un cero real y su par se presentan como sigue:

$$z = \rho, \quad z = \frac{1}{\rho}$$

Lo anterior hace notar que los ceros de un filtro FIR exhibe una simetría con respecto al círculo unitario.

La principal diferencia entre los cuatro tipos de filtro FIR es con respecto al número de ceros en $z = 1$ y $z = -1$. A continuación se enlista este caso para cada uno de los tipos de filtro FIR.

- Tipo 1: Ninguno o un número par de ceros en $z = 1$ y $z = -1$.
- Tipo 2: Ninguno o un número par de ceros en $z = 1$ y un número impar de ceros en $z = -1$.
- Tipo 3: Un número impar de ceros en $z = 1$ y $z = -1$.
- Tipo 4: Un número impar de ceros en $z = 1$ y ninguno o un número par de ceros en $z = -1$.

La presencia de ceros en $z = \pm 1$ conlleva a limitaciones en el uso de filtros FIR para diseñar filtros. Por ejemplo, debido a que el filtro FIR tipo 2 siempre tiene un cero en $z = -1$, no puede ser utilizado para diseñar filtros pasoaltas. Por otro lado el filtro tipo 3 tiene ceros en $z = 1$ y $z = -1$, como resultado, no puede ser usado para diseñar filtros pasobajas, pasoaltas ni supresor de banda. De forma similar el filtro FIR tipo 4 no es apropiado para diseñar filtro pasobajas debido a la presencia de un cero en $z = 1$. Finalmente, el filtro tipo 1 no tiene muchas restricciones y puede ser utilizado para diseñar casi cualquier tipo de filtro.

3.6. Funciones ventanas en la respuesta del filtro FIR

Por definición, una secuencia de longitud finita simétrica alrededor de un punto medio se llama *función ventana*. El proceso de multiplicar una secuencia de longitud infinita por una función ventana se llama *ventaneo*. Se dice que la señal discreta creada mediante ventaneado es una señal *ventaneada*. [2]

Con el fin de mejorar la respuesta de un filtro, se utilizan diversas funciones ventanas. La influencia de las ventanas se muestra en la figura 3.7.

En la figura 3.7a se puede observar que si se tiene un número infinito de coeficientes en la función de transferencia, se obtiene la respuesta del filtro ideal. Debido a que los filtros ideales no pueden ser creados, se limitan el número de coeficientes con una ventana.

Si se utiliza una ventana rectangular se obtiene la respuesta que se muestra en la figura 3.7b. Para este caso se observa que la amplitud de la respuesta no es muy buena ya que su oscilación es muy grande en la banda de paso. Si se utiliza la ventana que propuso Hamming, Von Hann o Keiser, las oscilaciones logran disminuir, éstas controlan no sólo el número de muestras, como en el caso de la ventana rectangular, sino también el tamaño de las muestras, lo que se ve reflejado en los coeficientes de la función de tranferencia. Los nuevos coeficientes del filtro se determinan usando la ecuación $c'_w = w_m \cdot c_m$, donde w_m son los coeficientes de la ventana.

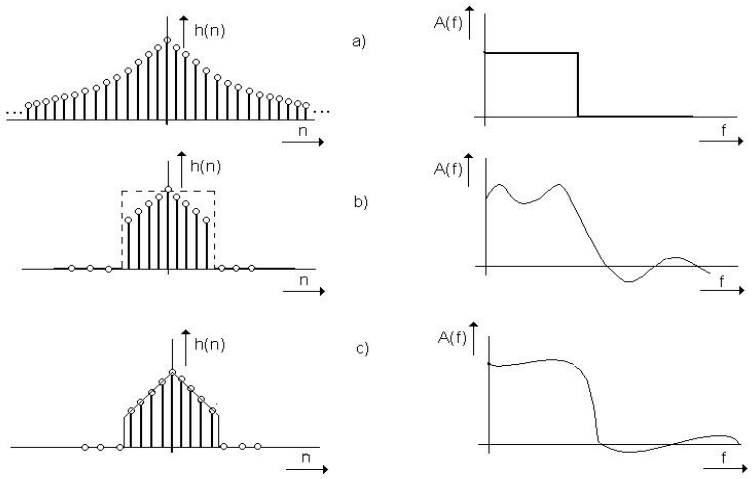


Figura 3.7: Influencia de las ventanas en la respuesta de un filtro

3.6.1. Ventana de Hamming

La ventana de Hamming se define mediante la ecuación 3.19

$$w_i = 0{,}54 + 0{,}46cos(\frac{2\pi i}{M}) \tag{3.19}$$

Para M=20, los coeficientes de la ventana de Hamming son los siguientes

$$\begin{aligned}
w_0^H &= 1 \\
w_1^H &= 0{,}54 + 0{,}46cos\tfrac{2\pi}{20} = 0{,}977486 \\
w_2^H &= 0{,}54 + 0{,}46cos\tfrac{4\pi}{20} = 0{,}9121478 \\
w_3^H &= 0{,}54 + 0{,}46cos\tfrac{6\pi}{20} = 0{,}8103812 \\
w_4^H &= 0{,}54 + 0{,}46cos\tfrac{8\pi}{20} = 0{,}6821478 \\
w_5^H &= 0{,}54 + 0{,}46cos\tfrac{10\pi}{20} = 0{,}5233846 \\
w_6^H &= 0{,}54 + 0{,}46cos\tfrac{12\pi}{20} = 0{,}3978522 \\
w_7^H &= 0{,}54 + 0{,}46cos\tfrac{14\pi}{20} = 0{,}2696188 \\
w_8^H &= 0{,}54 + 0{,}46cos\tfrac{16\pi}{20} = 0{,}1678522 \\
w_9^H &= 0{,}54 + 0{,}46cos\tfrac{18\pi}{20} = 0{,}102524 \\
w_{10}^H &= 0{,}54 + 0{,}46cos\tfrac{20\pi}{20} = 0{,}08
\end{aligned}$$

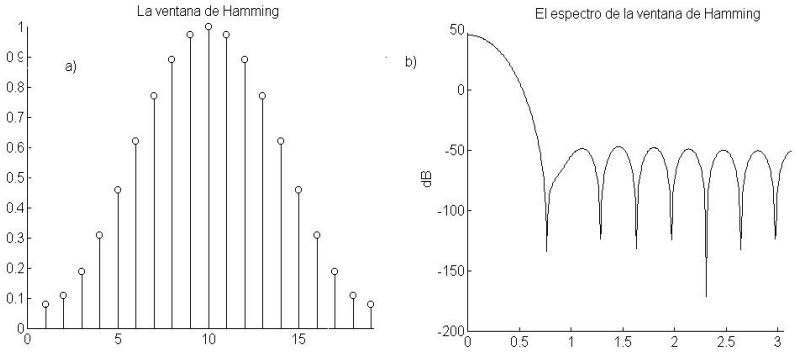


Figura 3.8: Ventana de Hamming y su Espectro

Si se tranforma $w_i(n)$ mediante la tranformada de Fourier se puede obtener el espectro de la ventana de Hamming. La ventana de Hamming en el dominio de la frecuencia se muestra en la ecuación (3.20).

$$W(f) = \frac{Mcos(\pi fM)}{\pi fM} \frac{0{,}54 - 0{,}08(fM)^2}{1 - (fM)^2} \tag{3.20}$$

El espectro de la ventana se muestra en la figura 3.8b, ahí se observa que la gananacia máxima en la banda de supresión siempre es menor a 50[dB]. La ventana propiamente, se presenta en la figura 3.8a.

Capítulo 4

Filtros con respuesta infinita al impulso unitario

En este capítulo abordaremos a los filtros digitales que, como su nombre lo indica, tienen un número infinito de términos no nulos en la salida, cuando en la entrada se coloca un impulso unitario. En general, la salida de los filtros IIR depende de las entradas actuales y pasadas, y además de las salidas en instantes anteriores.

La función de transferencia de los filtros digitales con la respuesta infinita al impulso (IIR) se puede escribir de la forma

$$H(z) = \frac{X(z)}{Y(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_k z^{-k}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_k z^{-k}} = \frac{\sum_{i=0}^{k} b_i z^{-i}}{1 + \sum_{i=1}^{k} a_i z^{-i}} \tag{4.1}$$

Aplicando la tranformada z inversa, se puede obtener la ecuación en diferencias.

$$y(n) = \sum_{i=0}^{k} b_i x(n-i) - \sum_{i=1}^{k} a_i y(n-i) \tag{4.2}$$

Para realizar la ecuación en diferencias, ecuación (4.2), se necesita:

- Sumador
- Elemento de Retardo
- Multiplicador

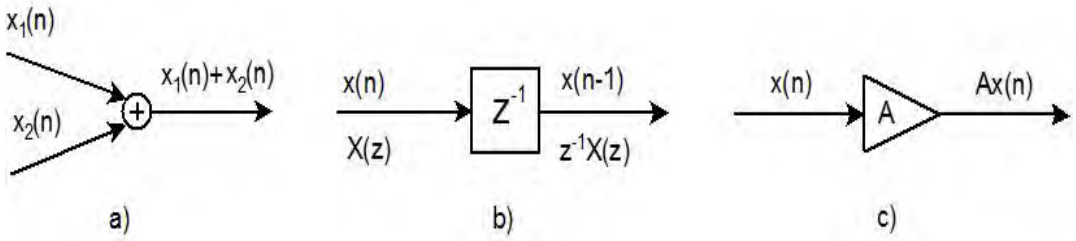


Figura 4.1: a)El sumador, b)El elemento de retardo y c)El multiplicador

4.1. Estructuras del Filtro IIR

Una función de transferencia de un filtro digital IIR de orden N se caracteriza por poseer $2N+1$ coeficientes únicos, y en general, requiere $2N+1$ miltiplicadores y $2N$ sumadores de dos entradas para su implementación. Como en el caso de la realización de los filtros FIR, las estructuras de los filtros IIR en las cuales los coeficientes de multiplicación son precisamente los coeficientes de la función de transferencia, son llamadas *Estructuras en la forma directa*. A continuación se describe el desarrollo de estas estructuras [3].

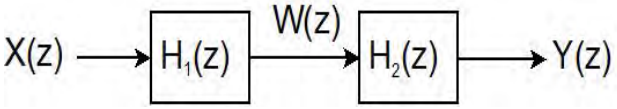


Figura 4.2: Una posible realización del filtro IIR

Considere un filtro de segundo orden, caracterizado por la función de tranferencia

$$H(z) = \frac{P(z)}{D(z)} = \frac{p_0 + p_1 z^{-1} + p_2 z^{-2}}{1 + d_1 z^{-1} + d_2 z^{-1}},$$

la cual se puede implementar en forma de cascada con dos secciones, como se muestra en la figura 4.2, donde

$$H_1(z) = \frac{W(z)}{X(z)} = P(z) = p_0 + p_1 z^{-1} + p_2 z^{-2}$$

y

$$H_2(z) = \frac{Y(z)}{W(z)} = \frac{1}{D(z)} = \frac{1}{1 + d_1 z^{-1} + d_2 z^{-1}}$$

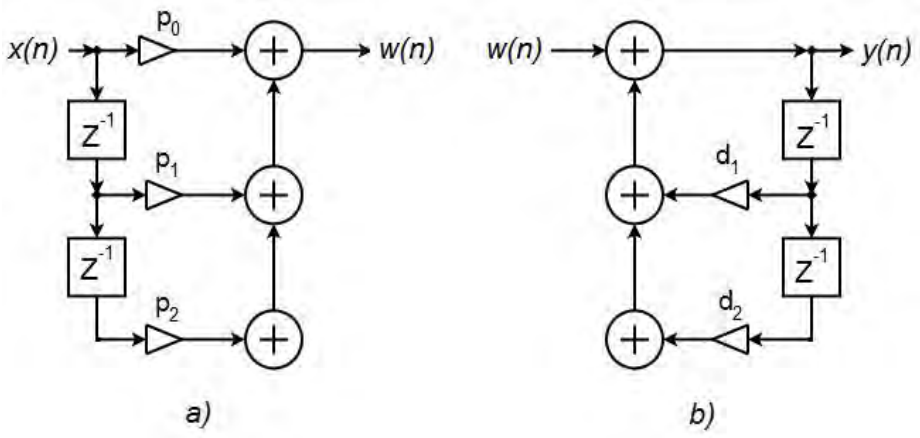


Figura 4.3: Realización de a)H_1 y b)H_2

La sección que corresponde a $H_1(z)$ puede ser visto como un filtro FIR y puede ser realizado como se muestra en la figura 4.3a. En el caso de $H_2(z)$, es necesario representarla en el dominio del tiempo.

$$y(n) = w(n) - d_1 y(n-1) - d_2 y(n-2) - d_3 y(n-3),$$

Una vez en el dominio del tiempo es más claro observar los coeficientes multiplicadores y los retrasos en el tiempo. La estructura de H_2 se muestra en la figura 4.3b.

Una estructura de cascada entre la figuras 4.3a y 4.3b da como resultado la realización de la función de transferencia del filtro IIR original. La estructura resultante se presenta el figura 4.4a y es comúnmente denominada *Estructura en la forma directa I*. La estructura transpuesta a la forma directa I se muestra en la figura 4.4b. Para obtener una estructura transpuesta se reemplazan nodos por sumadores y los sumadores por nodos y se cambian las direcciones los multiplicadores. La figura 4.5 muestra la forma en que se reemplazan los elementos del filtro. Un ejemplo de filtro IIR de la forma directa I y su transpuesta se muestra en la figura 4.6

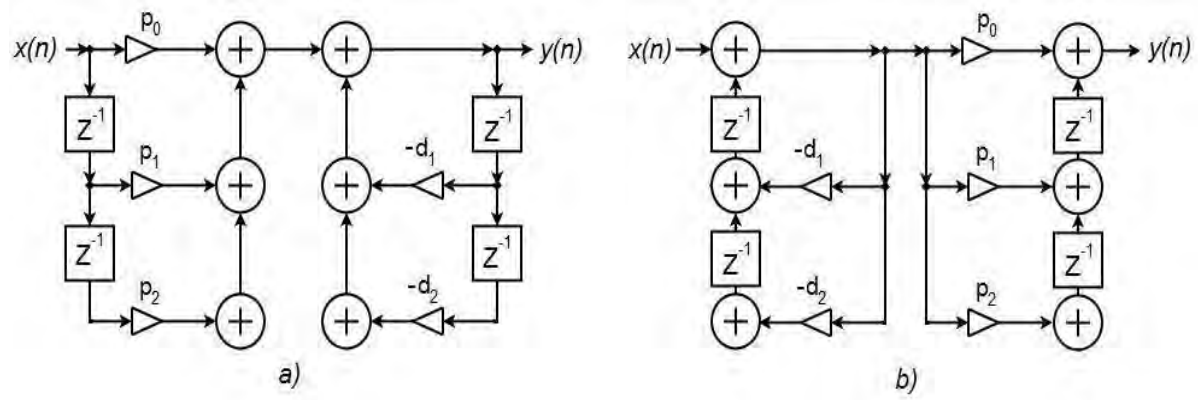


Figura 4.4: Forma directa I y su transpuesta. Filtro IIR

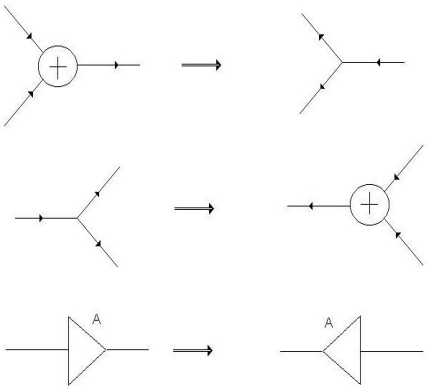


Figura 4.5: Reglas para obtener una estructura transpuesta de un filtro FIR

Es necesario hacer notar que todas las estructuras mencionadas anteriormente son no canónicas ya que emplean $2N$ retrasos para implementar un función de orden N. Para realizar una implementación canónica, se parte de que los nodos 1 y $1'$ son los mismos, por lo tanto, los elementos de retardo superiores pueden ser compartidos como se puede ver en la figura 4.6. Igualmente las

señales en los nodos 2 y 2′ son las mismas, por lo que también es posible compartirlos. La estructura resultante es ya una estructura canónica. Dicha estructura y su transpuesta son mostradas en la figura 4.7.

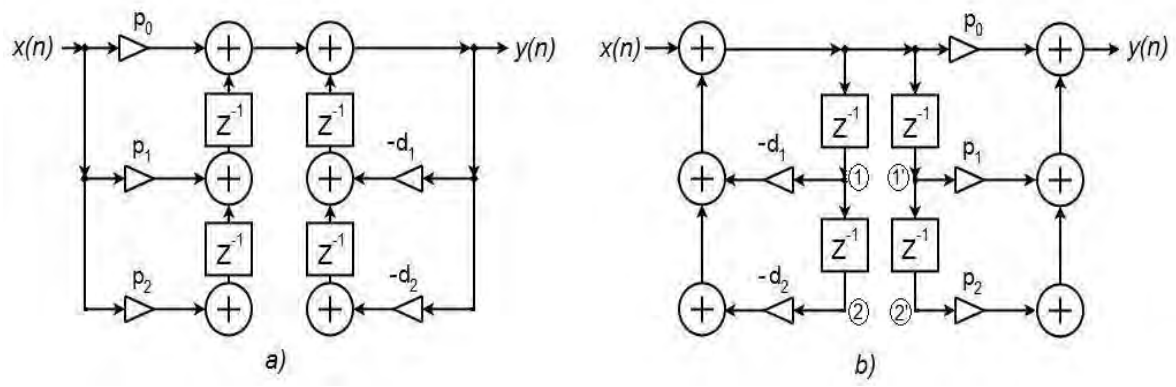


Figura 4.6: Conversión de una estructura a su transpuesta de la figura 4.4

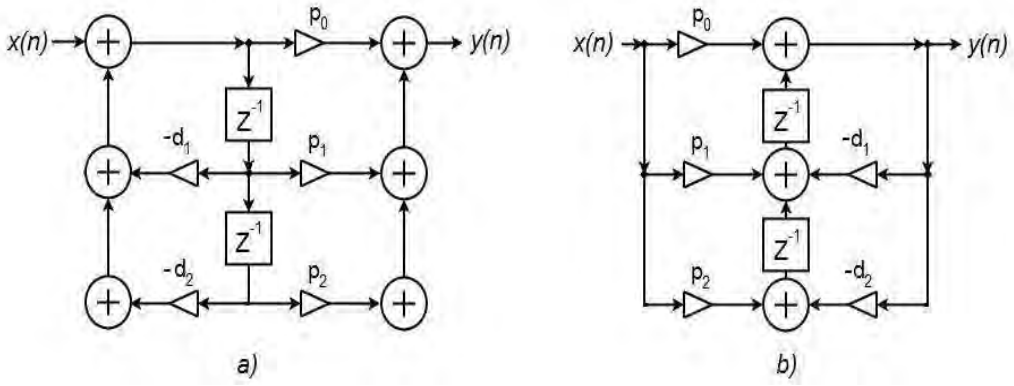


Figura 4.7: a)Estructura Canónica del filtro IIR y b) su transpuesta

4.2. Estructura en Cascada del Filtro IIR

A menudo, la estructura de un filtro digital es realizada expresando al polinomio numerador y al denominador de la función de tranferencia como un producto de polinomios de menor orden. A esta estructura se le denomina *Cascada*. De esta forma se puede considerar, por ejemplo a $H(z) = P(z)/D(z)$ expresado como

$$H(z) = \frac{P_1(z)P_2(z)P_3(z)}{D_1(z)D_2(z)D_3(z)} \tag{4.3}$$

Diferentes estructuras de cascadas se pueden obtener dependiendo de los pares polinomiales polo-cero elegidos. Algunos ejemplos se muestran en la figura 4.8. Se pueden obtener tambien, diferentes estructuras de cascadas con el simple cambio de orden de las secciones.

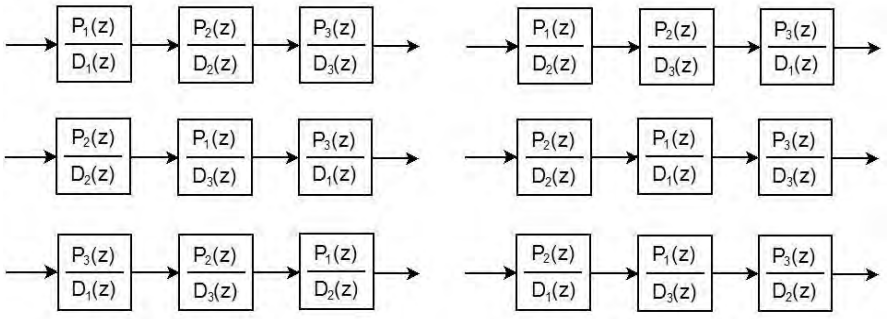


Figura 4.8: Diferentes formas de obtener una estructura en cascada

Normalmente, los polinomios son factorizados como producto de polinomios de primer orden y de segundo orden. En este caso $H(z)$ es expresada como

$$H(z) = p_0 \prod_{k=1}^{n} \frac{1 + \beta_{1k} z^{-1} + \beta_{2k} z^{-2}}{1 + \alpha_{1k} z^{-1} + \alpha_{2k} z^{-2}} \tag{4.4}$$

Para el caso de polinomios de primer orden se tiene que $\beta_{2k} = 0$ y $\alpha_{2k} = 0$. Una posible estructura de una función de transferecia de tercer orden

$$H(z) = p_0 \left(\frac{1 + \beta_{11} z^{-1}}{1 + \alpha_{11} z^{-1}} \right) \left(\frac{1 + \beta_{12} z^{-1} + \beta_{22} z^{-2}}{1 + \alpha_{12} z^{-1} + \alpha_{22} z^{-2}} \right)$$

se muestra en la figura 4.9

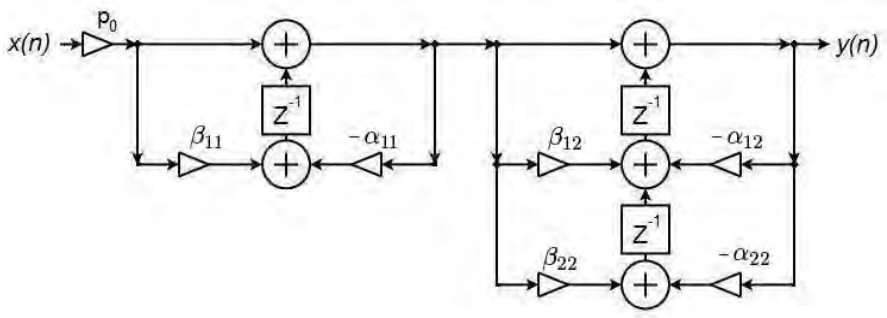


Figura 4.9: Una posible estructura en cascada para un filtro de tercer orden

4.3. Estructura en Paralelo del Filtro IIR

Un filtro IIR puede tener una estructura en paralelo haciendo uso de la expansión en fracciones parciales de su función de transferencia. [3]
Es común que la función de transferencia del filtro digital $G(z)$ sea una fracción propia con polos simples. Estos polos se localizan en $z = \lambda_l$, donde $1 \leq l \leq N$. La expansión en fracciones parciales, para este caso, es de la forma

$$G(z) = \sum_{l=1}^{N} \frac{\rho_l}{1 - \lambda_l z^{-1}} \tag{4.5}$$

donde la constante ρ_l se denomina *residuo* y esta dada por

$$\rho_l = (1 - \lambda_l z^{-1}) G(z) \mid_{z=\lambda_l}$$

La expansión en fracciones parciales de la función de transferencia en la forma de la ecuación (4.5) conduce a la *forma paralela I*. Por lo tanto, asumiendo polos simples $G(z)$ puede ser expresada de la forma

$$G(z) = \gamma_0 + \sum_k \left(\frac{\gamma_{0k} + \gamma_{1k} z^{-1}}{1 + \alpha_{1k} z^{-1} + \alpha_{2k} z^{-2}} \right) \tag{4.6}$$

si se trata de polos reales entonces, $\gamma_{1k} z^{-1} = 0$ y $\alpha_{2k} z^{-2} = 0$
La figura 4.10 muestra el ejemplo de un estructura en paralelo en la forma I para un filtro IIR de orden 3.

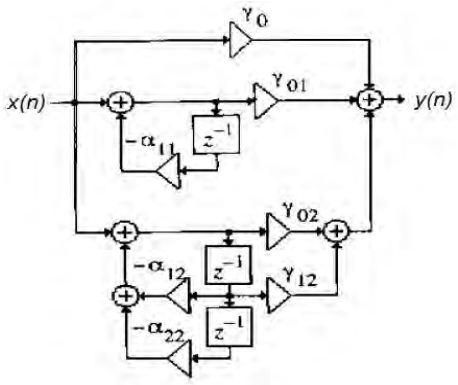


Figura 4.10: Forma paralela I para un filtro IIR de tercer orden.

Capítulo 5

Introducción a los sistemas multitasa

La característica común más importante en los filtros y los bancos de filtros usados en este escrito, es que usan procesamiento digital multitasa de señales. Para poder entender los sistemas multitasa, es escencial, primero, entender como el muestreo puede ser cambiado.

5.1. Reducción de la tasa de muestreo

La reducción de la tasa de muestreo de una señal se denomina *Decimación de la Tasa de Muestreo* o simplemente *Decimación*, esta consiste de dos etapas: La primera consiste en un filtro anti-aliasing y la segunda realiza la decimación o submuestreo.

5.1.1. Submuestreo

El submuestreo es el proceso en el cual la tasa de muestreo de una señal discreta $x(n)$ es reducida por un factor M. Lo anterior se logra tomando en cuenta solamente cada $M-$ésimo valor de la señal. La relación entre la señal resultante $y(m)$ y la señal original esta dada por:

$$y(m) = x(m \cdot M) \tag{5.1}$$

La figura 5.1 muestra la representación del flujo de una señal durante el submuestreo. El símbolo cuadrado con una flecha apuntando hacia abajo, se denomina *Submuestreador*. La señal $y(m)$ es la señal submuestreada con respecto a la señal $x(n)$.

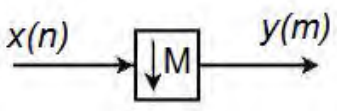


Figura 5.1: Proceso de Submuestreo.

En el dominio de z, se puede utilizar la transformada z de la señal original

$$X(z) = \sum_{n=-\infty}^{\infty} x(n) \cdot z^{-n}$$

para obtener la tranformada z de la señal $y(m)$, considerando también que la *fase offset,* λ, es igual a cero, es decir, durante el submuestreo, la primera muestra que se tomó como válida fue $x(0)$.

$$\begin{aligned} X_0^{(p)} &= \sum_{n=-\infty}^{\infty} x(m \cdot M) \cdot z^{-mM} \\ &= \sum_{n=-\infty}^{\infty} y(m) \cdot (z^M)^{-m} \\ X_0^{(p)} &= Y(z^M) = Y(z') \end{aligned} \tag{5.2}$$

La transformada z de la señal muestreada $Y(z^M)$, puede ser expresada usando componentes de modulación, para lo que es necesario dicho definir dicho término en el contexto de la tranformada z.

La modulación de una tranformada z es realizada por la multiplicación de la variable independiente z con el número W_M^k. Para el caso de $Y(z^M)$ se tiene que:

$$Y(z^M) = \frac{1}{M} \sum_{k=0}^{M-1} X(zW_M^k) \tag{5.3}$$

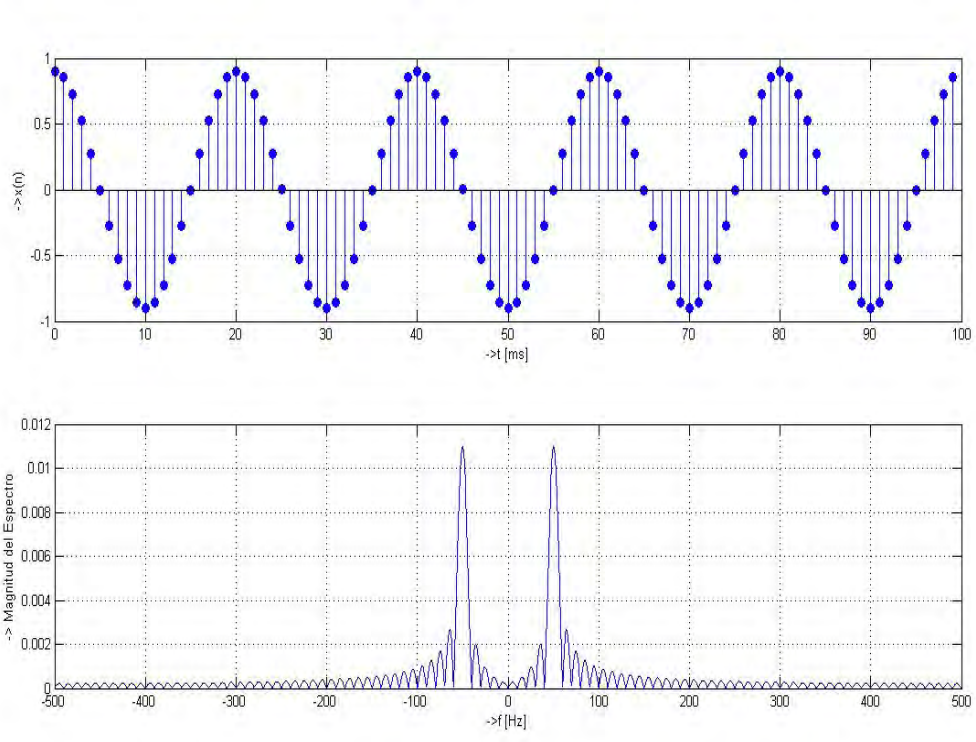


Figura 5.2: Señal original

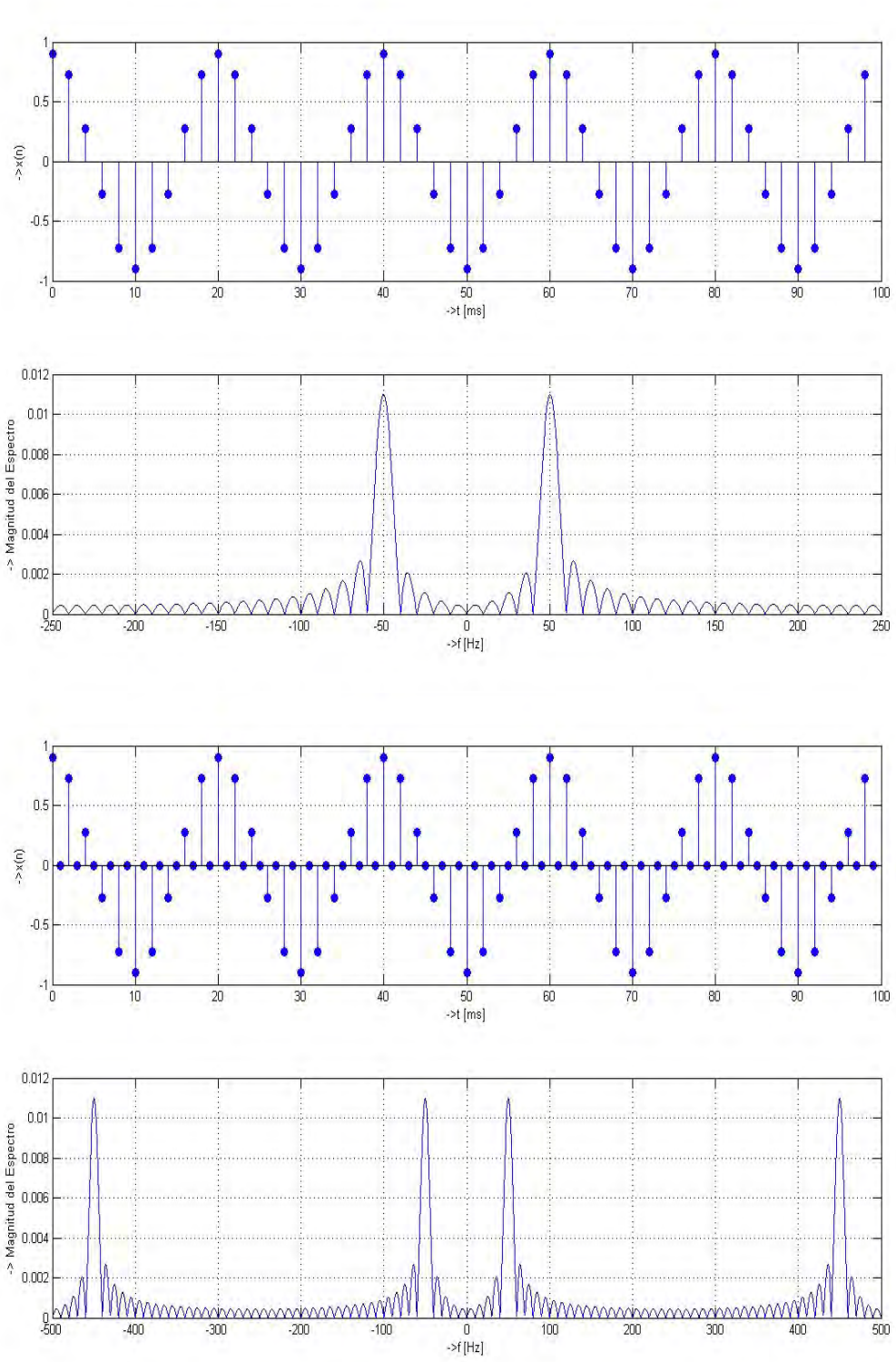


Figura 5.3: a)Señal submuestreada a partir de la señal en 5.2 y b) Señal sobremuestreada a partir de la señal submuestreada.

5.2. Aumento de la tasa de muestreo

Si algunas señales de banda angosta son combinadas para formar un señal de banda ancha, su tasa de muestreo debe primero ser aumentada. La tasa de muestreo también necesita ser incrementada cuando una señal de banda angosta será observada con una resolución muy fina en dominio del tiempo.
El aumento en la tasa de muestreo es denominado *Interpolación*, y consiste de un sobremuestreo seguido de un *filtro anti-imagen o interpolador*.

5.2.1. Sobremuestreo

La tasa de muestreo de una señal discreta $y(m)$ es aumentada por un factor L, colocando $L-1$ ceros igualmente espaciados entre cada par de muestras. La señal resultante $u(n)$ esta dada por

$$u(n) = \begin{cases} y(n/L) & \text{para } n = mL, \quad \text{m entero,} \\ 0 & \text{otro caso} \end{cases} \tag{5.4}$$

El símbolo correspondiente al proceso de sobremuestreo se muestre en la figura 5.4

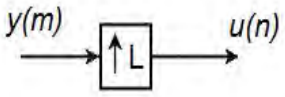


Figura 5.4: Proceso de Submuestreo.

La figura 5.5a muestra una señal $y(m)$ y la figura 5.5b la señal obtenida despues del sobremuestreo por un factor de 3.

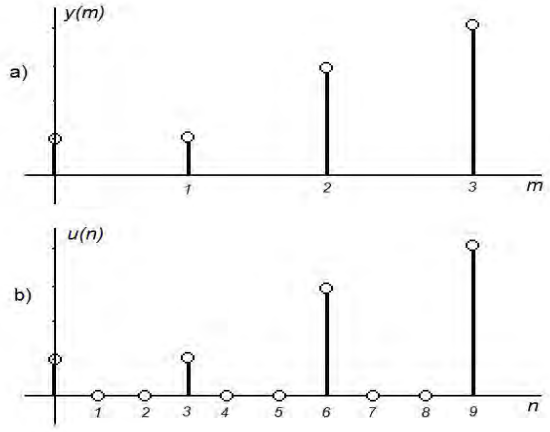


Figura 5.5: Sobremuestreo de una señal.

El sobremuestreo es el proceso inverso al submuestreo, por lo tanto, si la transformada z

$$Y(z) = \sum_{m=-\infty}^{\infty} y(m) z^{-m}$$

de la señal $y(m)$ antes del sobremuestreo es identificada con $Y(z)$ de la ecuación (5.2), y la transformada z

$$U(z) = \sum_{n=-\infty}^{\infty} u(n)z^{-n}$$

de la señal sobremuestreada $u(n)$ con $X_0^{(p)}(z^M)$ de la ecuación (5.2), se puede utilizar esa misma ecuación (5.2) para deducir lo siguiente:

$$U(z) = Y(z^L). \tag{5.5}$$

Un efecto muy importante que se presenta en la frecuencia durante el proceso de sobremuestreo, es el denominado *formación de imágenes*. Si un sobremuestreo es realizado con un factor L, en el dominio de la frecuencia se observará L veces el espectro de la señal original. Este efecto recibe su nombre debido a que se obtiene, en el espectro de salida, una *imagen* adicional del espectro de entrada. Para el caso general, un aumento en la tasa de muestreo con un factor L provocará $L-1$ imágenes adicionales del espectro de entrada. Un filtro pasobajas remueve las $L-1$ imágenes y, en efecto, *rellena* las muestras valuadas en cero con muestras con valores interpolados.

Como ejemplo de decimación e interpolación en conjunto, considere la señal que se muestra en la figura 5.2. Después de un submuestreo la señal resultante es la que se muestra en la figura 5.3a, note que en comparacin con la señal la única diferencia visible es la frecuencia de muestreo. Si a la señal submuestreada, figura 5.3a, se le aplica un sobremuestreo, se obtiene la señal mostrada en la figura 5.3b. En el espectro de la señal sobremuestrada se observa la formación de frecuencias imágenes de las cuales ya se hizo mención en el párrafo anterior.

5.3. Equivalencias

Como se verá más adelante, un sistema multitasa esta conformado por interconexiones de dispositivos que alteran la tasa de muestreo muestreo, en este caso submuestreadores y sobremuestreadores. En muchas aplicaciones estos dispositivos aparecen en un estructura de cascada. A menudo se intercambian las posiciones de los dispositivos para obtener una mejor eficiencia computacional.

5.3.1. Cascada de Sobremuestreador y Submuestreador

El sobremuestreador o el submuestreador pueden ser usados para cambiar la tasa de muestreo de una señal solamente por un factor entero. Si se desea implementar un cambio de tasa de muestro con un factor fraccionario, es común usar un submuestreador y un sobremuestreador en cascada. Dichos dispositivos, de factor M y L respectivamente, pueden ser intercambiados sin ocasionar ningún cambio en la relación antrada salida, si y sólo si M y L son *primos relativos*, es decir, M y L no deben tener un factor común que sea entero mayor a 1.

5.3.2. Identidades Nobles

Otras dos equivalencias muy simples pero muy utilizadas en el procesamiento multitasa son las mostradas en la figura 5.6. Estas identidades nobles nos permiten mover los dispositivos de alteración de tasa de muestreo para así obtener una posición mas ventajosa en cuestión de eficiencia.

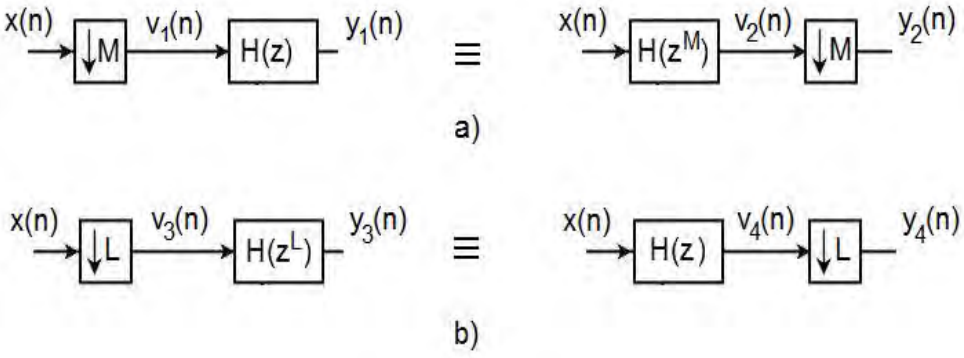


Figura 5.6: Equivalencias en Cascada.

5.4. Requerimientos Computacionales

Los filtros decimadores e interpoladores pueden ser diseñados con filtros digitales IIR o FIR. En el caso de procesamiento digital de señales con tasas simples, los filtros IIR en el aspecto computacional, son en general, más eficientes que los filtros FIR.

La situación no es exactamente la misma en el caso del procesamiento multitasa, un ejemplo sencillo es el de suponer un filtro decimador de longitud N. Si la implementación se realiza con un filtro FIR en la forma directa, entonces

$$v(n) = \sum_{m=0}^{N-1} h(m)x(n-m)$$

Como el submuestreador conserva solo cada M-ésima muestra de $v(n)$ en su salida, es suficiente calcular $v(n)$ usando la ecuación anterior sólo para valores de n que sean múltiplo de M y saltar las muestra restantes. Esto deja un ahorro computacional con un factor de M.

Por otro lado, si se usa un filtro IIR, la complicación se nota inmediatamente al observar la forma de su función de transferencia, lo que ocasiona trabajar con una señal intermedia que para ser calculada necesita todos los valores de n.

Para el caso de un filtro interpolador, los argumentos explicados anteriormente se mantienen. Si la función de tranferencia $H(z)$ del filtro se implementa con filtros FIR, entonces el ahorro en cálculos es por un factor de L ya que, en este caso $v(n)$ tiene $L-1$ ceros entre dos valores consecutivos que son diferentes de cero.

5.5. Descomposición Polifase

Como se vió en la sección anterior, un decimador o interpolador empleando filtros FIR pasobajas puede ser eficiente computacionalmente, adicionalmente a esto, si los filtros FIR se realizan usando la descomposición polifase, descrita en secciones anteriores, se puede obtener una reducción mayor de complejidad. A continuación explicaremos la descomposición polifase de nuevo, pero de forma más general, incluyendo su aplicación en la eficiente realización del interpolador y decimador.

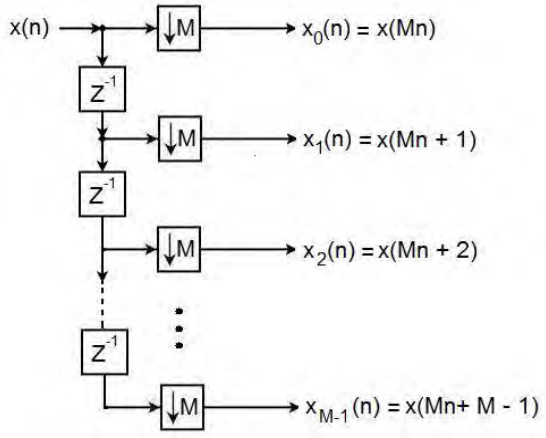


Figura 5.7: Una interpretación estructural de la descomposición polifase de M bandas de una secuencia $x(n)$

5.5.1. La descomposición

Si se considera un secuencia arbitraria $x(z)$ con transformada z:

$$X(z) = \sum_{n=-\infty}^{\infty} x(n)z^{-n}$$

se puede escribir $X(z)$ como

$$X(z) = \sum_{k=0}^{M-1} z^{-1} X_k(z^M), \tag{5.6}$$

donde

$$X_k(z) = \sum_{n=-\infty}^{\infty} x_k(n)z^{-n} = \sum_{n=-\infty}^{\infty} x(Mn+k)z^{-n}$$

$$0 \leq k \leq M-1$$

Las subsecuencias $x_k(n)$ se llaman componentes polifase de la secuencia madre $x(n)$, y las funciones $X_k(z)$, dadas por la transformada z de $x_k(n)$, se denominan componentes polifase de $X(z)$ [6]. La relación entre la subsecuencias $x_k(n)$ y la secuencia original $x(n)$ esta dada por

$$x_k(n) = \ x(Mn+k), \quad 0 \leq k \leq M-1$$

La ecuación (5.6) se puede rescribir en la forma matricial como:

$$X(z) = \begin{bmatrix} 1 & z^{-1} & \cdots & z^{-(M-1)} \end{bmatrix} \begin{bmatrix} X_0(z^M) \\ X_1(z^M) \\ \cdot \\ \cdot \\ \cdot \\ X_{M-1}(z^M) \end{bmatrix}$$

Una interpretación multitasa estructural de la descomposición polifase se muestra en la figura 5.7

Como se vio en el capítulo 3, la descomposición polifase de una función de transferencia de un filtro FIR puede ser desarrollada por inspección. Además de esa forma existen algunos otros métodos, a continuación se explicará brevemente algunas variaciones de la realización en paralelo de un filtro FIR basado en la descomposición polifase.
Contrario al caso de los filtros FIR, la descomposición polifase de una función de transferencia de un filtro IIR $H(z) = P(z)/D(z)$ no es tan fácil de realizar. Una forma de llegar a una descomposición polifase de M ramas de $H(z)$ es expresando la función de transferencia en la forma $P'(z)/D'(z^M)$ multiplicando el denominador $D(z)$ y el numerador $P(z)$ con un polimonio apropiado y luego aplicar una descomposición polifase de M ramas a $P(z)$.

A partir de lo mencionado en el párrafo anterior se puede observar que dicho método incrementa el orden total de la complejidad de $H(z)$, sin embargo en ciertas estructuras multitasas este método puede resultar muy eficiente.

5.5.2. Estructuras de filtros FIR basadas en descomposición polifase

Considere una descomposición polifase de M ramas de $H(z)$ dada por

$$X(z) = \sum_{k=0}^{M-1} z^{-1} E_k(z^M), \tag{5.7}$$

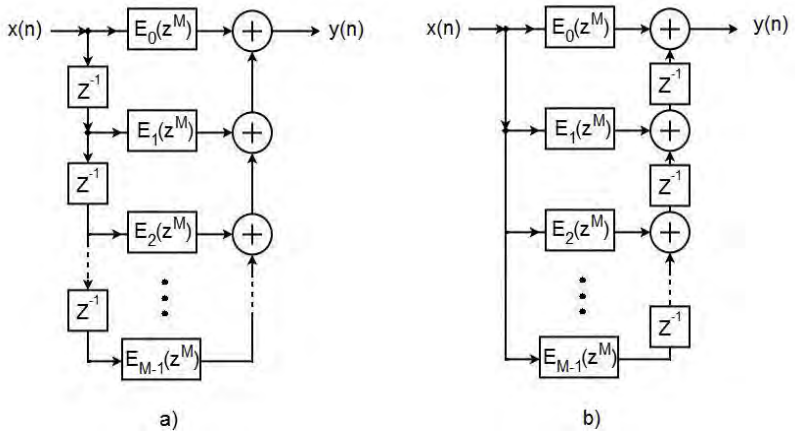


Figura 5.8: a) Realización directa de un filtro FIR basada en la descomposición polifase Tipo I y b) su transpuesta

Una realización directa basada en la ecuación (5.7) se muestra en la figura 5.8a. La transpuesta de esta realización es la indicada en la figura 5.8b. Una representación alternativa de la estructura transpuesta de la figura 5.8b se puede obtener usando la notación

$$R_l(z^M) = E_{M-1-l}(z^M), \quad 0 \leq l \leq M-1 \tag{5.8}$$

resultando en la estructura mostrada en la figura 5.9. La correspondiente descomposición polifase está entonces dada por

$$H(z) = \sum_{l=0}^{M-1} z^{-(M-1-l)} R_l(z^M). \tag{5.9}$$

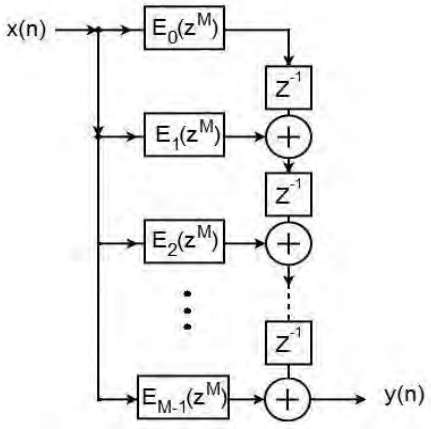


Figura 5.9: Realización de un filtro FIR basado en la descomposición polifase tipo II.

Para diferenciar las dos descomposiciones de las ecuaciones (5.8) y (5.9), la primera se denomina descomposición polifase tipo I y la segunda descomposición polifase tipo II.

5.5.3. Estructuras del decimador e interpolador eficientes computacionalmente

Es posible obtener estructuras de interpolador y decimador eficientes computacionalmente aplicando una descomposición polifase a su función de transferencia correspondiente.

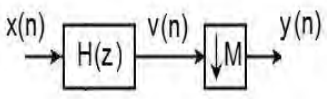


Figura 5.10: Decimador o Submuestreador

Considerando la aplicación de la descomposición polifase en el filtro de decimación de la figura 5.10, la estructura completa del decimador es la forma que se muestra en la figura 5.11a, cuando el filtro paso bajas $H(z)$ es realizado como en la figura 5.8. Una realización equivalente es la mostrada en la figura 5.11b, obtenida usando la equivalencia de cascada número 1 de la figura 5.6a. Esta última realización es más eficiente computacionalmente que la estructura de la figura 5.11a; para ilustrar este hecho es necesario asumir que el filtro de decimación $H(z)$ es un filtro FIR de longitud N con un periódo de muestreo de entrada $T = 1$. Ya que la salida $y(n)$ del decimador es obtenida submuestreando la salida del filro $v(n)$ por un factor de M, es necesario sólo calcular $v(n)$ en $n = \ldots, -2M, -M, 0, M, 2M, \ldots$. Los requerimientos de cálculo son entonces N multiplicaciones y $(N-1)$ sumas por muestra de salida calculada. Sin embargo, si n crece, las señales almacenadas en los elementos de retardos cambian, como resultado de esto, todos los cálculos necesitan ser completados en un periodo de muestreo, mientras que para los siguientes $(M-1)$ periódos, las unidades aritméticas se mantienen sin actividad.
Ahora, si se considera la estructura de la figura 5.11b y si la longitud de los subfiltros $E_k(z)$ es N_k, entonces $N = \sum_{k=0}^{M-1}$. Los requerimientos de cálculo para el $k-$ésimo subfiltro son N_k multiplicaciones y $N_k - 1$ sumas por muestra de salida, por lo tanto, la estructura completa tiene entonces $\sum_{k=0}^{-1} N_k = N$ multiplicaciones y $\sum_{k=0}^{-1}(N_k - 1) + (M-1) = N - 1$ sumas por muestra de

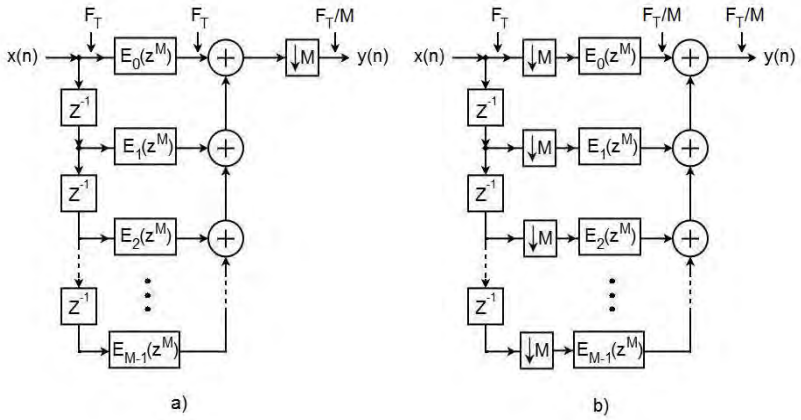


Figura 5.11: a) Implementación de un decimador basado en una descomposición polifase tipo I y b)estructura del decimadore eficiente computacionalmente. En las figuras se muestran las tasa de muestreo.

salida en el decimador. la ventaja en esta estructura es que la unidades aritméticas funcionan en cada periodo de muestreo de salida, lo que significa M veces que el periodo de muestreo de entrada. La misma eficiencia se puede obtener para el caso cuando se implementa un interpolador usando descomposición polifase. Una mayor eficiencia en las estructuras de interpoladores y decimadores se puede lograr si se aprovecha la simetría de los coeficientes del filtro de $H(z)$, considere por ejemplo, la realización de un decimador de factor 3 ($M = 3$) usando un filtro FIR pasobajas de longitud 12 con una respuesta simétrica al impulso.

$$H(z) = h[0] + h[1]z^{-1} + h[2]z^{-2} + h3]z^{-3} + h[4]z^{-4} + h[5]z^{-5} + h[5]z^{-6}$$
$$+h[4]z^{-7} + h[3]z^{-8} + h[2]z^{-9} + h[1]z^{-10} + h[0]z^{-11}$$

Una descomposición polifase de la función de transferencia de arriba tendría los siguientes subfiltros:

$$E_0(z) = h[0] + h[3]z^{-1} + h[5]z^{-2} + h[2]z^{-3}$$
$$E_1(z) = h[1] + h[4]z^{-1} + h[4]z^{-2} + h[1]z^{-3}$$
$$E_2(z) = h[2] + h[5]z^{-1} + h[3]z^{-2} + h[0]z^{-3}$$

Se puede observar que los subfiltros $E_1(z)$ sigue conservando una respuesta simétrica al impulso, además la respuesta al impulso de $E_2(z)$ es la respuesta imagen de $E_0(z)$. Una realización eficiente computacionalmente puede ser creada usando sólo seis multiplicadores y 11 sumadores de dos entradas.

Capítulo 6

Bancos de filtros

Muchos sistemas multitasa emplean un *banco de filtros*. Existen aplicaciones donde es deseable separar una señal en un grupo de señales subbanda ocupando, normalmente sin traslape, porciones de la banda de frecuencia original. En otras aplicaciones suele ser necesario combinar varias señales subbanda para formar una sola señal compuesta que ocupe todo el rango de Nyquist. Para este fin, los bancos de filtros digitales juegan un rol muy importante.

Un Banco de filtros es un grupo de filtros digitales pasobanda con una entrada en común o con una suma en la salida, como se muestra en la figura 6.1. La estructura de la figura 6.1a es denominada *Banco de filtros de Análisis de M bandas*, donde los filtros $H_i(z)$ se conocen como *Filtros de Análisis*. Esta estructura es utilizada para descomponer la señal de entrada en M señales $v_i(n)$ de subbandas.

La estructura opuesta al banco de filtros de análisis, a través del cual un grupo de señales $\hat{v}_i(n)$ de subbanda, usualmente perteneciente a frecuencias contiguas, se combinan para formar un señal $y(n)$, se denomina *Banco de filtros de Síntesis*. La figura 6.1b muestra un banco de filtros de síntesis de L bandas donde cada filtro $F_i(z)$ se conoce como *Filtro de Síntesis*.

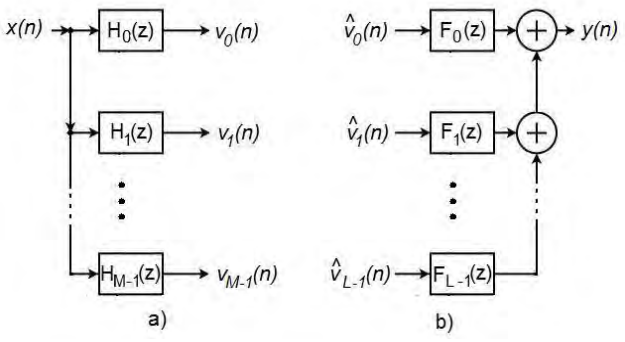


Figura 6.1: a)Banco de Filtros de Análisis y b) Banco de Filtros de Síntesis.

6.1. Banco de filtros de análisis y síntesis

6.1.1. Banco de filtros de análisis: Dos canales

La forma más común de descomponer una señal es en una componente de alta frecuencia y una componente baja frecuencia. En el banco de filtros de la figura 6.2 la señal de entrada $X(z)$ es procesada simultáneamente por un filtro pasobajas $H_0(z)$ y un filtro pasoaltas $H_1(z)$. El rango de frecuencias disponible, de $\Omega = 0$ hasta $\Omega = \pi$, el cual es la mitad de la tasa de muestreo, es comúnmente divido en dos mitades, como se muestra en la figura 6.3, de este modo las señales filtradas tienen un ancho de banda aproximada de $b = \pi/2$, y la tasa de muestreo es de esta forma dividida a la mitad. En este caso se puede aceptar una pequeña cantidad de traslapae (*aliasing*).

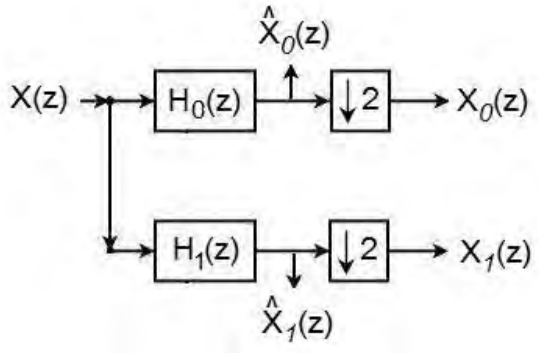


Figura 6.2: Banco de Filtros de Análisis de dos canales.

Las dos señales filtradas en la figura 6.2 son

$$\hat{X}_0(z) = X(z) \cdot H_0(z)$$
$$\hat{X}_1(z) = X(z) \cdot H_1(z)$$

submuestreando con $M = 2$ las señales de subbanda quedan

$$X_0(z) = \frac{1}{2}X(z^{\frac{1}{2}})H_0(z^{\frac{1}{2}}) + \frac{1}{2}X(-z^{\frac{1}{2}})H_0(-z^{\frac{1}{2}}) \tag{6.1}$$

$$X_1(z) = \frac{1}{2}X(z^{\frac{1}{2}})H_1(z^{\frac{1}{2}}) + \frac{1}{2}X(-z^{\frac{1}{2}})H_1(-z^{\frac{1}{2}}) \tag{6.2}$$

las cuales pueden ser escritas en forma matricial:

$$\begin{bmatrix} X_0(z) \\ X_1(z) \end{bmatrix} = \frac{1}{2}\begin{bmatrix} H_0(z^{\frac{1}{2}}) & H_0(-z^{\frac{1}{2}}) \\ H_1(z^{\frac{1}{2}}) & H_1(-z^{\frac{1}{2}}) \end{bmatrix} \cdot \begin{bmatrix} X(z^{\frac{1}{2}}) \\ X(-z^{\frac{1}{2}}) \end{bmatrix} \tag{6.3}$$

Las figuras 6.3b y 6.3d muestran las señales $\hat{X}_0(z)$ y $\hat{X}_1(z)$ en el dominio de la frecuencia. El eje de la frecuencia tiene una escala con respecto a la tasa de muestreo en la entrada.

El submuestreo produce el espectro que se presenta en la figura 6.3c y 6.3e. La señal pasoalta $X_1(z)$ pasa dentro la banda base $0 \leq \Omega \leq \pi$, con respecto a la tasa de muestreo reducida, y es invertida en frecuencia Los componentes espectrales en (6.1) y (6.2), los cuales dependen de $X(z^{\frac{1}{2}})$, se encuentran en la banda base. Los componentes espectrales que dependen de $X(-z^{\frac{1}{2}})$ son repeticiones periodicas de los anteriores. Debido a que las señales filtradas no son siempre de banda limitada a π un poco de aliasing aparece en la banda base.

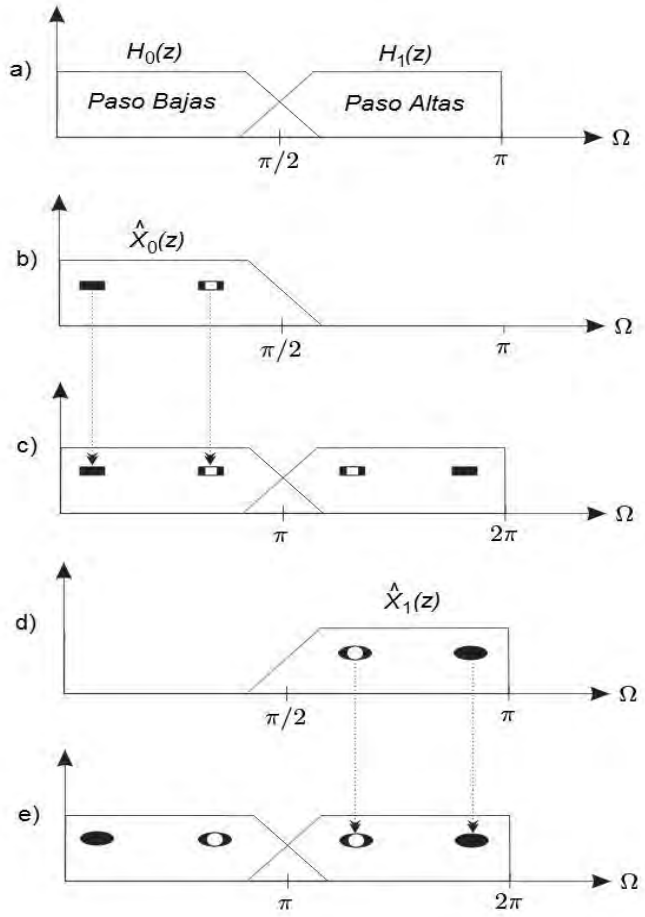


Figura 6.3: a)Función de Transferencia, b y d)Espectro de la señal filtrada, c y e)Espectro de la señal submuestreada en el banco de filtros de análisis.

6.1.2. Banco de filtros de Síntesis: Dos canales

La figura 6.4 muestra un banco de filtros de síntesis de dos canales con un filtro paso bajas $G_0(z)$ y un filtro paso altas $G_1(z)$. Este banco es recíproco del banco mostrado en la figura 6.2 las características fundamentales de los dos filtros , $G_1(z)$ y $G_1(z)$, son las mismas con respecto a los del banco de análisis. Después del sobremuestreo de las señales $X_0(z)$ y $X_1(z)$, el filtro paso bajas elimina parte sustancial del espectro imagen de la señal paso bajas $X_0(z)$ en el rango $\pi/2 \leq \Omega \leq \pi$. El filtro paso altas $G_1(z)$, mientras tanto, elimina la mayoría del espectro imagen de la señal $X_1(z)$ que está en el rango $0 \leq \Omega \leq \pi/2$. Debido a que la respuesta en frecuencia de la dos señales se translapan, el espectro imagen no es totalmente eliminado.
La señal de salida $\hat{X}(z)$ del banco de filtros de síntesis puede ser expresado usando (5.5) y viendo la figura 6.4, con lo que se tiene

$$\hat{X}(z) = G_0(z) \cdot X_0(z^2) + G_1(z) \cdot X_1(z^2)$$

Esta relación puede ser también expresada en forma matricial como

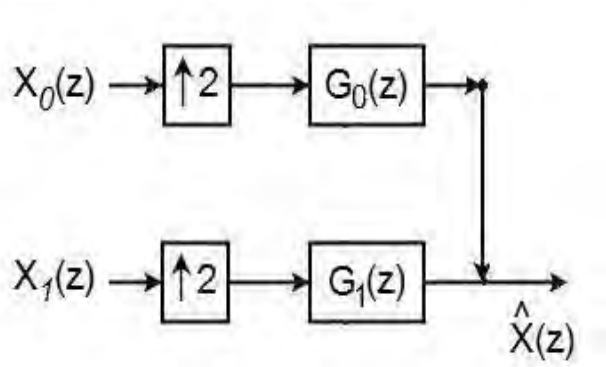


Figura 6.4: Banco de Filtros de Síntesis de dos canales.

$$\hat{X}(z) = \begin{bmatrix} G_0(z) & G_1(z) \end{bmatrix} \cdot \begin{bmatrix} X_0(z^2) \\ X_1(z^2) \end{bmatrix} \tag{6.4}$$

6.2. Banco de Filtros Espejo en Cuadratura (QMF)

Los bancos de filtros de espejo en cuadratura son bancos de filtros de dos canales con codificación de subbanda (SBC) y respuesta en frecuencia con potencia complementaria.

6.2.1. Banco de filtros de dos canales con codificación de subbanda

Un banco de filtros con codificación de subbanda está formado por un banco de filtros de análisis seguido de un bando de filtros de síntesis. La figura 6.5 muestra un banco de filtros de dos canales SBC. El banco de análisis, con los filtros $H_0(z)$ y $H_1(z)$ descompone la señal de entrada $X(z)$ en las señales de subbanda $X_0(z)$ y $X_1(z)$, seguidamente se encuentra el banco de síntesis con los filtros $G_0(z)$ y $G_1(z)$, los cuales reconstruyen la señal de salida a partir de las señales de subbanda.

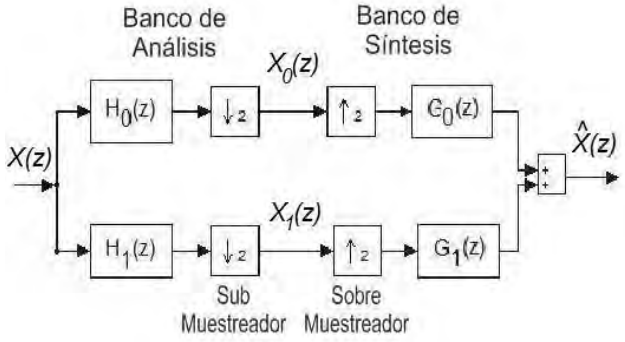


Figura 6.5: Banco de Filtros de dos canales SBC.

En los bancos de filtros criticamente muestreados, el número de muestras por unidad de tiempo del conjunto total de señales de subbanda es igual al número de muestras de entrada por unidad

de tiempo. Sin embargo, la potencia de las señales de subbanda es generalmente menor a la de la señal original. Por otro lado, para un determinado error de cuantización, existe una ganancia de codificación si se codifican las señales de subbanda en lugar de la señal original. Las señales codificadas pueden ser utilizadas para almacenamiento o transmisión. Después de la decodificación, la señal original es reconstruida. Todo lo anterior se convierte, entonces, en los requerimientos fundamentales para bancos de filtros SBC: Los bancos de filtros de análisis y síntesis deben ser diseñados para producir una señal $\hat{X}(z)$ que sea lo más cercana posible, o inclusive, exactamente igual a la señal original $X(z)$. Además, es importante tener una buena selectividad en frecuencia, es decir, que la suma de la potencia de las señales de subbanda sea prácticamente igual a la potencia de la señal original.
Si se sustituye la ecuación (6.3) en la ecuación (6.4), se obtiene la relación entre la señal de entrada $X(z)$ y la salida $\hat{X}(z)$:

$$\hat{X}(z) = \begin{bmatrix} G_0(z) & G_1(z) \end{bmatrix} \cdot \frac{1}{2} \begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_1(-z) \end{bmatrix} \cdot \begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{6.5}$$

que puede ser expandida como

$$\hat{X}(z) = \frac{1}{2}[G_0(z)H_0(z) + G_1(z)H_1(z)]X(z) + \frac{1}{2}[G_0(z)H_0(-z) + G_1(z)H_1(-z)]X(-z) =$$

$$F_0(z)X(z) + F_0(z)X(-z)$$

donde la función $F_0(z)$ describe las caracteristicas de tranferencia del banco de filtros. La función $F_1(z)$ denota los componentes de aliasing, los cuales son producidos por traslape de la respuesta en frecuencia. Si $F_1(z)$ es igual a cero, entonces se dice que se obtuvo un banco de filtros libre de aliasing (alias-free filter bank).

$$F_0(z) = \frac{1}{2}G_0(z)H_0(z) + \frac{1}{2}G_1(z)H_1(z) = z^{-k} \tag{6.6}$$

La función $F_0(z)$ denota la calidad de la reconstrucción. Si sólo se presentan retrasos, por ejemplo $F_0(z) = z^{-k}$, se dice que el filtro realiza una reconstrucción perfecta. Escribiendo estos dos requirimientos en forma matricial, se tiene

$$\frac{1}{2} \begin{bmatrix} G_0(z) & G_1(-z) \end{bmatrix} \begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_1(-z) \end{bmatrix} = \begin{bmatrix} z^{-k} & 0 \end{bmatrix} \tag{6.7}$$

El banco de filtros de análisis y de síntesis deben ser elegidos de tal manera que la ecuación (6.7) se satisfaga lo más cercanamente posible. Para realizar esto, el requerimiento de buena selectividad debe ser tomado en cuenta.

6.2.2. Bancos QMF estándar

Los primeros filtros sugeridos para satisfacer la ecuación (6.7), se denominaron *Filtros Espejo en Cuadratura (QMF)*. [5] Utilizando un prototipo pasobajas, a continuación se especifícan los siguientes filtros de análisis y síntesis

$$H_0(z) = H(z) \tag{6.8}$$

$$H_1(z) = H(-z) \tag{6.9}$$

$$G_0(z) = 2H(z) \tag{6.10}$$

$$G_1(z) = -2H(-z) \tag{6.11}$$

Con operaciones aritméticas se puede comprobar que la condición $F_1(z) = 0$ se cumple,en este caso se puede decir que los componentes de aliasing se cancelan entre sí.

El factor de 2 en los filtros de síntesis son cancelados por el factor 1/2 que es introducido por el submuestro. Se puede observar también que $H(-z)$ es pasoaltas si H(z) es pasobajas, ya que un cambio de signo en z es equivalente a un desplazamiento de π en la respuesta en frecuencia.

Si se substituye de la ecuación (6.8) a la (6.11) en (6.6) se obtiene una condición para la reconstrucción de la señal:

$$H^2(z) - H^2(-z) = z^{-k} \tag{6.12}$$

Para logar una perfecta reconstrucción, el prototipo $H(z)$ debe satisfacer la condición (6.12)

6.3. Banco de filtros FIR de dos canales con reconstrucción perfecta

Un banco de filtros de dos canales, de reconstrucción perfecta y con fitros FIR de fase lineal puede ser diseñado si se cumple:

$$\mid H_0(e^{jw}) \mid^2 + \mid H_1(e^{jw}) \mid^2 = 1$$

donde $H_0(e^{jw})$ y $H_1(e^{jw})$ son filtros de análisis.

6.3.1. Matrices de Modulación

Para un banco de filtros de dos canales la relación entre la señal de entrada y de salida se expresa mediante la ecuación:

$$Y(z) = \frac{1}{2}[H_0(z)G_0(z) + H_1(z)G_1(z)]X(z) + \frac{1}{2}[H_0(-z)G_0(z) + H_1(-z)G_1(z)]X(-z)$$

la cual puede ser expresada en forma matricial:

$$Y(z) = \frac{1}{2}[G_0(z)G_1(z)] \begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_0(-z) \end{bmatrix} \begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{6.13}$$

La ecuación (6.13) puede escribirse

$$Y(-z) = \frac{1}{2}[G_0(-z)G_1(-z)] \begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_0(-z) \end{bmatrix} \begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{6.14}$$

combinando las ecuaciones (6.13) y (6.14) se obtiene

$$\begin{bmatrix} Y(z) \\ Y(-z) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} G_0(z) & G_1(z) \\ G_0(-z) & G_1(-z) \end{bmatrix} \begin{bmatrix} H_0(z) & H_0(-z) \\ H_1(z) & H_0(-z) \end{bmatrix} \begin{bmatrix} X(z) \\ X(-z) \end{bmatrix}$$

$$\begin{bmatrix} Y(z) \\ Y(-z) \end{bmatrix} = \frac{1}{2} G^{(m)}(z)[H^{(m)}(z)]^T \cdot \begin{bmatrix} X(z) \\ X(-z) \end{bmatrix} \tag{6.15}$$

donde

$$H^{(m)}(z) = \begin{bmatrix} H_0(z) & H_1(z) \\ H_0(-z) & H_1(-z) \end{bmatrix}$$

es la matriz de modulación de análisis y

$$G^{(m)}(z) = \begin{bmatrix} G_0(z) & G_1(z) \\ G_0(-z) & G_1(-z) \end{bmatrix}$$

es la matriz de modulación de síntesis.

6.3.2. Condición para reconstrucción perfecta

Para la reconstrucción perfecta se debe tener

$$Y(z) = z^{-l} \cdot X(z)$$

y consecuentemente

$$Y(-z) = (-z)^{-l} \cdot X(-z) \tag{6.16}$$

Si se sustituye la ecuación (6.16) en la (6.15) se obtiene:

$$G^{(m)}(z) \cdot [H^m(z)]^T = 2 \begin{bmatrix} z^{-l} & 0 \\ 0 & (-z)^{-l} \end{bmatrix} \tag{6.17}$$

que puede ser expresada como:

$$G^{(m)}(z) = 2 \begin{bmatrix} z^{-l} & 0 \\ 0 & (-z)^{-l} \end{bmatrix} \left(\left[H^{(m)}(z) \right]^T \right)^{-1} \tag{6.18}$$

Desde la ecuación (6.18) se puede obtener las ecuaciones para calcular $G_0(z)$ y $G_1(z)$ si se conoce $H_0(z)$ y $H_1(z)$:

$$G_0(z) = \frac{2}{c} z^{l-k} H_1(z)$$

$$G_1(z) = -\frac{2}{c} z^{-(l-k)} H_0(-z)$$

6.3.3. Bancos de Filtros Biortogonales

Para un banco de filtros de reconstrucción perfecta, $P(z)$, denomidado *filtro producto normalizado*, debe ser paso bajas de banda media con fase cero. P(z) es un polinomio simétrico de la forma

$$P(z) = 1 + p_1(z + z^{-1}) + p_3(z^3 + z^{-3}) + p_5(z^5 + z^{-5}) + \cdots$$

La forma general de $P(z)$ está dada por la ecuación (6.19)

$$P(z) = (1 + z^{-1})^m \cdot (1 + z)^m \cdot R(z) \tag{6.19}$$

donde $R(z)$ es un polinomio simétrico, con

$$R(e^{j\theta}) \geq 0$$

Estos clase de filtros de banda media se denominan *Binomial* o también *filtro Maximamente Plano*. Si $R(z)$ tiene la forma

$$R(z) = r_0 + \sum_{s=1}^{m-1} r_s(z^s + z^{-s})$$

se dice que es de *grado minimo*, en este caso $R(z)$ resulta de suma importancia.

Por ejemplo si $m = 1$ y $R(z) = \frac{1}{2}$, de la ecuación 6.19 se obtiene

$$P(z) = \frac{1}{2}(z + 2 + z^{-1})$$

$$P(z) = \frac{1}{2}z(1 + z^{-1})(1 + z^{-1}) = z^l H_0(z) \cdot G_0(z)$$

Si se elige $l = 1$ entonces

$$H_0(z) = \frac{1}{\sqrt{2}}(1 + z^{-1})$$

$$G_0(z) = \frac{1}{\sqrt{2}}(1 + z^{-1})$$

lográndose así el orden más bajo.

Para el caso $m = 2$ y $R(z) = az + b + az^{-1}$ entonces,

$$P(z) = (1 + z^{-1})(1 + z)^2 \cdot (az + b + az^{-1})$$

$$P(z) = az^3 + (4a + b)z^2 + (7a + 4b)z + (8a + 6b) + (7a + 4b)z^{-1} + (4a + b)z^{-2} + az^3$$

Para que $P(z)$ sea un polinómio simétrico, el coeficiente en z^0 debe ser 1 y las potencias pares de z no deben estar presentes, por lo tanto,

$$\begin{array}{l} 8a + 6b = 1 \\ 4a + b = 0 \quad \Rightarrow a = -\frac{1}{16}\ b = \frac{1}{4} \end{array} \tag{6.20}$$

entonces se tiene,

$$P(z) = [1 + \frac{1}{2}(z + z^{-1})]^2 \cdot [1 - \frac{1}{4}(z + z^{-1})]$$

$$P(z) = \frac{1}{16}z^3(1 + 2z^{-1} + z^{-2})^2 \cdot (-1 + 4z^{-1} - z^{-2}) = G_0 \cdot H_0 \cdot c \cdot z^l$$

La separación de P(z) a $G_0(z)$ y $H_0(z)$ puede ser por algunas de los métodos que se presentan a continuación:

Separación Le Gall 3/5

$$
\begin{aligned}
H_0(z) &= \tfrac{1}{2}(1 + 2z^{-1} + z^{-2}) \\
G_0(z) &= \tfrac{1}{8}(1 + 2z^{-1} + z^{-2})(-1 + 4z^{-1} - z^{-2}) \\
H_1(z) &= \tfrac{1}{8}(-1 - 2z^{-1} + 6z^{-2} - 2z^{-3} - z^{-4}) \\
G_1(z) &= -\tfrac{1}{2}(1 - 2z^{-1} + z^{-2})
\end{aligned}
\tag{6.21}
$$

Analizando el circuito en la figura 6.5 con las funciones de transferencia de la ecuación 6.21, mediante el Simulink, se obtuvieron las gráficas en las figuras 6.6 y 6.7

En la figura (6.6) se muestra la señal de voz en la entrada y a la salida, mientras que en la figura (6.7) se muestra el espectrogama de la señal de voz en la entrada y en la salida para el caso de la separación de LeGall 3/5. De los espectrogramas de la figura 6.7 se puede observar que el correspondiente a la señal de salida del banco de filtros es idéntico al espectograma de la señal de voz de la entrada.

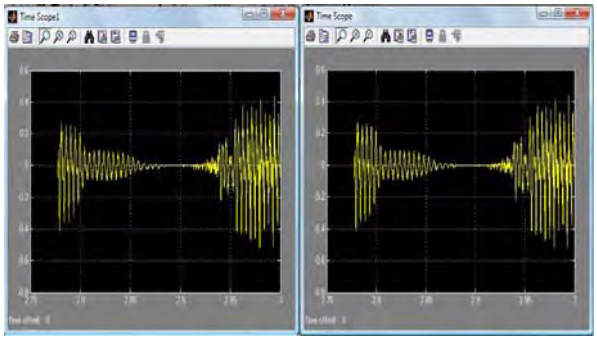


Figura 6.6: Señal de entrada y salida en el tiempo. Le Gall 3/5

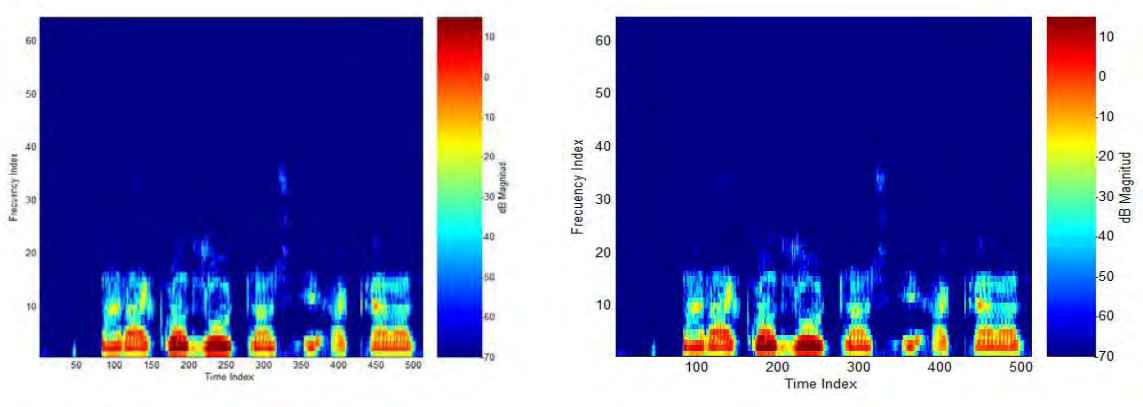


Figura 6.7: Señal de entrada y salida en la frecuencia. Le Gall 3/5

Separación de Daubechies 4/4

La separación del polinomio $P(z)$ a las funciones de tranferencia $H_0(z)$ y $G_0(z)$ es la siguiente,

$$
\begin{aligned}
H_0(z) &= \tfrac{1}{8}(1 + 3z^{-1} + 3z^{-2} + z^{-3}) \\
G_0(z) &= \tfrac{1}{2}(-1 + 3z^{-1} + 3z^{-2} - z^{-3}) \\
H_1(z) &= \tfrac{1}{8}(-1 - 3z^{-1} + 3z^{-2} + z^{-3}) \\
G_1(z) &= \tfrac{1}{2}(-1 + 3z^{-1} - 3z^{-2} + z^{-3})
\end{aligned}
\tag{6.22}
$$

Analizando el circuito de la figura 6.5, mediante Simulink, con las funciones de transferencia de la ecuación (6.22), se obtuvieron las gráficas que se muestran en las figuras 6.8 y 6.9.
En la figura (6.8) se muestra la señal de voz en la entrada y a la salida, mientras que en la figura (6.9) se muestra el espectrogama de la señal de voz en la entrada y en la salida para el caso de la separación de Daubechies 4/4.

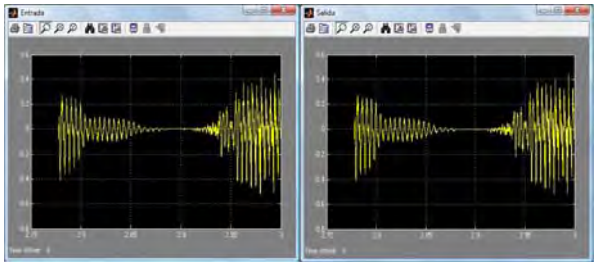


Figura 6.8: Señal de entrada y salida en el tiempo. Daubechies 4/4

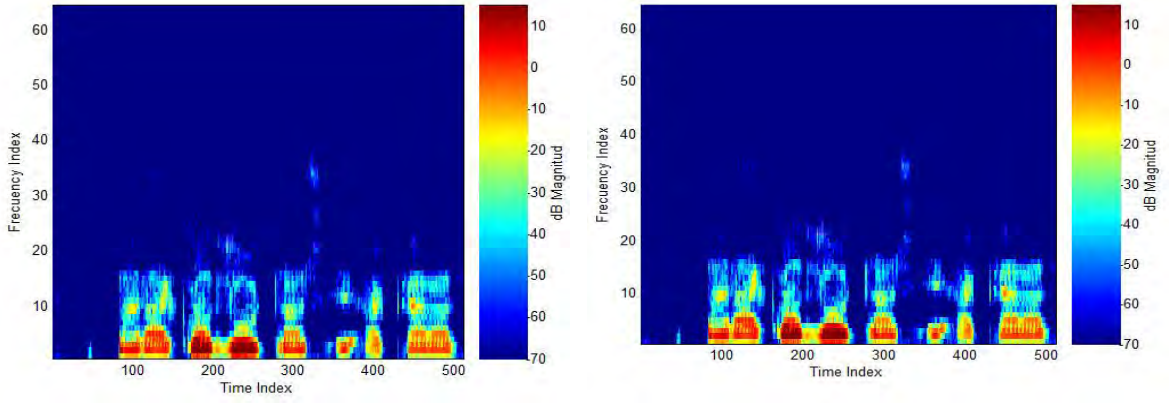


Figura 6.9: Señal de entrada y salida en la frecuencia. Daubechies 4/4

Como se puede ver de la figura 6.9 los espectrogramas de la entrada y de la salida del banco de filtros son idénticos. En el espectro de la salida no faltan ni las frecuencias bajas ni las frecuencias altas a pesar de que sólo se transmitió cada segunda muestra de la señal de entrada. También de la figura 6.8 se ve que la respuesta en el tiempo es idéntica a la señal de entrada.

6.4. Banco de Filtros con estructura de árbol: Cuatro Canales.

En la sección pasada se analizó el banco de filtros de dos canales y se desarrollaron las condiciones para satisfacer, con los filtros de análisis y de síntesis, la recostrucción perfecta. También es posible

desarrollar un banco de filtros de análisis y/o síntesis multibanda insertando uno o más bancos QMF de dos canales. Además si un banco QMF es de reconstrucción perfecta, la estructura multibanda generada tambien posee la propiedad de reconstrucción perfecta.

6.4.1. Banco de filtros con igual banda de paso

Si se inserta un banco QMF de dos canales maximamente decimado en cada canal de otro banco QMF de dos canales maximamente decimado, entre el submuestreador y el sobremuestreador, se puede generar un banco QMF de cuatro canales maximamente decimado, como el que se muestra en la figura 6.10. Debido a que los bancos de filtros de análisis y de síntesis tiene ahora una forma como de árbol. El sistema completo es a menudo denominado *Banco de filtros con estructura de árbol*.

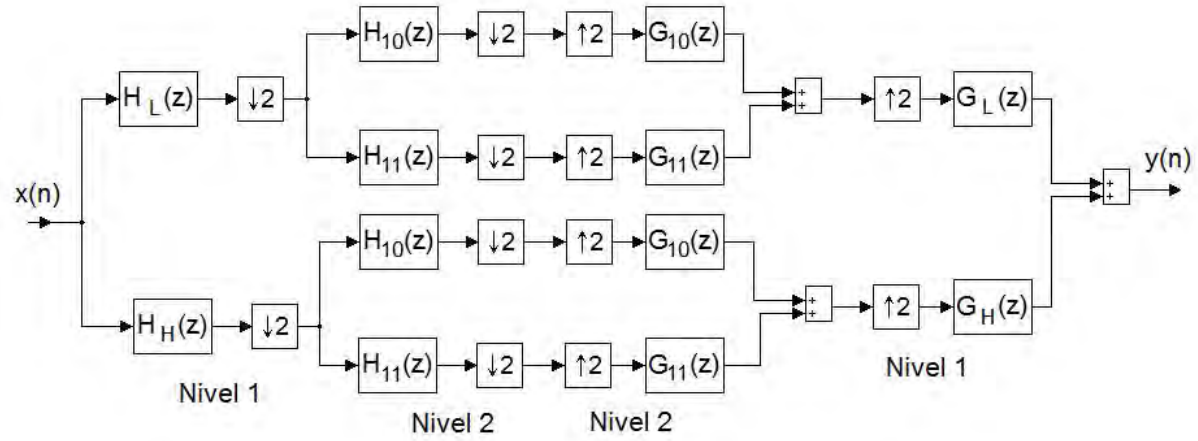


Figura 6.10: Banco QMF de cuatro canales.

Es necesario hacer notar que en el figura 6.10, los bancos QMF de dos canales en el segundo nivel no tienen que ser identicos. Sin embargo, si existen diferentes bancos QMF con diferentes filtros de análisis y síntesis, para compensar la desigualdad de ganancias y retardos de los dos sistemas de dos canales, es necesario insertar retardos adicionales con el valor apropiado a la mitad para asegurar una recostrucción perfecta del sistema completo de cuatro canales.
Una representación equivalente del sistema QMF de cuatro canales de la figura 6.10 se muestra en la figura 6.11. Los filtros de análisis y de síntesis en la representación equivalente están relacionados con los de la estructura de la figura 6.10 de la siguiente manera:

$$\begin{aligned} H_0(z) &= H_L(z)H_{10}(z^2), \quad H_1(z) = H_L(z)H_{11}(z^2) \\ H_2(z) &= H_H(z)H_{10}(z^2), \quad H_3(z) = H_H(z)H_{11}(z^2) \\ G_0(z) &= G_L(z)G_{10}(z^2), \quad G_1(z) = G_L(z)H_{11}(z^2) \\ G_2(z) &= G_H(z)G_{10}(z^2), \quad G_3(z) = G_H(z)H_{11}(z^2) \end{aligned} \tag{6.23}$$

De las ecuaciones en (6.23) se puede ver que cada filtro de análisis $H_i(z)$ es una cascada de dos filtros, uno con una banda de paso y una banda de supresión simples y otro con dos bandas de

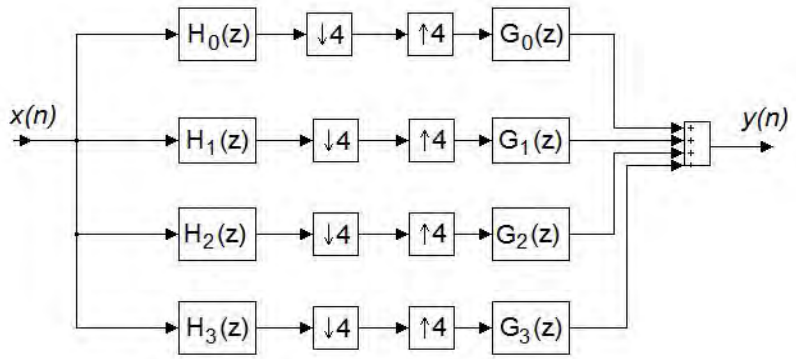


Figura 6.11: Banco QMF de cuatro canales.

paso y dos bandas de supresión. La banda de paso de la cascada es el rango de frecuencia donde las bandas de paso de los dos filtro se traslapan, por otra parte, la banda de supresión de la cascada está formada por tres diferentes rangos de frecuencia, en dos de estos rangos, la banda de paso de un filtro coincide con la banda de supresión del otro, mientras que en el tercer rango, las dos bandas de supresión coninciden. Como resultado de lo anterior, la ganancia de la cascada no es igual en cada una de las tres regiones de la banda de supresión provoncando una atenuación característica dispareja en dicha banda. Este tipo de comportamiento debe ser tomado en cuenta en el diseño del banco de filtros con estructura de árbol.

Bancos QMF con más de cuatro canales pueden ser fácilmente construidos. Es necesario hacer notar que el número de canales resultantes a partir de este método está restringido a una potencia de 2, por ejemplo $L = 2^v$. Además como se muestra en el figura 6.11, los filtros en la rama de análisis (síntesis) tienen bandas de paso de igual ancho dado por π/L. Sin embargo, con una simple modificación de la forma de realización, es posible diseñar bancos QMF con filtros de análisis y síntesis teniendo diferentes bandas.

6.5. Banco de Filtros de L Canales

La estructura básica de un banco de filtros QMF de L canales es el que se muestra en la figura 6.12.

6.5.1. Representación Polifase

Considerando la representación polifase del $k-$ésimo filtro de análisis $H_k(z)$:

$$H_k(z) = \sum_{l=0}^{L-1} z^{-l} E_{kl}(z^L) \quad 0 \leq k \leq L-1$$

Una representación matricial del grupo de ecuaciones propuestas arriba estaría dada por

$$\mathbf{h}(z) = \mathbf{E}(z^L)\mathbf{e}(z)$$

donde

$$\mathbf{h}(z) = \left[\begin{array}{cccc} H_0(z) & H_1(z) & \cdots & H_{L-1}(z), \end{array} \right]^t,$$

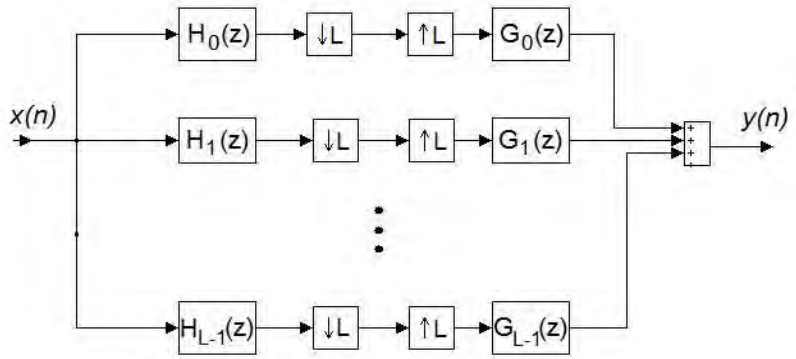


Figura 6.12: Estructura básica del banco de filtros QMF de L canales.

$$\mathbf{e}(z) = \begin{bmatrix} 1 & z^{-1} & \cdots & z^{L-1} \end{bmatrix}^t$$

y

$$\mathbf{E}(z) = \begin{bmatrix} E_{00}(z) & E_{01}(z) & \cdots & E_{0,L-1}(z) \\ E_{10}(z) & E_{11}(z) & \cdots & E_{1,L-1}(z) \\ \vdots & \vdots & \ddots & \vdots \\ E_{L-1,0}(z) & E_{L-1,1}(z) & \cdots & E_{L-1,L-1}(z) \end{bmatrix}$$

la matriz $\mathbf{E}(z)$ definida arriba es llamada *Matriz de componentes polifase Tipo I*. La figura 6.13a muestra el banco de filtros de análisis con una reresentación polifase Tipo I.

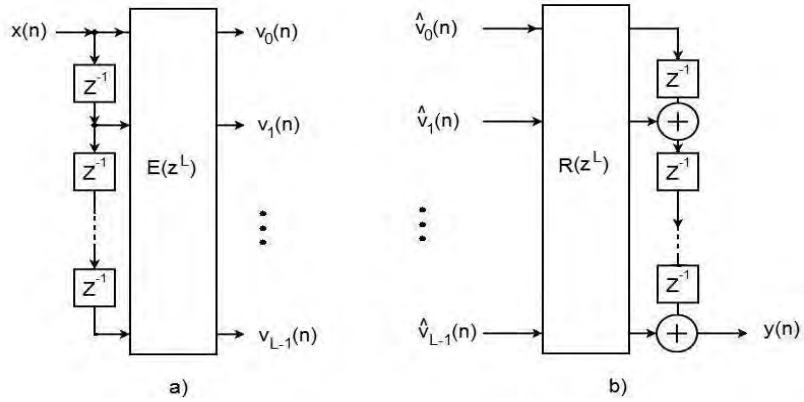


Figura 6.13: a) Representación polifase Tipo I para el banco de filtros de análisis. b)Representación polifase Tipo II para el banco de filtros de Síntesis.

Asimismo, se puede representar los L filtros de síntesis en una forma polifase Tipo II:

$$G_k(z) = \sum_{l=0}^{L-1} z^{-(L-1-l)} \mathbf{R}_{lk}(z^L), \quad 0 \leq k \leq L-1$$

En forma matricial, el grupo de ecuaciones de arriba se puede reescribir como

$$\mathbf{g}^t(z) = z^{-(L-1)} \tilde{e}(z) \mathbf{R}(z^L)$$

donde

$$\mathbf{g}(z) = \left[\begin{array}{cccc} G_0(z) & G_1(z) & \cdots & G_{L-1}(z), \end{array} \right]^t,$$

$$\tilde{\mathbf{e}}(z) = \left[\begin{array}{cccc} 1 & z & \cdots & z^{L-1} \end{array} \right] = \tilde{\mathbf{e}}^t(z^{-1}),$$

y

$$\mathbf{R}(z) = \left[\begin{array}{cccc} R_{00}(z) & R_{01}(z) & \cdots & R_{0,L-1}(z) \\ R_{10}(z) & R_{11}(z) & \cdots & R_{1,L-1}(z) \\ \vdots & \vdots & \ddots & \vdots \\ R_{L-1,0}(z) & R_{L-1,1}(z) & \cdots & R_{L-1,L-1}(z) \end{array} \right]$$

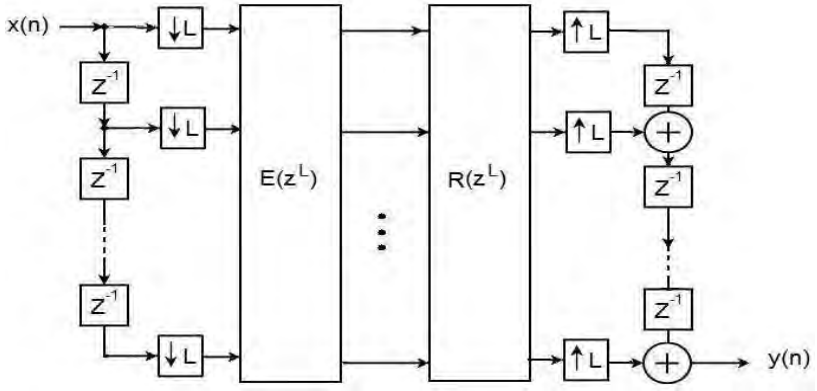


Figura 6.14: Estructura del Banco QMF de L canales basado en las representaciones polifase de los bancos de filtros de análisis y síntesis

La matriz $\mathbf{R}$(z) definida anteriormente es llamada *matriz de componentes polifase tipo II*. La figura 6.13 muestra la representación polifase tipo II del banco de filtros de síntesis. Haciendo uso de las representaciones polifase de la figura 6.13 en la figura 6.12 y de las equivalencias en cascada de la figura 5.6, se obtiene una realización equivalente del banco QMF de L canales, el cual se muestra en la figura 6.14.

6.5.2. Condición para reconstrucción perfecta

Si las matrices de los componentes polifase de la figura 6.14 satisfacen la relación

$$\mathbf{R}(z)\mathbf{E}(z) = c\mathbf{I} \tag{6.24}$$

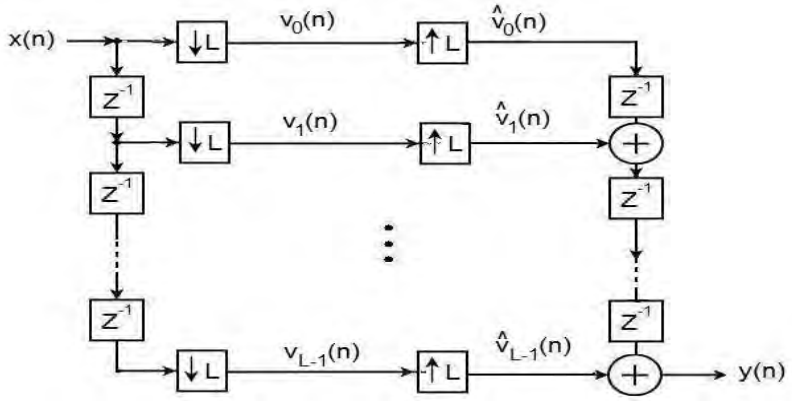


Figura 6.15: Estructura del Banco QMF de L canales basado en las representaciones polifase de los bancos de filtros de análisis y síntesis

donde $\mathbf{I}$ es una matriz identidad de LxL y c es una constante, la estructura de la figura 6.14 se reduce a la que se muestra en la figura 6.15. Comparando la figura 6.15 y 6.12 se observa que la primera puede ser considerada como un caso especial de un banco QMF de L canales si considera

$$H_k(z) = z^k, \quad G_k(z) = z^{-(L-1-k)}, 0 \leq k \leq L-1. \tag{6.25}$$

Por lo tanto, una condición más general para la condición perfecta está dada por

$$\mathbf{R}(z)\mathbf{E}(z) = cz^{-K}\mathbf{I} \tag{6.26}$$

donde K es un entero no negativo.

Capítulo 7

Microprocesador TMS320C6713

Existe una gran cantidad de aplicaciones dentro del procesamiento digital de señales, en áreas, tales como: radar, sonar, voz, comunicaciones, telefonía, medicina, control, sismología, imágenes, etc. Las herramientas que han permitido obtener soluciones reales y eficientes son el desarrollo de las tecnologías de programación y los microprocesadores de procesamiento digital de señales.

En la actualidad se ha presentado un amplio crecimiento en la industria digital esto se debe en gran medida al desarrollo de algoritmos de procesamiento digital de señales. Dentro de muchas áreas se requiere tener aplicaciones, tales como filtrado, compresión, análisis en frecuencia, entre otras. El uso de sistemas digitales permite realizar estas tareas con una computadora digital o un microprocesador. En este capítulo se explicarán las características del microprocesador que se utilizar en el diseño del banco de filtros.

7.1. Características del TMS320C6713

El dispositivo TMS320C6713, ejecuta un máximo de 8 instrucciones, de 32 bits por ciclo. El CPU contiene 32 registros de propósito general, de 32 bits y 8 unidades funcionales. Este dispositivo tienen un conjunto completo de herramientas de desarrollo y optimización, que incluyen un compilador C eficiente, un optimizador de ensamble para simplificar la planificación y programación del lenguaje ensamblador, y un depurador con interfase gráfica, basada en Windows, para visualizar las características de ejecución en el código fuente. Además, contiene una tarjeta de emulación de hardware compatible con la interfase del emulador TI XDS510.

Las características del dispositivo TMS320C6713, incluyen

1. Un CPU avanzado VLINW (very long intruction word) con 8 unidades funcionales, que incluyen 2 multiplicadores y 6 ALU's (Unidades Lógico Aritméticas).
   a) Ejecuta un máximo de 8 instrucciones por ciclo, 10 veces más que los DSP's típicos.
   b) Permite rápido tiempo de desarrollo en diseños con código RISC altamente efectivos.
2. Empaquetado de instrucción
   a) Obtiene el tamaño del código equivalente a las 8 instrucciones ejecutadas serialmente o en paralelo.
   b) Reduce el tamaño del código y el consumo de energía.

3. Ejecución condicional de todas las instrucciones.

   a) Reduce los saltos costosos.

   b) Incrementa el paralelismo para mantener en alto desempeño

4. Ejecuta el código programado, en unidades funcionales independientes.

5. Proporciona soporte eficiente de memoria para una variedad de aplicaciones de 8, 16 y 32 bits de datos.

6. Maneja operaciones artiméticas de 40 bits, adicionando presición extra a vocoders y otras aplicaciones computacionalmente intensivas.

7. Proporciona soporte de normalización y saturación, en operaciones aritméticas claves.

8. Soporta operaciones comunes, halladas en aplicaciones de control y manipulación de datos, como manipulación de campos, extracción de instrucción, activación, desactivación y conteo de bits.

Además, el TMS320C6713 tiene las siguientes características:

- Un máximo de 1336 MIPS (millones de instrucciones por segundo), a 167 MHz.
- Un máximo de 1 G FLOPS (operaciones de punto flotante por segundo), a 167 MHz, para operaciones de precisión simple.
- Un máximo de 250 M FLOPS a 167 MHz, para operaciones de doble precisión.
- Un máximo de 688 M FLOPS a 167 MHz, para operaciones de multiplicación y acumulación.
- Soporte de Hardware para operaciones de punto flotante, de simple y doble presición.
- Multiplicación entera de 32x32 bits, con resultado de 32 o 64 bits.

El dispositivo TMS320C6713 tiene una gran variedad de opciones de memoria y periféricos tal como: Amplia memoria RAM para ejecución rápida de algoritmos, acceso a la memoria por medio del puerto host, controlador multicanal DMA, puertos serie multicanal y un timer de 32 bits.

7.2. Arquitectura del dispositivo TMS320C67X

El procesador TMS320C6713, consiste en tres partes: el CPU, los periféricos y la memoria. Además ocho unidades funcionales operan en paralelo (seis ALU's y dos multiplicadores), con dos conjuntos similares de cuatro unidades funcionales básicas. Las unidades se comunican usando un camino cruzado entre dos clasificaciones de registros, cada uno de los cuales contiene 16 registros de 32 bits. La figura (7.1) muestra el diagrama de bloques del dispositivos TMS320C6713.

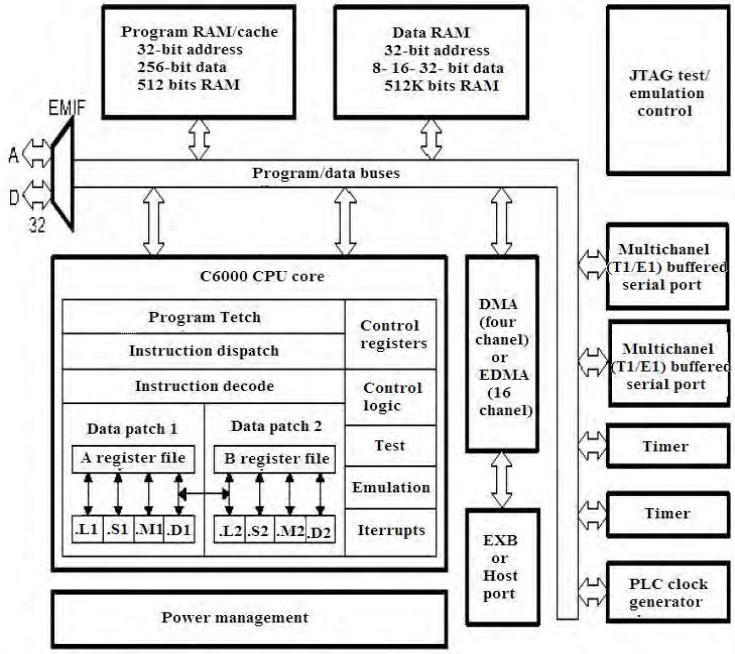


Figura 7.1: Diagrama de bloques de TMS320C6713.

7.2.1. Unidad de procesamiento Central (CPU)

El CPU contiene:

- Unidad fetch de programa
- Unidad de despacho de instrucción
- Unidad de decodificación de instrucción
- 32 registros de 32 bits
- Dos caminos de datos, cada uno con cuatro unidades funcionales
- Registros de control
- Lógica de control
- Lógica de interrupción

El CPU tiene dos caminos de datos (A y B), cada camino tiene cuatro unidades funcionales (.L, .S, .M y .D) y un archivo de registros de 32 bits (register file). Las unidades funcionales ejecutan operaciones de lógica, corrimiento, multiplicación y direccionamiento de datos. Todas las instrucciones aceptan operaciones de carga y almacenamiento sobre los registros. Las dos unidades de direccionamiento de datos (.D1 y .D2) son exclusivamente responsables de toda la transferencia de datos entre los archivos de registros y la memoria.

7.2.2. Caminos de datos del CPU

Los caminos de datos del CPU consisten de: dos archivos de registros de propósito general (A y B), ocho unidades funcionales (.L1, .L2, .S1, .S2, .M1, .M2, .D1 y .D2), dos caminos de lectura de memoria (LD1 y LD2), dos caminos de almacenamiento en memoria (ST1 y ST2), dos caminos cruzados entre los archivos de registros(1X y 2X) y dos caminos de direccionamiento de datos (DA1 y DA2). La figura 7.2 muestra el camino de los datos del CPU.

7.2.3. Archivos de registros de propósito general (register files)

Hay dos archivos de registros de propósito general (A y B) en los caminos de datos del TMS320C6713. Cada uno de esos archivos contiene 16 registros de 32 bits (A0-A15) para el archivo A y (B0-B15) para el archivo B. Los registros de propósito general pueden ser usados para manejar datos o punteros de direccionamiento de estos. Los registros A1, A2, B0, B1 y B2 pueden ser utilizados como registros de condición. Los registros A4-A7 y B4-B7 pueden ser usados para el direccionamiento circular.

Los archivos de registros de propósito general soportan datos de 32 y 40 bits de punto fijo. Los datos de 32 bits, pueden estar contenidos en cualquier registro de propósito general. Los datos de 40 bits están contenidos en dos registros; los 32 bits menos significativos del dato (LSB) son colocados en un registro par y los restantes ocho bits mas significativos (MSB) son colocados en los ocho bits menos significativos del registro próximo superior (que es siempre un registro impar). El TMS320C6713 también usa ese par de registros para colocar valores de punto flotante de doble precisión de 64 bits.

7.2.4. Unidades funcionales

Las ocho unidades funcionales en los caminos de datos del TMS320C6713 pueden ser divididas en dos grupos de cuatro; cada unidad funcional, en un camino de datos, es casi idéntica a la unidad correspondiente, en el otro camino de datos por ejemplo, .L es muy similar a .L2. Las unidades funcionales son descritas en la tabla 7.1

La mayoría de los caminos en el CPU, soportan operaciones de 32 bits, y algunas soportan operaciones largas (40 bits). Cada unidad funcional tiene su propio puerto de escritura de 32 bits, en un archivo de registros de propósito general. Todas las unidades terminan en 1 (por ejemplo, .L1) cuando se refiere al archivo de registros A y en 2 cuando se refiere al archivo de registros B. Cada unidad funcional tiene dos puertos de lectura de 32 bits, para cada operando src1 y src2. Cuatro unidades (.L1, .L2, .S1 y .S2) tienen un puerto extra de 8 bits para ejecutar operaciones de 40 bits. Debido a que cada unidad tiene su propio puerto de escritura de 32 bits, las ocho unidades pueden ser usadas en paralelo en cada ciclo.

7.2.5. Archivos de registros de control del TMS320C6713

Una unidad (.S2) puede leer y escribir hacia los registros de control. Mostrados en la figura 7.2. La tabla 7.1, menciona y describe, los registros de control contenidos en el archivo de registros de control. Cada registro es accesado con la instrucción MVC.

El TMS320C6713 posee tres registros de configuración adicionales, para soportar operaciones de punto flotante tabla ref. Los registros especifican los modos de redondeo de punto flotante para las unidades .M y .L. También contienen campos de bit para advertir si src1 y src2 son NaN (no es un

Unidad Funcional	Operaciones en punto fijo	Operaciones en punto flotante
Unidad .L(.L1, .L2)	Operaciones aritméticas y comparación de 32 y 40 bits. Cuenta de 0s o 1s mas a la izguierda, para 32 bits. Normalización de 32 y 40 bits.	Operaciones lógicas Operaciones de conversión $DP \rightarrow SP$, $INT \rightarrow DP$, $INT \rightarrow SP$
Unidad .S(.S1, .S2)	Operaciones aritméticas de 32 bits Corrimientos de 32/40 bits y operaciones campos de bits en 32 bits. Operaciones lógicas de 32 bits. Saltos. Generación de constantes. Transferencia de registros de/hacia registros (solamente.S2)	Comparación recíproca. Operaciones de raíz cuadrada Operaciones de valor absoluto. Operaciones de Conversión SP a DP
Unidad .M(.M1,.M2)	Operaciónes de multiplicación de 16x16 bits	Operaciones de multiplicación de 32x32 bits. Operaciones de multiplicación de punto flotante
Unidad .D(.D1,.D2)	Sumas, restas y cálculos de direccionamiento circular de 32 bits Carga y almacenamiento con offset constante de 5 bits. Carga y almacenamiento con offset constante de 15 bits. (solo .D2)	Lectura de palabras dobles con offset constante de 5 bits

Cuadro 7.1: Unidades funcionales y las operaciones realizádas

número) o números desnormalizados. Además si resulta overflow o underflow, es inexacto, infinito o invalido. Hay campos que advierten si una división por cero fue ejecutada o si una comparación fue ejecutada con un NaN.

7.2.6. Caminos entre archivos de registros (Register File Cross Paths)

Cada unidad funcional lee directamente de y escribe directamente hacia el archivo de registros, dentro de su propio camino de datos. Esto es, las unidades .L1, .S1, .D1 y .M1 escriben en el archivo de registros A y las unidades .L2, .S2, .D2 y .M2 escriben en el archivo de registros B. Los archivos de registros son conectados a las unidades funcionales del archivo de registros opuesto, através de los caminos cruzados 1X y 2X. Esos caminos cruzados permiten a las unidades funcionales, de un camino de datos, accesar a operandos de 32 bits, del lado opuesto. El camino cruzado 1X permite a las unidades funcionales del camino de datos A, leer su operando fuente del archivo de registros B. El camino cruzado 2X permite a las unidades funcionales del camino de datos B, leer su operando fuente del archivo de registros A.

7.2.7. Caminos de Memoria, Cargas y Almacenamiento

Hay dos caminos de 32 bits, para leer los datos de memoria en los registros de almacenamiento: LD1 para el archivo de registros A y LD2 para el archivo de registros B. El TMS320C6713 también tiene un segundo camino de carga de 32 bits para ambos archivos de registros A y B. Este segundo camino permite a la instrucción LDDW leer simultáneamente dos registros de 32 bits en los lados

Abreviatura	Nombre	Descripción
AMR	Registro de modo de direccionamiento	Especifica si utiliza direccionamiento lineal o circular para cada uno de los ocho registros; también contiene el tamaño para el direccionamiento circular
CSR	Registro de control de estado	Contiene el bit de interrupción global, los bits de control del cache y otros bits de control de estado misceláneos
IFR	Registro de bandera de interrupción	Despliega el estado de las interrupciones
ISR	Registros de para activar interrupción	Permite activar interrupciones manualmente
ICR	Registro para interrupción	Permite limpiar interrupciones pendientes manualmente
IER	Registro para retorno de interrupción	Permite habilitar / deshabilitar interrupciones individuales
NRP	Puntero de retorno de interrupción no mascarable	Contiene la dirección de retorno de una interrupción no mascarable
PCE1	Contador del programa, fase E1	Contiene la dirección del paquete fetch (contiene el paquete de ejecución del pipeline) en la etapa E1.

Cuadro 7.2: Registros de control

Abreviatura	Nombre	Descripción
FADCR	Registro de configuración del sumador del punto flotante	Especifica el modo underflow, modo de redondeo, NaN y otras excepciones para la unidad .L
FAUCR	Registro de configuración auxiliar de punto flotante	Especifica modos de underflow, modos de redondeo, NaN y otras excepciones para la unidad .S
FMCR	Registro de configuración del multiplicador de punto flotante	Especifica modos de underflow, modos de redondeo, NaN y otras excepciones para la unidad .M

Cuadro 7.3: Registros de control extendidos para el TMS320C6713

Unidad .L	Unidad .M	Unidad .S		Unidad .D	
ABS	MPY	ADD	SET		ADD STB(15-bitt offset) Solo .S2
ADD	MPYU	ADDK	SHL	ADDAB	STH (15-bitt offset) Solo .S2
ADDU	MPYUS	ADD2	SHR	ADDAH	STW(15-bitt offset) Solo .S2
AND	MPYSU	AND	SHRU	ADDAW	SUB
CMPEQ	MPYH	B disp	SHRL	LDB	SUBAB
CMPGT	MPYHU	B IRP	SUB	LDBU	SUBAH
CMPGTU	MPYHUS	B NRP	SUBU	LDH	SUBAW
CMPLT	MPYHSU	B reg	SUB2	LDHU	ZERO
CMPLTU	MPYHL	CLR	XOR	LDW	
LMBD	MPYHLU	EXT	ZERO	LDB	
MV	MPYHULS	EXTU		LDBU	
NEG	MPYHSLU	MV		LDH	
NORM	MPYLH	MVC		LDHU	
NOT	MPYLHU	MVK		LDW	
OR	MPYLUHS	MVKH		MV	
SADD	MPYLSHU	MVLKH		STB	
SAT	SMPY	NEG		STH	
SSUB	SMPYHL	NOT		STW	
SUB	SMPYLH	OR			
SUBU	SMPYH				
SUBC					
XOR					
ZERO					

Cuadro 7.4: Mapeo de instrucciones de punto fijo y unidades funcionales

A y B. Existen además dos caminos de 32 bits, ST1 y ST2, para almacenar valores de los registros a la memoria, para cada archivo de registros. Los caminos de lectura largos .L y .S son compartidos con los caminos de almacenamiento.

7.2.8. Caminos de direccionamiento de datos

Los caminos de direccionamiento de datos (DA1 y DA2) mostrados en la figura 7.2 colocados fuera de las unidades .D, permiten generar direcciones de datos de un archivo de registros. Con eso se sostienen cargas y almacenamientos en memoria, desde el otro archivo de registros. Sin embargo, las cargas y almacenamientos ejecutados en paralelo, debe cargar a y de el mismo archivo de registro. Aunque también existe la alternativa de que ambos usen un camino cruzado al registro opuesto.

7.2.9. Mapeo entre instrucciones y Unidades Funcionales

La tabla 7.4 muestra el mapeo entre las instrucciones y las unidades funcionales para las instrucciones de punto fijo del TMS320C6713. La tabla 7.5 muestra el mapeo entre las instrucciones y las unidades funcionales para las instrucciones de punto flotante del TMS320C6713.

Unidad .L	Unidad .M	Unidad .S	Unidad .D
ADDDP	MPYDP	ABSDP	ADDAD
ADDSP	MPYI	ABSSP	LDDW
DPINT	MPYID	CMPEQDP	
DPSP	MPYSP	CMPEQSP	
INTDP		CMPGTDP	
INTDPU	CMPGTSP		
INTSP		CMLTDP	
INTSPU	CMPLTSP		
SPINT		RCPDP	
SPTRUNC		RCPSP	
SUBDP		RSQRDP	
SUBSP		RSQRSP	
SPDP			

Cuadro 7.5: Mapeo de instrucciones de punto flotante y unidades funcionales

Tipo de direccionamiento	Ninguna modificación del registro de dirección	Preincremento o predecremento del registro del dirección	Postincremento o postdecremento del registro de dirección
Registro indirecto	$*R$	$*++R$	$*R++$
		$*--R$	$*R--CMPEQDP$
Registro relativo	$*+R[ucst5]$	$*++R[ucst5]$	$*R++[ucst5]$
	$*-R[ucst5]$	$*--R[ucst5]$	$*R--[ucst5]$
Base + index	$*+R[offsetR]$	$*++R[offsetR]$	$*R++[offsetR]$
	$*-R[offsetR]$	$*--R[offsetR]$	$*R--[offsetR]$

Cuadro 7.6: Generación de direccionamiento indirecto para Load/Store

7.3. Modos de direccionamiento

Los modos de direccionamiento son lineales por default para el TMS320C6713 aunque también existe el modo de direccionamiento circular. El modo de direccionamiento se especifica con el registro modo de direccionamiento (AMR).

Con todos los registros se puede ejecutar el direccionamiento lineal. Solo en ocho de ellos se puede ejecutar el direccionamiento circular: del A4 a A7 (que son usados por la unidad .D1) y del B4 a B7 (que son usados por la unidad D2). Ninguna otra unidades puede ejecutar direccionamiento circular. Las instrucciones LDB/LDH/LDW, STB/STH/STW, ADDAB/ADDAH/ADDAW, y SUBAB/SUBAH/SUBAW se apoyan en el registro AMR, para determinar que tipo de calculo del direccionamiento es ejecutado por esos registros.

El CPU del dispositivo TMS320C6713 tienen arquitectura de carga/almacenamiento, lo que significa que la única manera de accesar datos en memoria es con la instrucción de carga o almacenamiento. La tabla 7.6 muestra la síntaxis de un direccionamiento indirecto, para una localización en memoria.

7.4. Interrupciones

El CPU del dispositivo TMS320C6713 tienen 14 interrupciones. Estas son reset, la interrupción no mascarable (NMI) e interrupciones de la 4 a la 15. Estas interrupciones corresponden a las señales RESET, NMI e INT4-INT15 respectivamente, sobre los límites del CPU. En el mismo dispositivo TMS320C6713, estas señales pueden estar ligadas directamente a lo pines del dispositivo, conectando periféricos al chip, o pueden ser desactivadas permanentemente, cuando están ligadas e inactivas en el chip. Generalmente, RESET y NMI son conectadas directamente a los pines del dispositivo. Las características del servicio de interrupción incluyen:

- El pin IACK del CPU es usado para confirmar la recepción de una petición de interrupción
- Los pines INUM0 INUM3 indican el vector de interrupción que está siendo utilizado
- Los vectores de interrupción son reubicables
- Los vectores de interrupción consisten de un paquete fetch. Con los paquetes se proporciona un rápido servicio.

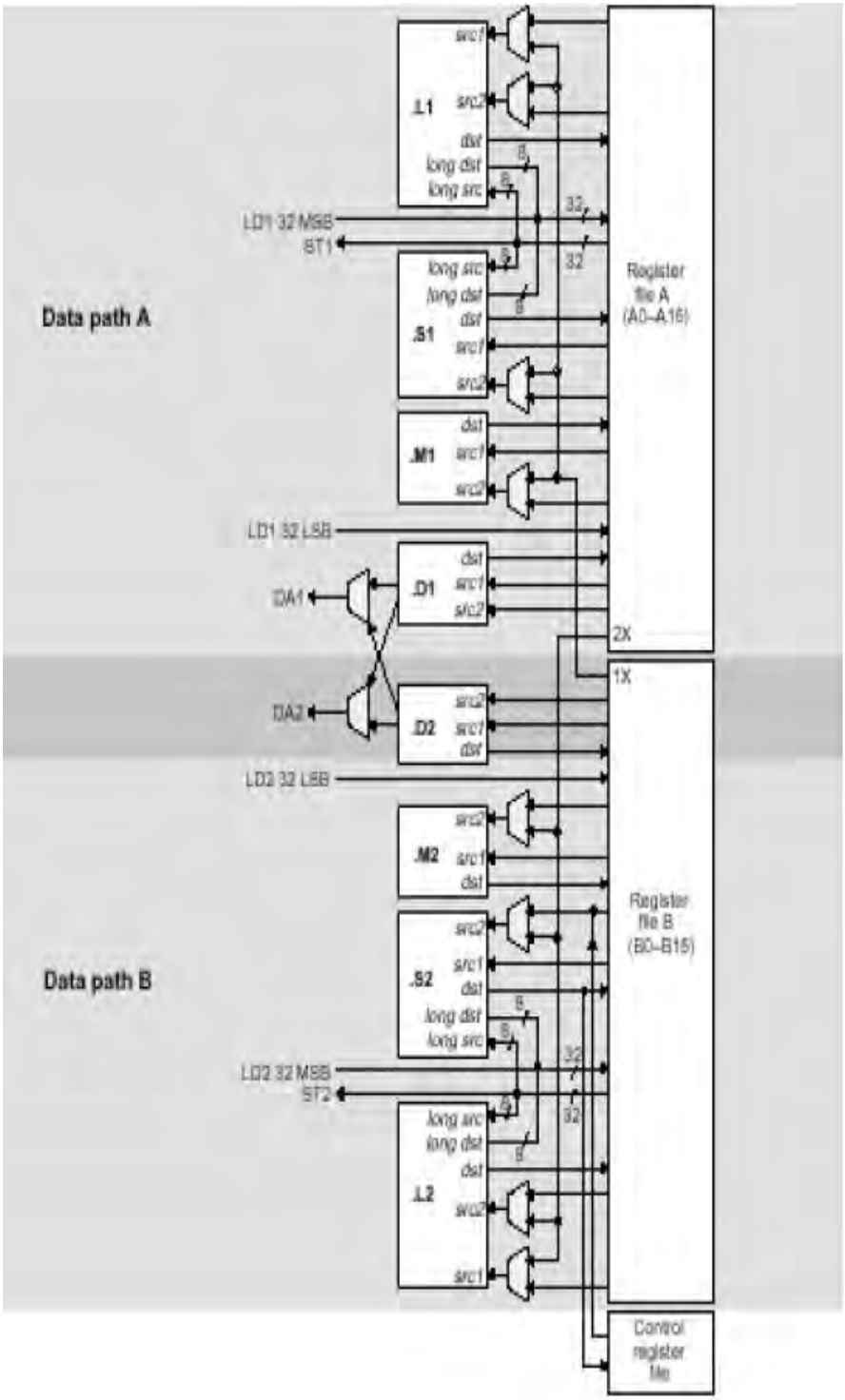


Figura 7.2: Camino de datos del TMS320C6713.

Capítulo 8

Diseño e Implementación

Una vez cubierta la parte teórica de este trabajo, en el presente capítulo se dará a conocer el proceso de diseño e implementación del banco de filtros para procesar audio. Primero se explicará el procedimiento seguido para el diseño de cada uno de los filtros del banco y posteriormente se abordará el proceso de implementación.

8.1. Diseño del Banco de Filtros

El objetivo es diseñar un banco de filtros de cuatro canales para un codificador de audio, lo más sencillo en este caso es la estructura de árbol o multinivel, como se vio en la sección 6.4. En dicha sección se mencionó lo sencillo que resulta pasar de una estructura QMF de dos canales a una QMF de cuatro canales, se dijo también que si la estructura incial era de reconstrucción perfecta, la resultante lo era también. Por tanto, lo imporante ahora es conseguir un estructura de banco de filtros de dos canales con reconstrucción perfecta.

8.1.1. Diseño de los filtros del banco

El lenguaje de alto nivel MATLAB posee la instrucción `[h0,h1,g0,g1] = firpr2chfb(n,fp)` que permite obtener los coeficientes de filtros tipo FIR que forman parte del banco de análisis ($h0$ y $h1$) y de síntesis ($g0$) y ($g1$) de un banco de filtros con rescontrucción perfecta. Para poder obtener dichos coeficientes sólo es necesario llamar a la instrucción mencionada especificando el orden deseado de los filtros y la frecuencia de corte de los filtros pasobajas ($h0$ y $g0$).

Haciendo uso de dicha instrucción y usando un orden de filtro FIR igual a 7 y bandas de paso iguales, los siguientes coeficientes fueron obtenidos:

```
h0 =

    0.5410    0.3064    0.1002   -0.0715   -0.0917    0.0366    0.1455   -0.2593


h1 =

    0.2593    0.1455   -0.0366   -0.0917    0.0715    0.1002   -0.3064    0.5410


g0 =

   -0.5186    0.2910    0.0731   -0.1834   -0.1430    0.2004    0.6128    1.0820
```

```
g1 =

    1.0820   -0.6128    0.2004    0.1430   -0.1834   -0.0731    0.2910    0.5186
```

Estos coeficientes dan lugar a la función de transferencia de cada uno de los filtros del banco de dos canales de reconstrucción perfecta, como se muestra a continuación.

$$
\begin{aligned}
H_0(z) &= 0{,}5410+0{,}3064z^{-1}+0{,}1002z^{-2}-0{,}0715z^{-3}-0{,}0917z^{-4}+0{,}0366z^{-5}+0{,}1455z^{-6}-0{,}2593z^{-7} \\
H_1(z) &= 0{,}2593+0{,}1455z^{-1}-0{,}0366z^{-2}-0{,}0917+z^{-3}+0{,}0715z^{-4}+0{,}1002z^{-5}-0{,}3064z^{-6}+0{,}5410z^{-7} \\
G_0(z) &= -0{,}5186+0{,}2910z^{-1}+0{,}0731z^{-2}-0{,}1834z^{-3}-0{,}1430z^{-4}+0{,}2004z^{-5}+0{,}6128z^{-6}+1{,}0820z^{-7} \\
G_1(z) &= 1{,}0820-0{,}6128z^{-1}+0{,}2004z^{-2}+0{,}1430z^{-3}-0{,}1834z^{-4}-0{,}0731z^{-5}0{,}2910z^{-6}0{,}5186z^{-7}
\end{aligned}
\tag{8.1}
$$

La respuesta en frecuencia de cada de los filtros se muestra en la figura 8.1, donde se aprecia que la banda de cada uno es precisamente la mitad del ancho de banda disponible.

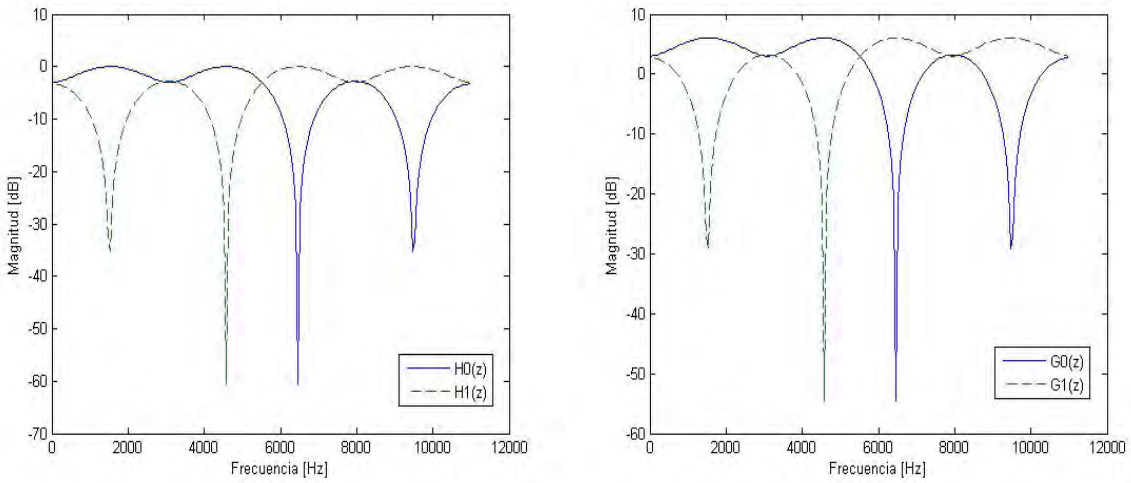


Figura 8.1: Respuesta en frecuencia de los filtros a)$H_0(z)$ y $H_1(z)$, b)$G_0(z)$ y $G_1(z)$.

Si se duda de que los bancos obtenidos conformen un banco de recostrucción perfecta, la instrucción `pr=0.5*conv(g0,h0)+0.5*conv(g1,h1)` puede ser utilizada. Al aplicar dicha instricción a los filtros obtenidos se obtiene lo siguiente:

```
pr =

  Columns 1 through 12

         0   -0.0015    0   -0.0106    0   -0.0047    0    0.9998    0   -0.0047    0   -0.0106

  Columns 13 through 15

         0   -0.0015    0
```

En el octavo elemento del vector resultante se observa un valor muy cercano a uno que no aparece en ningún otro elemento y que permite asegurar una muy buena reconstrucción de la señal original.

8.1.2. Obtención del banco de 4 canales: Estructura de Árbol

Una vez diseñado el banco de filtros de dos canales con resconstrucción perfecta, para crear un banco de cuatro canales con las mismas características, sólo es necesario insertar un banco idéntico entre el submuestreador y el sobremuestreador, es decir, entre el banco de análisis y el de síntesis del banco de dos canales original. La figura 8.2 muestra el banco resultante, en el que se observan dos niveles diferentes.

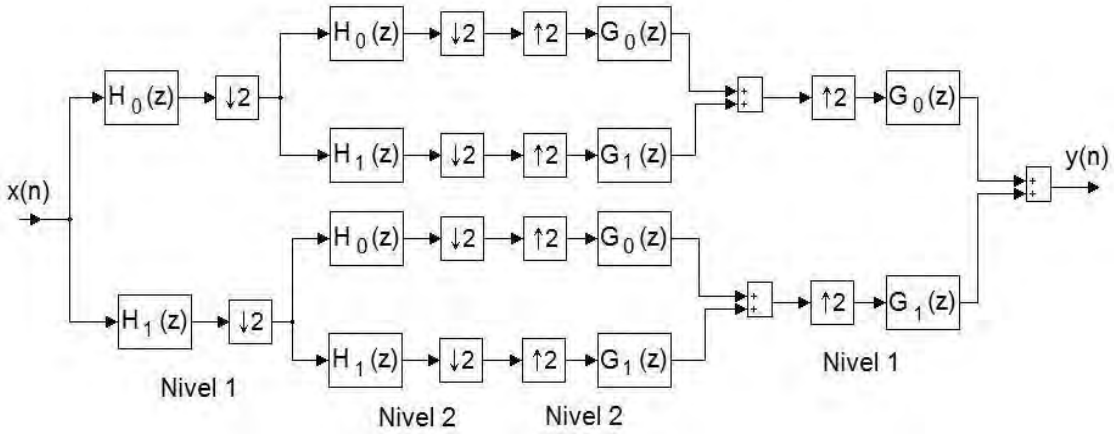


Figura 8.2: Banco de filtros del 4 canales formado a partir de un banco de 2 canales.

8.1.3. Simulación para Banco de 4 canales con estructura de árbol

Para verficar el buen diseño y funcionamiento del banco de filtros obtenido, la simulación resulta ser una herramienta muy importante, para llevar acabo esta tarea se hizo uso del programa de computación *Simulink*, el cual pertenece al familia *Mathworks* y trabaja muy de cerca con MATLAB. A continuación se muestra la estructura del banco de filtros diseñado dentro de Simulink (Figura 8.3).

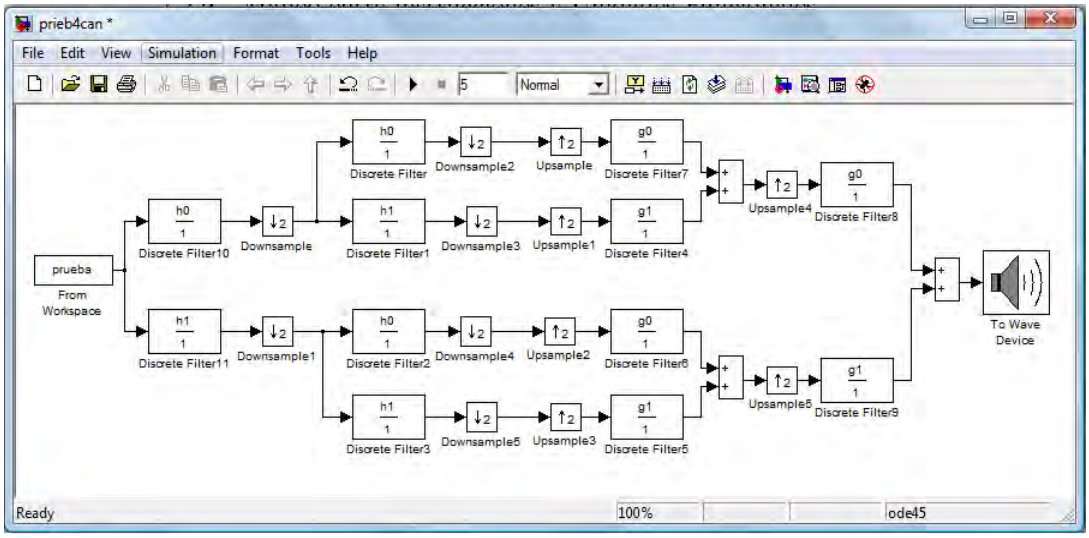


Figura 8.3: Diseño del banco de filtros del 4 canales con estructura de árbol en Simulink.

Como la señal que se pretende procesar es audio, es necesario cargar un archivo de audio en el ambiente de Simulink, lo cual se hace con el siguiente código en Matlab. Este código permite leer un archivo de audio y convertir sus datos en una secuencia de datos que pueda ser entendida por Simulink.

```
NomArch='sonido.wav';
fs=22050;
ts=1/fs;
tsamp=3;
N=tsamp*fs+1;  % numero de muestras
Samp=wavread(NomArch,N);
t=0:ts:(ts*(size(Samp,1)-1));
t=t';
clear prueba
prueba(:,1)=t;
prueba(:,2)=Samp;
```

Una vez cargado el archivo de audio necesario para la simulación, sólo falta iniciar la simulación propiamente. Es posible constatar si la simulación es un éxito de diferentes formas: con el bloque *To Wave Device* es posible escuchar la señal a la salida o la entrada del banco y así, de forma un tanto empírica, verificar el funcionamiento del banco; con el bloque *Time Scope* se puede verificar si las señales de entrada y de salida son parecidas en el tiempo; finalmente con un arreglo de bloques es posible comparar las señales entrada/salida en el dominio de la frecuencia, que finalmente es la mejor forma de observar si el banco recupera la señal de forma adecuada. A continuación se muestran los resultados obtenidos de la simulación del banco de filtros de 4 canales con estructura de árbol utilizando las dos ultimas herramientas descritas.

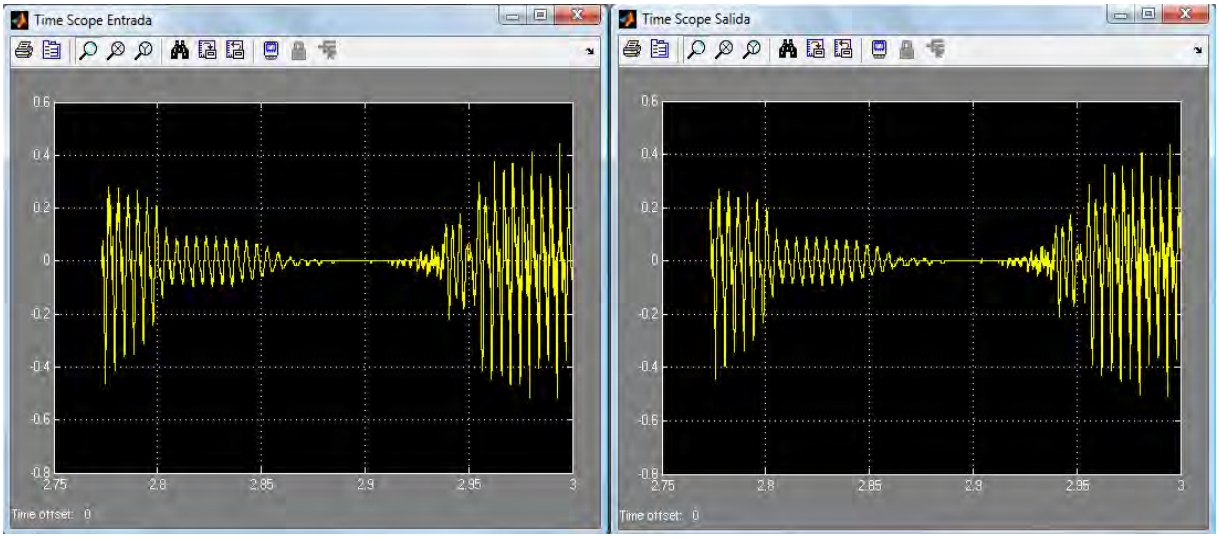


Figura 8.4: Señales de entrada y salida en el banco de filtros del 4 canales con estructura de árbol en Simulink.

En ambas figuras (8.4 y 8.5) se observa que las señales de entrada y de salida, en el tiempo y en la frecuencia, son practicamente iguales, lo que implica que el banco de filtros simulado presenta un reconstrucción perfecta.

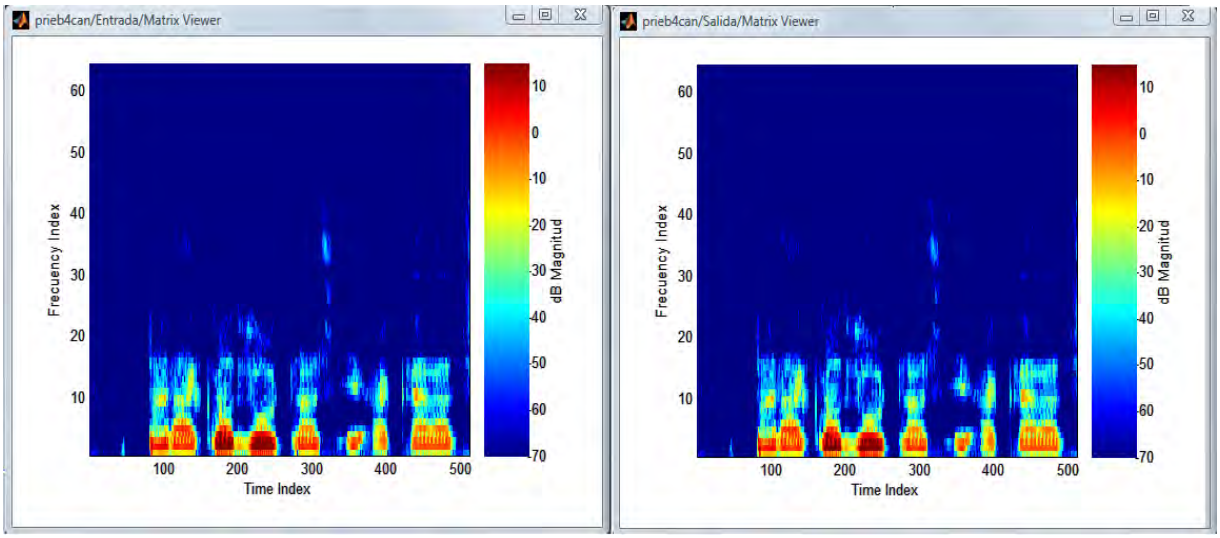


Figura 8.5: Espectros de entrada y salida en el banco de filtros del 4 canales con estructura de árbol en Simulink.

8.1.4. Conversión al banco equivalente de 4 canales

En la sección 6.4 también se menciona que es posible crear una estructura equivalente a la mostrada en la figura 8.2 utilizando las igualdades presentadas en (6.23). Es así como se llega a los filtros del banco equivalente, quedando como sigue:

$$\begin{aligned} H_{00}(z) &= H_0(z)H_0(z^2), \quad H_{01}(z) = H_0(z)H_1(z^2) \\ H_{10}(z) &= H_1(z)H_0(z^2), \quad H_{11}(z) = H_1(z)H_1(z^2) \\ G_{00}(z) &= G_0(z)G_0(z^2), \quad G_{01}(z) = G_0(z)H_1(z^2) \\ G_{10}(z) &= G_1(z)G_0(z^2), \quad G_{11}(z) = G_1(z)H_1(z^2) \end{aligned} \tag{8.2}$$

La estructura resultante se muestra el figura 8.6

Aplicando las ecuaciones en (8.2) y con la ayuda de Matlab, se obtiene la funciones de transferencia de los filtros del banco de L canales. Enseguida se muestran dichas funciones de transferencia.

```
H00 =

  Columns 1 through 11

   0.2927    0.1658    0.2200    0.0552    0.0353    0.0286    0.0220   -0.1582   -0.0214   -0.0988    0.0317

  Columns 12 through 20

  -0.0108    0.0804    0.0572   -0.0021    0.0147   -0.0080   -0.0042    0.0212   -0.0377

H01 =

  Columns 1 through 11

   0.1403    0.0795    0.1047    0.0260   -0.0290   -0.0121   -0.0289   -0.0874    0.0540   -0.0106    0.0645
```

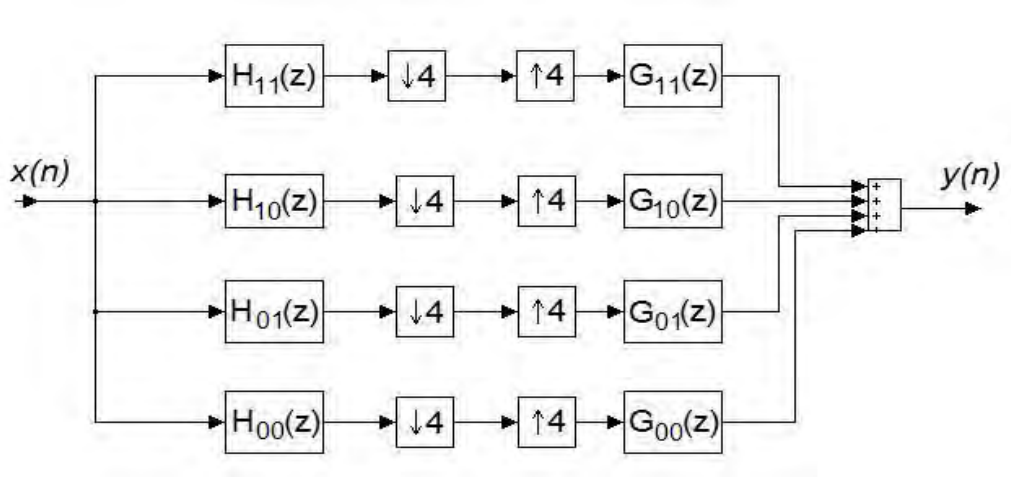


Figura 8.6: Banco de filtros del 4 canales equivalente.

```
  Columns 12 through 20

   0.0317   -0.1756   -0.0747   -0.0295    0.0070    0.0427   -0.0372   -0.0446    0.0795

H10 =

  Columns 1 through 11

   0.1403    0.0787    0.0597   -0.0050    0.0535    0.0407   -0.1661    0.3038   -0.1079    0.1690   -0.0230

  Columns 12 through 20

   0.0608    0.0517   -0.0301    0.0254   -0.0593   -0.0008    0.0344   -0.0446    0.0787

H11 =

  Columns 1 through 11

   0.0672    0.0377    0.0282   -0.0026    0.0037    0.0073   -0.0915    0.1449   -0.0253    0.0939    0.0280

  Columns 12 through 20

  -0.0209   -0.0499   -0.0962   -0.0035    0.0768   -0.0526    0.0235    0.0939   -0.1658

G00 =

  Columns 1 through 11

   0.2689   -0.1509   -0.1888    0.1798    0.0575   -0.1360   -0.2590   -0.5696    0.2286    0.3216   -0.0433

  Columns 12 through 20

   0.1269   -0.3951   -0.0855   -0.0715   -0.2270    0.0352    0.3397    0.3756    0.6631

G01 =

  Columns 1 through 11

  -0.5611    0.3149    0.3969   -0.3768   -0.3035    0.3876    0.6912    1.0528   -0.2987   -0.7025    0.1269
```

```
  Columns 12 through 20

   0.2579   -0.0424    0.2161   -0.0807   -0.2665   -0.0864   -0.0208    0.1783    0.3149

G10 =

  Columns 1 through 11

  -0.5611    0.3178    0.2109   -0.2525    0.2326    0.0347   -0.3881   -0.1674   -0.1203    0.2070    0.2431

  Columns 12 through 20

  -0.0919    0.6761   -0.4315    0.0444   -0.0012   -0.0541    0.0591    0.1783    0.3178

G11 =

  Columns 1 through 11

   1.1708   -0.6631   -0.4462    0.5303   -0.1044   -0.2896    0.6222    0.5470   -0.3849   -0.1996   -0.0838

  Columns 12 through 20

   0.1121    0.3755   -0.1012    0.0184   -0.0481   -0.0747   -0.0592    0.0847    0.1509
```

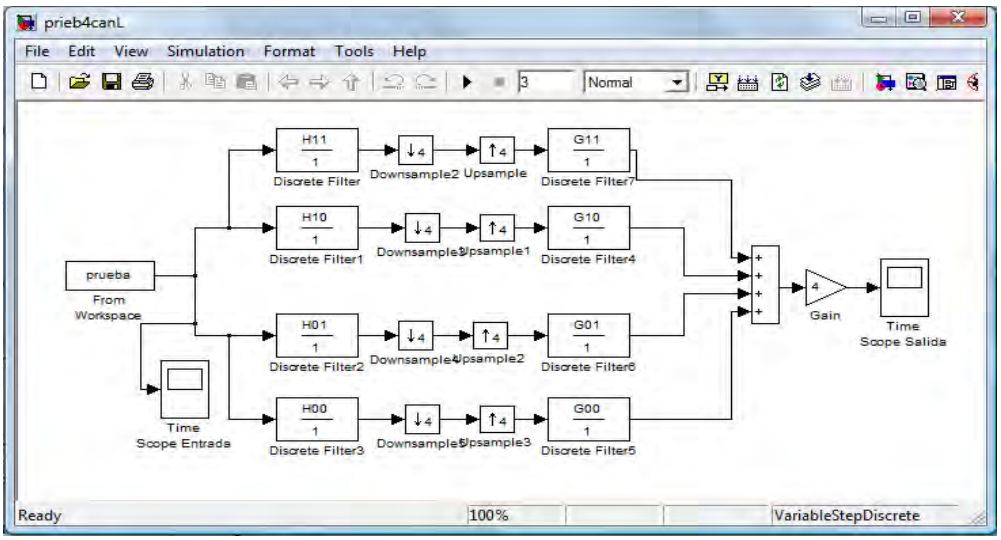


Figura 8.7: Diseño del banco de filtros de 4 canales equivalente en Simulink.

8.1.5. Simulación para el Banco de 4 canales equivalente

Una vez obtenido los filtros del banco de filtros de 4 canales a partir del banco con estructura de árbol, es necesario verificar el buen diseño y funcionamiento del banco de filtros mediante la simulación de dicho banco, como se realizó en el caso anterior. La figura 8.7 muestra el diseño del banco de filtros de 4 canales en el ambiente de *Simulink*.

Al correr la simulación por tres segundos en *Simulink* se obtienen las señales de entrada y salida en el tiempo y su espectrograma correspondiente. Los resultados obtenidos se muestran en las figuras 8.8a y 8.8b.

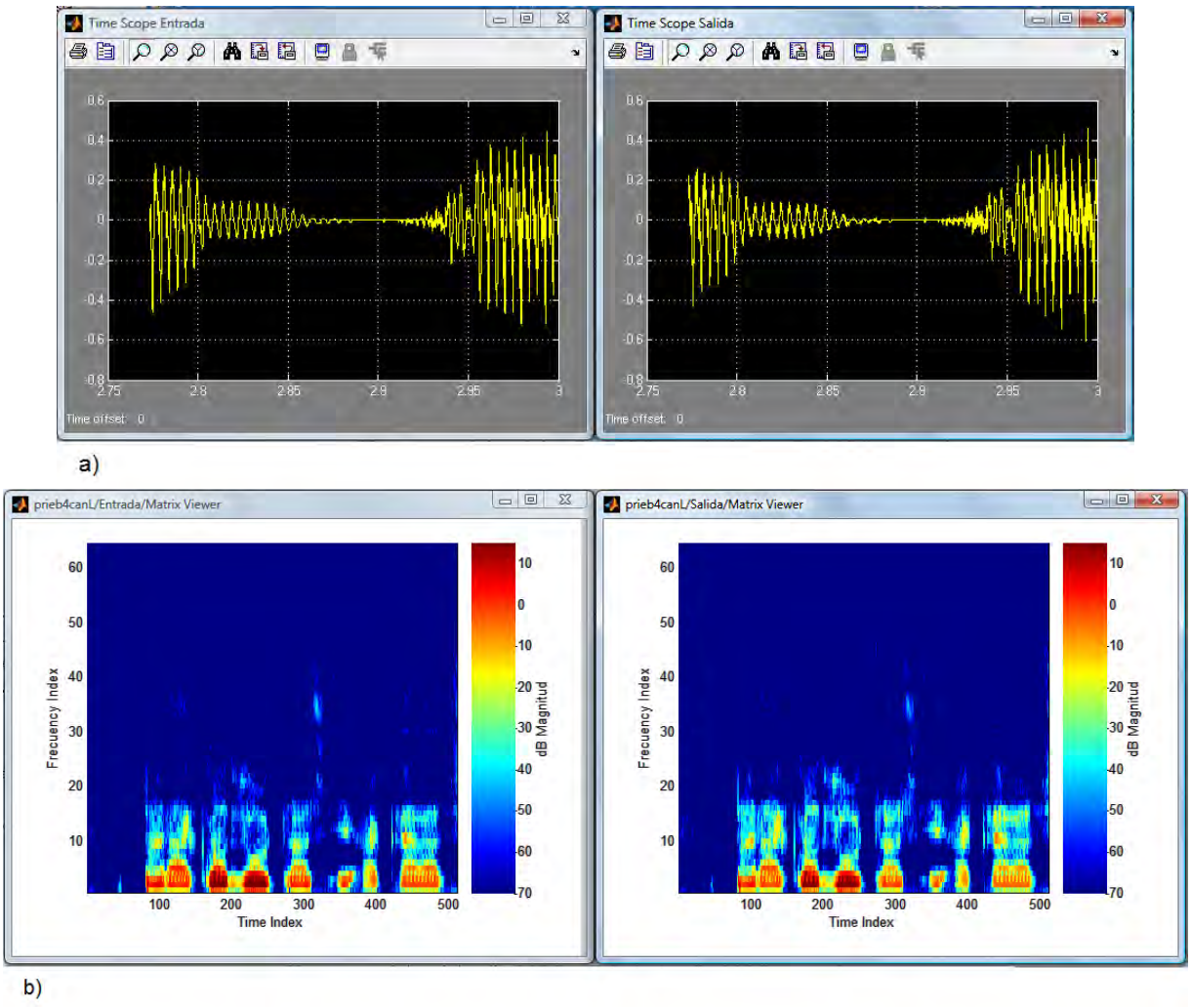


Figura 8.8: Señales de entrada y salida en el tiempo y espectrograma del banco de filtros del 4 canales en Simulink.

En los resultados se puede observar que practicamente no existe diferencia entre las señales de entrada y salida en el tiempo y frecuencia, lo que conlleva a concluir que al igual que el caso anterior este banco es de reconstrucción perfecta.

8.2. Implementación del Banco de Filtros

El siguiente paso después de la simulación es la implementación, la cual es posible realizar con la ayuda de la herramienta *Real Time Workshop*, perteneciente también a la familia de *Mathwoks*.

8.2.1. Implementación del banco de filtros de cuatro canales: Estructura de árbol

Debido a que el microprocesador a usar es el TMS320C6713, es necesario insertar en el diseño de *Simulink*, realizado anteriormente, elementos y configuraciones de acuerdo al microprocesador mencionado.

Primero es necesario insertar el bloque *C6713DSK* que representa las configuraciones para poner en marcha el microprocesador, dichas configuraciones son realizadas por *Simulink* automáticamente al insertar el bloque mencionado. Después, se agrega en la entrada el bloque *ADC*, que tiene como función convertir la señal analógica del micrófono a una señal digital, para que pueda ser procesada por el micropocesador. Es importante mencionar que es necesario especificar, en este último bloque, la tasa de muestreo a la cual se va a trabajar.

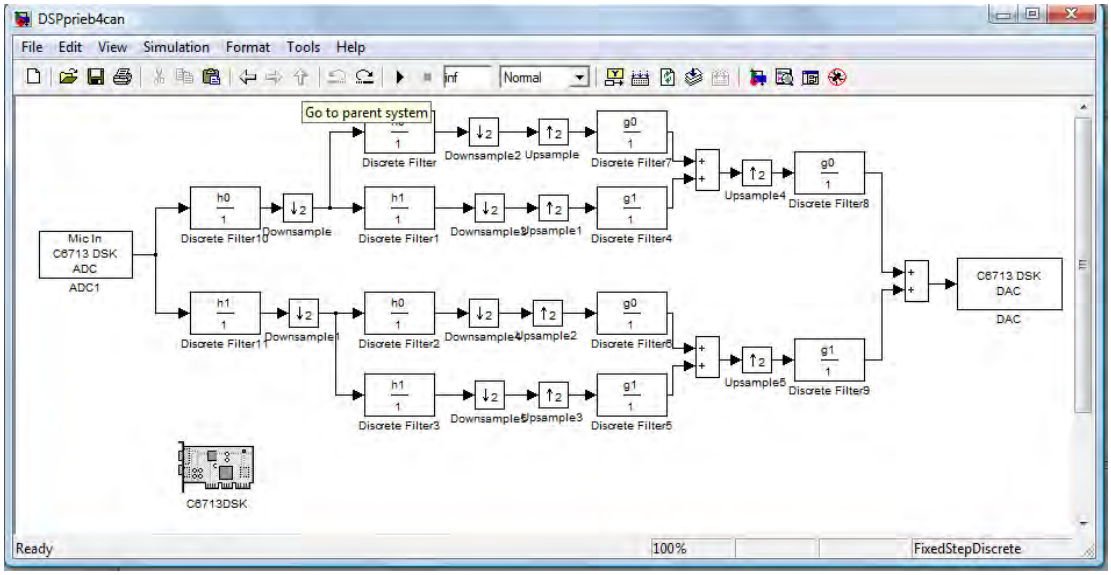


Figura 8.9: Implementación del banco de filtros de 4 canales estructura de árbol en Simulink.

Del mismo modo se requiere insertar un bloque *DAC* en la salida, que tiene la tarea de convertir la señal digital ya procesada en una señal analógica disponible para ser escuchada.

El diseño del banco de filtros de cuatro canales con estructura de árbol antes de ser implementado y con los elementos antes descritos se muestra a continuación en la figura 8.9

En el menú *tools* de *Simulink* es posible seleccionar la opción *Real Time Workshop/Build model*. Por medio de esta opción es que el diseño del banco de filtros inicia su implementación. Primero se crean códigos en el lenguaje de programación C del banco de filtros y posteriormente se establece un vínculo entre *Matlab* y el programa *Code Composer* el cual pone en funcionamiento la tarjeta del microprocesador y genera los códigos en ensamblador a partir de los códigos en C generados por *Simulink*.

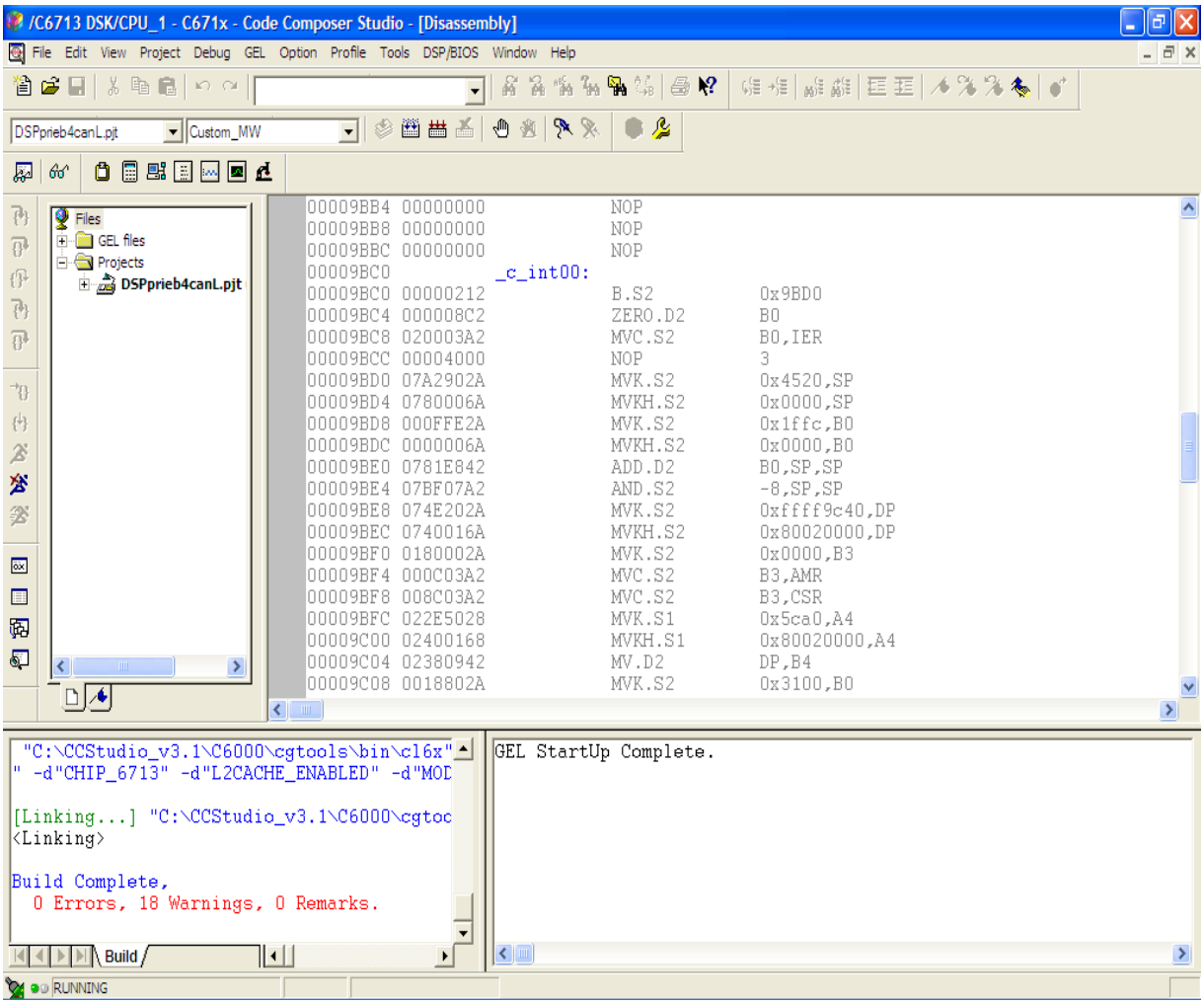


Figura 8.10: Ejecución del banco de filtros en Code Composer.

Se puede observar en la figura 8.10 la pantalla del programa cuando la tarjeta está en uso y el banco de filtros se está ejecutando, sí en ese momento, en la entrada de la tarjeta se conecta un micrófono y en la salida una bocina o audifonos se puede corroborar que el banco de filtros está funcionando correctamente.

Los códigos generados por *Real Time Workshop* se presentan en el apéndice A y con ayuda de los comentarios ahí escritos y algunos conocimientos del lenguaje C, es posible entenderlos de forma correcta.

8.2.2. Implementación del banco de filtros equivalente de 4 canales

Como en el caso anterior, en el diseño del banco de filtros equivalente de 4 canales se insertan los mismos elementos para poder implementar dicho banco. Esta nueva configuración se puede observar en la figura 8.11.

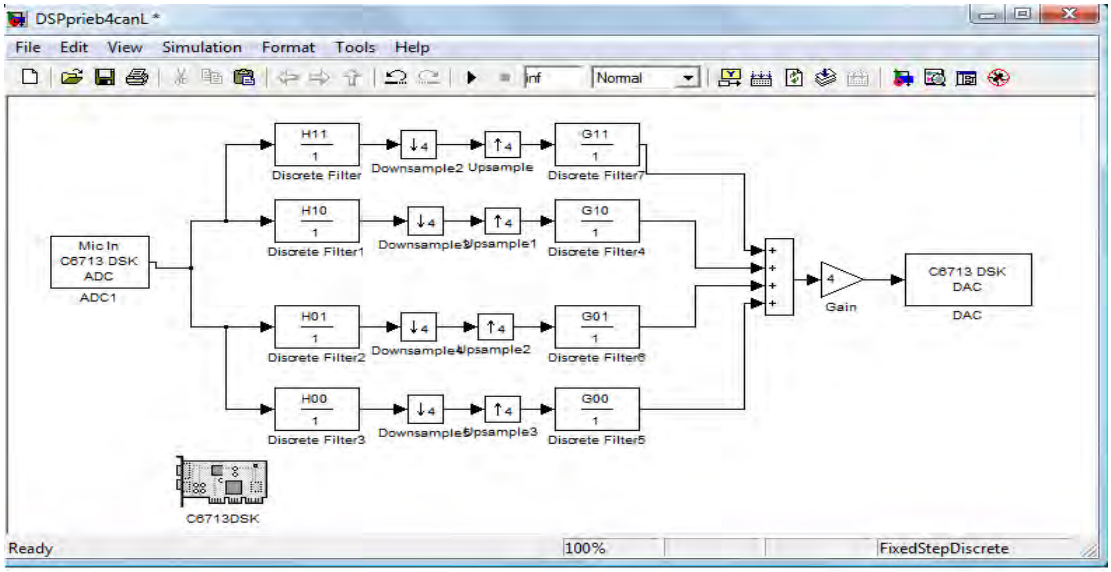


Figura 8.11: Impletación del banco de filtros equivalente de 4 canales en Simulink.

Capítulo 9

Conclusiones

Una vez diseñados e implementados los bancos de filtros de 4 canales con resconstrucción perfecta para codificador de audio de los cuales trata esta tesis, es momento de reportar los puntos relevantes de este trabajo y las conclusiones a las que se ha llegado, asi como verificar si todo el trabajo realizado tuvo los resultados esparados.

Tal como la literatura acerca de los bancos de filtros y el procesamiento multitasa menciona, es posible diseñar y obtener bancos de filtros de dos o cuatro canales que recuperen la señal procesada de forma adecuada, es decir, que presenten un reconstrucción perfecta independientemente del orden de los filtros que para ese objetivo se utilicen. Sin embargo, el hecho de usar filtros con un orden alto mejora la eficiencia del propio filtro, cuestión relevante cuando se trata de filtros FIR. Lo anterior fue corrobarado durante la simulación e implementación de los bancos de filtros diseñados.

A lo largo de este trabajo se presentaron diferentes tipos de bancos de filtros, los que se consideraron relevantes. Si bien existen estructuras de bancos que son muy eficientes computacionalmente, como la estructura polifase, los bancos implementados se realizaron debido a su facilidad para comprenderlos y que sin ser completamente eficientes funcionan de forma correcta en implementaciones en tiempo real y para el tipo de señal digital implementada, como se muestra en el capítulo de implementación de este trabajo.

El papel jugado por los lenguajes de alto nivel en esta tesis, y en el avance y desarrollo de una investigación en general, es de suma relevancia. Estos programas de computadora permitieron agilizar el trabajo ya que apoyaron durante todas las etapas de la investigación: Diseño, Simulación e Implementación, además de resultar ser muy amigables, es decir, fáciles de operar. La obtención de las funciones de transferencias de los filtros de cada banco de filtros no habría sido tan rápida sin la ayuda de *Matlab*, el cual posee herramientas e instrucciones que permiten el diseño eficaz de filtros FIR y IIR. La simulación no habría sido posible sin *Simulink* y la implementacion habría tomado largo tiempo sin la herramienta *Real Time Workshop* que permite crear códigos en lenguaje C a partir del modelo creado en simlunik, que a su vez son utilizados por Code Composer Studio para generar archivos en lenguaje ensamblador que permiten poner en funcionamiento al Microprocesador utilizado (TMS320C6713).

Respecto a los resultados obtenidos, en primera instancia, se puede considerar los dos bancos de filtros implementados, uno de 4 canales con estructura multinivel o de árbol y otro que corresponde a una estructura equivalente del primer caso. Por otra parte, la metodología con la que fue realizada la implementación es tan flexible que es posible modificar los filtros del banco rápidamente y realizar un implementacion diferente en cuestión de minutos. Lo anterior permite que la implementación

y metodología antes mencionadas puedan ser utilizadas en el laboratorio correspondiente y así, de forma muy dinámica, permitirle a un alumno interactuar con un microprocesador, así como verificar y poner en práctica sus conocimientos sobre, filtros, procesamiento multitasa, banco de filtros, interpolación y/o decimación. Evidentemente, es necesario a partir del material que el presente trabajo ofrece, diseñar una serie de actividades determinadas para el correcto uso del material y un aprovechamiento eficiente de parte del alumno.

Finalmente, con todo lo mencionado anteriormente, es posible considerar que los objetivos planteados para el presente trabajo se cubrieron de forma satisfactoria, cuestión que se puede constatar con las implementaciones ya mencionadas y mediante lo reportado mediante este documento.

Índice de figuras

Apéndice A

Diseño de los Bancos de Filtros

A.1. Código en Matlab utilizado para diseñar los filtros de los bancos de filtros implementados

```
%Diseno de filtros para el Banco de Filtros Multinivel

[h0 h1 g0 g1]= firpr2chfb(7,0.49999)
fm=22050;

[hh0,w]=freqz(h0,1,200);
[hh1,w]=freqz(h1,1,200);
[hg0,w]=freqz(g0,1,200);
[hg1,w]=freqz(g1,1,200);

f=fm*w/(2*pi);

figure
plot(f,20*log10(abs(hh0)),'-',f,20*log10(abs(hh1)),'--')
xlabel('frecuencia [Hz]')
ylabel('Magnitud [dB]')

figure
plot(f,20*log10(abs(hg0)),'-',f,20*log10(abs(hg1)),'--')
xlabel('frecuencia [Hz]')
ylabel('Magnitud [dB]')

%Verificacion de la reconstruccion perfecta
n=0:11;
pr=0.5*conv(g0,h0)+0.5*conv(g1,h1);


%Conversion a Banco de Filtros de 4 bandas

H02=[h0(1) 0 h0(2) 0 h0(3) 0 h0(4) 0 h0(5) 0 h0(6) 0 h0(7)];
```

```
H12=[h1(1) 0 h1(2) 0 h1(3) 0 h1(4) 0 h1(5) 0 h1(6) 0 h1(7)];
G02=[g0(1) 0 g0(2) 0 g0(3) 0 g0(4) 0 g0(5) 0 g0(6) 0 g0(7)];
G12=[g1(1) 0 g1(2) 0 g1(3) 0 g1(4) 0 g1(5) 0 g1(6) 0 g1(7)];

H00=conv(h0,H02);
H01=conv(h0,H12);
H10=conv(h1,H02);
H11=conv(h1,H12);
G00=conv(g0,G02);
G01=conv(g0,G12);
G10=conv(g1,G02);
G11=conv(g1,G12);
```

Apéndice B

Implementación de los Banco de Filtros

B.1. Códigos en C generados por Real-Time Workshop

B.1.1. Datos

```
/*
 * File: DSPprieb4can_data.c
 *
 * Real-Time Workshop code generated for Simulink model DSPprieb4can.
 *
 * Model version                        : 1.5
 * Real-Time Workshop file version      : 6.5  (R2006b)  03-Aug-2006
 * Real-Time Workshop file generated on : Wed Apr 15 08:40:44 2009
 * TLC version                          : 6.5 (Aug  3 2006)
 * C source code generated on           : Wed Apr 15 08:40:51 2009
 */
#include "DSPprieb4can.h"
#include "DSPprieb4can_private.h"

/* Block parameters (auto storage) */

#pragma DATA_ALIGN(DSPprieb4can_P,8)

Parameters_DSPprieb4can DSPprieb4can_P = {
  /*  DiscreteFilter10_A : '<Root>/Discrete Filter10'
   */
  { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

  /*  DiscreteFilter10_C : '<Root>/Discrete Filter10'
   */
  { 3.0641673148813753E-001, 1.0021204742470907E-001, -7.1520341073669461E-002,
    -9.1702675155071856E-002, 3.6565640701629203E-002, 1.4550456329120948E-001,
    -2.5929204477517348E-001 },
```

```
5.4101948363811370E-001,            /* DiscreteFilter10_D : '<Root>/Discrete Filter10'
                                     */
0.0,                                /* Downsample_IC : '<Root>/Downsample'
                                     */

/*  DiscreteFilter_A : '<Root>/Discrete Filter'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter_C : '<Root>/Discrete Filter'
 */
{ 3.0641673148813753E-001, 1.0021204742470907E-001, -7.1520341073669461E-002,
  -9.1702675155071856E-002, 3.6565640701629203E-002, 1.4550456329120948E-001,
  -2.5929204477517348E-001 },
5.4101948363811370E-001,            /* DiscreteFilter_D : '<Root>/Discrete Filter'
                                     */
0.0,                                /* Downsample2_IC : '<Root>/Downsample2'
                                     */
0.0,                                /* Upsample_IC : '<Root>/Upsample'
                                     */

/*  DiscreteFilter7_A : '<Root>/Discrete Filter7'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter7_C : '<Root>/Discrete Filter7'
 */
{ 2.9100912658241895E-001, 7.3131281403258405E-002, -1.8340535031014371E-001,
  -1.4304068214733892E-001, 2.0042409484941814E-001, 6.1283346297627506E-001,
  1.0820389672762274E+000 },
-5.1858408955034696E-001,           /* DiscreteFilter7_D : '<Root>/Discrete Filter7'
                                     */

/*  DiscreteFilter1_A : '<Root>/Discrete Filter1'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter1_C : '<Root>/Discrete Filter1'
 */
{ 1.4550456329120948E-001, -3.6565640701629203E-002, -9.1702675155071856E-002,
  7.1520341073669461E-002, 1.0021204742470907E-001, -3.0641673148813753E-001,
  5.4101948363811370E-001 },
2.5929204477517348E-001,            /* DiscreteFilter1_D : '<Root>/Discrete Filter1'
                                     */
0.0,                                /* Downsample3_IC : '<Root>/Downsample3'
                                     */
0.0,                                /* Upsample1_IC : '<Root>/Upsample1'
                                     */
```

```
/*  DiscreteFilter4_A : '<Root>/Discrete Filter4'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter4_C : '<Root>/Discrete Filter4'
 */
{ -6.1283346297627506E-001, 2.0042409484941814E-001, 1.4304068214733892E-001,
  -1.8340535031014371E-001, -7.3131281403258405E-002, 2.9100912658241895E-001,
  5.1858408955034696E-001 },
1.0820389672762274E+000,            /* DiscreteFilter4_D : '<Root>/Discrete Filter4'
                                     */
0.0,                                /* Upsample4_IC : '<Root>/Upsample4'
                                     */

/*  DiscreteFilter8_A : '<Root>/Discrete Filter8'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter8_C : '<Root>/Discrete Filter8'
 */
{ 2.9100912658241895E-001, 7.3131281403258405E-002, -1.8340535031014371E-001,
  -1.4304068214733892E-001, 2.0042409484941814E-001, 6.1283346297627506E-001,
  1.0820389672762274E+000 },
-5.1858408955034696E-001,           /* DiscreteFilter8_D : '<Root>/Discrete Filter8'
                                     */

/*  DiscreteFilter11_A : '<Root>/Discrete Filter11'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter11_C : '<Root>/Discrete Filter11'
 */
{ 1.4550456329120948E-001, -3.6565640701629203E-002, -9.1702675155071856E-002,
  7.1520341073669461E-002, 1.0021204742470907E-001, -3.0641673148813753E-001,
  5.4101948363811370E-001 },
2.5929204477517348E-001,            /* DiscreteFilter11_D : '<Root>/Discrete Filter11'
                                     */
0.0,                                /* Downsample1_IC : '<Root>/Downsample1'
                                     */

/*  DiscreteFilter2_A : '<Root>/Discrete Filter2'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter2_C : '<Root>/Discrete Filter2'
 */
{ 3.0641673148813753E-001, 1.0021204742470907E-001, -7.1520341073669461E-002,
```

```
  -9.1702675155071856E-002, 3.6565640701629203E-002, 1.4550456329120948E-001,
  -2.5929204477517348E-001 },
5.4101948363811370E-001,            /* DiscreteFilter2_D : '<Root>/Discrete Filter2'
                                     */
0.0,                                /* Downsample4_IC : '<Root>/Downsample4'
                                     */
0.0,                                /* Upsample2_IC : '<Root>/Upsample2'
                                     */

/*  DiscreteFilter6_A : '<Root>/Discrete Filter6'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter6_C : '<Root>/Discrete Filter6'
 */
{ 2.9100912658241895E-001, 7.3131281403258405E-002, -1.8340535031014371E-001,
  -1.4304068214733892E-001, 2.0042409484941814E-001, 6.1283346297627506E-001,
  1.0820389672762274E+000 },
-5.1858408955034696E-001,           /* DiscreteFilter6_D : '<Root>/Discrete Filter6'
                                     */

/*  DiscreteFilter3_A : '<Root>/Discrete Filter3'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter3_C : '<Root>/Discrete Filter3'
 */
{ 1.4550456329120948E-001, -3.6565640701629203E-002, -9.1702675155071856E-002,
  7.1520341073669461E-002, 1.0021204742470907E-001, -3.0641673148813753E-001,
  5.4101948363811370E-001 },
2.5929204477517348E-001,            /* DiscreteFilter3_D : '<Root>/Discrete Filter3'
                                     */
0.0,                                /* Downsample5_IC : '<Root>/Downsample5'
                                     */
0.0,                                /* Upsample3_IC : '<Root>/Upsample3'
                                     */

/*  DiscreteFilter5_A : '<Root>/Discrete Filter5'
 */
{ 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

/*  DiscreteFilter5_C : '<Root>/Discrete Filter5'
 */
{ -6.1283346297627506E-001, 2.0042409484941814E-001, 1.4304068214733892E-001,
  -1.8340535031014371E-001, -7.3131281403258405E-002, 2.9100912658241895E-001,
  5.1858408955034696E-001 },
1.0820389672762274E+000,            /* DiscreteFilter5_D : '<Root>/Discrete Filter5'
                                     */
```

```
  0.0,                                 /* Upsample5_IC : '<Root>/Upsample5'
                                        */

  /*  DiscreteFilter9_A : '<Root>/Discrete Filter9'
   */
  { 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 },

  /*  DiscreteFilter9_C : '<Root>/Discrete Filter9'
   */
  { -6.1283346297627506E-001, 2.0042409484941814E-001, 1.4304068214733892E-001,
    -1.8340535031014371E-001, -7.3131281403258405E-002, 2.9100912658241895E-001,
    5.1858408955034696E-001 },
  1.0820389672762274E+000              /* DiscreteFilter9_D : '<Root>/Discrete Filter9'
                                        */
};

/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

B.1.2. La función main()

```
/*
 * File: DSPprieb4can_main.c
 *
 * Real-Time Workshop code generated for Simulink model DSPprieb4can.
 *
 * Model version                        : 1.5
 * Real-Time Workshop file version      : 6.5  (R2006b)  03-Aug-2006
 * Real-Time Workshop file generated on : Wed Apr 15 08:40:44 2009
 * TLC version                          : 6.5 (Aug  3 2006)
 * C source code generated on           : Wed Apr 15 08:40:51 2009
 */
#include "DSPprieb4cancfg.h"
#include "rtwtypes.h"
#include "MW_c6xxx_csl.h"
#include "c6000_main.h"
#include "DSPprieb4can.h"
#include "DSPprieb4can_private.h"
#define DSK_CPLD_BASE                  0x90080000
#define DSK_USER_REG                   0
#define NUM_SUBRATE_TASKS              (2)

SEM_Handle taskSEMList[NUM_SUBRATE_TASKS];
```

```
extern void TSK_prolog(TSK_Handle hTask);
extern void TSK_epilog(TSK_Handle hTask);

/* Function: exitprocessing ---------------------------------
 *
 * Abstract:
 *      Perform various tasks at program exit.
 */
void exitprocessing()
{
  disable_DMA();
  disable_interrupts();
  UTL_halt();
}

void tSubRate_1_TSK_fcn(void)
{
  TSK_prolog( TSK_self() );
  while (1) {
    SEM_pend(taskSEMList[0], SYS_FOREVER);
    DSPprieb4can_step(1);
  }
}

void tSubRate_2_TSK_fcn(void)
{
  TSK_prolog( TSK_self() );
  while (1) {
    SEM_pend(taskSEMList[1], SYS_FOREVER);
    DSPprieb4can_step(2);
  }
}

void tBaseRateTSK_fcn(void)
{
  int_T i;
  volatile int_T isErrStatus;
  TSK_prolog( TSK_self() );

  /* Initialize semaphore list */
  taskSEMList[0] = &rtTaskSemaphore_1;
  taskSEMList[1] = &rtTaskSemaphore_2;
  isErrStatus = rtmGetErrorStatus(DSPprieb4can_M) != NULL;
  while (!isErrStatus ) {
    /* Wait for the next timer interrupt */
    if (SEM_pend(&rtClockSEM, 0) != FALSE ) {
      /* turn on board LED(s) */
      *(volatile unsigned char *)(DSK_CPLD_BASE + DSK_USER_REG) |= 0x0F;
```

```
    } else {
      SEM_pend(&rtClockSEM, SYS_FOREVER);
    }

    for (i = 1; i <= 2.0; i++) {
      if (rtmStepTask(DSPprieb4can_M, i)) {
        SEM_post(taskSEMList[i - 1]);
        if (SEM_pend(taskSEMList[i - 1], 0) != FALSE ) {
          SEM_post(taskSEMList[i - 1]);

          /* turn on board LED(s) */
          *(volatile unsigned char *)(DSK_CPLD_BASE + DSK_USER_REG) |= 0x0F;
        }
      }
    }

    DSPprieb4can_step(0);
    isErrStatus = rtmGetErrorStatus(DSPprieb4can_M) != NULL;
  }                                    /* while */

  SEM_post(&stopSEM);
}

void main(void)
{

#ifdef L2CACHE_ENABLED

  turnOn_L2Cache();

#endif

  LOG_printf(&LOG_MW1, "**starting the model**");

  /* Drop out of main() and enter DSP/BIOS Kernel */
}

//
// TSK prolog/epilog functions.
//
void TSK_prolog(TSK_Handle hTask)
{

#ifdef ENET_SOCKET_CALLS

  fdOpenSession( hTask );

#endif
```

```
}

void TSK_epilog(TSK_Handle hTask)
{

#ifdef ENET_SOCKET_CALLS

  fdCloseSession( hTask );

#endif

}

//
// This task is run at the highest priority. It is used to
// initialize the model and also to monitor stopping conditions.
// OS executes this task immidiatey after falling out of main().
//
void initTerminateTSK_fcn(void)
{
  /* Perform model initialization here. */
  DSPprieb4can_initialize(1);

  /* Start real-time clock and enable interrupts. */
  enable_interrupts();

  /* this timer is used for clocking base rate task */
  configureTimers();

  /* Wait for a stopping condition. */
  SEM_pend(&stopSEM, SYS_FOREVER);

  /* We have acquired the STOP semaphore. Perform model termination. */
  /* Suspend syncronous tasks */
  {
    TSK_epilog( &tBaseRateTSK );
    TSK_setpri( &tBaseRateTSK, -1 );
    TSK_epilog( &tSubRate_1_TSK );
    TSK_setpri( &tSubRate_1_TSK, -1);
    TSK_epilog( &tSubRate_2_TSK );
    TSK_setpri( &tSubRate_2_TSK, -1);
  }

  /* Disable rt_OneStep() here */

  /* Terminate model */
  DSPprieb4can_terminate();
```

```
  targetTerminate();

  /* Print out an exit message */
  LOG_printf(&LOG_MW1, "Stopping the model.");
  exitprocessing();
}

/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

B.1.3. Proceso completo del banco de filtros

```
/*
 * File: DSPprieb4can.c
 *
 * Real-Time Workshop code generated for Simulink model DSPprieb4can.
 *
 * Model version                        : 1.5
 * Real-Time Workshop file version      : 6.5  (R2006b)  03-Aug-2006
 * Real-Time Workshop file generated on : Wed Apr 15 08:40:44 2009
 * TLC version                          : 6.5 (Aug  3 2006)
 * C source code generated on           : Wed Apr 15 08:40:51 2009
 */
#include "DSPprieb4can.h"
#include "DSPprieb4can_private.h"

/* Block signals (auto storage) */

#pragma DATA_ALIGN(DSPprieb4can_B,8)

BlockIO_DSPprieb4can DSPprieb4can_B;

/* Block states (auto storage) */
#pragma DATA_ALIGN(DSPprieb4can_DWork,8)

D_Work_DSPprieb4can DSPprieb4can_DWork;

/* Real-time model */
RT_MODEL_DSPprieb4can DSPprieb4can_M_;
RT_MODEL_DSPprieb4can *DSPprieb4can_M = &DSPprieb4can_M_;
static void rate_monotonic_scheduler(void);

/* Set which subrates need to run this base step (base rate always runs).
 * This function must be called prior to calling the model step function
 * in order to "remember" which rates need to run this base step.  The
```

```
 * buffering of events allows for overlapping preemption.
 */
void DSPprieb4can_SetEventsForThisBaseStep(boolean_T *eventFlags)
{
  /* Task runs when its counter is zero, computed via rtmStepTask macro */
  eventFlags[1] = rtmStepTask(DSPprieb4can_M, 1);
  eventFlags[2] = rtmStepTask(DSPprieb4can_M, 2);
}

/* This function updates active task flag for each subrate
 * and rate transition flags for tasks that exchagne data.
 * The function assumes rate-monotonic multitasking scheduler.
 * The function must be called at model base rate so that
 * the generated code self-manages all its subrates and rate
 * transition flags.
 */
static void rate_monotonic_scheduler(void)
{
  /* To ensure a safe and deterministic data transfer between two rates,
   * data is transfered at the priority of a fast task and the frequency
   * of the slow task.  The following flags indicate when the data transfer
   * happens.  That is, a rate interaction flag is set true when both rates
   * will run, and false otherwise.
   */

  /* tid 0 shares data with slower tid rate: 1 */
  DSPprieb4can_M->Timing.RateInteraction.TID0_1 =
    (DSPprieb4can_M->Timing.TaskCounters.TID[1] == 0);

  /* tid 1 shares data with slower tid rate: 2 */
  if (DSPprieb4can_M->Timing.TaskCounters.TID[1] == 0) {
    DSPprieb4can_M->Timing.RateInteraction.TID1_2 =
      (DSPprieb4can_M->Timing.TaskCounters.TID[2] == 0);
  }

  /* Compute which subrates run during the next base time step.  Subrates
   * are an integer multiple of the base rate counter.  Therefore, the subtask
   * counter is reset when it reaches its limit (zero means run).
   */
  if (++DSPprieb4can_M->Timing.TaskCounters.TID[1] == 2) {/* Sample time: [0.00025s, 0.0s] */
    DSPprieb4can_M->Timing.TaskCounters.TID[1] = 0;
  }

  if (++DSPprieb4can_M->Timing.TaskCounters.TID[2] == 4) {/* Sample time: [0.0005s, 0.0s] */
    DSPprieb4can_M->Timing.TaskCounters.TID[2] = 0;
  }
}
```

```
/* Model step function for TID0 */
void DSPprieb4can_step0(void)          /* Sample time: [0.000125s, 0.0s] */
{
  /* local block i/o variables */
  real_T rtb_ADC1;
  real_T rtb_Upsample4;
  real_T rtb_Upsample5;
  real_T rtb_DiscreteFilter8;
  real_T rtb_DiscreteFilter9;

  {                                    /* Sample time: [0.000125s, 0.0s] */
    rate_monotonic_scheduler();
  }

  /* S-Function Block: <Root>/ADC1 (c6416dsk_adc) */
  {
    const real_T ADCScaleFactor = 1.0 / 2147483648.0;
    int32_T *blkAdcBuffPtr;

    // Retrieve ADC buffer pointer and invalidate CACHE
    blkAdcBuffPtr = (int32_T *) getAdcBuff();
    CACHE_wbInvL2( (void *) blkAdcBuffPtr, 8, CACHE_WAIT );
    rtb_ADC1 = (real_T)*blkAdcBuffPtr * ADCScaleFactor;

    /* Skip Right side for mono mode */
    blkAdcBuffPtr += 2;
  }

  /* DiscreteFilter: '<Root>/Discrete Filter10' */
  DSPprieb4can_B.DiscreteFilter10 = DSPprieb4can_P.DiscreteFilter10_D*rtb_ADC1;

  {
    int_T nx = 7;
    const real_T *x = &DSPprieb4can_DWork.DiscreteFilter10_DSTATE[0];
    const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter10_C[0];
    while (nx--) {
      DSPprieb4can_B.DiscreteFilter10 += (*Cmtx) * (*x++);
      Cmtx += 1;
    }
  }

  /* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample' */
  {
    if (DSPprieb4can_M->Timing.RateInteraction.TID0_1) {
      DSPprieb4can_B.Downsample = DSPprieb4can_B.DiscreteFilter10;
    }
  }
```

```
/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample4' */
{
  real_T *buf;

  /* special sample hit - toggle read buffer */
  if (DSPprieb4can_M->Timing.RateInteraction.TID0_1) {
    DSPprieb4can_DWork.Upsample4_ReadBufIdx =
      !(DSPprieb4can_DWork.Upsample4_ReadBufIdx);
  }

  buf = (DSPprieb4can_DWork.Upsample4_ReadBufIdx)
    ? &DSPprieb4can_DWork.Upsample4_Buffer[0] + 1
    : &DSPprieb4can_DWork.Upsample4_Buffer[0];

  {
    real_T *y = (real_T *)&rtb_Upsample4;
    int_T c = DSPprieb4can_DWork.Upsample4_Count;
    if (c++ == 0) {
      *y++ = *buf;
    } else {
      *y++ = 0.0;
    }

    if (c == 2)
      c = 0;

    /* Update counters for next sample hit */
    DSPprieb4can_DWork.Upsample4_Count = c;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter8' */
rtb_DiscreteFilter8 = (DSPprieb4can_P.DiscreteFilter8_D)*rtb_Upsample4;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter8_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter8_C[0];
  while (nx--) {
    rtb_DiscreteFilter8 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter11' */
DSPprieb4can_B.DiscreteFilter11 = DSPprieb4can_P.DiscreteFilter11_D*rtb_ADC1;

{
```

```
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter11_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter11_C[0];
  while (nx--) {
    DSPprieb4can_B.DiscreteFilter11 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample1' */
{
  if (DSPprieb4can_M->Timing.RateInteraction.TID0_1) {
    DSPprieb4can_B.Downsample1 = DSPprieb4can_B.DiscreteFilter11;
  }
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample5' */
{
  real_T *buf;

  /* special sample hit - toggle read buffer */
  if (DSPprieb4can_M->Timing.RateInteraction.TID0_1) {
    DSPprieb4can_DWork.Upsample5_ReadBufIdx =
      !(DSPprieb4can_DWork.Upsample5_ReadBufIdx);
  }

  buf = (DSPprieb4can_DWork.Upsample5_ReadBufIdx)
    ? &DSPprieb4can_DWork.Upsample5_Buffer[0] + 1
    : &DSPprieb4can_DWork.Upsample5_Buffer[0];

  {
    real_T *y = (real_T *)&rtb_Upsample5;
    int_T c = DSPprieb4can_DWork.Upsample5_Count;
    if (c++ == 0) {
      *y++ = *buf;
    } else {
      *y++ = 0.0;
    }

    if (c == 2)
      c = 0;

    /* Update counters for next sample hit */
    DSPprieb4can_DWork.Upsample5_Count = c;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter9' */
```

```
rtb_DiscreteFilter9 = DSPprieb4can_P.DiscreteFilter9_D*rtb_Upsample5;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter9_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter9_C[0];
  while (nx--) {
    rtb_DiscreteFilter9 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Sum: '<Root>/Sum2' */
DSPprieb4can_B.Sum2 = rtb_DiscreteFilter8 + rtb_DiscreteFilter9;

/* S-Function Block: <Root>/DAC (c6416dsk_dac) */
{
  const real_T DACScaleFactor = 2147483648.0;
  void *blkDacBuffPtr;
  int32_T *outPtr;
  blkDacBuffPtr = getDacBuff();
  outPtr = (int32_T *) blkDacBuffPtr;

  /* Left */
  *outPtr = (int32_T)(DSPprieb4can_B.Sum2 * DACScaleFactor);

  /* Right */
  *(outPtr+1) = *outPtr;             /* Copy same word to RHS for mono mode. */
  outPtr += 2;
  CACHE_wbL2( (void *) blkDacBuffPtr, 8, CACHE_WAIT );
}

/* DiscreteFilter Block: '<Root>/Discrete Filter10' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter10_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter10_DSTATE[0];
  real_T xtmp = rtb_ADC1;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter8' */
{
```

```
    int_T i;
    const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter8_A[0];
    real_T *x = &DSPprieb4can_DWork.DiscreteFilter8_DSTATE[0];
    real_T xtmp = rtb_Upsample4;
    for (i=6; i>0; i--) {
      xtmp += Amtx[i]*x[i];
      x[i] = x[i-1];
    }

    x[0] = xtmp + Amtx[0]*x[0];
  }

  /* DiscreteFilter Block: '<Root>/Discrete Filter11' */
  {
    int_T i;
    const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter11_A[0];
    real_T *x = &DSPprieb4can_DWork.DiscreteFilter11_DSTATE[0];
    real_T xtmp = rtb_ADC1;
    for (i=6; i>0; i--) {
      xtmp += Amtx[i]*x[i];
      x[i] = x[i-1];
    }

    x[0] = xtmp + Amtx[0]*x[0];
  }

  /* DiscreteFilter Block: '<Root>/Discrete Filter9' */
  {
    int_T i;
    const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter9_A[0];
    real_T *x = &DSPprieb4can_DWork.DiscreteFilter9_DSTATE[0];
    real_T xtmp = rtb_Upsample5;
    for (i=6; i>0; i--) {
      xtmp += Amtx[i]*x[i];
      x[i] = x[i-1];
    }

    x[0] = xtmp + Amtx[0]*x[0];
  }
}

/* Model step function for TID1 */
void DSPprieb4can_step1(void)          /* Sample time: [0.00025s, 0.0s] */
{
  /* local block i/o variables */
  real_T rtb_Upsample;
  real_T rtb_Upsample1;
  real_T rtb_Upsample2;
```

```
real_T rtb_Upsample3;
real_T rtb_DiscreteFilter5;
real_T rtb_DiscreteFilter6;

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample' */
{
}

/* DiscreteFilter: '<Root>/Discrete Filter' */
DSPprieb4can_B.DiscreteFilter = DSPprieb4can_P.DiscreteFilter_D*
  DSPprieb4can_B.Downsample;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter_C[0];
  while (nx--) {
    DSPprieb4can_B.DiscreteFilter += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample2' */
{
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_B.Downsample2 = DSPprieb4can_B.DiscreteFilter;
  }
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample' */
{
  real_T *buf;

  /* special sample hit - toggle read buffer */
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_DWork.Upsample_ReadBufIdx =
      !(DSPprieb4can_DWork.Upsample_ReadBufIdx);
  }

  buf = (DSPprieb4can_DWork.Upsample_ReadBufIdx)
    ? &DSPprieb4can_DWork.Upsample_Buffer[0] + 1
    : &DSPprieb4can_DWork.Upsample_Buffer[0];

  {
    real_T *y = (real_T *)&rtb_Upsample;
    int_T c = DSPprieb4can_DWork.Upsample_Count;
    if (c++ == 0) {
      *y++ = *buf;
```

```
    } else {
      *y++ = 0.0;
    }

    if (c == 2)
      c = 0;

    /* Update counters for next sample hit */
    DSPprieb4can_DWork.Upsample_Count = c;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter7' */
rtb_DiscreteFilter5 = (DSPprieb4can_P.DiscreteFilter7_D)*rtb_Upsample;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter7_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter7_C[0];
  while (nx--) {
    rtb_DiscreteFilter5 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter1' */
DSPprieb4can_B.DiscreteFilter1 = DSPprieb4can_P.DiscreteFilter1_D*
  DSPprieb4can_B.Downsample;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter1_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter1_C[0];
  while (nx--) {
    DSPprieb4can_B.DiscreteFilter1 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample3' */
{
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_B.Downsample3 = DSPprieb4can_B.DiscreteFilter1;
  }
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample1' */
{
```

```
    real_T *buf;

    /* special sample hit - toggle read buffer */
    if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
      DSPprieb4can_DWork.Upsample1_ReadBufIdx =
        !(DSPprieb4can_DWork.Upsample1_ReadBufIdx);
    }

    buf = (DSPprieb4can_DWork.Upsample1_ReadBufIdx)
      ? &DSPprieb4can_DWork.Upsample1_Buffer[0] + 1
      : &DSPprieb4can_DWork.Upsample1_Buffer[0];

    {
      real_T *y = (real_T *)&rtb_Upsample1;
      int_T c = DSPprieb4can_DWork.Upsample1_Count;
      if (c++ == 0) {
        *y++ = *buf;
      } else {
        *y++ = 0.0;
      }

      if (c == 2)
        c = 0;

      /* Update counters for next sample hit */
      DSPprieb4can_DWork.Upsample1_Count = c;
    }
  }

  /* DiscreteFilter: '<Root>/Discrete Filter4' */
  rtb_DiscreteFilter6 = DSPprieb4can_P.DiscreteFilter4_D*rtb_Upsample1;

  {
    int_T nx = 7;
    const real_T *x = &DSPprieb4can_DWork.DiscreteFilter4_DSTATE[0];
    const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter4_C[0];
    while (nx--) {
      rtb_DiscreteFilter6 += (*Cmtx) * (*x++);
      Cmtx += 1;
    }
  }

  /* Sum: '<Root>/Sum' */
  DSPprieb4can_B.Sum = rtb_DiscreteFilter5 + rtb_DiscreteFilter6;

  /* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample4' */
  {
    real_T *inBuf = (DSPprieb4can_DWork.Upsample4_WriteBufIdx)
```

```
    ? &DSPprieb4can_DWork.Upsample4_Buffer[0] + 1 :
    &DSPprieb4can_DWork.Upsample4_Buffer[0];
  *inBuf = *(real_T *)&DSPprieb4can_B.Sum;

  /* toggle write buffer */
  DSPprieb4can_DWork.Upsample4_WriteBufIdx =
    !(DSPprieb4can_DWork.Upsample4_WriteBufIdx);
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample1' */
{
}

/* DiscreteFilter: '<Root>/Discrete Filter2' */
DSPprieb4can_B.DiscreteFilter2 = DSPprieb4can_P.DiscreteFilter2_D*
  DSPprieb4can_B.Downsample1;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter2_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter2_C[0];
  while (nx--) {
    DSPprieb4can_B.DiscreteFilter2 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample4' */
{
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_B.Downsample4 = DSPprieb4can_B.DiscreteFilter2;
  }
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample2' */
{
  real_T *buf;

  /* special sample hit - toggle read buffer */
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_DWork.Upsample2_ReadBufIdx =
      !(DSPprieb4can_DWork.Upsample2_ReadBufIdx);
  }

  buf = (DSPprieb4can_DWork.Upsample2_ReadBufIdx)
    ? &DSPprieb4can_DWork.Upsample2_Buffer[0] + 1
    : &DSPprieb4can_DWork.Upsample2_Buffer[0];
```

```
  {
    real_T *y = (real_T *)&rtb_Upsample2;
    int_T c = DSPprieb4can_DWork.Upsample2_Count;
    if (c++ == 0) {
      *y++ = *buf;
    } else {
      *y++ = 0.0;
    }

    if (c == 2)
      c = 0;

    /* Update counters for next sample hit */
    DSPprieb4can_DWork.Upsample2_Count = c;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter6' */
rtb_DiscreteFilter6 = (DSPprieb4can_P.DiscreteFilter6_D)*rtb_Upsample2;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter6_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter6_C[0];
  while (nx--) {
    rtb_DiscreteFilter6 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter3' */
DSPprieb4can_B.DiscreteFilter3 = DSPprieb4can_P.DiscreteFilter3_D*
  DSPprieb4can_B.Downsample1;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter3_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter3_C[0];
  while (nx--) {
    DSPprieb4can_B.DiscreteFilter3 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample5' */
{
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_B.Downsample5 = DSPprieb4can_B.DiscreteFilter3;
```

```
  }
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample3' */
{
  real_T *buf;

  /* special sample hit - toggle read buffer */
  if (DSPprieb4can_M->Timing.RateInteraction.TID1_2) {
    DSPprieb4can_DWork.Upsample3_ReadBufIdx =
      !(DSPprieb4can_DWork.Upsample3_ReadBufIdx);
  }

  buf = (DSPprieb4can_DWork.Upsample3_ReadBufIdx)
    ? &DSPprieb4can_DWork.Upsample3_Buffer[0] + 1
    : &DSPprieb4can_DWork.Upsample3_Buffer[0];

  {
    real_T *y = (real_T *)&rtb_Upsample3;
    int_T c = DSPprieb4can_DWork.Upsample3_Count;
    if (c++ == 0) {
      *y++ = *buf;
    } else {
      *y++ = 0.0;
    }

    if (c == 2)
      c = 0;

    /* Update counters for next sample hit */
    DSPprieb4can_DWork.Upsample3_Count = c;
  }
}

/* DiscreteFilter: '<Root>/Discrete Filter5' */
rtb_DiscreteFilter5 = DSPprieb4can_P.DiscreteFilter5_D*rtb_Upsample3;

{
  int_T nx = 7;
  const real_T *x = &DSPprieb4can_DWork.DiscreteFilter5_DSTATE[0];
  const real_T *Cmtx = &DSPprieb4can_P.DiscreteFilter5_C[0];
  while (nx--) {
    rtb_DiscreteFilter5 += (*Cmtx) * (*x++);
    Cmtx += 1;
  }
}

/* Sum: '<Root>/Sum1' */
```

```
DSPprieb4can_B.Sum1 = rtb_DiscreteFilter6 + rtb_DiscreteFilter5;

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample5' */
{
  real_T *inBuf = (DSPprieb4can_DWork.Upsample5_WriteBufIdx)
    ? &DSPprieb4can_DWork.Upsample5_Buffer[0] + 1 :
    &DSPprieb4can_DWork.Upsample5_Buffer[0];
  *inBuf = *(real_T *)&DSPprieb4can_B.Sum1;

  /* toggle write buffer */
  DSPprieb4can_DWork.Upsample5_WriteBufIdx =
    !(DSPprieb4can_DWork.Upsample5_WriteBufIdx);
}

/* DiscreteFilter Block: '<Root>/Discrete Filter' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter_DSTATE[0];
  real_T xtmp = DSPprieb4can_B.Downsample;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter7' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter7_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter7_DSTATE[0];
  real_T xtmp = rtb_Upsample;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter1' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter1_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter1_DSTATE[0];
  real_T xtmp = DSPprieb4can_B.Downsample;
```

```
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter4' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter4_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter4_DSTATE[0];
  real_T xtmp = rtb_Upsample1;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter2' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter2_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter2_DSTATE[0];
  real_T xtmp = DSPprieb4can_B.Downsample1;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
}

/* DiscreteFilter Block: '<Root>/Discrete Filter6' */
{
  int_T i;
  const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter6_A[0];
  real_T *x = &DSPprieb4can_DWork.DiscreteFilter6_DSTATE[0];
  real_T xtmp = rtb_Upsample2;
  for (i=6; i>0; i--) {
    xtmp += Amtx[i]*x[i];
    x[i] = x[i-1];
  }

  x[0] = xtmp + Amtx[0]*x[0];
```

```
  }

  /* DiscreteFilter Block: '<Root>/Discrete Filter3' */
  {
    int_T i;
    const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter3_A[0];
    real_T *x = &DSPprieb4can_DWork.DiscreteFilter3_DSTATE[0];
    real_T xtmp = DSPprieb4can_B.Downsample1;
    for (i=6; i>0; i--) {
      xtmp += Amtx[i]*x[i];
      x[i] = x[i-1];
    }

    x[0] = xtmp + Amtx[0]*x[0];
  }

  /* DiscreteFilter Block: '<Root>/Discrete Filter5' */
  {
    int_T i;
    const real_T *Amtx = &DSPprieb4can_P.DiscreteFilter5_A[0];
    real_T *x = &DSPprieb4can_DWork.DiscreteFilter5_DSTATE[0];
    real_T xtmp = rtb_Upsample3;
    for (i=6; i>0; i--) {
      xtmp += Amtx[i]*x[i];
      x[i] = x[i-1];
    }

    x[0] = xtmp + Amtx[0]*x[0];
  }
}

/* Model step function for TID2 */
void DSPprieb4can_step2(void)          /* Sample time: [0.0005s, 0.0s] */
{
  /* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample2' */
  {
  }

  /* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample' */
  {
    real_T *inBuf = (DSPprieb4can_DWork.Upsample_WriteBufIdx)
      ? &DSPprieb4can_DWork.Upsample_Buffer[0] + 1 :
      &DSPprieb4can_DWork.Upsample_Buffer[0];
    *inBuf = *(real_T *)&DSPprieb4can_B.Downsample2;

    /* toggle write buffer */
    DSPprieb4can_DWork.Upsample_WriteBufIdx =
      !(DSPprieb4can_DWork.Upsample_WriteBufIdx);
```

```
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample3' */
{
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample1' */
{
  real_T *inBuf = (DSPprieb4can_DWork.Upsample1_WriteBufIdx)
    ? &DSPprieb4can_DWork.Upsample1_Buffer[0] + 1 :
    &DSPprieb4can_DWork.Upsample1_Buffer[0];
  *inBuf = *(real_T *)&DSPprieb4can_B.Downsample3;

  /* toggle write buffer */
  DSPprieb4can_DWork.Upsample1_WriteBufIdx =
    !(DSPprieb4can_DWork.Upsample1_WriteBufIdx);
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample4' */
{
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample2' */
{
  real_T *inBuf = (DSPprieb4can_DWork.Upsample2_WriteBufIdx)
    ? &DSPprieb4can_DWork.Upsample2_Buffer[0] + 1 :
    &DSPprieb4can_DWork.Upsample2_Buffer[0];
  *inBuf = *(real_T *)&DSPprieb4can_B.Downsample4;

  /* toggle write buffer */
  DSPprieb4can_DWork.Upsample2_WriteBufIdx =
    !(DSPprieb4can_DWork.Upsample2_WriteBufIdx);
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample5' */
{
}

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample3' */
{
  real_T *inBuf = (DSPprieb4can_DWork.Upsample3_WriteBufIdx)
    ? &DSPprieb4can_DWork.Upsample3_Buffer[0] + 1 :
    &DSPprieb4can_DWork.Upsample3_Buffer[0];
  *inBuf = *(real_T *)&DSPprieb4can_B.Downsample5;

  /* toggle write buffer */
  DSPprieb4can_DWork.Upsample3_WriteBufIdx =
    !(DSPprieb4can_DWork.Upsample3_WriteBufIdx);
```

```
  }
}

void DSPprieb4can_step(int_T tid)
{
  switch (tid) {
   case 0 :
    DSPprieb4can_step0();
    break;

   case 1 :
    DSPprieb4can_step1();
    break;

   case 2 :
    DSPprieb4can_step2();
    break;

   default :
    break;
  }
}

/* Model initialize function */
void DSPprieb4can_initialize(boolean_T firstTime)
{
  (void)firstTime;

  /* Registration code */
  /* initialize real-time model */
  (void) memset((char_T *)DSPprieb4can_M,0,
                sizeof(RT_MODEL_DSPprieb4can));

  /* block I/O */
  {
    int_T i;
    void *pVoidBlockIORegion;
    pVoidBlockIORegion = (void *)(&DSPprieb4can_B.DiscreteFilter10);
    for (i = 0; i < 15; i++) {
      ((real_T*)pVoidBlockIORegion)[i] = 0.0;
    }
  }

  /* states (dwork) */
  (void) memset((char_T *) &DSPprieb4can_DWork,0,
                sizeof(D_Work_DSPprieb4can));

  {
```

```
  int_T i;
  real_T *dwork_ptr = (real_T *) &DSPprieb4can_DWork.DiscreteFilter10_DSTATE[0];
  for (i = 0; i < 108; i++) {
    dwork_ptr[i] = 0.0;
  }
}

codec_init();

/* S-Function Block: <Root>/ADC1 (c6416dsk_adc) */
config_codec_input();

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample' */
{
  DSPprieb4can_DWork.Downsample_EnableSysFlag = 0;
  DSPprieb4can_DWork.Downsample_DisableTimeTick = 0;
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample2' */
{
  DSPprieb4can_DWork.Downsample2_EnableSysFlag = 0;
  DSPprieb4can_DWork.Downsample2_DisableTimeTick = 0;
}

{
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample3' */
{
  DSPprieb4can_DWork.Downsample3_EnableSysFlag = 0;
  DSPprieb4can_DWork.Downsample3_DisableTimeTick = 0;
}

{
}

{
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample1' */
{
  DSPprieb4can_DWork.Downsample1_EnableSysFlag = 0;
  DSPprieb4can_DWork.Downsample1_DisableTimeTick = 0;
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample4' */
{
  DSPprieb4can_DWork.Downsample4_EnableSysFlag = 0;
```

```
  DSPprieb4can_DWork.Downsample4_DisableTimeTick = 0;
}

{
}

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample5' */
{
  DSPprieb4can_DWork.Downsample5_EnableSysFlag = 0;
  DSPprieb4can_DWork.Downsample5_DisableTimeTick = 0;
}

{
}

{
}

/* S-Function Block: <Root>/DAC (c6416dsk_dac) */
config_codec_output();

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample' */
if (DSPprieb4can_DWork.Downsample_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample_Buffer[0];
  DSPprieb4can_DWork.Downsample_pInBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample_Count = 0;
}

MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample_Buffer[0], (const
  byte_T *)&DSPprieb4can_P.Downsample_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample2' */
if (DSPprieb4can_DWork.Downsample2_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample2_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample2_Buffer[0];
  DSPprieb4can_DWork.Downsample2_pInBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample2_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample2_Count = 0;
}
```

```
MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample2_Buffer[0], (
  const byte_T *)&DSPprieb4can_P.Downsample2_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample' */
MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample_Buffer[0], (const
  byte_T *)&DSPprieb4can_P.Upsample_IC, 1, sizeof(real_T));
DSPprieb4can_DWork.Upsample_InputIdx = 0;
DSPprieb4can_DWork.Upsample_Count = 0;
DSPprieb4can_DWork.Upsample_ReadBufIdx = 1;
DSPprieb4can_DWork.Upsample_WriteBufIdx = 1;

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample3' */
if (DSPprieb4can_DWork.Downsample3_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample3_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample3_Buffer[0];
  DSPprieb4can_DWork.Downsample3_pInBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample3_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample3_Count = 0;
}

MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample3_Buffer[0], (
  const byte_T *)&DSPprieb4can_P.Downsample3_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample1' */
MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample1_Buffer[0], (const
  byte_T *)&DSPprieb4can_P.Upsample1_IC, 1, sizeof(real_T));
DSPprieb4can_DWork.Upsample1_InputIdx = 0;
DSPprieb4can_DWork.Upsample1_Count = 0;
DSPprieb4can_DWork.Upsample1_ReadBufIdx = 1;
DSPprieb4can_DWork.Upsample1_WriteBufIdx = 1;

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample4' */
MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample4_Buffer[0], (const
  byte_T *)&DSPprieb4can_P.Upsample4_IC, 1, sizeof(real_T));
DSPprieb4can_DWork.Upsample4_InputIdx = 0;
DSPprieb4can_DWork.Upsample4_Count = 0;
DSPprieb4can_DWork.Upsample4_ReadBufIdx = 1;
DSPprieb4can_DWork.Upsample4_WriteBufIdx = 1;

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample1' */
if (DSPprieb4can_DWork.Downsample1_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample1_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample1_Buffer[0];
  DSPprieb4can_DWork.Downsample1_pInBuf = (byte_T *)
```

```
    &DSPprieb4can_DWork.Downsample1_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample1_Count = 0;
}

MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample1_Buffer[0], (
  const byte_T *)&DSPprieb4can_P.Downsample1_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample4' */
if (DSPprieb4can_DWork.Downsample4_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample4_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample4_Buffer[0];
  DSPprieb4can_DWork.Downsample4_pInBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample4_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample4_Count = 0;
}

MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample4_Buffer[0], (
  const byte_T *)&DSPprieb4can_P.Downsample4_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample2' */
MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample2_Buffer[0], (const
  byte_T *)&DSPprieb4can_P.Upsample2_IC, 1, sizeof(real_T));
DSPprieb4can_DWork.Upsample2_InputIdx = 0;
DSPprieb4can_DWork.Upsample2_Count = 0;
DSPprieb4can_DWork.Upsample2_ReadBufIdx = 1;
DSPprieb4can_DWork.Upsample2_WriteBufIdx = 1;

/* Signal Processing Blockset Downsample (sdspdsamp2) - '<Root>/Downsample5' */
if (DSPprieb4can_DWork.Downsample5_EnableSysFlag == 0) {
  /* Initialize input and output buffer pointers */
  DSPprieb4can_DWork.Downsample5_pOutBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample5_Buffer[0];
  DSPprieb4can_DWork.Downsample5_pInBuf = (byte_T *)
    &DSPprieb4can_DWork.Downsample5_Buffer[0] + 1 * sizeof(real_T);

  /* Initialize counter */
  DSPprieb4can_DWork.Downsample5_Count = 0;
}

MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Downsample5_Buffer[0], (
  const byte_T *)&DSPprieb4can_P.Downsample5_IC, 1, sizeof(real_T));

/* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample3' */
```

```
  MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample3_Buffer[0], (const
    byte_T *)&DSPprieb4can_P.Upsample3_IC, 1, sizeof(real_T));
  DSPprieb4can_DWork.Upsample3_InputIdx = 0;
  DSPprieb4can_DWork.Upsample3_Count = 0;
  DSPprieb4can_DWork.Upsample3_ReadBufIdx = 1;
  DSPprieb4can_DWork.Upsample3_WriteBufIdx = 1;

  /* Signal Processing Blockset Upsample (sdspupsamp2) - '<Root>/Upsample5' */
  MWDSP_CopyScalarICs((byte_T *)&DSPprieb4can_DWork.Upsample5_Buffer[0], (const
    byte_T *)&DSPprieb4can_P.Upsample5_IC, 1, sizeof(real_T));
  DSPprieb4can_DWork.Upsample5_InputIdx = 0;
  DSPprieb4can_DWork.Upsample5_Count = 0;
  DSPprieb4can_DWork.Upsample5_ReadBufIdx = 1;
  DSPprieb4can_DWork.Upsample5_WriteBufIdx = 1;

  {
  }

  {
  }

  {
  }

  {
  }

  {
  }

  {
  }
}

/* Model terminate function */
void DSPprieb4can_terminate(void)
{
  /* (no terminate code required) */
}

/* File trailer for Real-Time Workshop generated code.
 *
 * [EOF]
 */
```

Bibliografía

[1] N. J. Fliege: *Multirate Digital Signal Processing*. John Wiley Sons, Ltd, New York 2000.

[2] Douglas K. Lindner: *Introducción a las señales y a los sistemas*. McGraw-Hill. Caracas, Venezuela. 2002

[3] Sanjit K. Mitra: *Digital Signal Processing. A computer-based approach*. 3rd Edition. McGraw-Hill. New York, 2006.

[4] Bohumil Psenicka: *Procesamiento de Señales. Filtros Pasivos, Activos y Digitales*. UNAM FI. México D.F.

[5] D. Esteban, C. Galand: *Application of Quadrature Mirror Filters to Split Band Voice Coding Schemes* Proc IEEE ICASSP'77, May 1997.

[6] M. Bellanger, G. Bonnerot, and M. Coudreuse: *Digital filtering by poliphase network: application to sample-rate alteration and filter banks.* IEEE Trans. Acoustics, Speech and Signal Processing, ASSP-24:109-114, April 1976.

[7] Alan V. Oppenheim and Alan S. Willsky *Signals and systems*, Pearson Education,1998.

[8] B. Psenicka, F, García Ugalde y V.F RuÍz. *Practical Design of Digital Filtres Using the Pascal Matrix.*, 2006.

[9] Rabiner R. and Gold B. *Theory and Applications of Digital Signal Processing*, Prentice-Hall, 1975.

[10] B. Psenicka, O. Nieto y V. López *Prácticas de laboratorio con microprocesadordes TMS320C6711*