UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

SIMULACIÓN PARA PLATAFORMA XILINX (FPGA) E IMPLEMENTACIÓN HARDWARE (DSP TEXAS INSTRUMENTS TMS320C6711) EN PUNTO FIJO Y PUNTO FLOTANTE DE UN CODIFICADOR DE VOZ LPC-10 PARA BAJAS TASAS DE BITS

TESIS

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO ELECTRÓNICO

PRESENTADO POR:

FELIPE DANIEL ARGANDOÑA MARTINEZ

PROMOCIÓN
2001 – II

LIMA – PERÚ
2008

SIMULACIÓN PARA PLATAFORMA XILINX (FPGA) E IMPLEMENTACIÓN HARDWARE (DSP TEXAS INSTRUMENTS TMS320C6711) EN PUNTO FIJO Y PUNTO FLOTANTE DE UN CODIFICADOR DE VOZ LPC-10 PARA BAJAS TASAS DE BITS

Agradezco a Dios sobre todas las cosas.

A mis Padres: Román y Noemí
Al Dr. Jorge Del Carpio Salinas
y a los compañeros del Centro de Investigación y Desarrollo CIDFIEE

SUMARIO

El presente trabajo de tesis describe el estudio, simulación (para plataforma FPGA Xilinx) e implementación en hardware (DSP) en dos versiones: punto fijo de 16 bits y punto flotante, de un codificador de voz basado en el esquema LPC-10 para canales de banda angosta.

En primer lugar, se realiza un estudio del marco teórico en el que se basan los codificadores de voz, luego se describen las técnicas de procesamiento de voz más importantes que conforman el presente trabajo, destacando entre ellas el algoritmo SIFT para cálculo óptimo de la frecuencia fundamental de la voz. Seguidamente se describen los detalles de la simulación y de la implementación en DSP en las dos versiones (punto fijo y punto flotante).

En base al estudio y correcto conocimiento de todos los bloques constituyentes del esquema LPC-10, se proponen mejoras (en el algoritmo SIFT) y optimizaciones tanto en el ámbito algorítmico, como en el ámbito del procesador DSP. Con ello se pretende obtener una mayor calidad de voz decodificada (sintética), sin comprometer seriamente la tasa de compresión del sistema ni el tiempo de ejecución del algoritmo, de tal forma que pueda ser empleado en sistemas de transmisión digital de datos de banda angosta.

La implementación en hardware constituye un aporte importante del presente trabajo. El codificador es implementado sobre la tarjeta de desarrollo DSK TMS320C6711 de Texas Instruments en las dos versiones (punto fijo y punto flotante).

Finalmente se evalúa la calidad de la voz sintética del codificador mediante pruebas subjetivas (Test MOS).

INDICE

INTRODUCCION

El propósito del presente trabajo de tesis apunta en primer lugar a estudiar, desarrollar, simular e implementar un codificador de voz basado en el esquema de predicción lineal LPC 10, de calidad aceptable en función de los requerimientos de un sistema de comunicaciones de banda angosta. Conjuntamente, se proponen técnicas de mejoras y optimización apuntando a obtener una calidad aceptable de la voz decodificada con el codificador propuesto. Una parte importante es la simulación del codificador propuesto, el cual se realiza en la plataforma System Generator para dispositivos FPGA de la familia Xilinx, lo que servirá de base para futuras implementaciones hardware en dichos dispositivos.

La implementación del codificador propuesto en hardware (procesador de señales DSP) en tiempo real en dos versiones (punto fijo de 16 bits y punto flotante) constituye la parte principal del presenta trabajo de tesis. Lo anterior implica diseñar modularmente los algoritmos de codificación y de decodificación de voz, apuntando a realizar mejoras futuras y/o aplicaciones de valor agregado (Ej. criptografía). Con esto se busca demostrar la operatividad del codificador de voz en tiempo real, lo cual constituye una base fundamental para la implementación de sistemas de comunicaciones reales.

Finalmente la evaluación del codificador con los estándares subjetivos de medición establecidos (Test MOS) sirve para medir la calidad del codificador implementado.

El capítulo 1 presenta el esquema general, el contexto, la justificación y los objetivos del presente trabajo.

El capítulo 2 describe los fundamentos de la codificación de voz, los codificadores de voz, y varios conceptos básicos claves que son aplicados en muchos esquemas de codificación de voz. Así mismo, el capítulo 2 también nos ofrece información teórica acerca de la producción de la voz, su modelado y las propiedades del sistema de audición humano. Dichos conceptos juegan un papel crucial en el diseño de los codificadores modernos de voz.

El capítulo 3 describe las técnicas básicas que se emplean en el procesamiento de las señales de voz. Se describen también los algoritmos para el cálculo de los parámetros mas importantes en el modelado de las señales de voz como son: el pitch, determinación de segmentos "sonoros" o "sordos", densidad espectral de potencia, etc. También se presenta el estudio de la predicción lineal.

El capítulo 4 describe en detalle el algoritmo SIFT, el cual es un algoritmo eficiente para el cálculo óptimo de la frecuencia fundamental de la voz y que constituye además una parte importante en el presente trabajo.

El capítulo 5 ofrece una breve descripción de los dispositivos FPGA, de los modos de generación de software para FPGA's de la familia Xilinx y al final hay una descripción del modo de trabajo en System Generator. Adicionalmente este capítulo muestra un comparativo entre el modo de programación en punto fijo y punto flotante.

El capítulo 6 presenta la descripción detallada del codificador implementado en System Generator, así como los resultados correspondientes para cada bloque que conforma el codificador o decodificador.

El capítulo 7 presenta la descripción de la tarjeta de desarrollo DSP TMS320C6711 de Texas Instruments. Este capítulo muestra además el flujo de generación de software para dicho dispositivo DSP.

El capítulo 8 presenta la descripción detallada del codificador implementado en las dos versiones: punto fijo de 16 bits y punto flotante, en el DSP TMS320C6711, así como los resultados correspondientes para cada bloque desde la entrada en PCM-16 bits / 8 KHz hasta la señal decodificada a la salida del decodificador. Cabe destacar que cada uno de los bloques viene descrito con los fundamentos teóricos necesarios, fórmulas y diagramas de flujo de los algoritmos correspondientes y a su vez se presentan resultados (capturas del Code Composer Studio) los cuales respaldan la teoría subyacente y confirman en la práctica la correcta operación del codificador.

El capítulo 9 presenta la evaluación subjetiva del codificador. Se presentan los resultados de una seria de pruebas subjetivas (Test MOS) del codificador para ambas versiones del codificador implementado.

Finalmente se listan las conclusiones del presente trabajo de tesis.

El autor del presente trabajo de tesis agradece al Centro de Investigación y Desarrollo de la Facultad de Ingeniería Eléctrica y Electrónica de la Universidad Nacional de Ingeniería y al Doctor Jorge Del Carpio por el apoyo en el desarrollo del presente trabajo.

CAPITULO I

ANTECEDENTES Y PLANTEAMIENTO DEL PROBLEMA

1.1 Antecedentes y Justificación

Debido a la masificación de las tecnologías para comunicación por voz, la codificación de la voz ha recibido, a través de los años, mucho interés por parte de la comunidad científica, las organizaciones de estándares y la industria, llegándose a obtener en la actualidad diversos modelos y soluciones (algunos muy sofisticados) orientados siempre a cumplir los requerimientos deseables en toda codificación de voz:

- Baja tasa de bits.
- Alta calidad de la voz.
- Buen rendimiento en presencia de otras señales (diferentes de las señales de voz).
- Bajo requerimiento de memoria.
- Baja complejidad computacional.
- Bajo retardo debido al tamaño del código, etc.

Es muy difícil que un modelo o solución satisfaga todos los requerimientos arriba mencionados, sino que, dependiendo de la aplicación en particular se debe hacer una evaluación costo-beneficio entre los distintos requerimientos y escoger los que más se adecuen a nuestro caso.

Una clasificación de los Codificadores de voz es de acuerdo a la tasa de bits, tal como se muestra en la TABLA Nº 1.1 (4).

TABLA Nº 1.1. Clasificación de los codificadores de acuerdo a la tasa de bits (4).

Categoría	Rango de la Tasa de Bits
Tasa de Bits Alta	> 15 kbps
Tasa de Bits Media	5 a 15 kbps
Tasa de Bits Baja	2 a 5 kbps
Tasa de Bits Muy Baja	< 2 kbps

Hoy en día la mayoría de los estándares son diseñados para tasas de bits menores que la Tasa de Bits media indicada en la TABLA Nº 1.1 y cada vez son más las aplicaciones que requieren tasas de bits bajas con una buena calidad de voz. Tales aplicaciones incluyen: videojuegos, sistemas inalámbricos, sistemas celulares, defensa, *Digital Mobile Radio*, *VoIP*, comunicación multimedia, etc. La TABLA Nº 1.2. muestra un resumen de los principales estándares de codificación de voz (4).

Un grupo de aplicaciones donde es primordial codificar la voz a bajas tasas de bits es la transmisión por el medio inalámbrico. Como se puede ver en la Fig. 1.2 muchos estándares están destinados para trabajar en el sistema celular el cual especifica un nivel de calidad mínimo "*Toll Quality*" que debe de ser cumplido por los desarrolladores de codificadores de voz. Por otro lado, las técnicas de codificación de voz nos ofrecen las herramientas necesarias para hacer nuestros propios codificadores de voz "a la medida" para aplicaciones específicas donde no es necesario ceñirse a las especificaciones de los estándares (tamaños de los *frames* por ejemplo), consiguiendo de esta manera dar una solución particular a un determinado problema.

El presente trabajo de tesis hace uso de las técnicas de codificación de voz para implementar en hardware un codificador de voz para transmisión digital de la voz a través de canales de banda angosta, esto es, con bajas tasas de bit disponible.

TABLA Nº 1.2. Estándares de codificación de voz (basado en (4)).

Año en que se implementó	Nombre del Estándar	Tasa de Bits (kbps)	Aplicaciones
1972	ITU - T G.711 PCM	64	Propósito general
1984	FS 1015 LPC	2,4	Comunicación segura
1987	ETSI GSM 6,10 RPE-LTP	13	Radio móvil digital
1990	ITU - T G.726 ADPCM	16, 24, 32,40	Propósito general
1990	TIA IS54 VSELP	7,95	Telefonía celular digital TDMA de norteamérica
1990	ETSI GSM 6,20 VSELP	5,6	Sistema celular GSM
1990	RCR STD-27B VSELP	6,7	Sistema celular japonés
1991	FS1016 CELP	4,8	Comunicación segura
1992	ITU - T G.728 LD-CELP	16	Propósito general
1993	TIA IS96 VBR-CELP	8,5, 4, 2, 0,8	Telefonía celular digital CDMA de norteamérica
1995	ITU - T G.723,1 MP-MLQ/ACELP	5,3, 6,3	Comunicaciones multmedia
1995	ITU - T G,729 CS-ACELP	8	Propósito general
1996	ETSI GSM EFR ACELP	12,2	Propósito general
1996	TIA IS641 ACELP	7,4	Telefonía celular digital TDMA de norteamérica
1997	FS MELP	2,4	Comunicación segura
1999	ETSI AMR-ACELP	12,2, 10,2, 7,95, 7,40, 6,70, 5,90, 5,15, 4,75	Propósito general
2000	AMR-WB	6.6 a 23.85	UMTS, GSM
2004	VMR-WB	0.8 a 13.3	De tasa variable. Telefonía celular digital CDMA

1.2 Soluciones comerciales en hardware para codificación de voz a bajas tasas de bits

Hoy en día son muchas las empresas que ofrecen soluciones hardware para una amplia variedad de aplicaciones de la codificación de la voz, sin embargo, una de las constantes de todas esas soluciones comerciales es el elevado costo de sus productos, tal como se puede apreciar en la TABLA Nº 1.3

TABLA Nº 1.3. Precios de algunos codificadores de voz comerciales (37, 38).

Compañía	Producto	Precio
Compandent	Vocoder MELP 2.4 kpbs	$1500
Compandent	Vocoder MELP 1.2 kpbs	> $2000
Compandent	Vocoder MELP 600 kpbs	> $2000
DSP Wizard	G.729.B Vocoder GSM	> $11000
Vocal Technologies	Vocoder MELP	> $2000

Ante esto, es fundamental que se pueda contar con una solución hecha a la medida y a bajo costo. El aporte del presente proyecto es precisamente diseñar e

implementar en hardware un codificador de voz específico para aplicaciones de bajas tasas de bit.

1.3 Posibles aplicaciones prácticas del codificador

Las aplicaciones de un codificador de voz para bajas tasas de bits son muchas: desde aplicativos de Internet, juegos en red, comunicaciones inalámbricas, etc. Una aplicación interesante es la transmisión digital de la voz por canales HF el cual es un medio que, entre otros limitantes, restringe el ancho de banda disponible, pero por otro lado los canales HF permiten comunicaciones de larga y muy larga distancia gracias a un fenómeno conocido como propagación ionosférica, consistente en la reflexión de las señales de radiofrecuencia en las capas altas de la atmósfera (la más importante de ellas situada a 250 Km. de altitud). El principal inconveniente de esta banda es que las señales transmitidas se encuentran expuestas a efectos de absorción atmosférica, elevado ruido y un acusado *multipath* (multicamino). Además, las condiciones de transmisión dependen de muchos factores (momento del día, estación del año, actividad de las manchas solares, tormentas ionosféricas, etc.). Lo anterior se representa en la Fig. 1.1.

En los canales presentes en la banda HF (3-30MHz) y dependiendo de la modulación digital empleada, se pueden alcanzar velocidades de transmisión de datos de 2.4 kbps, 4.8kpbs y hasta 9.6 kbps (si se usan técnicas sofisticadas de modulación digital). Son precisamente éstas bajas tasas de transmisión de datos las que nos condicionan a desarrollar codificadores de Baja Tasa de Bits, tal como el que se propone en el presente trabajo.

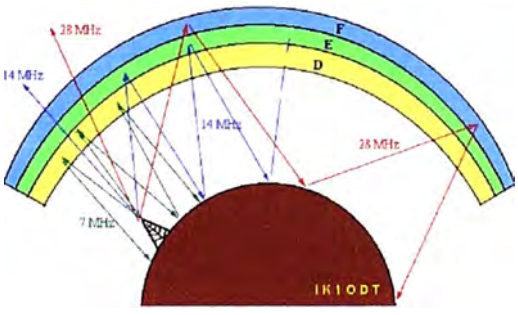


Fig. 1.1. Rebote de las ondas HF a través de la Ionosfera (39)

Por otro lado, tal como se comentó en la página anterior, se cuentan en el mercado con codificadores de voz comerciales (*vocoders*) independientes y equipos transreceptores que asimismo incluyen *vocoders* pero cuyos precios elevados los harían prohibitivos para un proyecto de desarrollo social en nuestro país. Por lo que es

imperativo el desarrollo local de codificadores de voz para bajas tasas de bits que puedan ser implementados en hardware, tal como se propone en el presente trabajo.

1.4 Plataformas hardware destino para implementar codificadores de voz

Hoy en día existen varias plataformas hardware donde se pueden implementar algoritmos para codificadores de voz, de entre ellos se pueden mencionar las siguientes:

- **Microprocesadores**. Estos dispositivos son potentes y generalmente son usados para implementar y soportar sistemas complejos que requieren alto rendimiento. Un microprocesador tal como de la familia Intel es mas que suficiente para soportar un codificador de voz que implemente un algoritmo altamente complejo. En estos casos por los, generalmente, altos costos de los microprocesadores, no amerita destinar un microprocesador para solamente implementar el codificador.

- **Microcontroladores**. Estos dispositivos son de menor potencia que los microprocesadores y son usados generalmente para labores de control de un sistema. Es posible implementar un codificador de voz en uno de estos dispositivos, pero la velocidad del mismo afectará el rendimiento de algoritmos de voz mas complejos.

- **DSP (Procesador Digital de Señales).** Estos dispositivos presentan una arquitectura destinada específicamente a soportar las estructuras básicas que conforman los algoritmos de procesamiento de señales en general, y entre ellos, los algoritmos de procesamiento de voz, por lo tanto son adecuados para la implementación de los codificadores de voz. Además su bajo costo los hace adecuados para la elaboración de productos electrónicos de consumo masivo.

- **FPGA (Field Programmable Gate Array).** Estos dispositivos se encuentran entre los más potentes que existen ya que no presentan una arquitectura fija tal como los DSP o microprocesadores, sino que permiten programar a nivel de hardware de tal forma que se puede optimizar un sistema (algoritmo) al máximo y solo depende de la pericia del diseñador. Si se tiene un algoritmo de voz muy complejo y se quiere hacer que se

ejecute en el menor tiempo posible, entonces una alternativa es implementarlo completamente en un dispositivo FPGA.

- **ASIC (Application Specific Integrated Circuits).** Esta alternativa es la mas costosa de todas ya que se tiene que diseñar un circuito integrado específico para la aplicación en cuestión, sin embargo es la más efectiva computacionalmente hablando ya que se puede hacer que el algoritmo del codificador de voz se ejecute incluso más rápido de lo que se ejecutaría si usásemos un dispositivo FPGA. Es una buena alternativa si es que nuestro diseño se implementará luego masivamente ya que en ese escenario debido al volumen de ventas, sí ameritaría la inversión en el diseño del circuito integrado.

El presente trabajo de tesis cubre dos plataformas de desarrollo hardware: DSP y FPGA. Para el caso del DSP se logra realizar la implementación completa del codificador en dos versiones: punto fijo y punto flotante. Mientras que en el caso del FPGA se logra realizar la simulación en el entorno System Generator de Xilinx, el cual es un paso importante para una futura implementación hardware en un FPGA de la familia Xilinx.

CAPITULO I I

FUNDAMENTOS DE LA CODIFICACION DE LA VOZ

2.1 Visión General de la Codificación de la Voz

La codificación de la voz persigue representar una señal de voz digitalizada usando la menor cantidad de bits como sea posible y tratando a su vez de mantener un nivel razonable de calidad de la voz de acuerdo a la aplicación en cuestión. Por ejemplo las aplicaciones de telefonía celular exigen un alto nivel de calidad de voz para satisfacer los requerimientos de usuarios finales. Por otro lado hay aplicaciones como por ejemplo los juegos en red (a través de la Internet) los cuales no son muy exigentes en cuanto a la calidad de la voz recibida o transmitida. *(4, 25)*

La codificación de la voz siempre ha sido y sigue siendo un campo de interés en la comunidad científica y cada año se siguen desarrollando teorías y técnicas para obtener codificadores de voz cada vez mas sofisticados.

Básicamente un codificador de voz es logrado mediante un algoritmo computacional. Dicho algoritmo puede ser ejecutado en un procesador específico como lo es un DSP, o también puede ser implementado puramente en hardware, como sería el caso de un FPGA. Ambas plataformas presentan ventajas y desventajas y depende del diseñador escoger la plataforma final de implementación.

Hay muchos esquemas, técnicas y estándares bien definidos y ampliamente usados dentro del campo de la codificación de la voz, los cuales pueden ser aplicados a una situación en particular sujeta a determinadas restricciones. La ubicación de un codificador de voz dentro de un sistema completo de comunicaciones es tal como se muestra en la Fig. 2.1 (ver bloques de color verde).

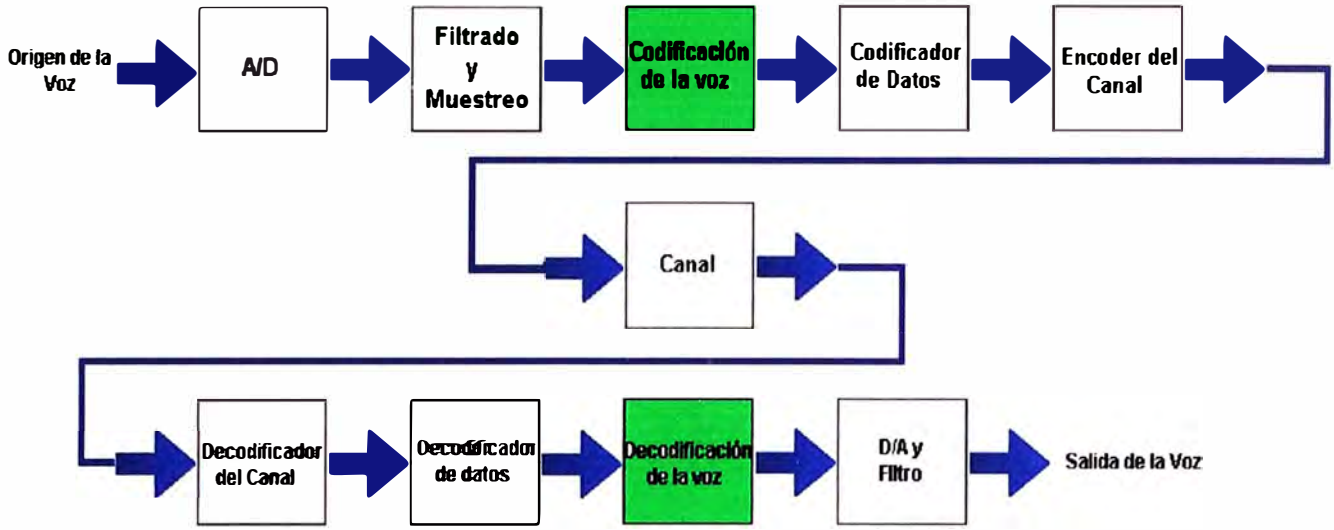


Fig. 2.1. Ubicación de un Codificador de Voz en un Sistema de Comunicación (4)

2.1.1 Características Optimas de un Codificador de Voz

Son las siguientes:

- *Baja Tasa de Bits*. Mientras mas baja sea la tasa de bits, menos ancho de banda es necesario para su transmisión, conduciendo a un sistema más eficiente. Este requerimiento esta en conflicto constante con otras propiedades del sistema, principalmente la calidad de la voz. En la práctica siempre se busca el equilibrio entre estos dos factores.
- *Alta Calidad de la Voz*. La voz decodificada debería de tener una calidad acorde a la aplicación específica. Hay varios parámetros en la percepción de la calidad de la voz: inteligibilidad, naturalidad, reconocimiento del hablante, etc.
- *Robustez frente a diferentes hablantes e idiomas*. La técnica empleada en el codificador de voz debe ser lo suficientemente robusta como para modelar eficientemente diferentes hablantes (hombres y mujeres adultos, y niños) y diferentes idiomas.
- *Robustez frente a errores de canal*. Es crucial para comunicaciones digitales donde los errores de canal tendrán un impacto negativo en la calidad de la voz.
- *Buena respuesta frente a señales que no son de voz*. Por ejemplo, en un sistema típico de telefonía, es típico encontrar tonos DTMF. Si bien es cierto, los codificadores no están diseñados para lidiar con este tipo de señales, deben responder de tal forma que no ocasionen ruidos molestos en el receptor.

- *Bajos requerimientos de memoria*. Con el fin de que sean realizables en la práctica sobre todo en procesadores de bajo precio, es crucial que el requerimiento de memoria sea mínimo.
- *Bajo retardo de codificación*. El codificador no debe introducir demasiado retardo ya que esto podría influir en la performance de todo el sistema. (4)

2.1.2 Clasificación de los Codificadores de Voz

Clasificación de acuerdo a la tasa de bits. Esta clasificación se resume en la TABLA Nº 1.1.

Clasificación por Técnica. Tenemos las siguientes:

- *Codificadores de forma de onda*. Estos codificadores hacen un intento para preservar la forma original de la forma de onda de la señal de entrada. Aquí, el codificador puede ser aplicado a cualquier señal de entrada. Son apropiados para altas tasas de bits ya que el rendimiento decrece drásticamente cuando se disminuye la tasa de bits. En la práctica, estos codificadores trabajan mejor a tasas de bits de 32 Kbps o superiores.
- *Codificadores Paramétricos*. Dentro del marco de trabajo de estos tipos de codificadores, se asume que la señal de voz es generada a partir de un modelo el cual es controlado por algunos parámetros. Durante la codificación, los parámetros del modelo son estimados de la señal de entrada, seguidamente estos parámetros son transmitidos como una secuencia de bits.
- *Codificadores Híbridos*. Este tipo de codificador combina la fuerza de un codificador de forma de onda con el de un codificador paramétrico. *(4, 25)*

2.2 Modelado y Producción de la Voz

La voz es una onda sonora de presión originada por movimientos controlados de estructuras anatómicas que conforman el sistema humano de producción de la voz tal como se puede apreciar en la Fig. 2.2.

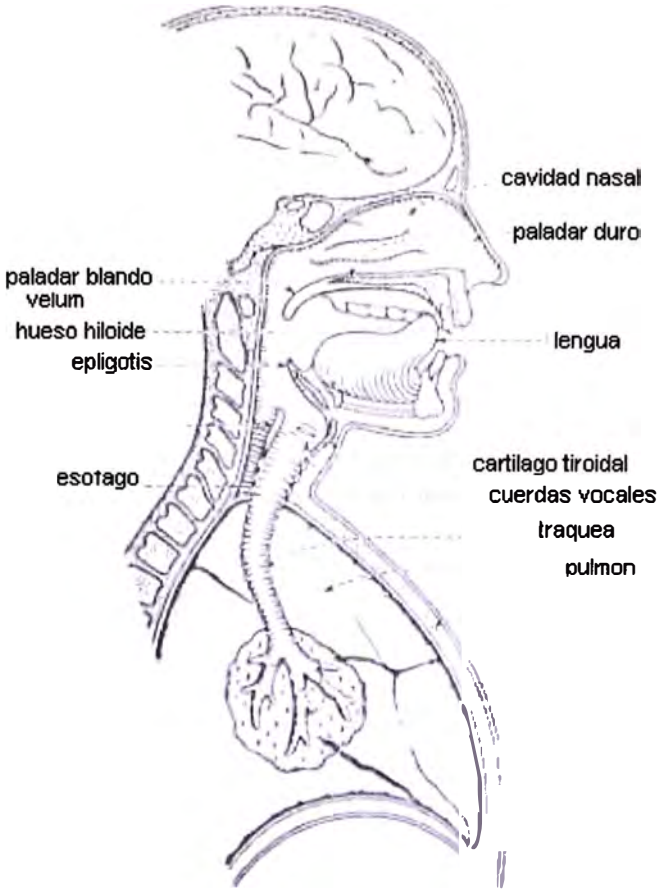


Fig. 2.2. Diagrama del sistema humano de producción de voz [44].

La voz es generada como una onda acústica que es radiada desde las fosas nasales y la boca cuando el aire es expulsado de los pulmones con el resultante flujo de aire perturbado por las constricciones dentro del cuerpo. Es muy útil interpretar la producción de la voz en términos de filtrado acústico. Las tres principales cavidades del sistema de producción de la voz son la nasal, oral y faríngea los cuales a su vez forman el filtro acústico principal. El filtro es excitado por el aire de los pulmones y es cargado en su salida principal por una impedancia de radiación asociada con los labios.

En la faringe encontramos uno de los componentes más importante del sistema de producción de la voz: las cuerdas vocales. La localización de las cuerdas vocales esta a la altura de la manzana de Adán. Las cuerdas vocales son un par de bandas de músculos y membranas mucosas que se abren y cierran rápidamente durante la producción de la voz. La velocidad a la cual las cuerdas se abren y cierran es única para cada persona y define la característica y personalidad de una voz en particular. (4,25)

Una señal de voz se puede clasificar como "sonora" o "sorda". Las señales "sonoras" son generadas cuando las cuerdas vocales vibran de tal forma que el flujo de aire desde los pulmones e interrumpido periódicamente, creando una secuencia de pulsos que excitan el tracto vocal. Cuando las cuerdas vocales están estacionarias, la turbulencia creada por el flujo de aire pasando a través de una contracción del tracto vocal genera sonidos "sordas". En el dominio del tiempo los sonidos "sonoros" se caracterizan por una fuerte periodicidad presente en la señal, con la frecuencia fundamental conocida como la frecuencia ***pitch***. Para los hombres, el pitch se encuentra en el rango de 50 a 250 Hz mientras que para mujeres el pitch abarca las frecuencias entre 120 y 500 Hz. Sonidos "sordos", por otro lado, no presentan ningún tipo de periodicidad y son esencialmente de naturaleza aleatoria (4, 25).

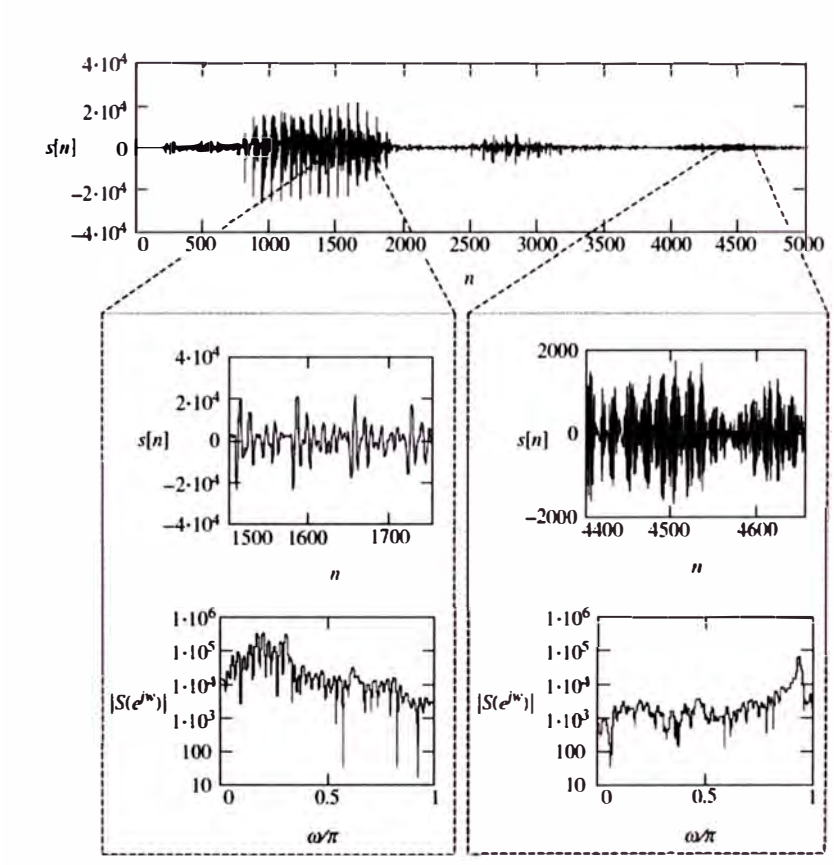


Fig. 2.3 Muestras de segmento de señales "sonoras" y "sordas" y sus espectros (4)

2.2.1 Estructura General de un Codificador de Voz

La Fig. 2.4 muestra un diagrama de bloques genérico de un codificador de voz de propósito general.

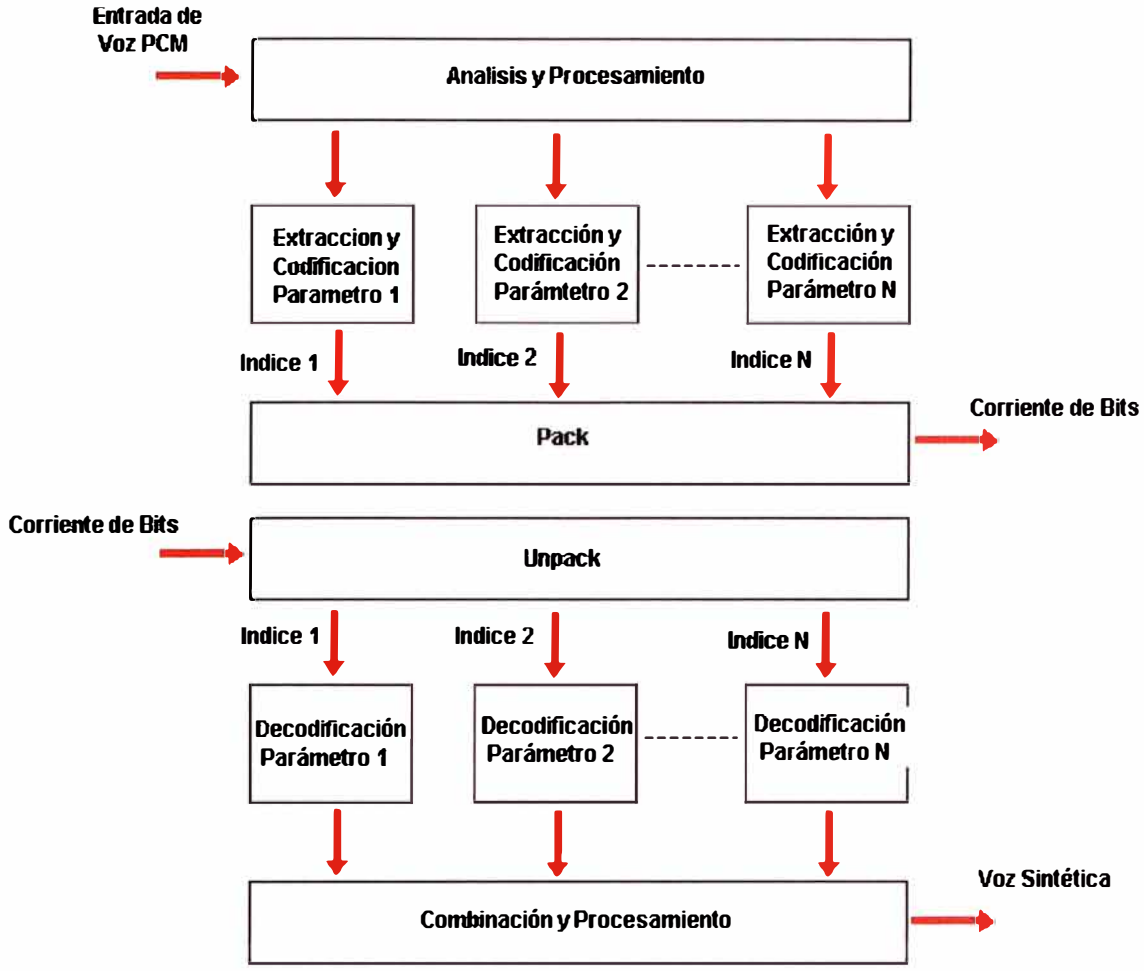


Fig. 2.4 Estructura General de un Codificador de Voz. (4)

2.3 Propiedades del sistema de audición humano

Para desarrollar un modelo eficiente de codificación de voz, es muy importante tener en cuenta las propiedades del sistema de audición humano para de esta forma aprovechar sus características en el modelo a desarrollar. (4)

La Fig. 2.5. muestra el diagrama del sistema de audición humano.

2.3.1 Umbral Absoluto

El umbral absoluto de un sonido es el nivel mínimo detectable de ese sonido en la ausencia de otros sonidos externos. De esta forma, el umbral absoluto caracteriza la cantidad de energía necesaria en un tono puro de tal forma que pueda ser detectado por un oyente en un ambiente sin ruido. La Fig. 2.6 muestra una curva típica de umbral absoluto.

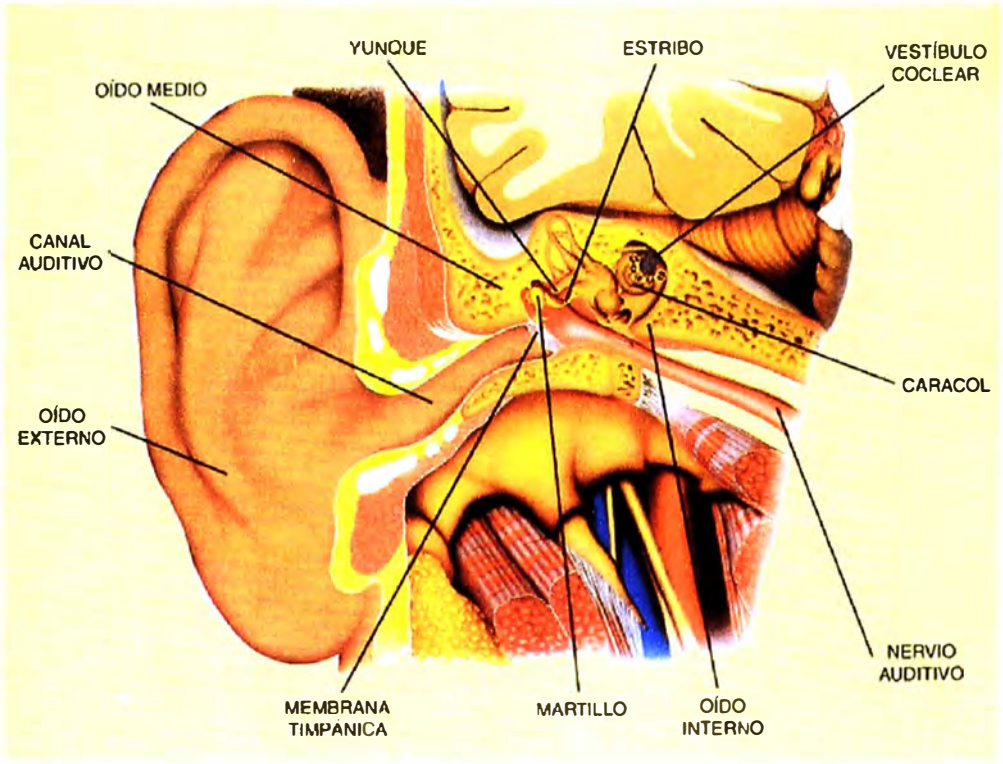


Fig. 2.5 Estructura del Sistema de Audición Humano (45)

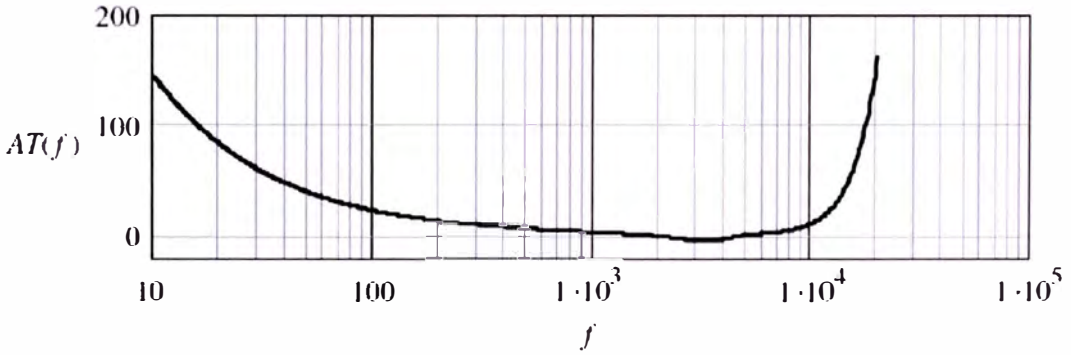


Fig. 2.6 Curva típica de umbral absoluto (4)

Algunas características del umbral absoluto se aplican en el desarrollo de codificadores. Se pueden mencionar las siguientes (4).

- Cualquier señal con una intensidad por debajo del umbral absoluto no necesita ser tomada en cuenta ya que no tiene impacto en la calidad del codificador de voz.
- La mayor parte de los recursos deben ser empleados para la representación de las señales dentro del rango de frecuencias mas sensible para el oído humano, el cual en términos generales se encuentra en el intervalo 1 – 4KHz.

2.3.2 Enmascaramiento

El enmascaramiento se refiere al hecho de que un determinado sonido puede ser inaudible en presencia de otros sonidos. La presencia de un único tono, por ejemplo, puede enmascarar las señales vecinas, siendo la capacidad de enmascaramiento inversamente proporcional a la diferencia absoluta en frecuencia. La Fig. 2.7 muestra un ejemplo donde un único tono genera una curva de enmascaramiento que ocasiona que cualquier otra señal con intensidad menor a esta sea imperceptible. (4)

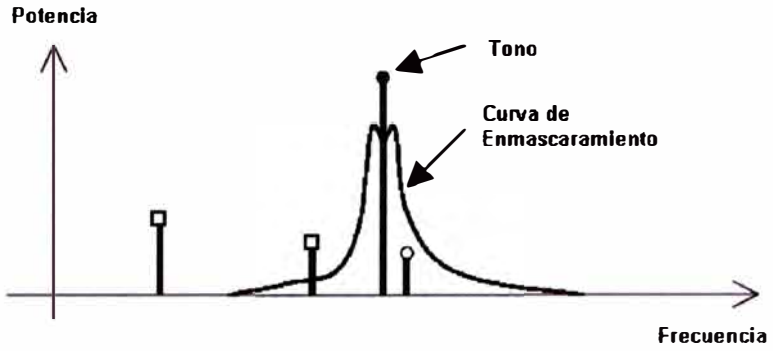


Fig. 2.7 Enmascaramiento debido a un tono. (4)

2.4 Estándares Actuales

A lo largo de los años ha habido muchos estándares en el área de codificación de la voz, para productos de consumo masivo principalmente (4).

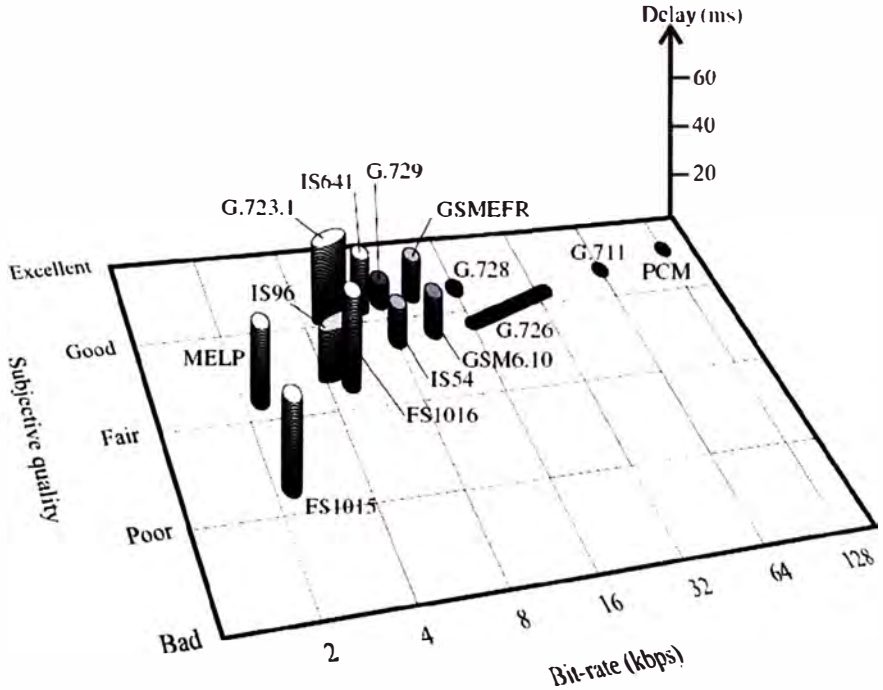


Fig. 2.8 Comparación de varios estándares por Bit-rate, Delay y Calidad (4]

CAPITULO III

TECNICAS DE PROCESAMIENTO DE LA VOZ

Se describen a continuación varios conceptos fundamentales y técnicas básicas y/o generales que se usarán a lo largo del presente trabajo de tesis. Cabe mencionar que existen muchas técnicas para procesar las señales de voz y aquí solo se tomarán en cuenta las importantes y que serán de utilidad en el presente trabajo.

3.1 Estimación de la Frecuencia Fundamental o Pitch

El pitch es un parámetro que identifica al locutor o hablante de tal forma que constituye uno de los parámetros más importantes en el análisis, síntesis y codificación de la voz El pitch es una característica única de cada persona. Los segmentos "sonoros" son generados cuando el flujo de aire proveniente de los pulmones es periódicamente interrumpido por los movimientos de las cuerdas vocales. El tiempo entre aperturas sucesivas de las cuerdas vocales es llamado periodo fundamental. Para los hombres el rango de la frecuencia fundamental se encuentra en el intervalo de 50 a 250 Hz mientras que para las mujeres el rango cae dentro del intervalo de 120 a 500 Hz. (4, 25,27)

3.1.1 Método de Autocorrelación

Es un método básico y directo para el cálculo del pitch, generalmente es usado para damos una primera aproximación al valor real porque tiene un margen de error considerable. El valor de la autocorrelación se calcula de acuerdo a la ecuación (3.1).

$$R[l,m] = \sum_{n=m-N+1}^{m} s[n]s[n-l] \qquad (3.1)$$

La autocorrelación refleja la similaridad entre el segmento *s[n], n = m-N+1 a m*, con respecto a la versión desplazada en el tiempo *s[n-l]*, donde *l* es un entero positivo que representa un lapso de tiempo. El rango de l es seleccionado de tal forma que cubra

un amplio rango de valores del periodo del pitch. Por ejemplo *l* = 20 a 147 (2.5 a 18.3 ms) y de esta forma las posibles frecuencias de pitch van de 54.5 Hz a 400 Hz (considerando un muestreo de 8KHz). Este rango de *l* es aplicable para la mayoría de hablantes y puede ser codificado con 7 bits. Finalmente se escoge el valor de *l* que corresponde al primer pico.

3.1.2 Función Magnitud de la Diferencia

También es usado para obtener una primera aproximación del valor real del pitch. La ventaja es que la función magnitud de la diferencia evita dichas multiplicaciones y es dada en la ecuación (3.2).

$$MDF[l,m] = \sum_{n=m-N+1}^{m} |s[n] - s[n-l]| \qquad (3.2)$$

En este el valor de *l* que minimiza la función MDF es la correspondiente a la estimación del pitch.

3.1.3 Pitch Fraccional

En aplicaciones reales y eficientes es necesario obtener el valor del pitch con la mayor precisión posible. Para lograr mas precisión en el calculo del pitch se han propuestos muchos métodos siendo los mas eficientes los basados en sistemas multitasa e interpolación. Uno de los métodos para hallar valores fraccionales del pitch es el Medan-Yair-Chazan. La técnica empleada por el presente trabajo de tesis se estudia en el capítulo 4, por ser de importancia.

3.2 Determinación de Segmentos "sonoros" y "sordos"

Por ejemplo, en el caso de los codificadores basados en el modelo Celp, no existe esta clasificación, lo que los hace más óptimos ya que no existe la restricción de clasificar un segmento como "sonoros" o "sordos". Sin embargo, en varios esquemas de codificación, sobretodo los basados en el esquema LPC 10 y sus derivados, la parte del algoritmo que determina si un determinado segmento es "sonoro" o "sordos" es crucial ya que es en base a esta determinación que el resto de estos algoritmos trabajan. Esto se revisa en el capítulo 4, ya que es de suma importancia para el presente trabajo.

3.3 Densidad Espectral de Potencia

La Densidad Espectral de Potencia (PSD en inglés) describe las características estadísticas de un proceso estocástico en el dominio de la frecuencia.

Para una señal determinística *x[n]*, la potencia promedio esta dada por la siguiente ecuación:

$$P = \lim_{N \to \infty} \frac{1}{2N+1} \sum_{n=-\infty}^{\infty} |x_N[n]|^2 = \frac{1}{2\pi} \int_{-\pi}^{\pi} \lim_{N \to \infty} \left(\frac{|X_N(e^{jw})|^2}{2N+1} \right) dw \tag{3.3}$$

Para un proceso estocástico x[n], la potencia promedio esta dada por la siguiente formula:

$$P = \frac{1}{2\pi} \int_{-\pi}^{\pi} \lim_{N \to \infty} \left(\frac{E\{|X_N(e^{jw})|^2\}}{2N+1} \right) dw \tag{3.4}$$

La Densidad Espectral de Potencia esta dada por:

$$S(e^{jw}) = \lim_{N \to \infty} \left(\frac{E\{|X_N(e^{jw})|^2\}}{2N+1} \right) \tag{3.5}$$

Para un proceso estocástico estacionario en el sentido amplio (WSS), se cumple la siguiente ecuación para la densidad espectral de potencia.(Nota: las señales de voz se modelan como de este tipo, es decir como WSS):

$$S(e^{jw}) = \sum_{l=-\infty}^{\infty} R[l] e^{-jwl} \qquad \text{y} \qquad R[l] = \frac{1}{2\pi} \int_{-\pi}^{\pi} S(e^{jw}) e^{jwl} dw \tag{3.6}$$

Para el ruido blanco: $S(e^{jw}) = \sigma^2$ $\qquad R[l] = \sigma^2 \delta[l]$ (3.7)

Varias relaciones útiles:

$$R_{YX}[l] = h[l] * R_X[l] \qquad R_{XY}[l] = h[-l] * R_X[l]$$
$$R_Y[l] = h[l] * R_{XY}[l] \qquad R_Y[l] = h[l] * h[l] * R_X[l] \tag{3.8}$$

3.3.1 Periodograma

Considerar una secuencia *x[n], n = 0,1,....N-1*. El periodograma $I_N(e^{jw})$ esta definido por:

$$I_N(e^{jw}) = \frac{1}{N}\left|X_N(e^{jw})\right|^2 \tag{3.9}$$

Con

$$X_N(e^{jw}) = \sum_{n=0}^{N-1} w[n]x[n]e^{-jwn} \tag{3.10}$$

La ventana w[n] es escogida adecuadamente para minimizar el derrame espectral (spectral leaking) debido al truncamiento de secuencias en secuencias finitas. Una ventana popular es la de Hamming.

La relación entre el periodograma y la autocorrelación de una señal esta dada por las siguientes ecuaciones:

$$I_N(e^{jw}) = \sum_{l=-(N-1)}^{N-1} R[l]e^{-jwl} \quad \text{y} \quad R[l] = \frac{1}{N}\sum_{m=0}^{N-1} w[m+l]w[m]x[m+l]x[m] \tag{3.11}$$

Tener en cuenta que el periodograma es una estimación de la PSD usando un número finito de muestras de la señal de entrada, siendo el estimado una función aproximada de la función real. En la práctica debido a que solo se cuentan con secuencias finitas, el periodograma ha llegado a ser una herramienta de análisis muy útil e importante.

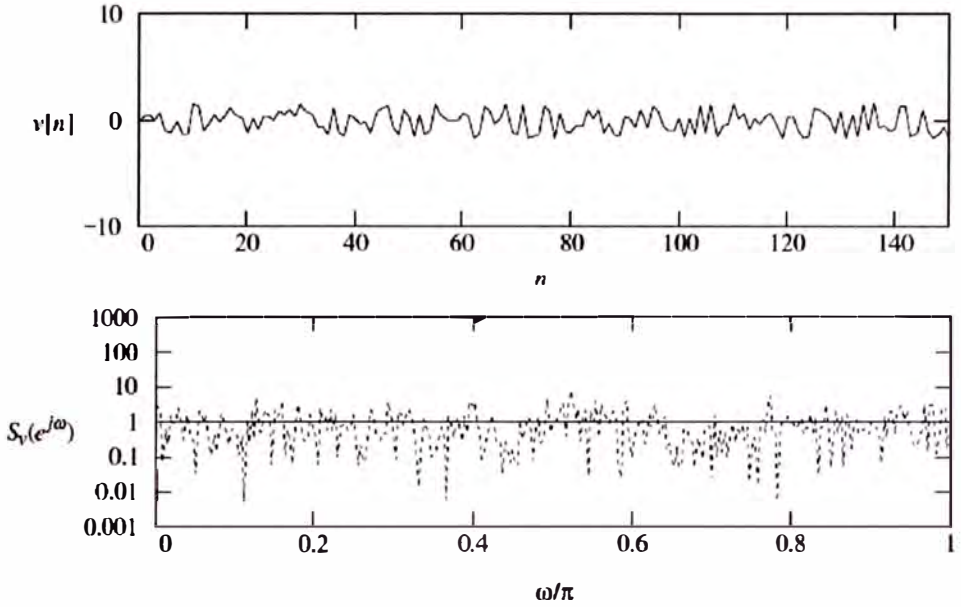


Fig. 3.1 Señal de entrada (arriba) y sus correspondientes (abajo) PSD (sólida) y Periodograma (puntcada). (4)

3.4 Modelo Autoregresivo

Una ecuación importante para entender las bases del modelo autoregresivo, la cual se deriva de las ecuaciones en (3.8), es la siguiente:

$$S_Y(e^{jw}) = \left|H(e^{jw})\right|^2 S_X(e^{jw}) \tag{3.12}$$

Ya que la densidad espectral de potencia de la salida del filtro esta dada por el espectro del ruido blanco (constante) multiplicado por la magnitud al cuadrado de la respuesta del filtro, se pueden producir señales aleatorias con una deseada característica espectral mediante la adecuada selección del polinomio en el denominador de la función filtro. (4, 22, 25)

Los valores de la secuencia *x[n], x[n-1], ... , x[n-M]* representan la realización de un proceso autoregresivo (AR) de orden M si este satisface la ecuación:

$$x[n] = -a_1 x[n-1] - a_2 x[n-2] - \ldots - a_M x[n-M] + v[n] \tag{3.13}$$

El valor presente del proceso *x[n]*, es igual a la combinación lineal de valores pasados del proceso mas una señal de error *v[n]*.

La función de transferencia de un analizador de un proceso AR es:

$$H_A(z) = \frac{V(z)}{X(Z)} = \sum_{l=0}^{M} a_l z^{-l} \tag{3.14}$$

La función de transferencia de un sintetizador de un proceso AR es:

$$H_S(z) = \frac{X(z)}{V(z)} = \frac{1}{H_A(z)} = \frac{1}{\sum_{l=0}^{M} a_l z^{-l}} \tag{3.15}$$

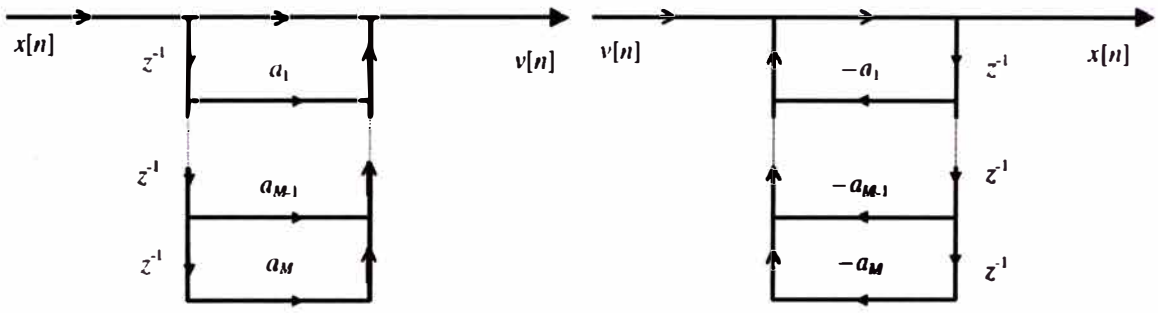


Fig. 3.2 Realizaciones de los filtros AR (analizador y sintetizador) (4)

3.4.1 Ecuación Normal o de Yule-Walker

Ya que *v[n]* representa ruido blanco, entonces no esta correlacionado con *x[n-l] para l ≥ 1*, es decir:

$$E\{v[n]x[n-l]\} = 0 \tag{3.16}$$

Multiplicando ambos lados de (3.16) por v[n] y tomando esperanzas.

$$E\{v[n]x[n]\} = \sigma_v^2 \tag{3.17}$$

De la última ecuación se tiene que la correlación cruzada entre *x[n]* y *v[n]* esta dada por la varianza de *v[n]*. Multiplicando otra vez ambos lados de (3.15) por *x[n-l], l=0,1…,M* y tomando la esperanza se obtiene la ecuación de Yule Walker.

3.4.2 Estimación de la Autocorrelación

Como se sabe, el periodograma es un estimado de la PSD. También se sabe que la función de autocorrelación y la PSD forman un par de transformada de Fourier (ver ecuaciones (3.6)). Basado en este hecho, la autocorrelación puede ser estimada primeramente de la señal, entonces se calcula la transformada de Fourier para la estimación del espectro. Como se verá luego, la función de autocorrelación juega un papel importante en el análisis de predicción lineal, el cual es un procedimiento para calcular los coeficientes de predicción lineal del modelo.

Por ello, es importante escoger un método adecuado que nos permita estimar de forma óptima el valor de la autocorrelación, el cual debe de ser estimado para cada segmento de voz. En general los métodos para la estimación de la autocorrelación pueden dividirse en recursivos y no recursivos, dependiendo del tipo de ventana que se use para la extracción de las muestras finitas.

Para una ventana de tamaño N, tenemos que la autocorrelación puede ser estimada de la siguiente ecuación:

$$R[l,m] = \frac{1}{N} \sum_{n=m-N+1+|l|}^{m} x[n]w[m-n]x[n-|l|]w[m-n+|l|] \tag{3.18}$$

En el presente trabajo se usa la ecuación (3.18) haciendo uso de una ventana de hamming:

$$w[n] = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right), \qquad 0 \le n \le N-1 \tag{3.19}$$

3.5 Predicción lineal y técnicas de predicción lineal

La predicción lineal (LP) constituye la base de casi todos los modernos codificadores de voz de hoy en día. Básicamente consiste en que las propiedades estadísticas de un segmento actual pueden ser modelados basados en las propiedades de un segmento anterior. Se basa en el modelo autoregresivo (AR), de hecho el análisis de predicción lineal es un procedimiento de estimación para encontrar los parámetros del modelo AR, dadas las muestras de la señal. (4, 22, 25)

3.5.1 Predicción Lineal

Se ilustra en la Fig. 3.3. La señal de ruido blanco *x[n]* es filtrada por el proceso sintetizador AR para obtener la señal *s[n]* (la señal AR) con los parámetros AR denotados por $\hat{a}_i$.

Un predictor lineal es usado para predecir s[n] basado en las M muestras pasadas mediante la siguiente ecuación:

$$\hat{s}[n] = -\sum_{i=1}^{M} a_i s[n-i] \tag{3.20}$$

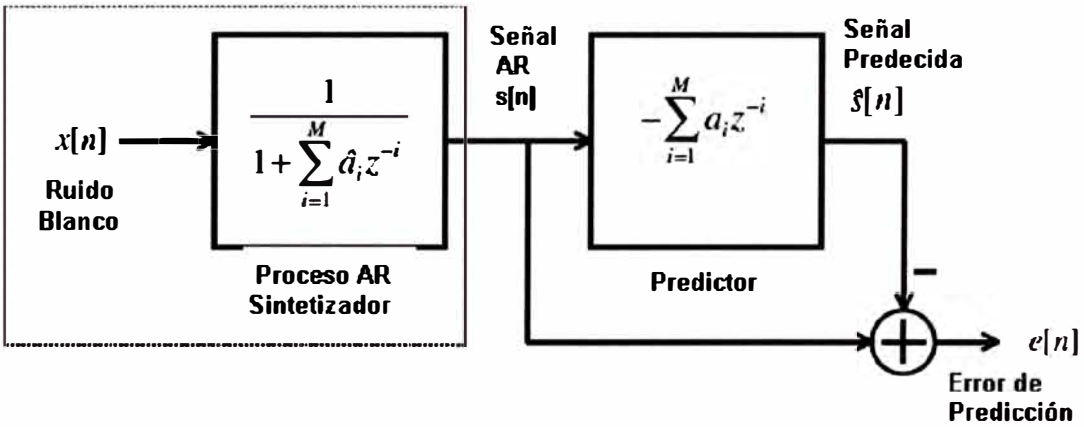


Fig. 3.3 Esquema de análisis de la Predicción Lineal (4)

Donde los $\hat{a}_i$ son los estimados de los parámetros AR y son referidos como los coeficientes LPC. La constante M es conocida como el orden del predictor.
El error de predicción es:

$$e[n] = s[n] - \hat{s}[n] \tag{3.21}$$

La Fig. 3.4 muestra la implementación y es conocido como el filtro de error-predicción. Este filtro toma la señal AR como entrada y produce la señal de error de predicción y el error como sus salidas.

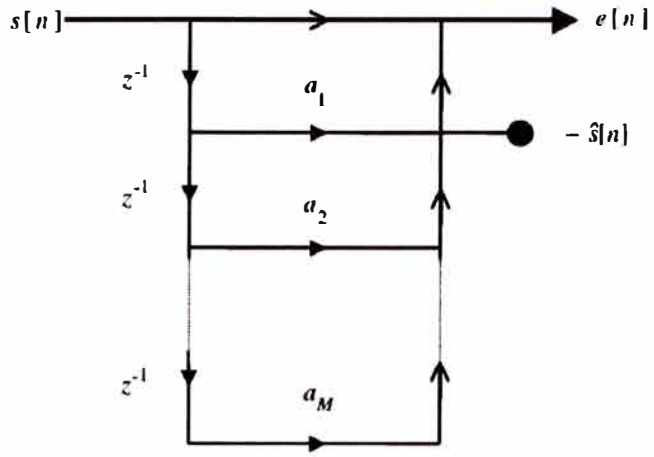


Fig. 3.4 Implementación del filtro de error-predicción (4)

3.5.2 Minimización del Error

Consiste en la estimación de los parámetros AR $\hat{a}_i$ de *s[n]*, siendo estos los coeficientes LPC. Para ejecutar la estimación, se debe establecer un criterio, en el presente caso es el error de predicción medio-cuadrático dada por la ecuación (3.22).

$$J = E\{e^2[n]\} = E\left\{\left(s[n] + \sum_{i=1}^{M} a_i s[n-i]\right)^2\right\} \tag{3.22}$$

Para minimizar la función J se deriva con respecto a los coeficientes predictivos.

$$\frac{\partial J}{\partial a_k} = 2E\left\{\left(s[n] + \sum_{i=1}^{M} a_i s[n-i]\right) s[n-k]\right\} = 0 \tag{3.23}$$

La solución de la ecuación (3.23) se da cuando los coeficientes predictivos resultantes son iguales a los correspondientes coeficientes del proceso AR $\hat{a}_i$ (ver Fig. 3.4).

3.5.3 Ecuación Normal

De la ecuación (3.23) se deriva la siguiente ecuación (3.24):

$$\sum_{i=1}^{M} a_i R_s[i-k] = -R_s[k] \tag{3.24}$$

En forma matricial: $\mathbf{R_s a = -r_s}$ (3.25)

La ganancia predictiva de un predictor esta dada por la siguiente ecuación:

$$PG = 10\log_{10}\left(\frac{\sigma_s^2}{\sigma_e^2}\right) = 10\log_{10}\left(\frac{E\{s^2[n]\}}{E\{e^2[n]\}}\right) \quad (3.26)$$

Un predictor óptimo es capaz de reducir la ganancia del error conduciendo a más altas ganancias predictivas.

3.5.4 Mínimo Error Predictivo Cuadrático Medio

Cuando los coeficientes LPC del filtro predictivo igualan a los coeficientes del proceso AR, entonces se tiene que el error de predicción es el mismo que el ruido blanco usado para generar la señal en el proceso AR. En esta situación óptima se cumple:

$$J\min = E\{e^2[n]\} = E\{x^2[n]\} = \sigma_X^2 \quad (3.27)$$

El mínimo error predictivo basado en los coeficientes de autocorrelación se da en (3.28).

$$J\min = \sigma_X^2 = R_s[0] + \sum_{i=1}^{M} a_i R_s[i] \quad (3.28)$$

Combinando la ecuación (3.27) con la (3.28) obtenemos la ecuación normal aumentada (3.29)

$$\begin{bmatrix} R_s[0] & r_s^T \\ r_s & R_s \end{bmatrix} \begin{bmatrix} 1 \\ a \end{bmatrix} = \begin{bmatrix} J\min \\ 0 \end{bmatrix} \quad (3.29)$$

3.6 Esquemas de análisis predictivo

El análisis Predictivo puede ser implementado de distinta manera dependiendo de los requerimientos del sistema en cuestión. Existen dos técnicas principales para la

codificación de la voz: predicción interna y predicción externa. Estos esquemas son ilustrados en la Fig. 3.5.

La razón por la cual la predicción externa puede ser usada es debido a que las propiedades estadísticas de la señal de voz cambian lentamente con el tiempo. Si el segmento no es excesivamente largo, sus propiedades pueden ser derivadas de segmentos anteriores.

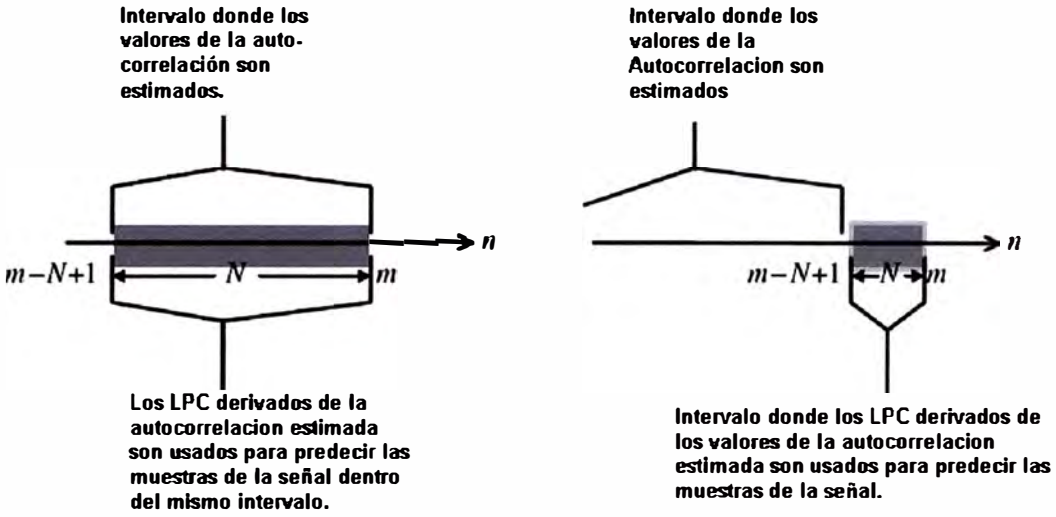


Fig. 3.5 Ilustración de la predicción interna (izquierda) y predicción externa (derecha) (4)

Varios algoritmos de codificación de voz usan predicción interna, donde los LPC's de un segmento en particular son derivados de la data perteneciente a ese segmento. De esta forma, los LPC's resultantes capturan las propiedades estadísticas de ese segmento. Las longitudes típicas de los segmentos de análisis varían de 160 a 240 muestras.

La predicción externa es usada principalmente en algoritmos donde el retardo del codificador es el principal problema. En este caso se puede usar una trama más corta (del orden de 20 muestras, por ejemplo en el estándar LD-CELP). En estos casos se usan técnicas recursivas para el calculo de la estimación de la autocorrelación de tal forma que los LPC's son derivados de las muestras comprendidas antes del instante $n=m-N+1$. (4, 22, 25)

3.7 Ganancia de Predicción

La ganancia de predicción está dada por la siguiente expresión:

$$PG[m] = 10\log_{10}\left(\frac{\sum_{n=m-N+1}^{m} s^2[n]}{\sum_{n=m-N+1}^{m} e^2[n]}\right) \tag{3.30}$$

Donde:

$$e[n] = s[n] - \hat{s}[n] = s[n] + \sum_{i=1}^{M} a_i[m]s[n-i], \quad n=m-M+1,\ldots..,m$$

Del esquema de predicción lineal se derivan observaciones muy importantes las cuales se resumen a continuación (4):

- *Para un orden de predicción dado, la ganancia predictiva promedio es mayor para segmentos "sonoros" que para segmentos "sordos".*

- *Para un segmento "sonoro", la ganancia de predicción asociada con un predictor que tiene un orden lo suficientemente grande como para cubrir un periodo de pitch es substancialmente más grande que la ganancia de predicción asociada a un predictor con un orden más pequeño que un periodo de pitch.*

- *Para remover la correlación entre muestras usando un predictor lineal, se requiere un orden de predicción mayor para segmentos "sonoros" que para segmentos "sordos". Para "blanquear" eficientemente un segmento "sonoro" se requiere que el orden del predictor sea mayor o igual al periodo pitch de dicho segmento de voz.* (4)

3.8 Algoritmo de Levinson-Durbin

La ecuación normal dada en (3.40) puede ser resuelta encontrando la inversa de la matriz ***Rs***. En general, la inversión de una matriz es computacionalmente exigente. Afortunadamente, la ecuación matricial puede ser resuelta por algoritmos eficientes

pasando por alto la inversión de dicha matriz. Dichos algoritmos aprovechan la estructura especial de la matriz de correlación.

El algoritmo de Levinson-Durbin se vale de dos propiedades claves de la matriz de autocorrelación:

- La matriz de correlación de un tamaño determinado contiene como subbloques todas las matrices de órdenes más bajos.
- La matriz de correlación es invariante a intercambios entre sus columnas o filas.

Las propiedades mencionadas corresponden a las matrices Toeplitz.

Entonces, basados en la explotación de la matriz Toeplitz, el algoritmo de Levinson – Durbin nos da como salidas los coeficientes LPC y RC. (4, 22, 25)

- Inicialización: $l = 0$,

$$J_0 = R[0]$$

- Recursión para $l = 1,2,3,\ldots,M$

Paso 1. Calcular el *l-esimo* RC.

$$k_l = \frac{1}{J_{l-1}}\left(R[l] + \sum_{i=1}^{l-1} a_i^{(l-1)} R[l-i]\right) \tag{3.31}$$

Paso 2. Calcular los LPC's para el predictor de orden *l*.

$$a_l^{(l)} = -k_l$$

$$a_i^{(l)} = a_i^{(l-1)} - k_l a_{l-i}^{(l-1)}, \quad i=1,2,\ldots,l-1 \tag{3.32}$$

Detener si $l = M$.

Paso 3. Calcular el **mínimo** error de predicción medio cuadrático asociado con el predictor de orden *l*.

$$J_l = J_{l-1}\left(1 - k_l^2\right) \tag{3.33}$$

l = l +1 y regresar al paso 1.

- El conjunto final de LPC's esta dado por

$$a_i = a_i^{(M)}, \qquad i=1,2,\ldots,M \tag{3.34}$$

Observar que en el proceso de solución, el conjunto de coeficientes RC (Reflection Coefficients) también es encontrado.

Características del Algoritmo de Levinson – Durbin:

- Es computacionalmente eficiente.
- Los coeficientes RC pueden ser usados para evaluar la estabilidad del sistema (basados en la propiedad de fase mínima).

El filtro de error de predicción con función de transferencia

$$A(z) = 1 + \sum_{i=1}^{M} a_i z^{-i} \tag{3.35}$$

Es un sistema de fase mínima si se cumple: $|k_i| < 1$, $i=1,2,\ldots,M$

3.8.1 Conversión de los Coeficientes de Reflexión a Coeficientes LPC's

Como se mencionó anteriormente, los coeficientes RC representan una forma alternativa de los coeficientes LPC. De hecho hay una correspondencia uno-a-uno entre ellos. Los coeficientes RC's poseen varias características deseables, haciéndolos

preferibles en varias situaciones prácticas. Las ecuaciones para convertir los coeficientes RC a coeficientes LPC son:

$$a_l^{(l)} = -k_l$$

$$a_i^{(l)} = a_i^{(l-1)} - k_l a_{l-i}^{l-1}, \quad i = 1,2,\ldots,l-1 \tag{3.36}$$

3.8.2 Conversión de los coeficientes LPC a coeficientes RC

Dado el conjunto de coeficientes LPC ai, i = 1,2,....,M, se desea encontrar los correspondientes coeficientes RC *ki*. Las ecuaciones que definen la conversión se da en (3.37).

$$kl = -a_l^{(l)} \tag{3.37}$$

$$a_i^{(l-1)} = \frac{a_i^{(l)} + k_l a_{l-i}^{(l)}}{1 - k_l^2} \quad i=1,2,\ldots,l-1 \tag{3.38}$$

3.9 Predicción Lineal Long-Term

Cuando se resuelve la ecuación para el modelo autoregresivo se encuentra que para modelar señales "sonoras" se necesita un orden alto para el filtro predictivo, lo cual es desventajoso. El problema radica en que ordenes altos de predictores resultan en excesivas tasas de bits de la voz codificada, lo cual no es deseable. Para superar el problema mencionado la literatura plantea el esquema de predicción long-term. Dicho esquema se muestra en la Fig. 3.6. El filtro de error de predicción long-term con entrada $e_s[n]$ y salida $e[n]$ tiene la siguiente función de transferencia (4, 22, 25):

$$H(z) = 1 + bz^{-T} \tag{3.39}$$

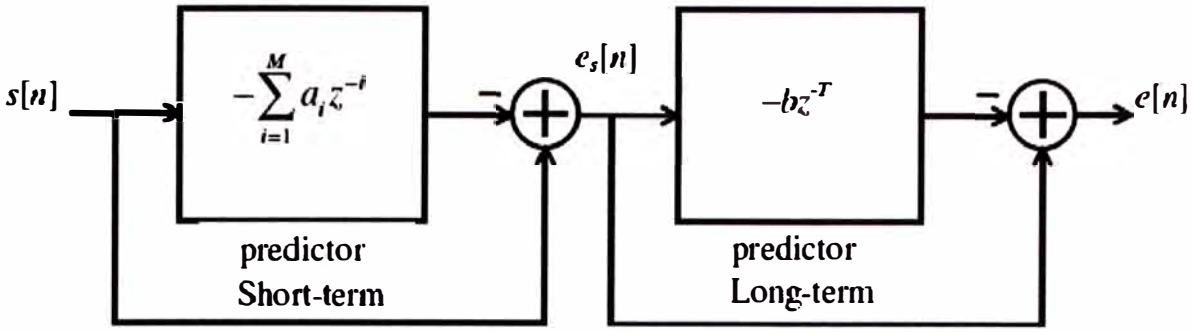


Fig. 3.6 Esquema de la predicción Long – Term (4)

El predictor long-term requiere el cálculo de dos nuevos coeficientes: el periodo del pitch T y la ganancia long-term b.

$$\hat{e}_S[n] = -\, b e_S[n-T] \tag{3.40}$$

Para encontrar b y T se recurre a la minimización del error cuadrático:

$$J = \sum_n \left(e_S[n] - \hat{e}_S[n] \right)^2 = \sum_n \left(e_S[n] + b e_S[n-T] \right)^2 \tag{3.41}$$

Diferenciando la ecuación (3.41) con respecto a b resulta en:

$$b = -\frac{\sum_n e_S[n] e_S[n-T]}{\sum_n e_S^2[n-T]} \tag{3.42}$$

Reemplazando (3.42) en (3.41), obtenemos el valor de J a ser minimizado:

$$J = \sum_n e_S^2[n] - \frac{\left(\sum_n e_S[n] e_S[n-T] \right)}{\sum_n e_S^2[n-T]} \tag{3.43}$$

Se hace un barrido de los valores de T en el rango de 20 a 147 y se escoge aquel que minimiza (3.43).

La efectividad del predictor long-term mejora cuando se usa un esquema de análisis basado en sub-segmentos. Dicho esquema es usado por varios codificadores entre ellos los basados en el estándar FS1016 (CELP). La gráfica se muestra en la Fig. 3.7.

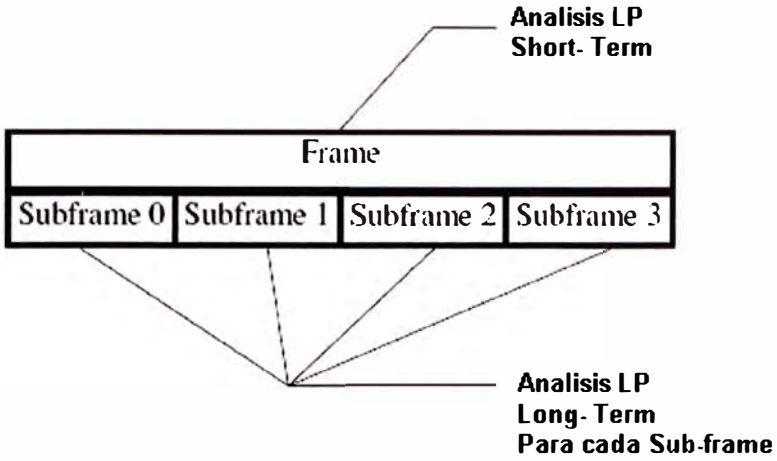


Fig. 3.7 Esquema de análisis LP basado en Sub-segmentos (4)

3.10 Filtros de Síntesis

Si el proceso de predicción es eficiente en conseguir que la señal de error tenga un alto grado de "blanqueamiento" entonces para recuperar la señal en el sintetizador solamente hace falta filtrar una señal generada por un generador de ruido blanco aleatorio a través de un filtro con función de transferencia (ver Fig. 3.8):

$$H(z) = \frac{1}{1 + \sum_{i=1}^{M} a_i z^{-i}} \tag{3.44}$$

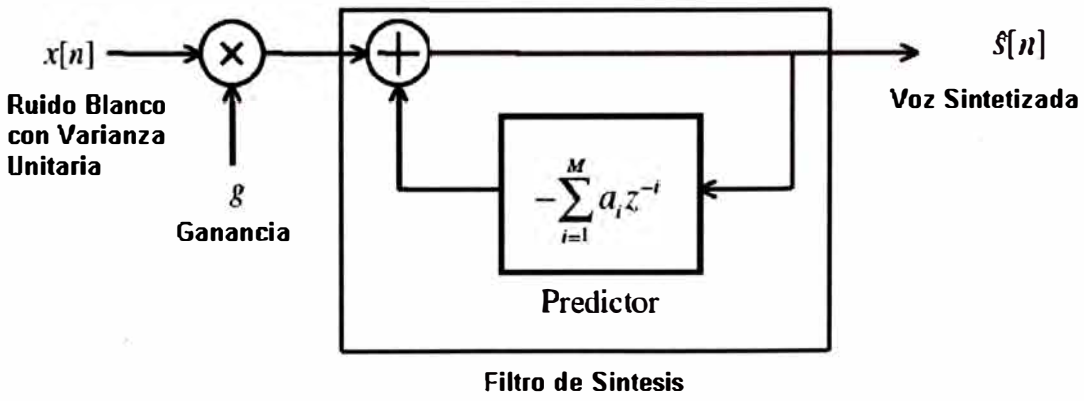


Fig. 3.8 Filtro de síntesis (4)

La ganancia se calcula mediante la siguiente ecuación.

$$g = \gamma \sqrt{R_s[0] + \sum_{i=1}^{M} a_i R_s[i]} \tag{3.45}$$

En el caso de predicción lineal long-term, el respectivo filtro de síntesis se muestra en la Fig. 3.9 (usado en el esquema CELP).

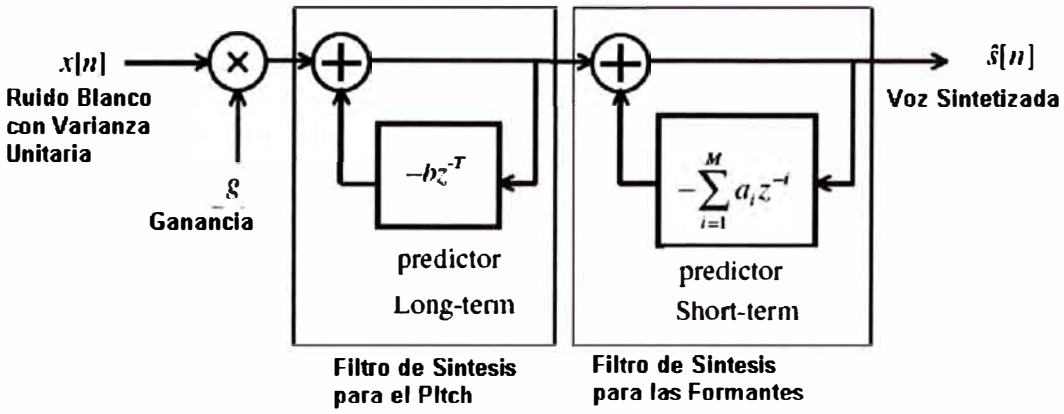


Fig. 3.9 Filtro de Síntesis para el caso de Predicción Long-Term (4)

3.11 Cuantización escalar de los coeficientes predictivos.

Antes de transmitir los parámetros que representan la voz decodificada, estos deben de ser cuantizados de manera apropiada para garantizar que el decodificador pueda recuperar los valores con máxima exactitud y cumplir los requerimientos sobretodo de estabilidad de los filtros de síntesis. Existen varios esquemas de cuantización. El que será usado en el presente trabajo de tesis es el esquema de cuantización de los coeficientes LPC a 8 bits. Cabe señalar que todos los esquemas de cuantización buscan minimizar la función de distorsión espectral.

3.11.1 Distorsión Espectral

Una manera de medir el grado de calidad de un esquema de cuantización es mediante la distorsión espectral debido a la cuantización. La función de sensitividad espectral es dada con el propósito de medir la magnitud del cambio del espectro recuperado debido a la variación de uno de los parámetros que definen dicho espectro.

La distorsión espectral entre dos PSD se da por la siguiente ecuación:

$$SD^2 = \frac{1}{n1 - n0} \sum_{n=n0}^{n1-1} \left[10\log_{10}\left(S_1\left(e^{j2\pi n/N}\right)\right) - 10\log_{10}\left(S_2\left(e^{j2\pi n/N}\right)\right) \right]^2 \qquad (3.46)$$

3.11.2 Función de Sensitividad Espectral

Esta función nos permite evaluar el impacto en la función de densidad espectral debido a variaciones en alguno de los parámetros cuantizados (LPC's, RC's o algún otro parámetro). La ecuación de la sensitividad espectral se da en (3.47).

$$\psi(x_0) = \left| \lim_{\Delta x \to 0} \frac{SD(x_0, x_0 + \Delta x)}{\Delta x} \right| \qquad (3.47)$$

Y la densidad espectral SD, está definida como sigue:

$$SD^2(x_0, x_0 + \Delta x) = \frac{1}{2\pi} \int_0^{2\pi} \left(10\log 10\left(S\left(e^{jw}, x_0\right)\right) - 10\log 10\left(S\left(e^{jw}, x_0 + \Delta x\right)\right)\right)^2 d\omega \qquad (3.48)$$

CAPITULO IV

ALGORITMO SIFT PARA EL CALCULO OPTIMO DE LA FRECUENCIA FUNDAMENTAL DE LA VOZ

En este capítulo se describe el algoritmo SIFT (*Simplified Inverse Filter Tracking*) el cual constituye una parte fundamental del bloque codificador en el presente trabajo. Al final se muestran algunos resultados.

4.1 Algoritmo SIFT

El algoritmo SIFT es una técnica de análisis simplificado basado en el esquema de filtro inverso, el cual retiene las ventajas de los métodos de autocorrelación y del análisis cepstral. El diagrama del algoritmo SIFT se muestra en la Fig. 4.1. (27)

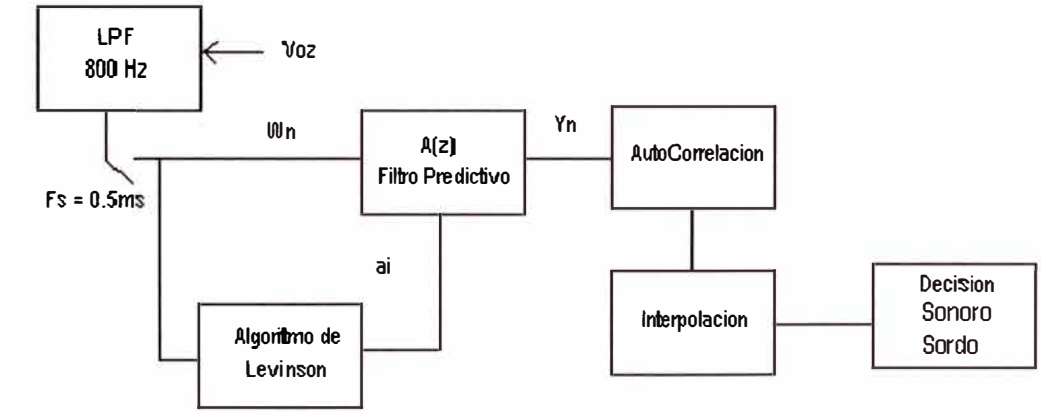


Fig. 4.1 Diagrama de Bloques del Algoritmo SIFT (27)

La voz entrante S_n (muestreada a 8.0 KHz) se filtra primero por un filtro pasabajo que tiene una frecuencia de corte de 0.8 KHz. Luego se muestrea a una tasa de 2 KHz (diezmado por un factor de 4) obteniéndose la señal diezmada W_n. A continuación se calculan los 5 primeros términos de la secuencia de autocorrelación en periodo corto (short-term) para cada ventana de 32ms de tamaño. Luego se ejecuta el algoritmo de Levinson para calcular los 4 primeros coeficientes predictivos { k_i }. Con dichos

coeficientes se procede a armar el filtro predictivo ("blanqueador") de orden 4 a través del cual pasará la señal diezmada. La salida Y_n del filtro predictivo es una señal de error. Se calcula la autocorrelación de esta secuencia de error y se ubica el valor pico y su índice correspondiente. A continuación se interpolan los valores alrededor del valor pico obtenido para obtener mejor resolución. Finalmente, basado en el pico encontrado, se hace una decisión para clasificar el segmento como "sonoro" o "sordo".

4.2 Prefiltrado (LPF de 800 Hz)

El filtrado es obligatorio antes de realizar un diezmado de la señal ya que evita que los espectros se solapen uno con el otro (aliasing). En este caso se usó un filtro IIR Chebyshev Tipo 1 de orden 5 (5 polos y 5 ceros), los coeficientes empleados fueron los siguientes(27):

num_ch=[0.001314939697 0.005259758789 0.007889638184 0.005259758789 0.001314939697];

den_ch =[1 -3.191026659642 4.149363076516 -2.570303903189 0.638454062317];

La respuesta en frecuencia del filtro se muestra en la Fig. 4.2.

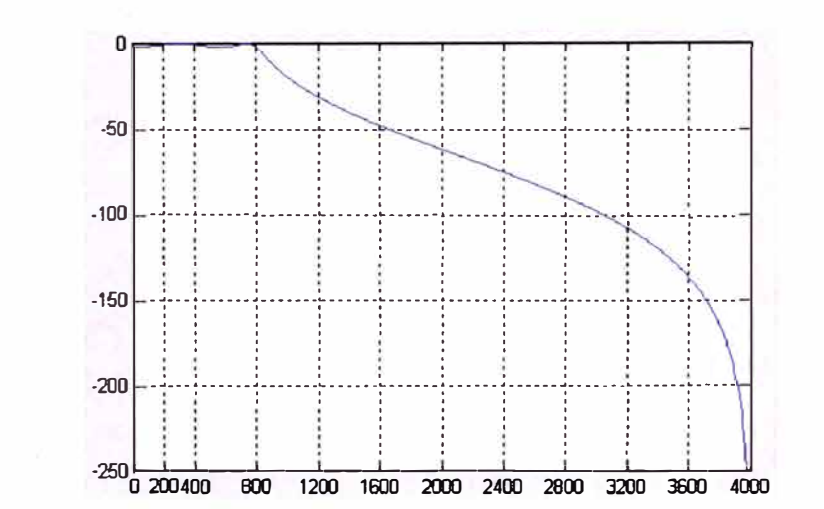


Fig. 4.2 Respuesta en Frecuencia del Filtro Chebyshev de Orden 5 empleado.

Se opta por el uso de un filtro IIR Chebyshev debido a que:

- El algoritmo SIFT no exige que el filtro sea altamente selectivo en frecuencia.

- Un filtro selectivo del tipo FIR por ejemplo requeriría más de 50 coeficientes.
- Un filtro Chebyshev de orden 5 se ejecuta mucho más rápido que filtros de órdenes más altos.
- La no linealidad de la fase no es crítico en los resultados finales.

4.3 Diezmado por un factor de 4

El algoritmo SIFT muestra que el cálculo preciso de la frecuencia fundamental es incluso posible con la tasa de muestreo de 2 KHz (8KHz / 4). El diezmado de 4 ocasiona que el número total de operaciones necesarias se reduzca grandemente.

En una de las implementaciones, el diezmado se realizó seleccionando, de la señal filtrada, las muestras espaciadas por 4 índices.

4.4 Filtro Inverso y Algoritmo de Levinson

De la señal diezmada calculamos los coeficientes predictivos {ki} y armamos el filtro inverso el cual hará un "blanqueamiento" de la señal diezmada. Un elemento crítico del algoritmo SIFT es la determinación del número M óptimo de coeficientes predictivos a ser calcula dos, ya que en este caso no se trata de un problema de estimación espectral sino de eliminar ciertas características de la señal diezmada y preservar otras, las cuales nos ayudarán a determinar el pitch con precisión. Si M es demasiado pequeño, se obtiene un muy pobre estimación de la estructura de resonancia dentro del rango (0, Fs/2). Si M es demasiado grande, entonces se obtiene la estructura de resonancia más detalles finos debidos a los múltiplos del pitch. Lo que se desea es obtener una estimación cercana de la estructura de resonancia e ignorar las características debido a los múltiplos del pitch. El algoritmo SIFT usa un valor de M=4.

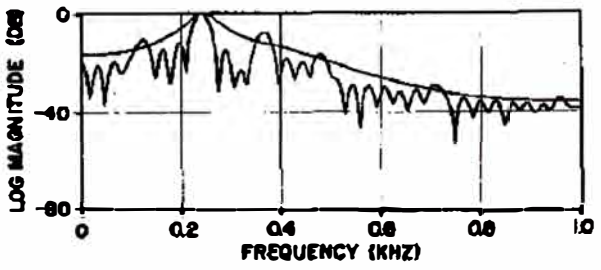


Fig. 4.3 Espectro de la entrada al filtro inverso y espectro del recíproco del filtro inverso(27)

Como se muestra en la Fig. 4.3, con un valor de M =4 se puede ver claramente el pico del primer formante en el primer espectro (el que es uniforme), también se puede ver en el otro espectro (el no uniforme) picos a múltiplos de 120 Hz para este caso.

Se puede decir también que, a la señal de salida del filtro inverso se le ha removido la componente de mayor resonancia dejando solo la información de la frecuencia fundamental superimpuesta a una constante. Es por eso que el filtro inverso puede ser considerado como uno de "pre-blanqueamiento" ya que no aplana por completo el espectro de la señal entrante, solo elimina las características debidas a las resonancias del tracto vocal pero reteniendo la estructura fina debido a pulsos glotales.

4.5 Autocorrelación de la señal de salida del Filtro Inverso

En esta etapa se calcula la autocorrelación de la señal de salida *Yn* del filtro inverso. Y se arma una secuencia r_n nueva de autocorrelación de la siguiente manera (27):

$$r_n = \begin{cases} \sum_{j=0}^{N-1-n} y_i y_{i+n} & n=0,1,\ldots,M/2-1 \\ 0 & n=M/2 \\ r_{M-n} & n=M/2+1,M/2+2,\ldots,M-1 \end{cases} \qquad (4.1)$$

Donde:

r_n: nueva secuencia de autocorrelación
Yi: secuencia de salida del filtro inverso
N: tamaño de la ventana (64)
M: tamaño de la secuencia r_n

De esta secuencia se consigue el primer estimado n = N', del valor de la frecuencia fundamental.

Los cálculos precisos de la frecuencia fundamental requieren una resolución de 0.1 a 0.15 ms en la escala de tiempos. Por ejemplo para un muestreo de T = 0.125 ms,

asumiendo que el pitch real sea de 6 ms, nos resultaría en un error de 1.74 Hz en el calculo. Mientras que en nuestro caso la frecuencia de muestreo ha sido diezmada y T = 0.5 ms lo cual nos arroja un error de 7 Hz. Ante esto es necesaria emplear alguna técnica de interpolación.

4.6 Interpolación de la secuencia r_n

El algoritmo SIFT emplea un algoritmo de interpolación basado en dos transformadas discretas de Fourier (FFT para mas comodidad), las cuales se simplifican en una matriz de 18 valores.(27)

- La señal diezmada es de tamaño N=64.
- La señal r_n es de tamaño $M = 2N = 128$.

Primero se calcula la transformada discreta de Fourier de la secuencia r_n esta secuencia R_k es de tamaño M. Dicha secuencia se convierte en una nueva secuencia R'_k de tamaño $M' = 4\ M = 512$. A esta última secuencia M' de tamaño 512 se le aplica la transformada inversa de Fourier para obtener la secuencia r'_n que viene a ser la interpolación de la secuencia original r_n.

Como se puede apreciar, realizar esta interpolación involucra realizar dos FFT de secuencias de tamaños considerables. Esto involucra un gran consumo de tiempo de ejecución.

Pero, el algoritmo SIFT realiza una simplificación mediante manipulaciones matemáticas para obtener solamente los valores que nos son útiles. Es decir, calcula solamente los valores interpolados próximos al valor estimado n = N' del pitch.

La ecuación simplificada luego de las manipulaciones matemáticas es la siguiente (27):

$$r_a = \frac{1}{M} \sum_{l=0}^{M/2-1} u_l \left[\frac{\sin \alpha (M-1)/2}{\sin \alpha / 2} + \frac{\sin \beta (M-1)/2}{\sin \beta / 2} \right] \qquad (4.2)$$

Donde:

$$\alpha = \frac{2\pi}{M}(l+a)$$

$$\beta = \frac{2\pi}{M}(l-a)$$

$$a = \frac{nM}{M'} \qquad n = 0,\ 1,\ \ldots,\ M' - 1$$

$$ul = \begin{cases} \frac{1}{2}r_0, l = 0 \\ r_l, l = 1,2,\ldots,M/2-1 \end{cases}$$

En nuestro caso $M / M' = 4$.

Como nuestro factor de interpolación es 4, vamos a tener en total 6 valores interpolados, 3 a la izquierda del valor estimado n = N' y 3 su derecha, tal como se muestra en la Fig. 4.4.

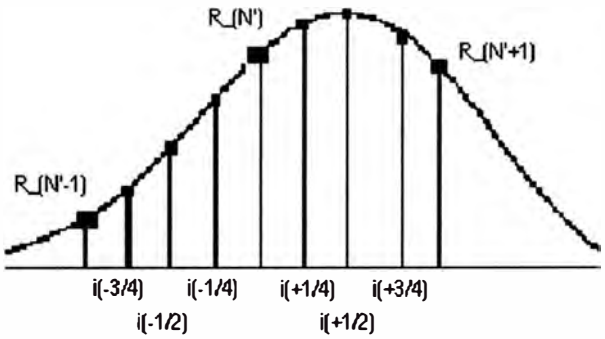


Fig. 4.4 Interpolación [basado en (27)]

A partir de la ecuación simplificada, el algoritmo SIFT emplea solamente dos vecinos para el cálculo de los 6 valores interpolados, es decir cada uno de los valores interpolados i(-3/4), ...i(+3/4) se derivan de los valores $R_{N'-1}$, $R_{N'}$ y $R_{N'+1}$.

Dicha relación se da en forma matricial (extendido de (27)):

$$\begin{bmatrix} i(-3/4) \\ i(-1/2) \\ i(-1/4) \\ i(1/4) \\ i(1/2) \\ i(3/4) \end{bmatrix} = \begin{bmatrix} 0,3169941646 & 0,8836639899 & -0,1636336748 \\ 0,6449978244 & 0,6285811759 & -0,2044876418 \\ 0,8952769119 & 0,3053885987 & -0,1341291693 \\ -0,1746897087 & 0,8947133641 & 0,3059247910 \\ -0,2201173475 & 0,6442108815 & 0,6293303746 \\ -0,1451785436 & 0,3164446326 & 0,8841874787 \end{bmatrix} \begin{bmatrix} R(N'-1) \\ R(N') \\ R(N'+1) \end{bmatrix} \qquad (4.3)$$

De esta forma obtenemos todos los valores interpolados.

Nuestra frecuencia fundamental resultante será:

$P = (N' + i) / 2$, donde i es el índice que corresponde al valor interpolado mas grande (i = -3/4, -1/2 .....1/2, 3/4).

Y finalmente: $$F = \frac{1}{P} \qquad (4.4)$$

4.7 Decisión para etiquetar la secuencia como "sonoro" o "sordo"

Para etiquetar un segmento como "sonoro" o "sordo", el algoritmo SIFT trabaja con el valor pico interpolado pero normalizado con respecto a r_0 , es decir $r(i) / r_0 = \varphi$, la descripción del criterio es el siguiente:

- Si $\varphi > 0.4$ el segmento j es clasificado como "sonoro". Pero si el segmento "j-1" fue "sordo" y el segmento "j-2" fue "sonoro" entonces el segmento "j-1" se reetiqueta como "sonoro".

- Si $\varphi < 0.4$, y los segmentos "j-1" y "j-2" no están etiquetados como "sonoros" entonces el segmento j se clasifica como "sordo". Caso contrario se verifica si $\varphi > 0.3$, si esto es cierto entonces el segmento j es etiquetado como "sonoro". (27)

4.8 Algunos Resultados

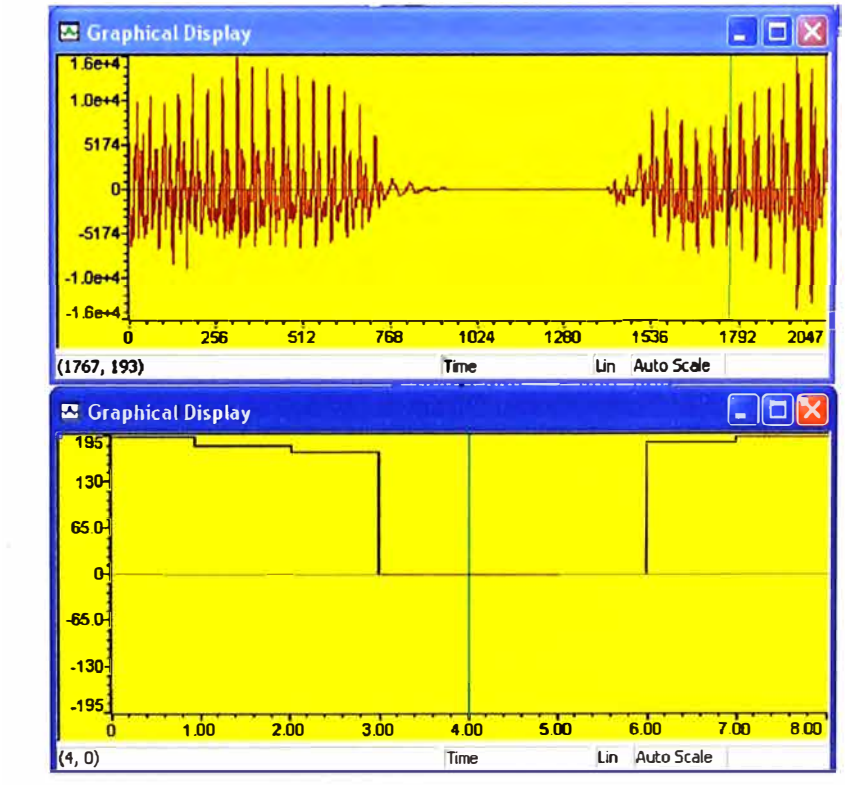


Fig. 4.5 Seguimiento del pitch para dos secuencias de voz usando el algoritmo SIFT

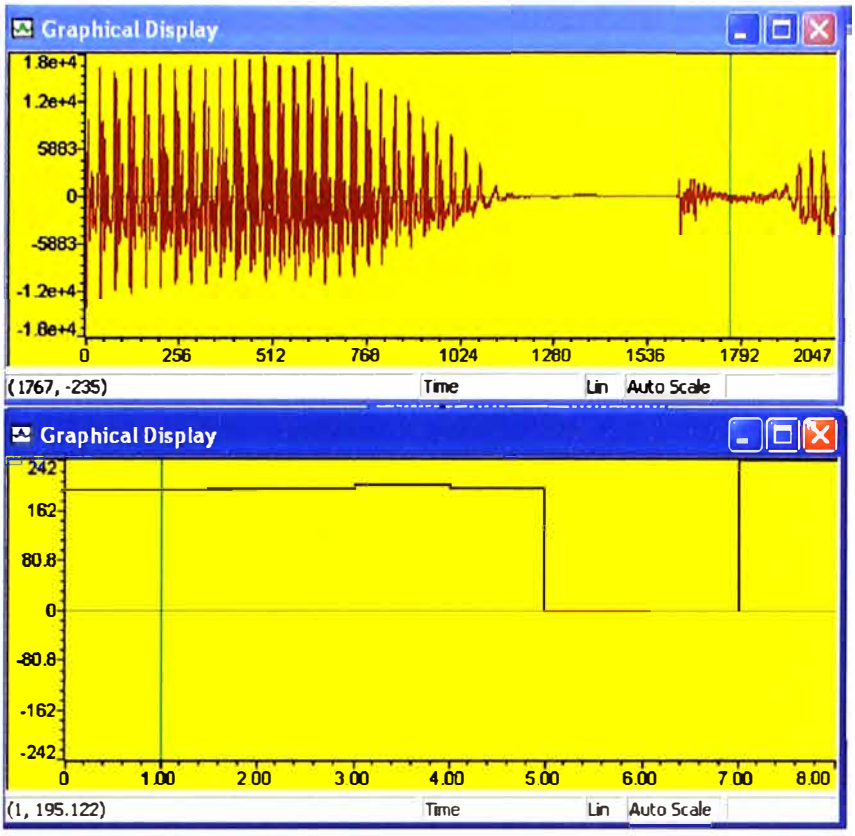


Fig. 4.5 Seguimiento (tracking) del pitch para dos secuencias de voz usando el algoritmo SIFT (continuación)

4.9 Resumen de las mejoras implementadas al algoritmo SIFT

Se resumen brevemente las mejoras realizadas al algoritmo SIFT. El algoritmo SIFT original genera una matriz (ver ecuación 4.3) de solamente 3 x 3. En la actual implementación, se han revisado totalmente las ecuaciones matemáticas que corresponden al algoritmo SIFT (27) y se ha encontrado que se puede lograr aun más precisión en los cálculos si se usa una matriz de 6 x 3 tal como se muestra en la expresión de la ecuación (4.3). Con lo cual tenemos una matriz de 18 términos cuando originalmente el algoritmo usaba una matriz de 9 términos.

Dicha nueva matriz se obtiene luego de analizar las expresiones en (4.1) y (4.2) y es importante destacar que no hay mayor complejidad adicional en el algoritmo por el hecho de duplicar el número de términos de la matriz en (4.3) mas bien lo que se ha ganado es mayor precisión en el cálculo del pitch y por ende mayor calidad en la voz decodificada.

CAPITULO V

DESCRIPCION DE LA PLATAFORMA SYSTEM GENERATOR DE XILINX

El presente capítulo describe primeramente los dispositivos FPGA de la familia Xilinx, en especial la tarjeta de desarrollo de Xilinx University Program: Virtex II Pro Development System XUPV2P. Seguidamente se describen las distintas opciones software que ofrece Xilinx para el desarrollo de aplicaciones sobre dispositivos FPGA, y en especial el System Generator. Finalmente se describen los bloques constituyentes del sistema implementado en System Generator y los correspondientes resultados de la simulación.

5.1 Dispositivos FPGA de XILINX

Un FPGA, cuyas siglas significan Field Programmable Gate Array es un dispositivo semiconductor que contiene componentes de lógica programable (usados para construir circuitos digitales reconfigurables) llamados "bloques lógicos" y también el grupo de interconexiones que unen los bloques lógicos. Dichos bloques lógicos pueden ser programados para realizar diversas funciones: desde compuertas lógicas, complejas funciones combinacionales (decodificadores, funciones matemáticas) hasta funciones complejas. En la mayoría de dispositivos FPGA los "bloques lógicos" también incluyen elementos de memoria o bloques de memoria. (31)

5.1.1 Arquitectura básica de un dispositivo FPGA de Xilinx

Un dispositivo FPGA está compuesto básicamente de los siguientes elementos y se puede apreciar en la Fig. 5.1.

- **CLB (Configurable Logia Blocks)**. Es un bloque básico que agrupa otros bloques fundamentales tales como los bloques SLICE y LUT. Esto permite que este bloque sea lo suficientemente flexible como para implementar cualquier otra función en el FPGA tal como memoria, lógica combinacional,

secuencial, máquinas de estado, etc. La figura Fig. 5.2 ilustra un bloque CLB.

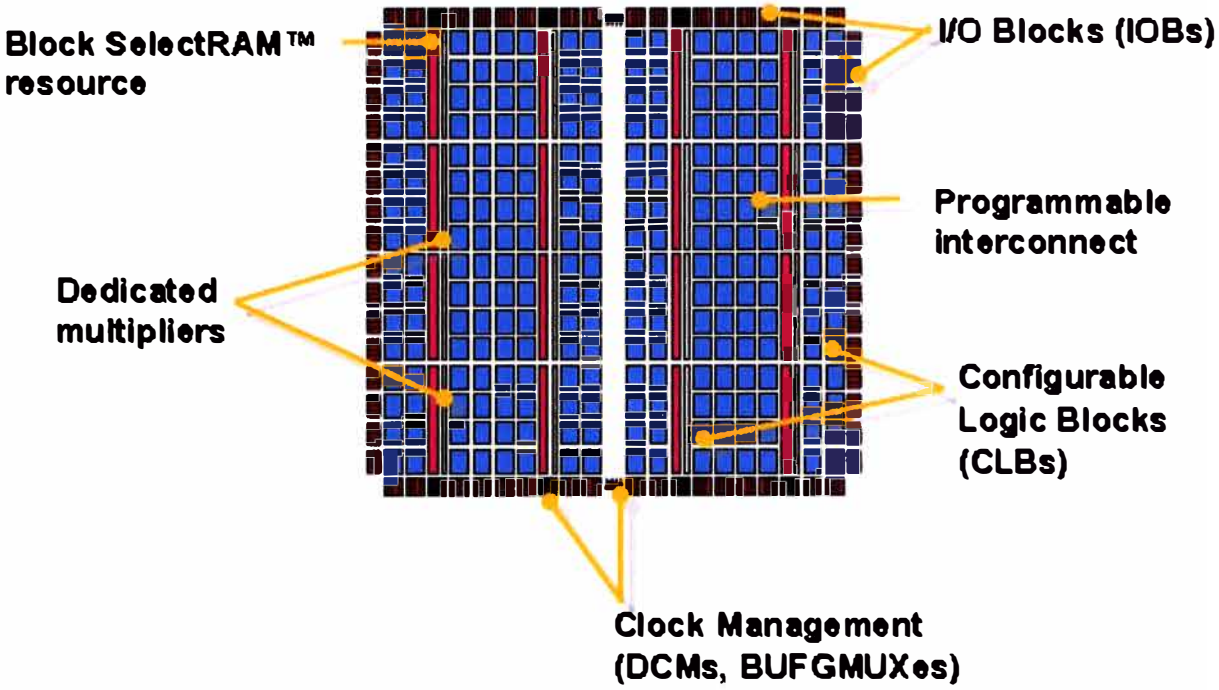


Fig. 5.1 Arquitectura básica de los dispositivos FPGA de la familia XILINX (31)

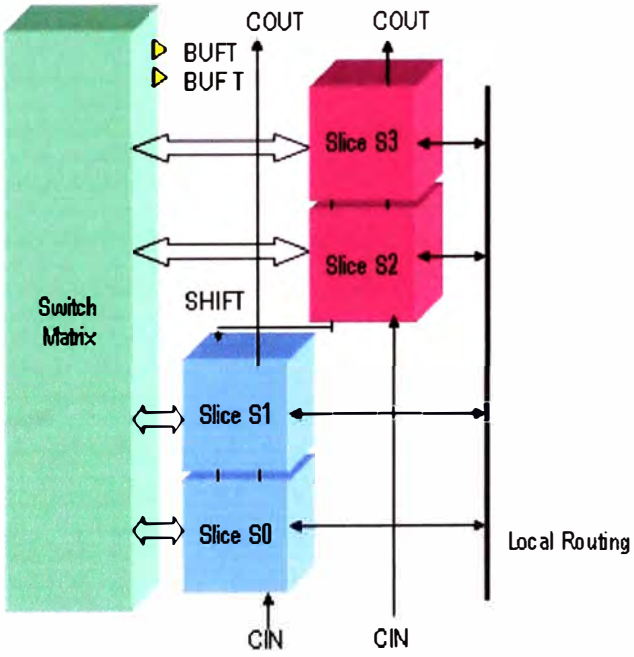


Fig. 5.2 Arquitectura básica de un CLB (31)

- **SLICE.** Cada *slice* agrupa LUT's y en conjunto con otros *slice* pueden implementar las funciones lógicas. Cada slice está conformada por LUT's flip-flops, y lógica adicional. La figura Fig. 5.3 muestra un slice.

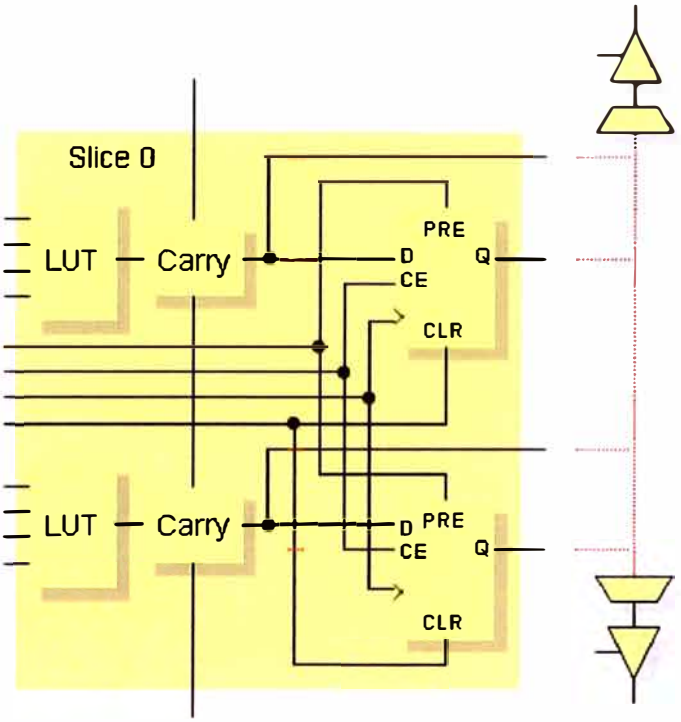


Fig. 5.3 Arquitectura básica de un SLICE (31)

- **LUT.** Significa Look Up Tables. La lógica combinacional básica está almacenada en dichas tablas de allí el nombre Look Up Tables. También se les conoce con el nombre de Generadores de Funciones. Su capacidad esta limitada por el número de entradas pero no por la complejidad de la función a desarrollar.

- **Multiplicadores dedicados** (Dedicated Multipliers) Son multiplicadores ya implementados dentro de la arquitectura del dispositivo FPGA. Pueden realizar multiplicaciones de 4x4, 8x8, 12x12 y 18x18. Físicamente se ubican cerca de los bloques de memoria.

- **DCM (Clock Management)**. Son controladores digitales de reloj. Los dispositivos FPGA de Xilinx contienen hasta 12 de estos controladores. Se localizan en los extremos superior e inferior del dispositivo. Los DCM

proveen las siguientes funcionalidades: función DLL (Delay-Locked Loop), DFS(Digital Frequency Synthesizer) y DPS(Digital Phase Shifter).

- **Bloques de memoria RAM distribuida.** Son bloques de memoria independientes y que pueden ser accesados desde cualquier aplicación. Hay diversas configuraciones dependiendo del dispositivo: 16K x 1, 8K x 2, 4K x 4, 2K x 9, etc.

- **Bloques de Interconexión.** Corresponde a las partes del dispositivo que une el resto de los componentes del FPGA.

Por ejemplo los dispositivos FPGA de la familia Virtex II de Xilinx presentan las características mostradas en la figura Fig. 5.4

Virtex-II Part Number	XC2V 40	XC2V 80	XC2V 250	XC2V 500	XC2V 1000	XC2V 1500	XC2V 2000	XC2V 3000	XC2V 4000	XC2V 6000	XC2V 8000
LUTs + FFs	512	1,024	3,072	6,144	10,240	15,360	21,504	28,672	46,080	67,584	93,184
BRAM (kb)	72	144	432	576	720	864	1,008	1,728	2,160	2,592	3,024
Multipliers	4	8	24	32	40	48	56	96	120	144	168
DCM Units	4	4	8	8	8	8	8	12	12	12	12

Fig. 5.4 Características de los dispositivos Virtex II de Xilinx (32)

5.1.2 Tarjeta de desarrollo Virtex II Pro Development System XUPV2P

Es un modulo de desarrollo que incluye el FPGA Virtex II Pro de Xilinx. Por los numerosos periféricos que contiene sirve para una amplia variedad de aplicaciones. La tarjeta puede emplearse tanto para diseño digital, sistema de desarrollos de microprocesadores o como un "host" para núcleos de procesadores embebidos y sistemas digitales complejos. (33)

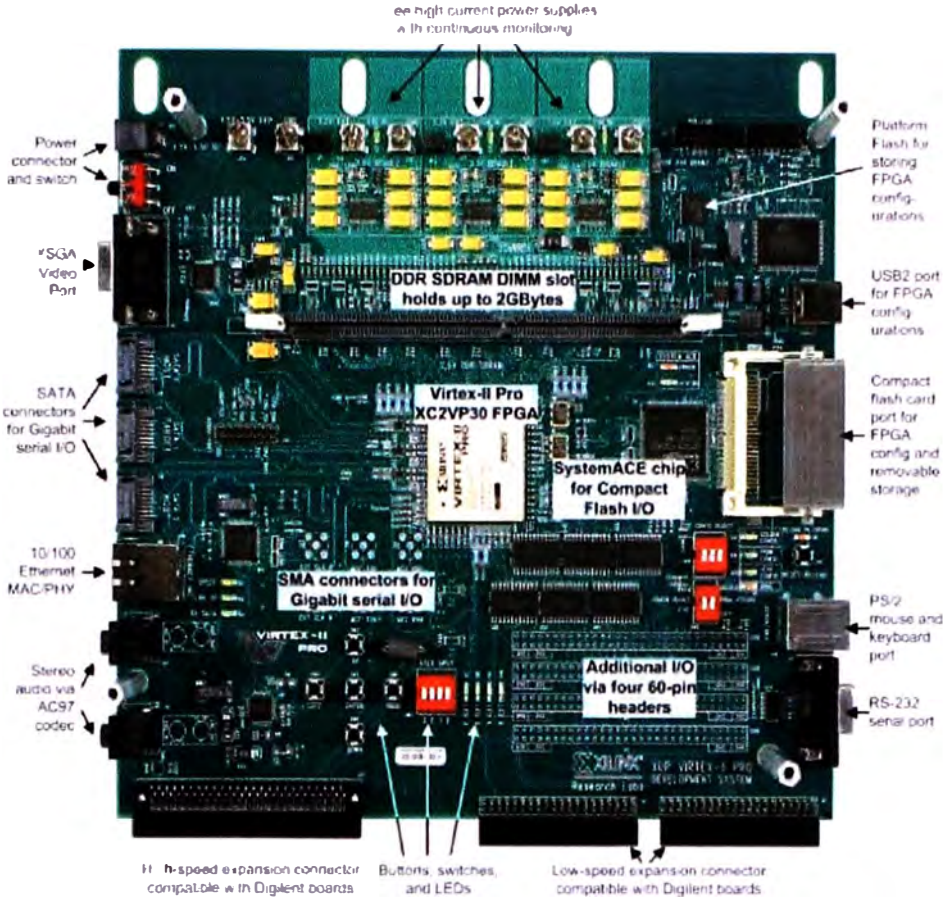


Figura 5.5 Tarjeta de Desarrollo Virtex II Pro (33)

De entre los múltiples periféricos que contiene la tarjeta de desarrollo, podemos destacar los siguientes: audio codec, salida de video XSGA, puerto 10/100 Ethernet, 2 puertos RS-232, 2 puertos serial ATA, puerto transreceptor Multi-Gigabit, expansión para 2GB de memoria DDR SRAM DIMM, puertos de expansión de alta velocidad, capacidad para insertar una memoria Flash, etc.

La figura Fig. 5.6 ilustra el diagrama de bloques de la tarjeta de desarrollo. Es preciso destacar que todos los dispositivos periféricos interactúan directamente con el dispositivo FPGA, tal como se muestra en la Fig. 5.6.

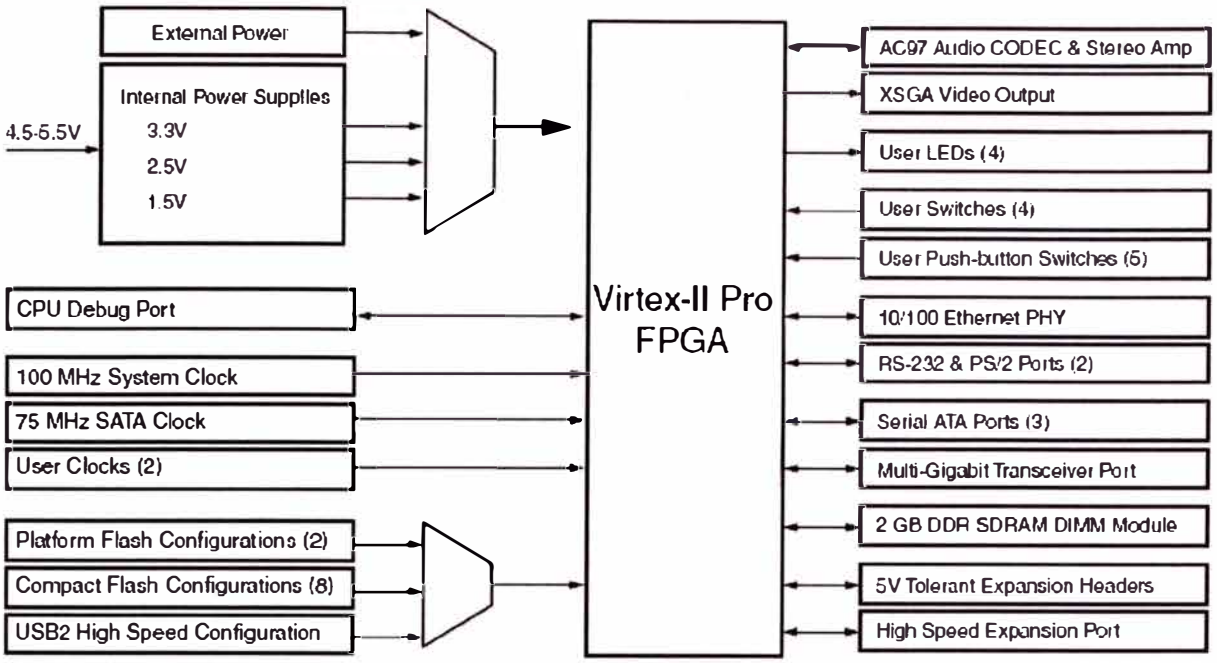


Figura 5.6 Diagrama de bloques de la tarjeta de desarrollo Virtex II Pro (33)

5.2 Software para Desarrollo de aplicaciones en dispositivos FPGA de Xilinx

Hoy por hoy existen muchas herramientas, compiladores, interfaces de desarrollo, y otros aplicativos que permiten realizar aplicaciones para los dispositivos FPGA de Xilinx, muchos de ellos son "third-party software" es decir son hechos por empresas "partner" de Xilinx, y por lo tanto no son gratuitas y no vienen incorporadas con el software básico proporcionado por Xilinx. De entre esas herramientas podemos mencionar compiladores de C, analizadores lógicos, sintetizadores, etc. Por otro lado, de las herramientas brindadas por Xilinx tenemos las siguientes (31, 32):

- ISE Foundation
- EDK
- ChipScope
- System Generator

5.2.1 ISE Foundation

La interfaz de desarrollo ISE integra todas las funcionalidades necesarias para desarrollar un diseño lógico completo sobre los dispositivos FPGA de Xilinx. De entre las muchas funcionalidades ofrecidas por el ISE, podemos mencionar las siguientes:

- Project Navigator. Del cual podemos visualizar todos los componentes que conforman nuestro proyecto.
- Administración del "timing".
- Administración de las restricciones ("constraints") .
- Herramientas de síntesis:
- Herramientas de implementación del diseño.
- Herramientas para administrar el esquemático del proyecto.
- Herramientas de mapeado.
- Herramientas de ruteo y ubicación (place & route)
- Herramientas de síntesis: XST (Xilinx) y también soporta las de otros fabricantes: Symplify, Precision, FPGA Compiler II
- Otras herramientas.

La interfaz ISE Foundation se muestra en la figura Fig. 5.7

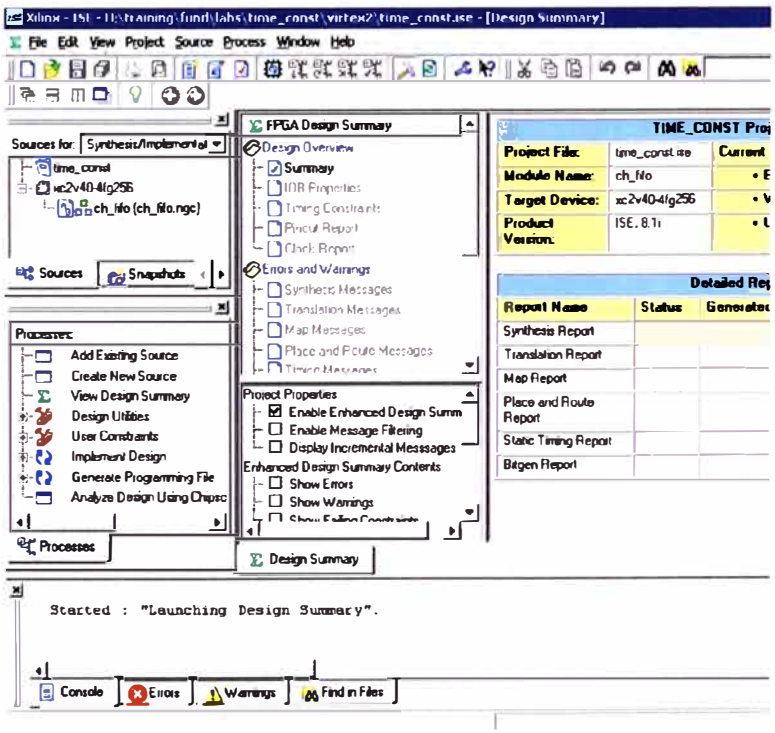


Figura 5.7 Entorno de desarrollo ISE Foundation (31)

5.2.2 EDK

El EDK (Embedded Development Kit) es un entorno de desarrollo que permite ampliar el ámbito de diseño sobre un dispositivo FPGA mediante la inclusión de procesadores embebidos de tal forma que se puedan desarrollar aplicaciones donde interactúen los procesadores embebidos y el diseño sobre el FPGA. Entre los procesadores embebidos podemos listar los siguientes: IBM PowerPC, Xilinx Microblaze y PicoBlaze. Además, el entorno EDK permite el manejo de los periféricos de todas las tarjetas de desarrollo para dispositivos Xilinx, evitando de esta forma, que el usuario tenga que diseñar "drivers" o controladores para cada periférico contenido en una determinada tarjeta de desarrollo como la XUPV2. (34)

El EDK interactúa tanto con el ISE como con otras herramientas de desarrollo tal como el System Generator, lo que permite hacer el diseño lo mas versátil posible. La figura Fig. 5.8 muestra el entorno de desarrollo del EDK.

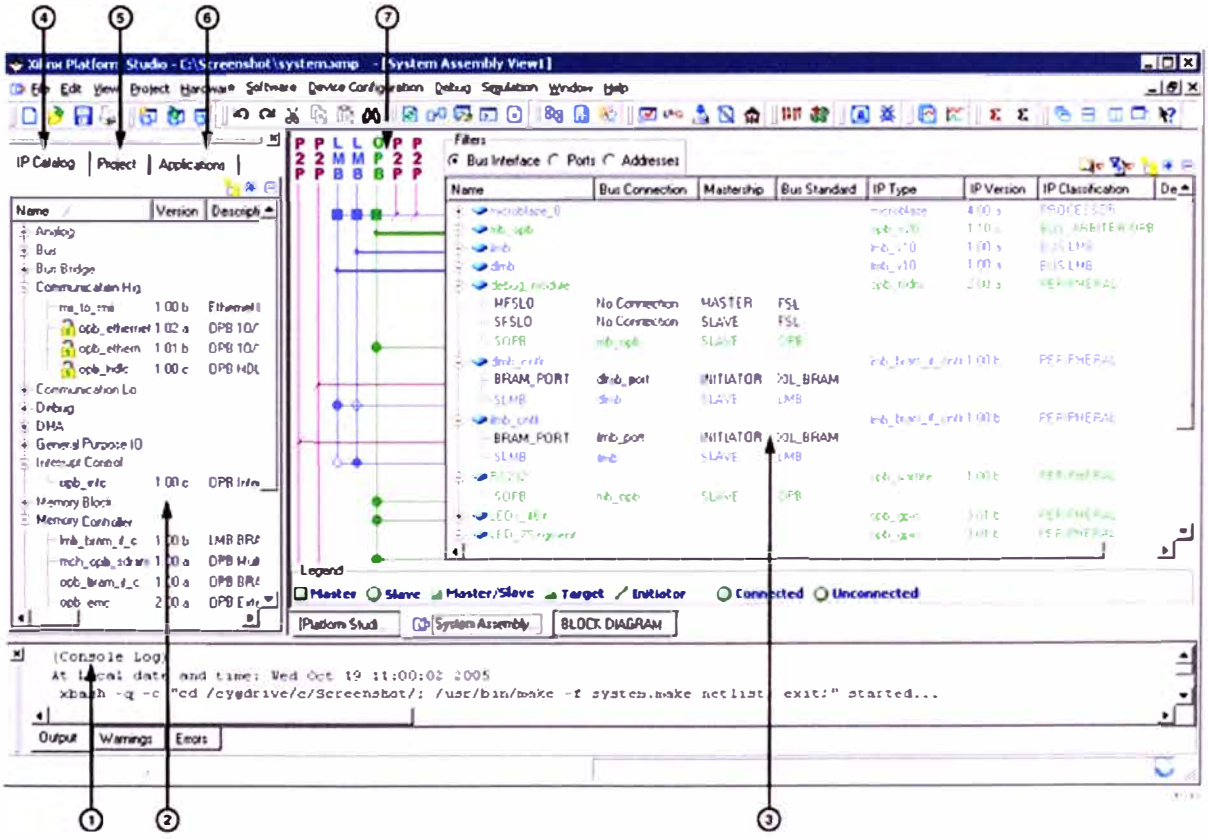


Figura 5.8 Entorno de desarrollo EDK (34)

5.2.3 ChipScope

Es una herramienta para realizar la depuración y verificación de las aplicaciones implementadas sobre el dispositivo FPGA, en tiempo real, es decir, podemos depurar nuestro diseño sin afectar su desempeño y en tiempo real. (34)

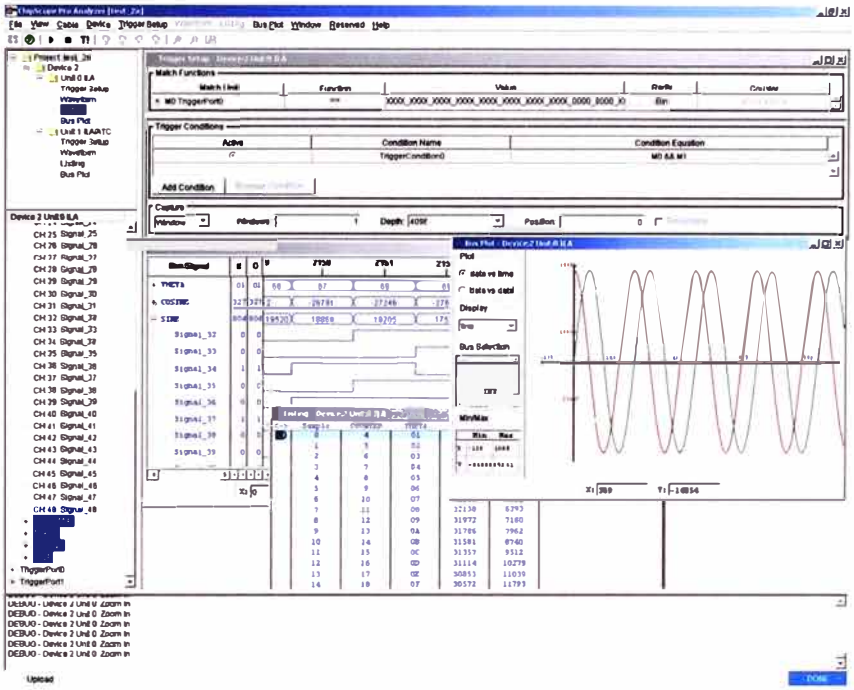


Figura 5.9 Entorno ChipScope [34]

5.3 System Generator

En esta sección se describen con más detalle las características y la forma de trabajo del System Generator, la cual es la plataforma sobre la que se implementa la simulación en el presente trabajo de tesis.

5.3.1 Tendencias en la Industria para el diseño de sistemas embebidos

La plataforma System Generator responde a las tendencias de la industria para el diseño de sistemas embebidos, dichas tendencias son las siguientes:

- Tendencia a desarrollos sobre plataformas de chips (FPGA, DSP) resultando en diseños altamente complejos.
- Los diseños requieren alta flexibilidad.
- Soporte para diversas metodologías de diseño.

- Retos en modelar e implementar plataformas enteras.
- Requerimiento de herramientas para verificación en lazo cerrado (Hardware-in-the-loop verification).

5.3.2 Integración con Simulink, Matlab

La plataforma System Generator se puede considerar como un conjunto de "plugins" de Simulink, solo que son especializados para construir diseños de lógica programable para dispositivos FPGA de Xilinx. Esta forma de trabajo ha venido siendo adoptada en los últimos años no solo por Xilinx sino por otros fabricantes de dispositivos FPGA como Altera. Incluso los fabricantes de DSP (Texas Instruments por ejemplo) han adoptando esta forma de trabajo, porque presenta muchas ventajas para el diseñador de entre las cuales podemos mencionar(32, 33, 35):

- Al integrase con el Simulink, los diseñadores se valen de todas las herramientas disponibles por Simulink: librerías matemáticas, librerías de procesamiento de señales, comunicaciones, control, y muchas mas.
- También están disponibles todos los bloques de generación de señales (sources) y de visualización de señales (sinks).
- Se puede trabajar en tiempo real, es decir ejecutar el diseño en paralelo tanto en el entorno Simulink como en el dispositivo FPGA y comparar ambos resultados.

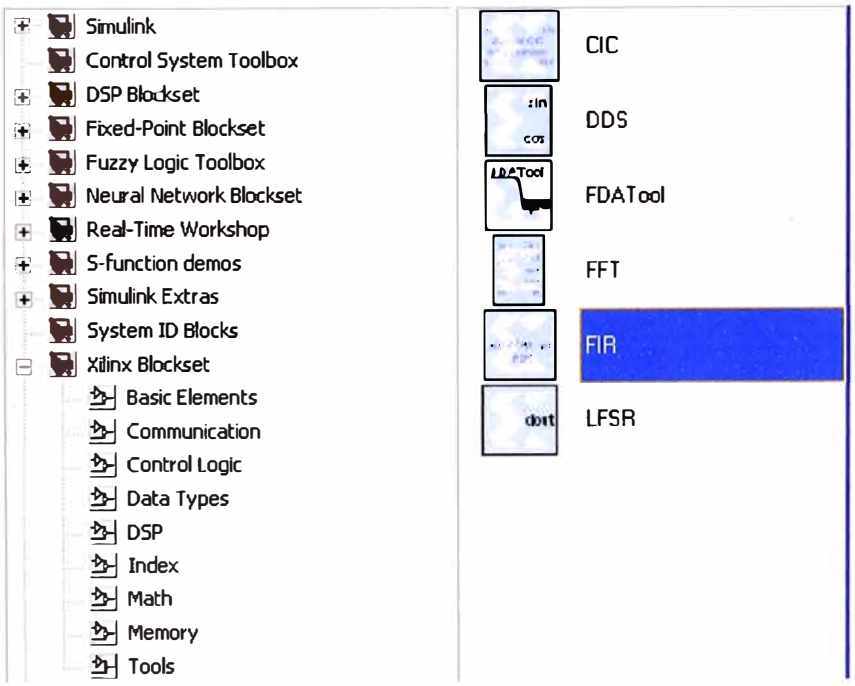


Figura 5.10 Librerías System Generator en Simulink (35)

La figura Fig. 5.10 muestra un "blockset" que contiene un conjunto de bloques de diseño los cuales se pueden usar para implementar un sistema. Dichos bloques no vienen incorporados en el software Matlab por sí solo sino que se tiene que adquirir una licencia de trabajo y descargar el System Generator de la página web de xilinx en la siguiente dirección: http://www.xilinx.com/ise/optional_prod/system_generator.htm. (35)

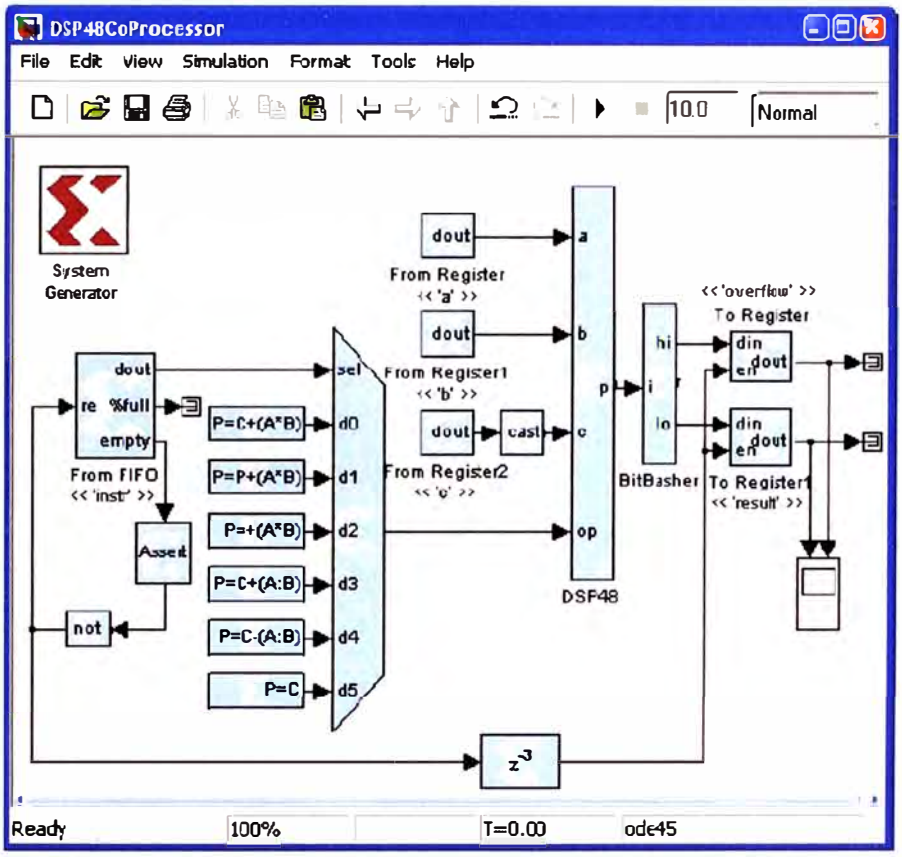


Figura 5.11 Diseño típico en System Generator (35)

La figura. Fig. 5.11 nos muestra un diseño típico en System Generator en el que se puede apreciar que el diseño se basa en los bloques de color morado que son los bloques Xilinx del System Generator.

Características de la forma de trabajo de System Generator:

- Flujo de diseño integrado desde el Simulink hacia el FPGA.
- Integra diferentes tecnologías: Matlab/Simulink, herramientas de síntesis HDL, librerías propietarias, herramientas de implementación para FPGA, etc.
- Precisión de bit y de ciclo a través del diseño y simulación.
- Abstracción aritmética: precisión arbitraria para operaciones de punto fijo, incluyendo cuantización y overflow.

5.3.3 Flujo de diseño en System Generator

La figura Fig. 5.12 muestra el flujo de diseño general de System Generator.

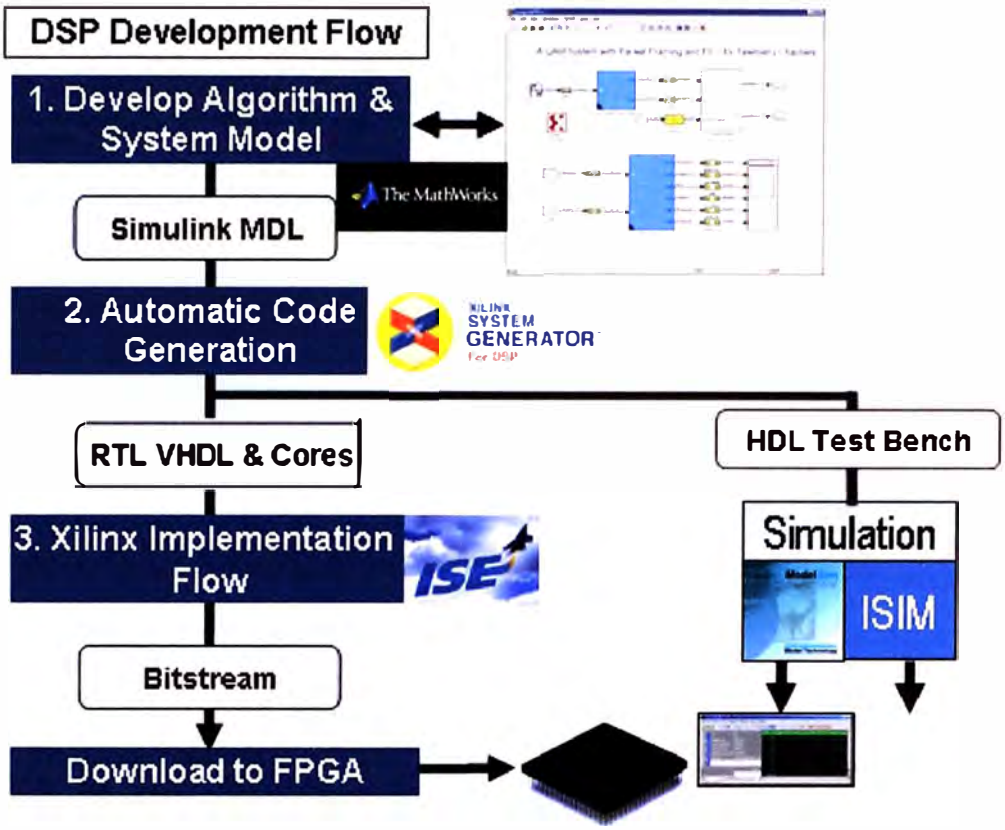


Figura 5.12 Flujo de diseño típico en System Generator (32)

El modelo diseñado en el Simulink, cuando es sintetizado e implementado con las herramientas del System Generator, genera una serie de archivos que son equivalentes a los que generaría si se estuviera diseñando el sistema en la herramienta ISE. Esto es así porque el System Generator invoca las mismas herramientas que son empleadas por el ISE (por ejemplo XST). Los archivos generados son los siguientes:

- Código generado en VHDL.
- Archivo .npl: es un archivo de proyecto de ISE el cual provee una entrada al flujo de diseño ISE, de esta forma se invocan las herramientas externas al System Generator.
- Archivo .xcf: es el archivo de restricciones ("constraints") el cual asiste en la implementación para obtener un máximo desempeño.
- Archivo testbench: el archivo testbench usa el estímulo producido por simulink y también compara los resultados de la simulación HDL y los de simulink.

- Archivos.do: Usados para simulación y compilación.

Básicamente, existen 4 formas diseño en System Generator:

En el primer escenario, la implementación solo se basa en bloques de Xilinx system Generator y a partir de ellos crea un diseño sintetizable, el cual puede ser implementado usando el entorno de desarrollo ISE. En este caso no se incluyen bloques definidos por el usuario. Esta es una forma rápida de diseño.

La segunda forma de trabajo es mediante el flujo de Co-simulación HDL. En este caso el diseño puede incluir "blackbox", código VHDL y también "IP core" (componentes propietarios) junto con los bloques convencionales de Xilinx y a partir de ellos se genera un diseño sintetizable el cual, así mismo, puede ser implementado usando el entorno ISE. Esta forma de trabajo también usa el bloque ModelSIM el cual invoca el simulador ModelSIM el cual simula el diseño. La salida del simulador es llevada de vuelta a Simulink y los resultados pueden ser visualizados mediante los bloques "sink" del Simulink.

La tercera forma de trabajo es mediante el flujo de diseño para verificación acelerada de "Hardware in the loop". En este aso también tenemos "blackbox", código VHDL, "IP core", bloques de interfaz hardware y bloques Xilinx. En este esquema se puede generar un diseño sintetizable el cual es implementado usando la herramienta xflow (invocado en segundo plano), el cual genera un archivo de bits (ejecutable) y un bloque de "hardware in the loop". Una vez que este nuevo bloque es añadido al diseño, el usuario puede correr la simulacion y obtener la respuesta en tiempo real tanto en el simulink como en el dispoitivo FPGA físico o real, para de esta forma ealuar el desempeño.

La cuarta forma de trabajo es para escenarios muy complejos donde se requiere el uso de los procesadores embebidos. En estos casos se manejan hasta 3 entornos de desarrollo simultáneamente: ISE, Simulink (System Generator) y EDK, para integrar todas las herramientas y generar un único flujo de desarrollo.

5.3.4 Creación de un diseño en System Generator

Crear un diseño en System Generator es directo, simplemente seleccionamos los bloques de la librería Xilinx que necesitaremos y los vamos colocando en la ventana de

diseño. Sin embargo, al momento de probar el sistema, se recurren a bloques específicos que permiten hacer la simulación con precisión de bit y reloj. Esto se describirá mas adelante. La creación de un diseño en System Generator se puede apreciar en la figura Fig. 5.13. (35)

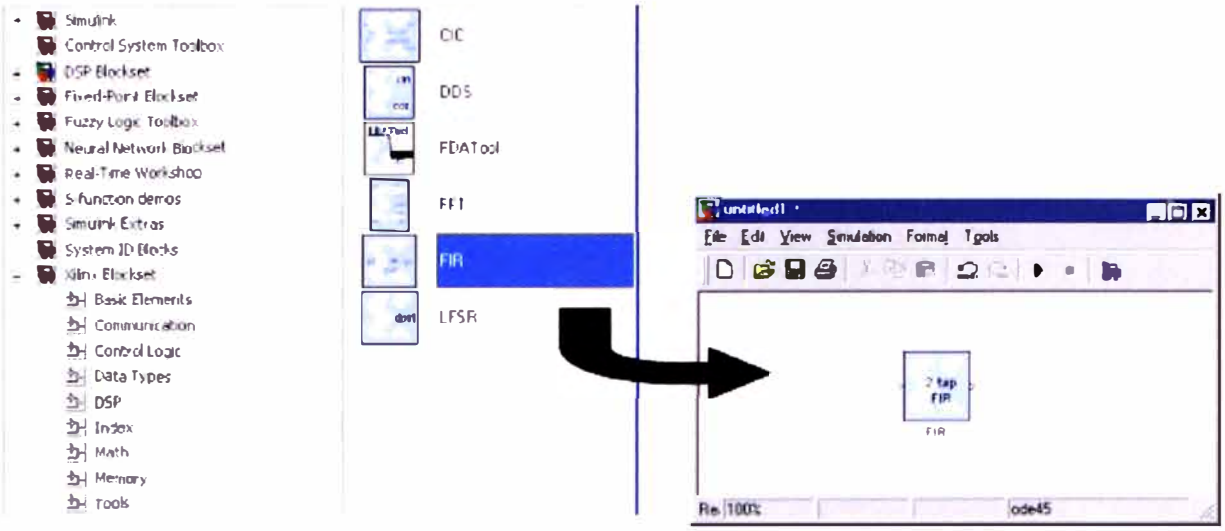


Figura 5.13 Creación de un diseño en System Generator (35)

CAPITULO VI

SIMULACION Y RESULTADOS DEL CODIFICADOR IMPLEMENTADO EN SYSTEM GENERATOR

En este capítulo se presenta una descripción detallada y paso a paso de los bloques que constituyen el diseño implementado tanto de la parte del codificador como del decodificador.

Así mismo, junto con la descripción de los bloques, se presentan los resultados de la simulación obtenidos en cada bloque del diseño implementado, resaltando los aspectos más importantes de los resultados.

Al final se presenta los resultados para la simulación de 0.5 segundo de voz, que equivale a 25 tramas de 160 muestras por trama (a 8000 Hz).

Todos los archivos correspondientes a la presente simulación se anexan en un CD junto con el presente trabajo.

6.1 Esquema General del Codificador Implementado

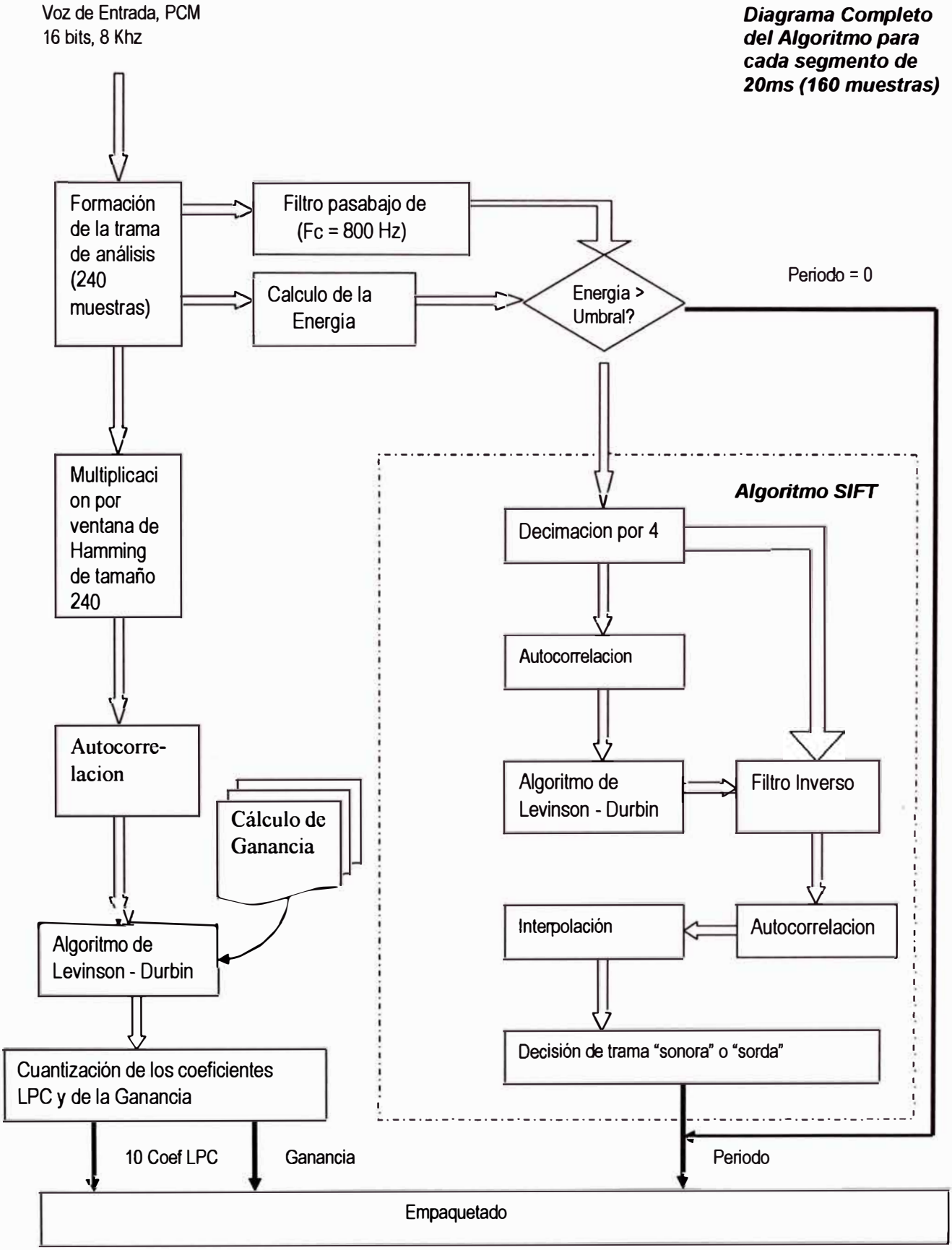


Figura 6.1 Diagrama general del codifcador implementado.

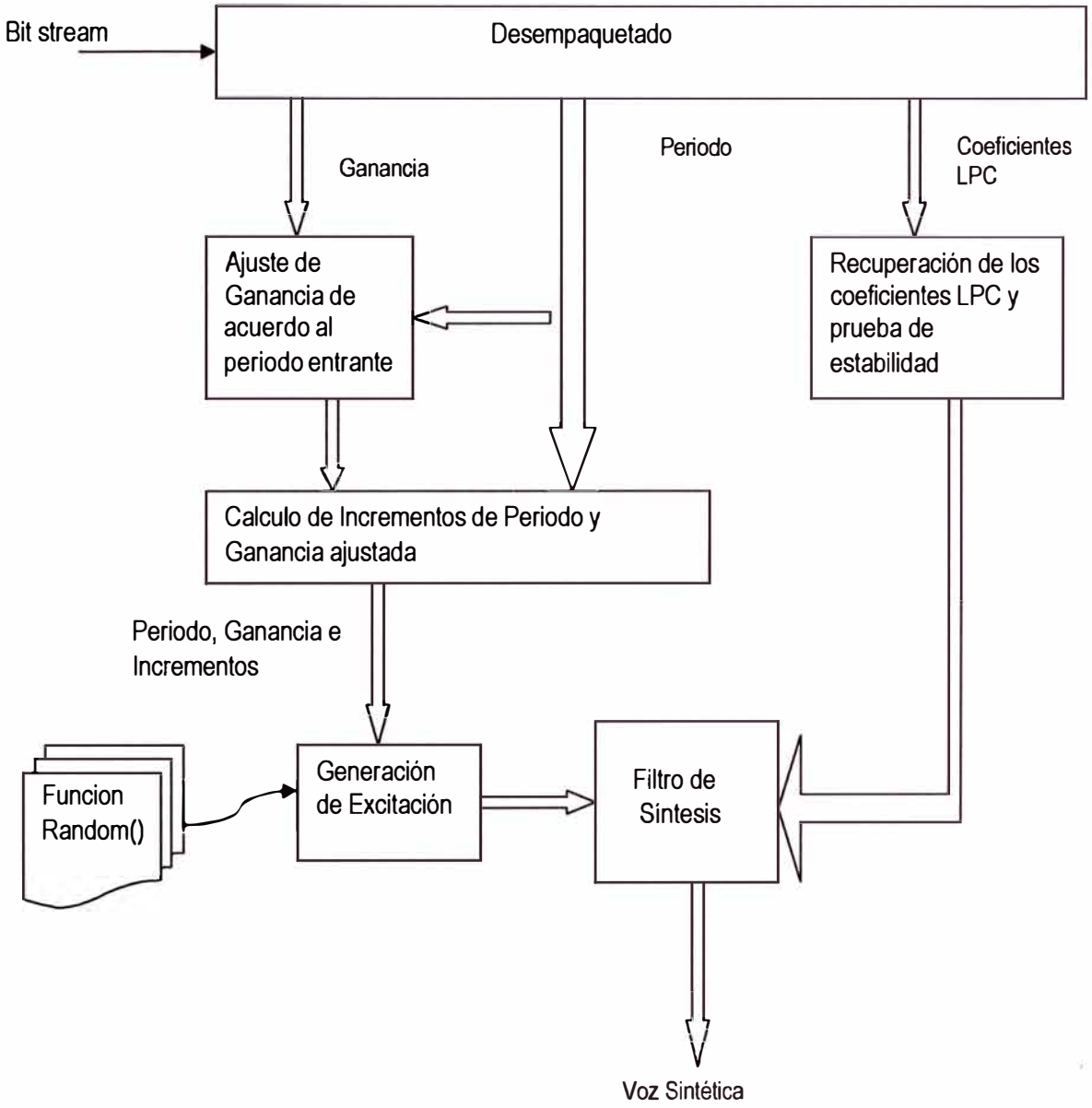


Fig. 6.2 Esquema General del Decodificador Implementado.

6.2 Diseño implementado en System Generator

La figura Fig. 6.3 nos muestra la vista panorámica del diseño implementado en System Generator. En las siguientes secciones se irán desarrollando y describiendo cada parte que constituye el diseño. Los manuales se pueden consultar en (31, 32, 33, 34, 35)

6.2.1 Valores globales de uso en el diseño

Los siguientes son valores globales los cuales nos servirán para desarrollar y describir los bloques en la presente implementación:

```
FC=800;     Frecuencia de Corte filtro pasabajo
FS=8000;    Frecuencia de muestreo de la señal de voz
DOWN = 4;   decimación
PITCHORDER=4;     numero de coeficientes lpc para algoritmo SIFT
WSCALE = 1.5863;  escala
LPC SAMPLES_PER_FRAME=160;    número de muestras de entrada

LPC FILTORDER = 10;     orden del filtro de síntesis
BUFLEN=LPC_SAMPLES_PER_FRAME*3/2; tamaño de la trama de análisis 8(240)

MINPIT = 50;      mínima frecuencia de pitch
MAXPIT = 300;     máxima frecuencia de pitch
MINPER= round(FS/(DOWN*MAXPIT) +0.5);     minimo periodo diezmado
MAXPER = round(FS/(DOWN*MINPIT) +0.5);    maximo periodo diezmado
```

6.2.2 System Generator Block

Se muestra en la figura Fig. 6.4. Cada modelo desarrollado en System Generator debe contener el bloque "System Generator" el cual se encuentra en la librería "Xilinx Blockset / Basic Elements". Este bloque es usado para configurar e invocar el "code generator" y además para resolver las escalas de tiempos entre el Simulink y el reloj que corre en el hardware FPGA. También, el bloque "System Generator" permite emplear varias opciones de compilación a diversos destinos (tarjetas destinos, módulos de evaluación, etc). El campo "FPGA clock period " define la frecuencia de reloj en hardware y el campo "Simulink System Period" define el tiempo en el entorno Simulink que corresponde a dicha frecuencia de reloj en hardware. Ocurre un error si el diseño no contiene uno de estos bloques. (35)

Hay que subrayar que el periodo “Simulink System Period” fue seteado a Tsysclk cuyo valor es de Tsysclk = (1/24000000)*15.

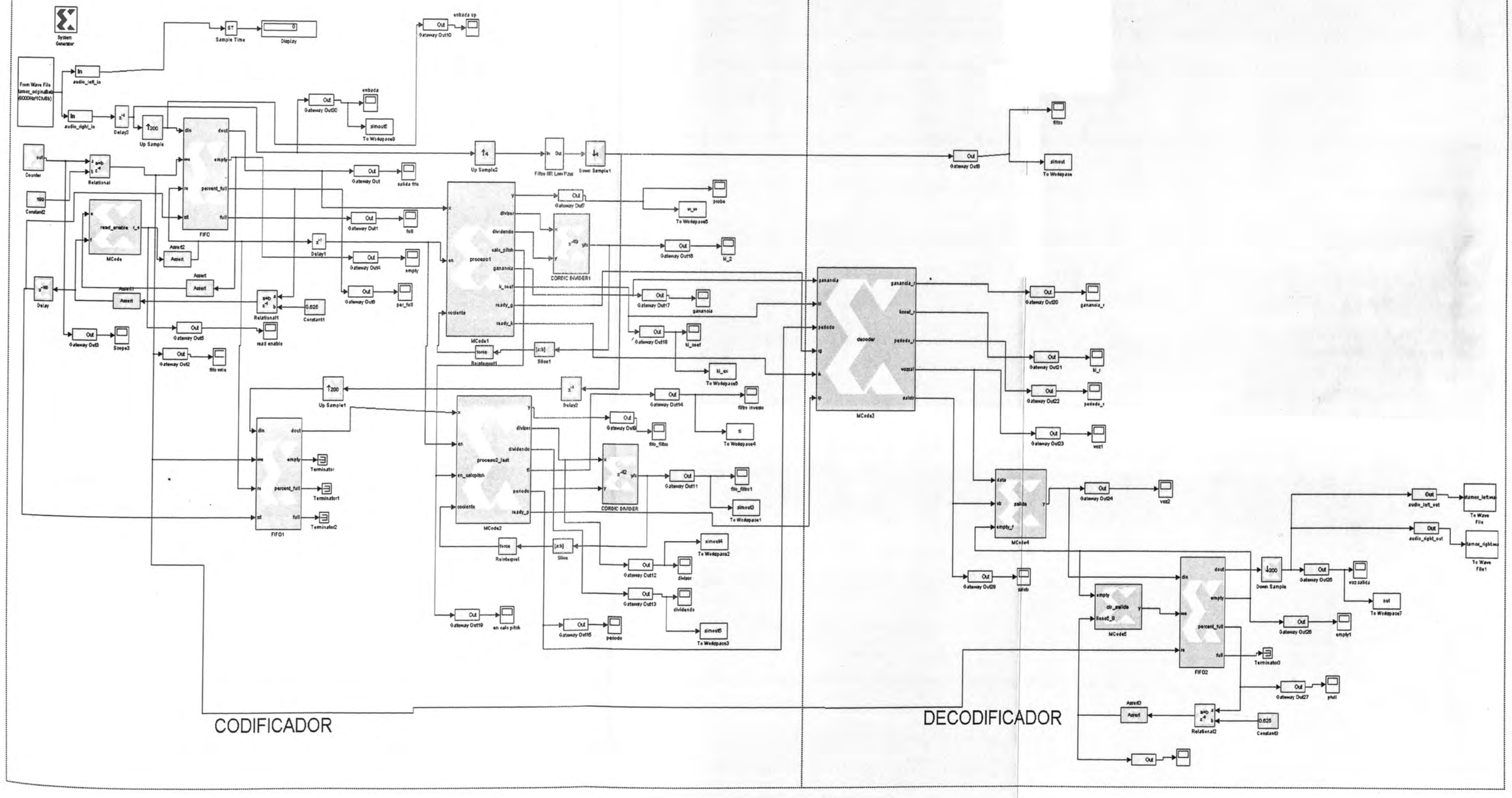


Fig. 6.3 Vista panoramica del diseño implementado en System Generator

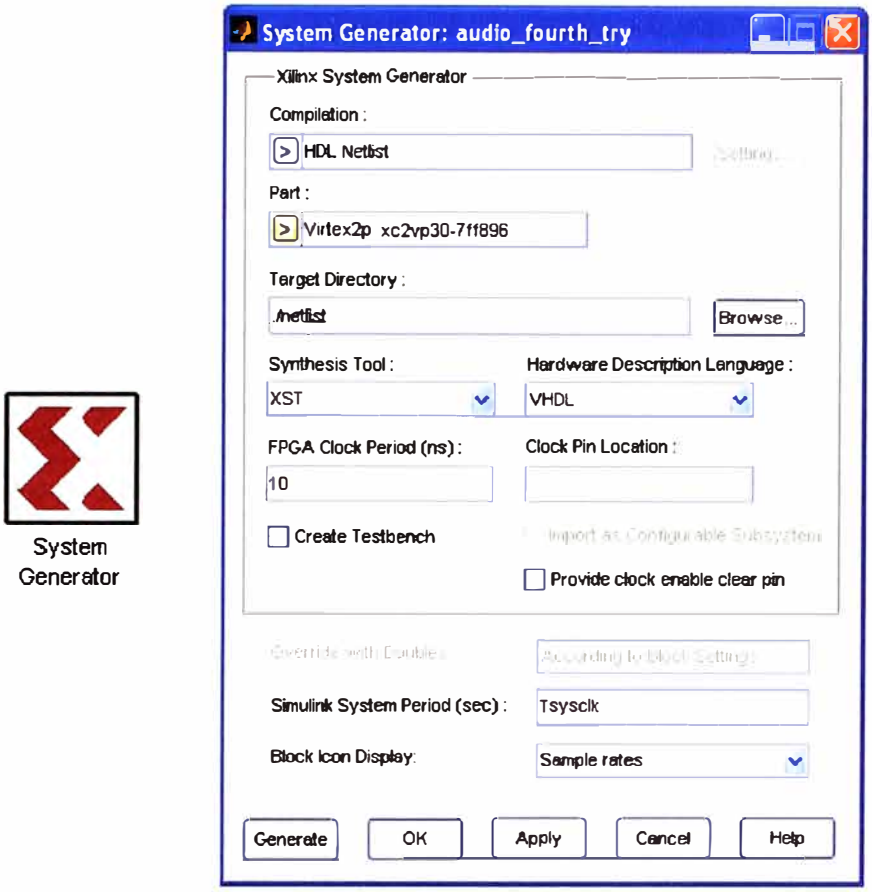


Fig. 6.4 Bloque "System Generator" y su cuadro de configuración.

6.2.3 Señal de entrada

El segmento de entrada del diseño se puede apreciar en la Fig. 6.5.

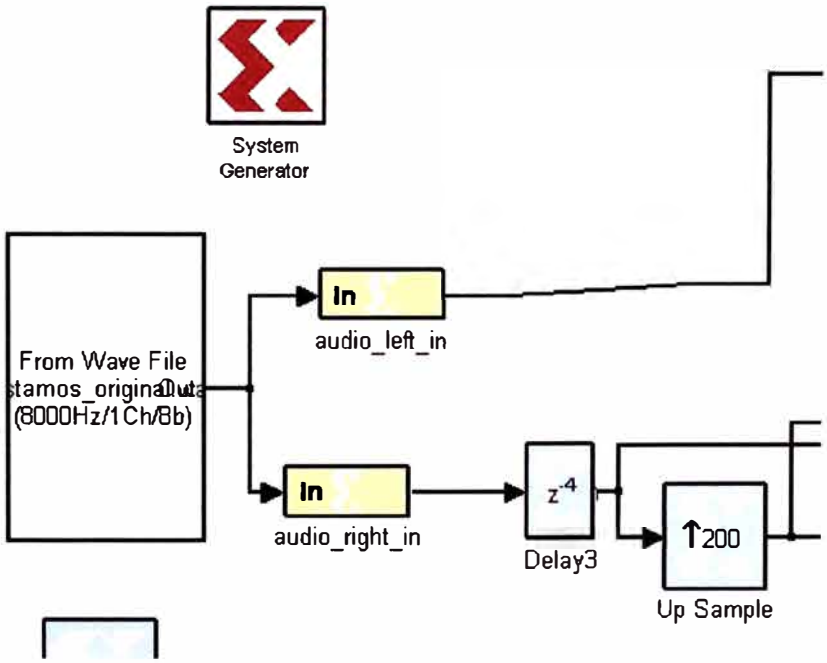


Fig. 6.5 Bloque de entrada del diseño.

En la figura Fig. 6.6 se puede apreciar el bloque "System Generator". El archivo de entrada se llama "estamos_original.wav" y corresponde a una secuencia de voz muestreada con una frecuencia de muestreo de 8000 Hz.

Se observa que el valor de periodo simulink configurado es de: Tsysclk = (1/24000000)*15, el cual se configura así en base a la frecuencia a la que se muestreará la voz entrante. Como la voz entrante debe muestrearse a 8000 Hz, entonces se configuran los periodos de muestreo de los bloques de entrada a 200* Tsysclk = 0,000125, lo cual equivale a una frecuencia de muestreo de 8000 Hz tal como se muestran en las figuras Fig. 6.7 y Fig. 6.8.

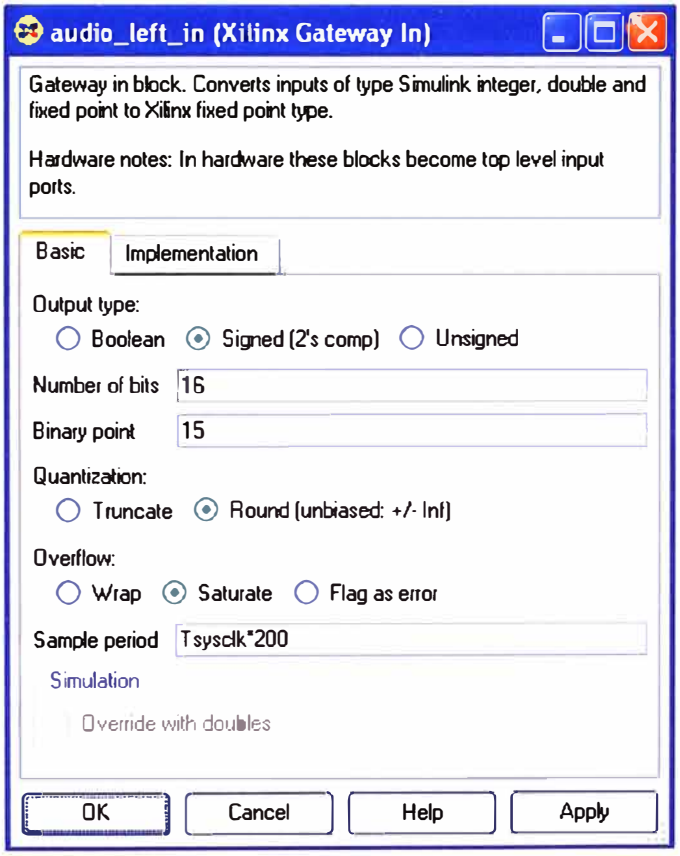


Fig. 6.6 Configuración de la frecuencia de muestreo para la señal de entrada.

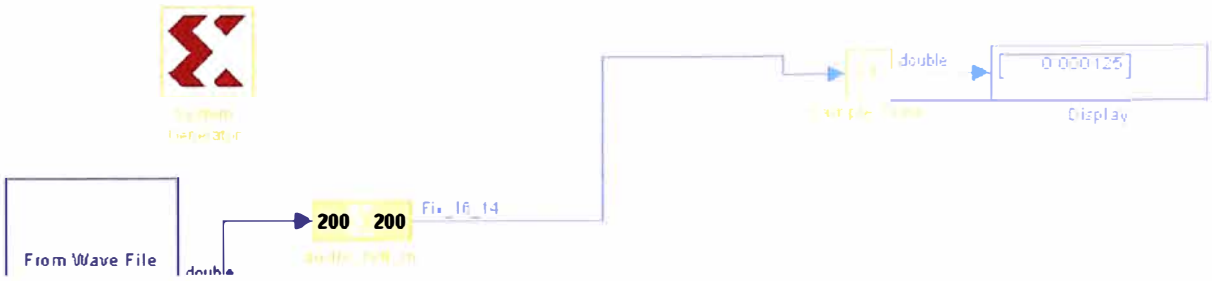


Fig. 6.7 Frecuencia de muestreo visualizada, la cual es 0.000125 segs (8000 Hz).

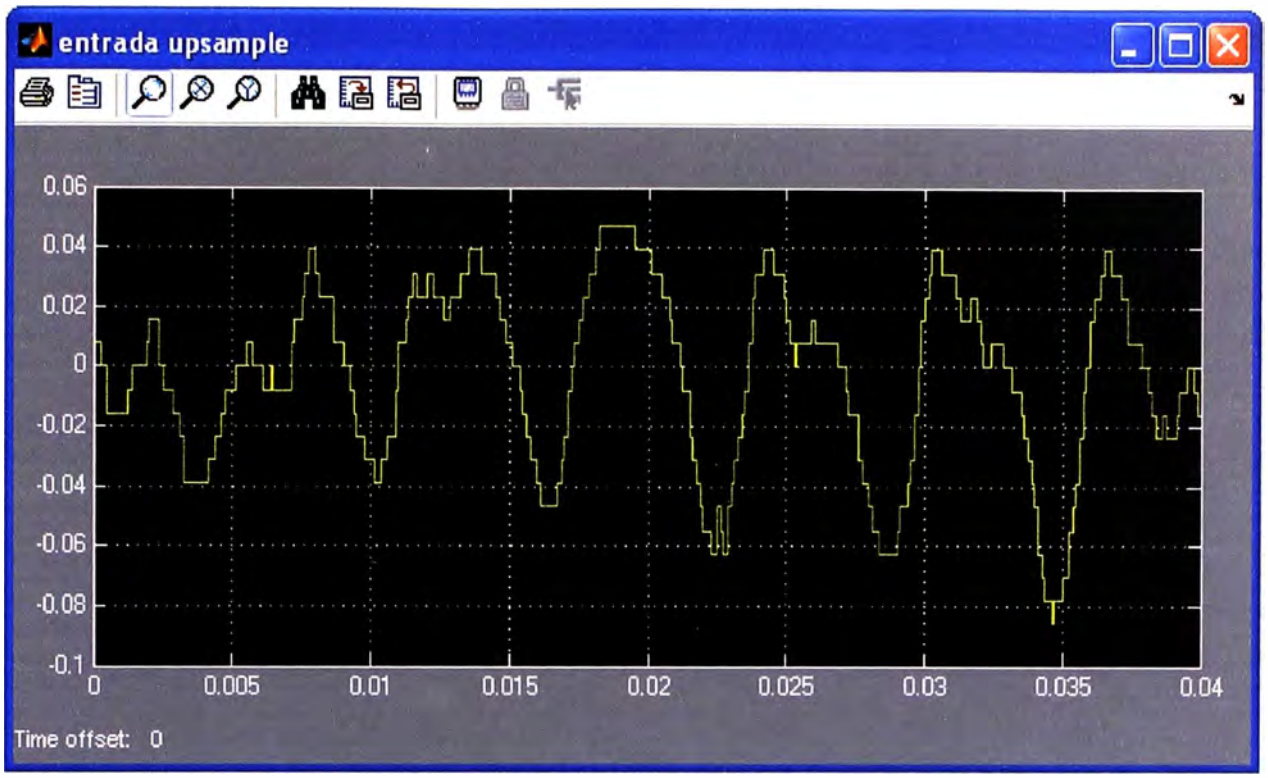


Fig. 6.8 Segmento de la Señal de entrada (muestreada a 8000 Hz)

6.2.4 Acondicionamiento y Control de la adquisición de muestras de entrada

Cada muestra de la señal de entrada es recibida cada 0.000125 (1/8000) segundos o 0.125 us. Para construir la trama de análisis que es de 240 muestras se necesitan 160 muestras de la señal de entrada antes de entrar a los siguientes bloques. Entonces se necesita un método para recolectar constantemente las muestras de la señal de entrada cada 1/8000 segundos sin interrumpir el desempeño del resto del algoritmo. La forma de trabajo se muestra en la figura Fig. 6.9 donde se puede apreciar que todo el algoritmo (codificador más decodificador) se debe ejecutar antes que se complete la recolección de las 160 muestras de la siguiente trama.

Fig. 6.9 Ilustración del manejo de tiempos

Se observa que para que el algoritmo se ejecute dentro del tiempo de recolección de 160 muestras, el periodo del reloj de trabajo del sistema debe ser mucho mas rápido que 1/8000. En la presente implementación se trabajó con un periodo dado por Tsysclk = (1/24000000)*15. Entonces, la sección de entrada se muestra en la figura Fig. 6.10.

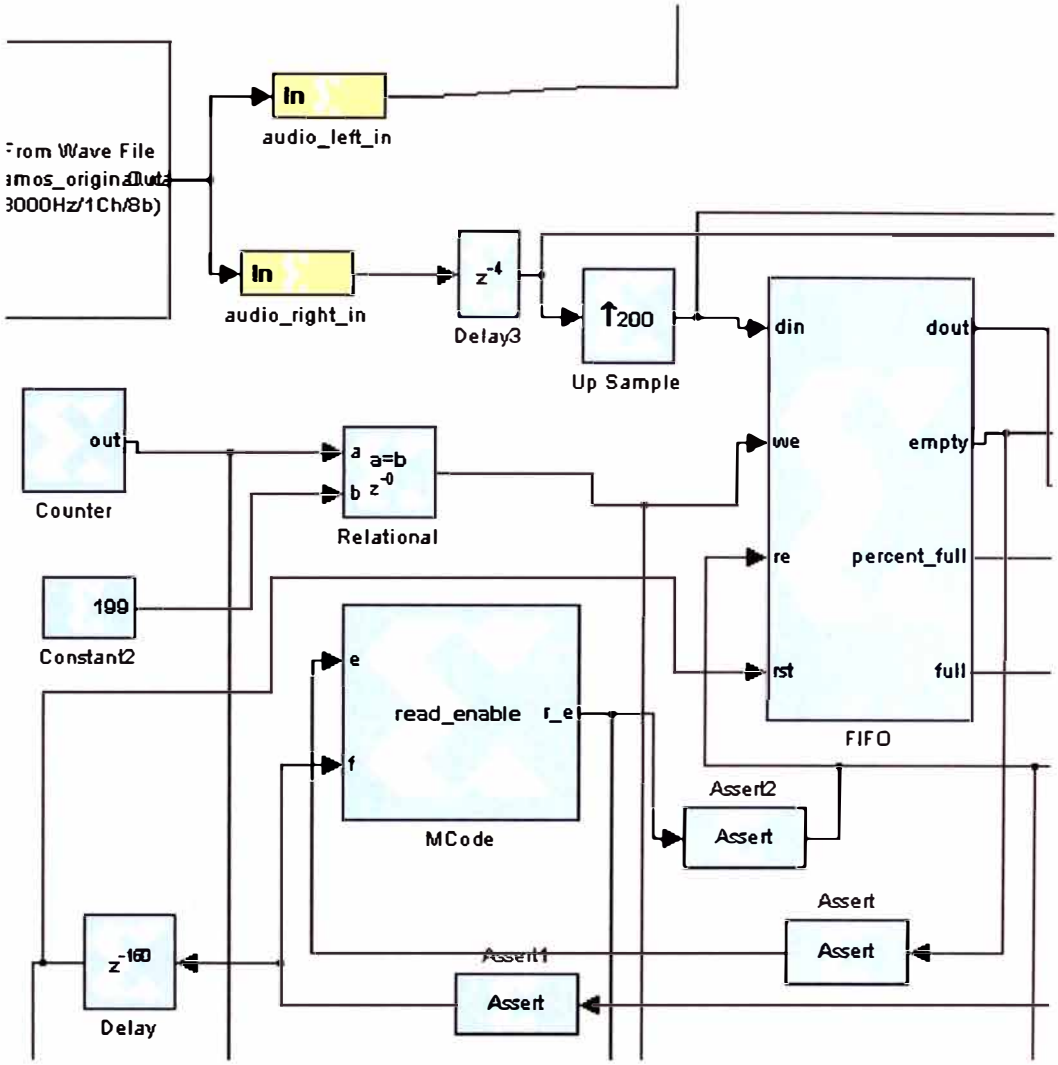


Fig. 6.10 Control y acondicionamiento de la adquisición de muestras.

A partir del bloque "Up Sample", el cual realiza un up-samplig por 200, el resto del sistema trabaja a dicha frecuencia, es decir a Tsysclk = ((1/24000000)*15)200 o equivalente a (1/8000)/200.

El **bloque FIFO** que se observa en la figura Fig. 6.10 realiza la recolección de los datos de entrada. Su configuración se muestra en la figura Fig. 6.11. Hay que observar que el bloque FIFO recibe datos a la frecuencia dada por (1/8000)/200 por lo que hay que incluir necesariamente una sección de control para sincronizar las muestras de entrada con la lectura de entrada del bloque FIFO.

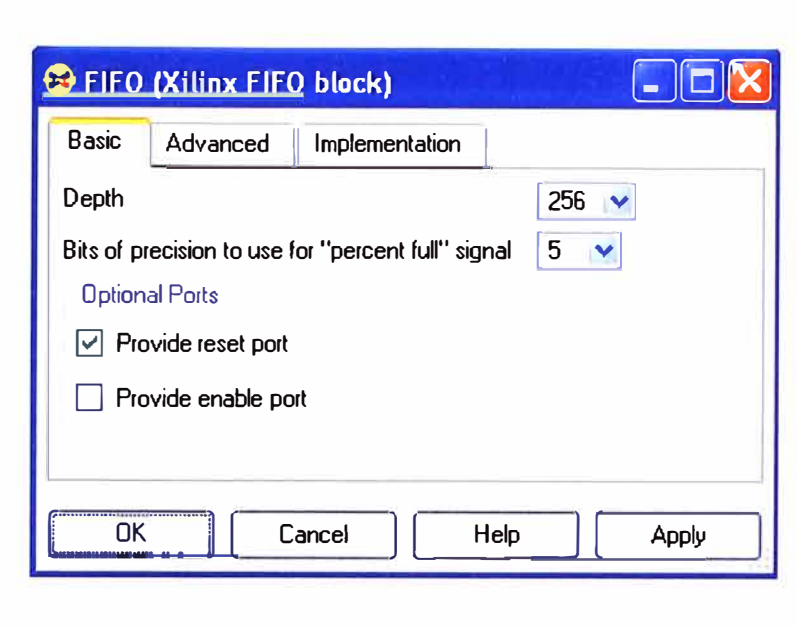


Fig. 6.11 Configuración del bloque FIFO de la Fig. 6.10.

Los bloques que se encargan de la sincronización son los que se muestran en la figura Fig. 6.10 y están conformados por :

Bloques “assert”
Bloque contador (counter).
Bloque relacional (relational)
Bloque retardo (delay).
Bloque m-code.

El objetivo de la lógica conformada por los bloques listados es hacer que el puerto de entrada del bloque FIFO sea habilitado solo una vez cada 200 ciclos dados por Tsysclk de tal forma que 200*Tsysclk nos da una frecuencia de 8000 muestras por segundo. De esta manera los datos que entran al bloque FIFO se van cargando a la frecuencia de 8000 muestras por segundo. Observar que el bloque FIFO tiene configurado un tamaño (depth) de 256 tal como se muestra en la figura Fig. 6.11, pero sin embargo solo se necesitan 160 muestras de entrada. Para controlar que el FIFO haya recolectado solo 160 muestras se observa el puerto de salida “percent_full” el cual da el nivel de llenado del FIFO. Como solo se requieren 160 muestras, entonces la salida del puerto “percent_full” que interesa es de 160/256 = 0.625. Cuando dicho puerto alcanza el valor de 0.625, entonces la lógica se entera que el bloque FIFO tiene 160 muestras por lo que se procede a vaciar el FIFO antes que ocurra la muestra 161. Lo anterior se resume y grafica en el diagrama de flujo de la figura Fig. 6.12.

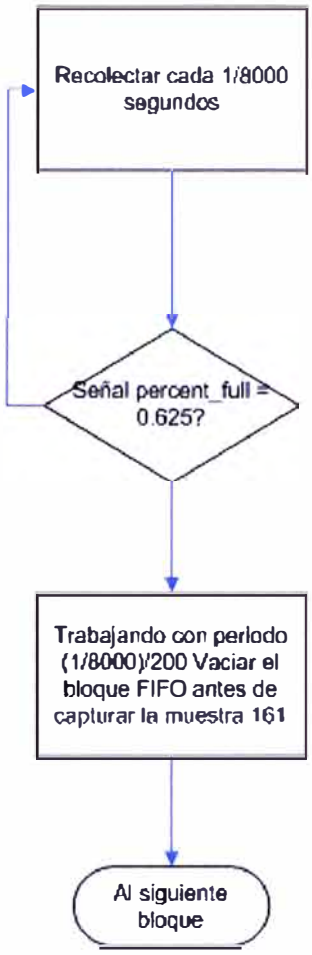


Fig. 6.12 Diagrama de flujo para el control de la adquisición

Las señales de control se muestran a continuación para dos tramas consecutivas. La figura Fig. 6.13 muestra la señal de habilitación de las lecturas a la entrada del bloque FIFO, la frecuencia de lectura es de 200*Tsyscl = 0,000125 o equivalentemente 1/8000 seg.

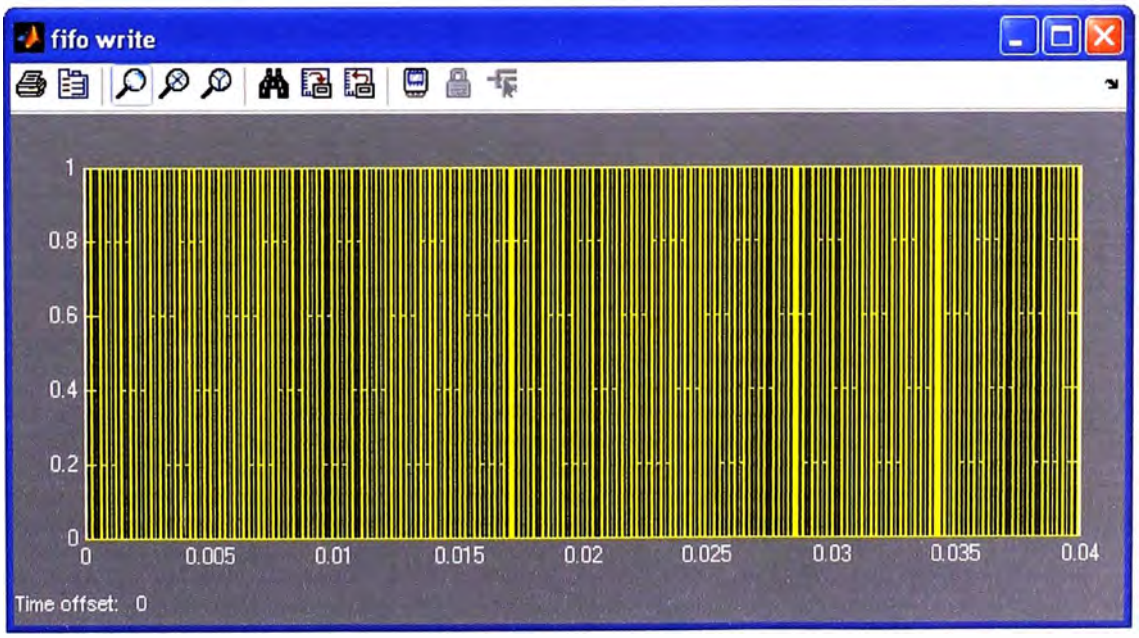


Fig. 6.13 Señal de habilitación de escritura al bloque FIFO.

La figura Fig. 6.14 muestra la señal "percent_full", la cual, como se comentó, marca el nivel de llenado del bloque FIFO. El valor que es de interés es de 0.625 (160 muestras de 256). Cada vez que el bloque FIFO alcanza ese nivel de llenado, se vacía por completo.

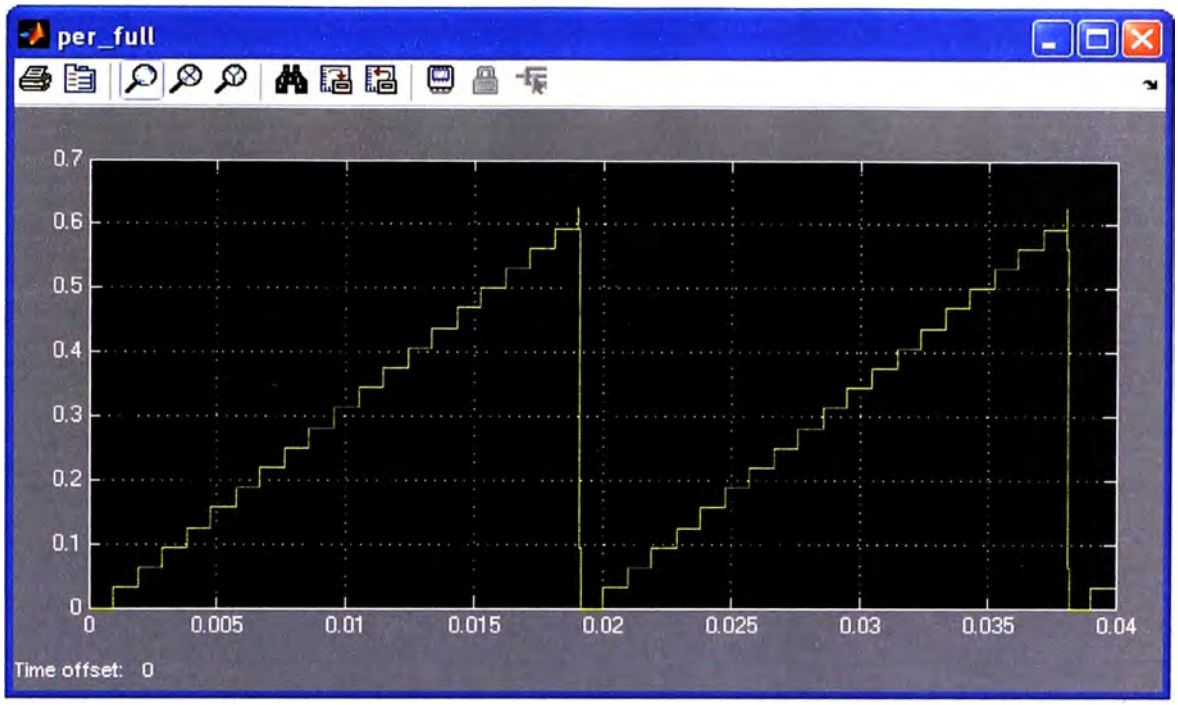


Fig. 6.14 Señal "percent_full" del bloque FIFO.

La figura Fig. 6.15 muestra que cuando el FIFO alcanza el nivel de llenado de 0.625 (160 muestras), la lógica conformada por el bloque m-code, activa la señal de lectura "read_enable", el cual permitirá vaciar el FIFO a la velocidad dada por el reloj Tsysclk, por lo que se consigue que el FIFO quede vacío antes que arrive la muestra 161. Esto se ilustra en la figura Fig. 6.16.

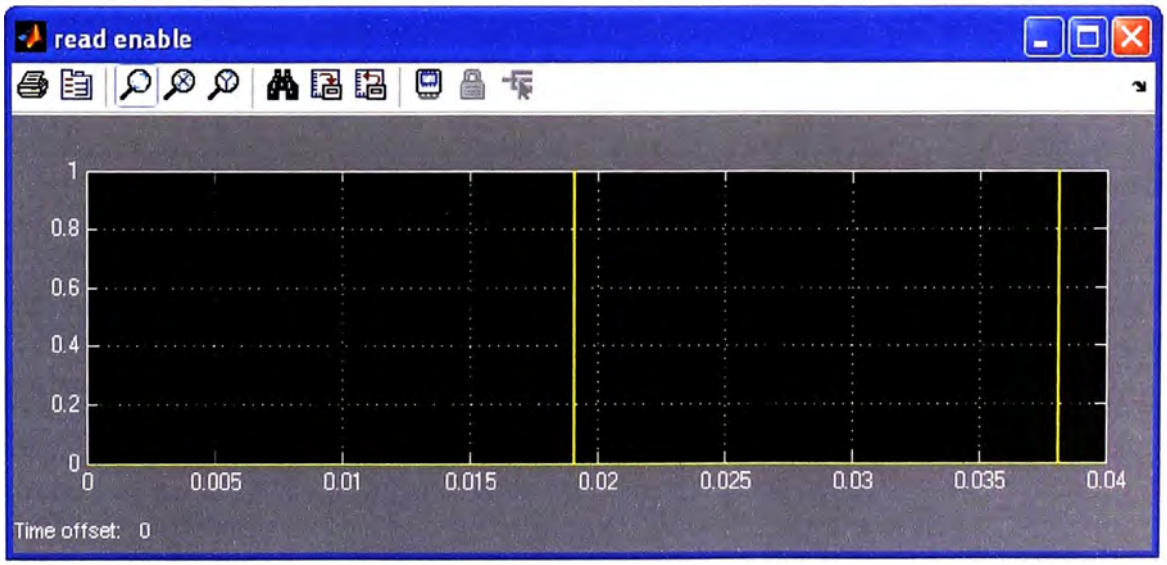


Fig. 6.15 Señal habilitación de lectura para el bloque FIFO.

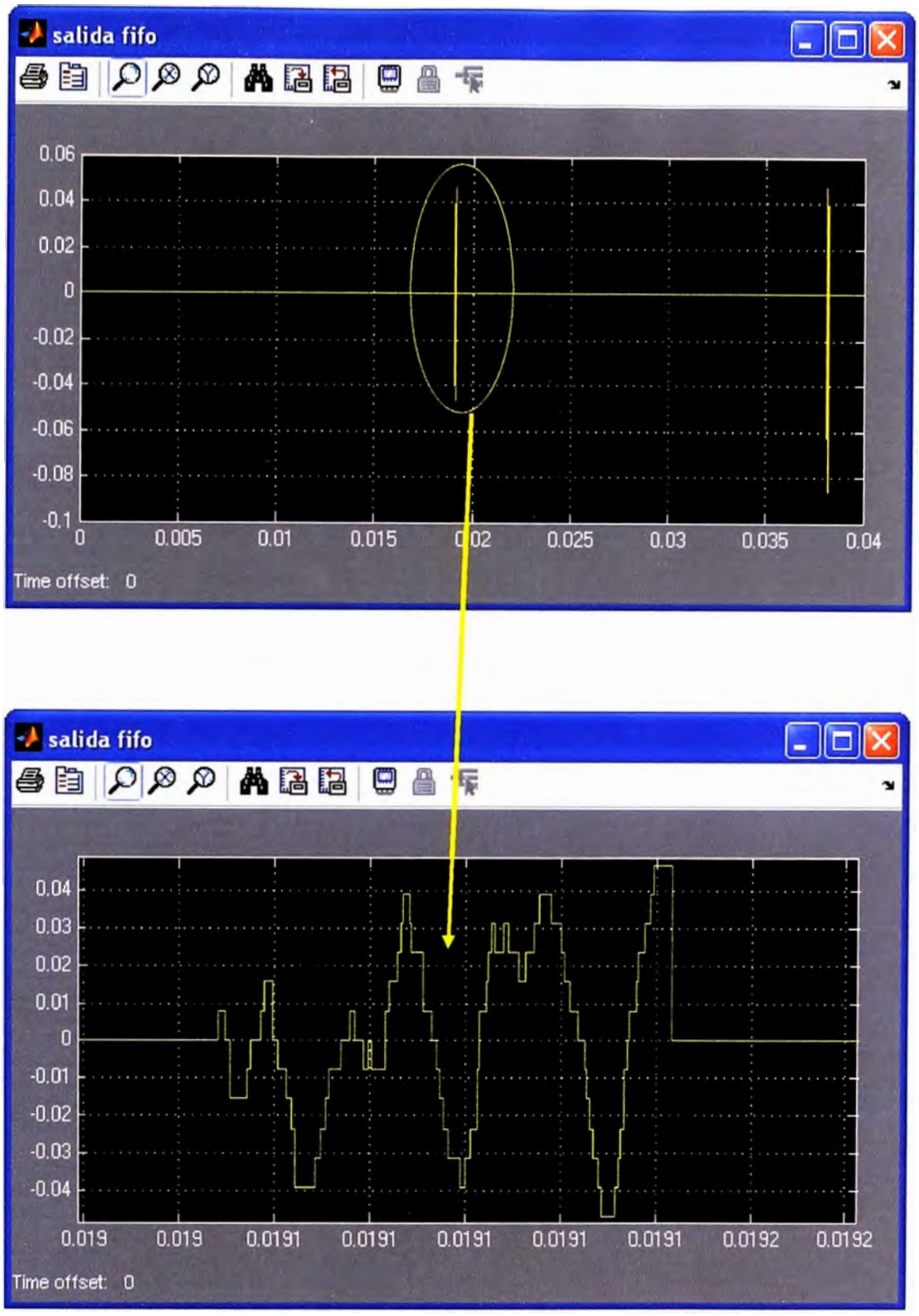


Fig. 6.16 Salida del bloque FIFO, donde se aprecia que las 160 muestras se leen en menos de 1/8000 s (0.125ms)

6.2.5 Filtro pasabajo

El filtro pasabajo es empleado previo al diezmado por 4. La frecuencia de corte es de 800 Hz la cual es adecuada para preservar los componentes de frecuencias correspondiente a la frecuencia fundamental (pitch). Ya que el diezmado es de 4, el filtro correspondiente debe tener frecuencia de corte de máximo 4000/4 = 1000 Hz. En el presente trabajo se ha tomado la frecuencia de corte de 800 Hz.

El filtro diseñado es un filtro de dos etapas de la foma dada en la expresión 6.1:

$$H(z) = \frac{1}{A_1(z)} \frac{1}{A_2(z)} = \frac{0.2069}{(1 - 0.7931z^{-1})} \frac{0.3032}{(1 - 1.4899z^{-1} + 0.7931z^{-2})}$$

(6.1)

La ubicación del filtro se muestra en la Fig. 6.17. Esta dentro de un subsistema.

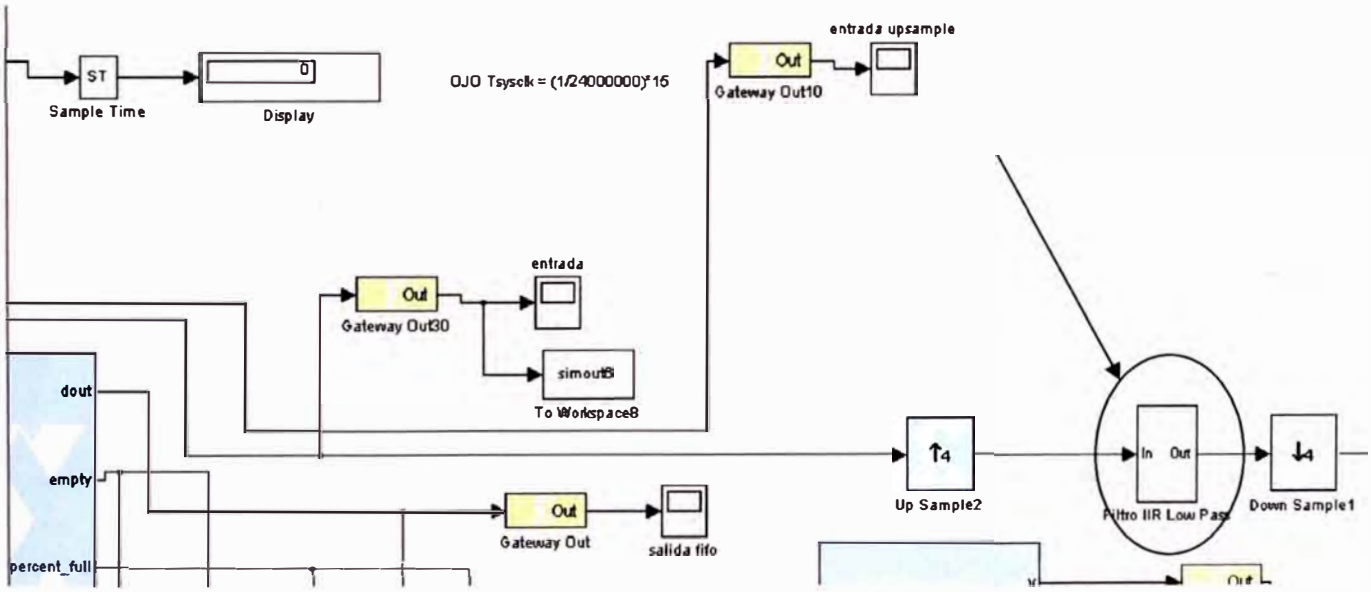


Fig. 6.17 Ubicación del filtro pasabajo.

El filtro implementado en System Generator se muestra en las figuras Fig. 6.18 y Fig. 6.19. La respuesta en frecuencia del filtro se muestra en la figura 6.20.

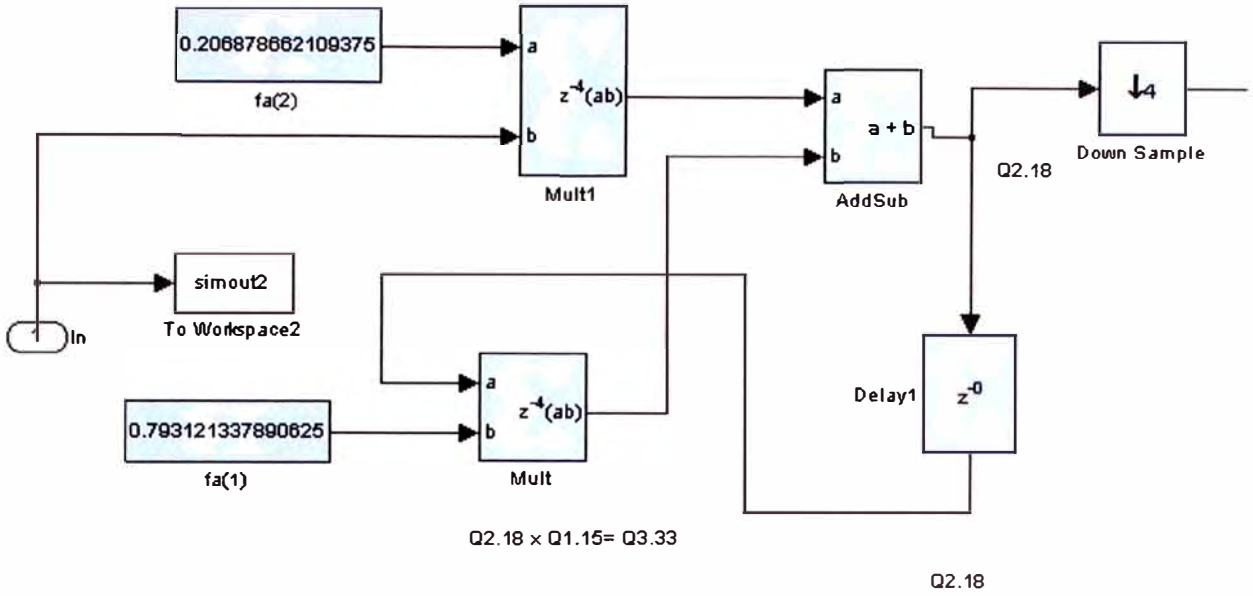


Fig. 6.18 Primera etapa Filtro pasabajo

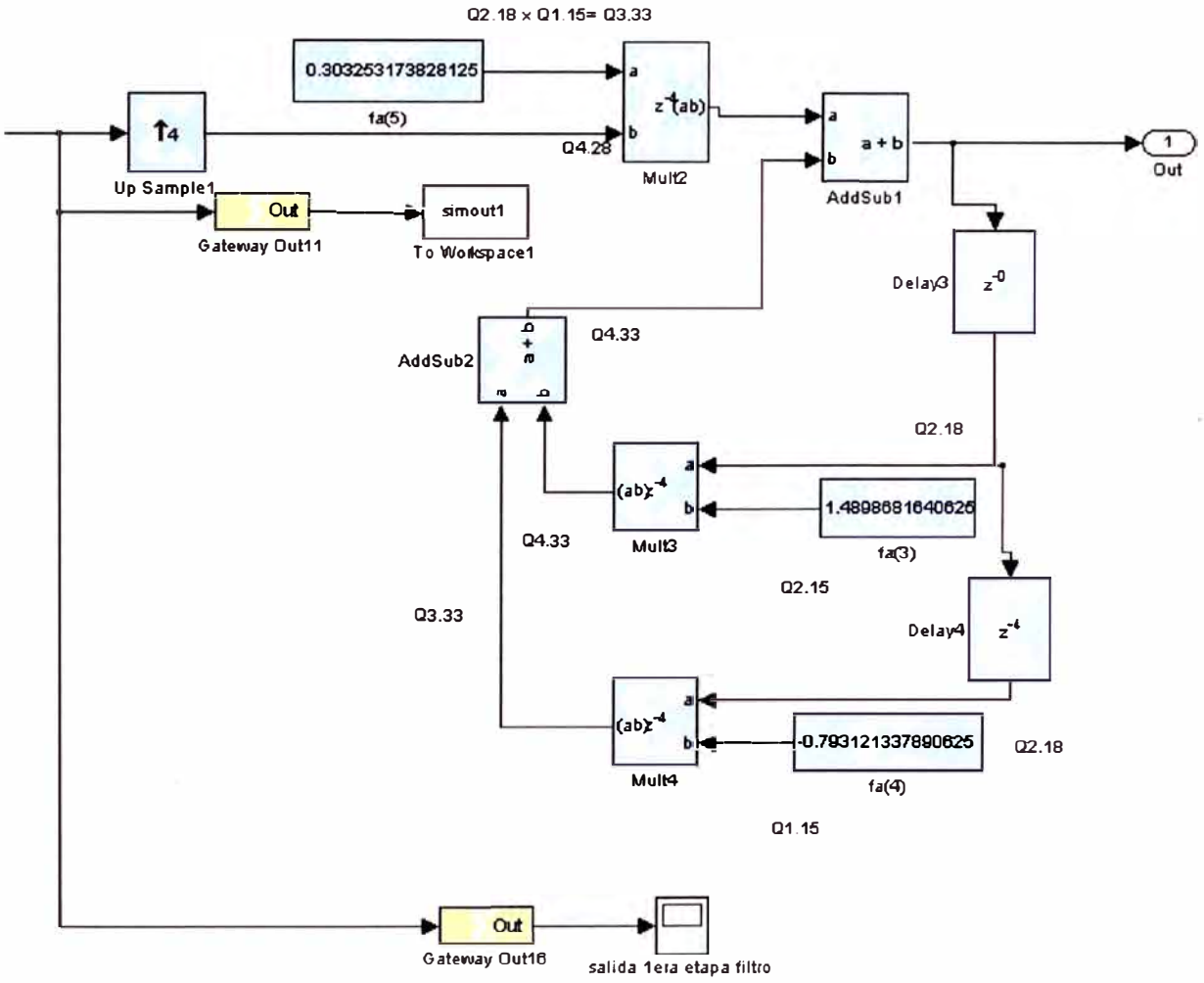


Fig. 6.19 Segunda etapa Filtro pasabajo

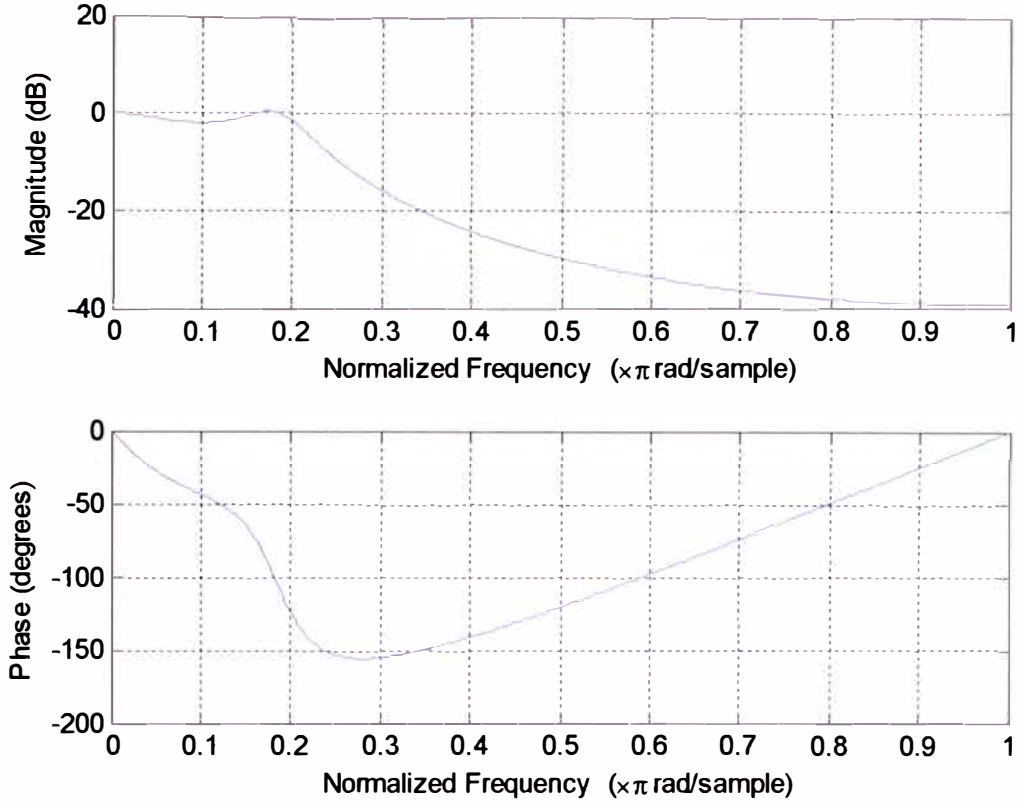


Fig. 6.20 Respuesta en frecuencia filtro pasabajo

Con relación al filtro pasabajo es importante resaltar lo siguiente:

- Ya que el algoritmo SIFT no exige que el filtro sea altamente selectivo en frecuencia, se puede usar el filtro diseñado.
- Un filtro selectivo del tipo FIR por ejemplo requeriría más de 50 coeficientes.
- El filtro diseñado se ejecuta mucho más rápido que filtros de órdenes más altos.
- La no linealidad de la fase no es crítico en los resultados finales.

El filtro implementado opera a 4 veces la frecuencia de muestreo, por lo que se hace uso de los bloques up-sample y down-sample para acondicionar los tiempos.

La salida del filtro pasabajo se muestra en la figura Fig. 6.21 para dos tramas consecutivas de entrada (mostradas en la figura Fig. 6.8).

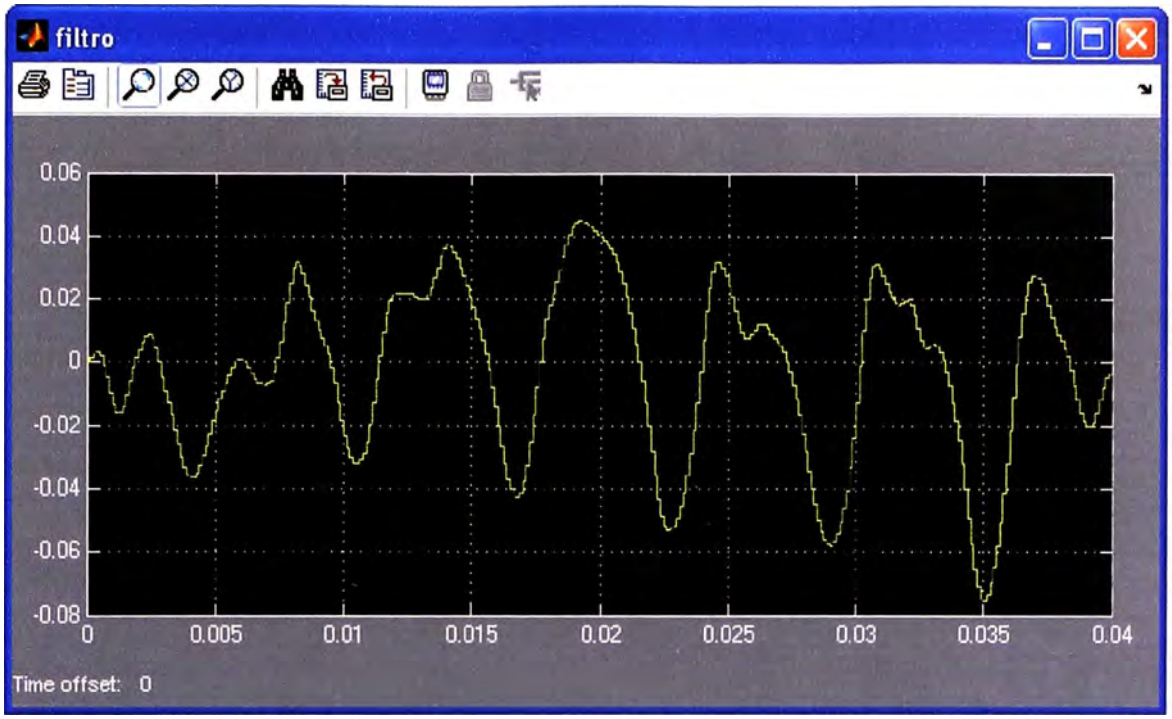


Fig. 6.21 Salida del filtro pasabajo para dos tramas consecutivas (ver Fig. 6.8)

6.2.6 Implementación del Algoritmo SIFT

Una vez que se tienen los datos filtrados por el filtro pasabajo, dichos datos pasan a la etapa que implementa el algoritmo SIFT. Dicha etapa se muestra en la figura Fig. 6.22.

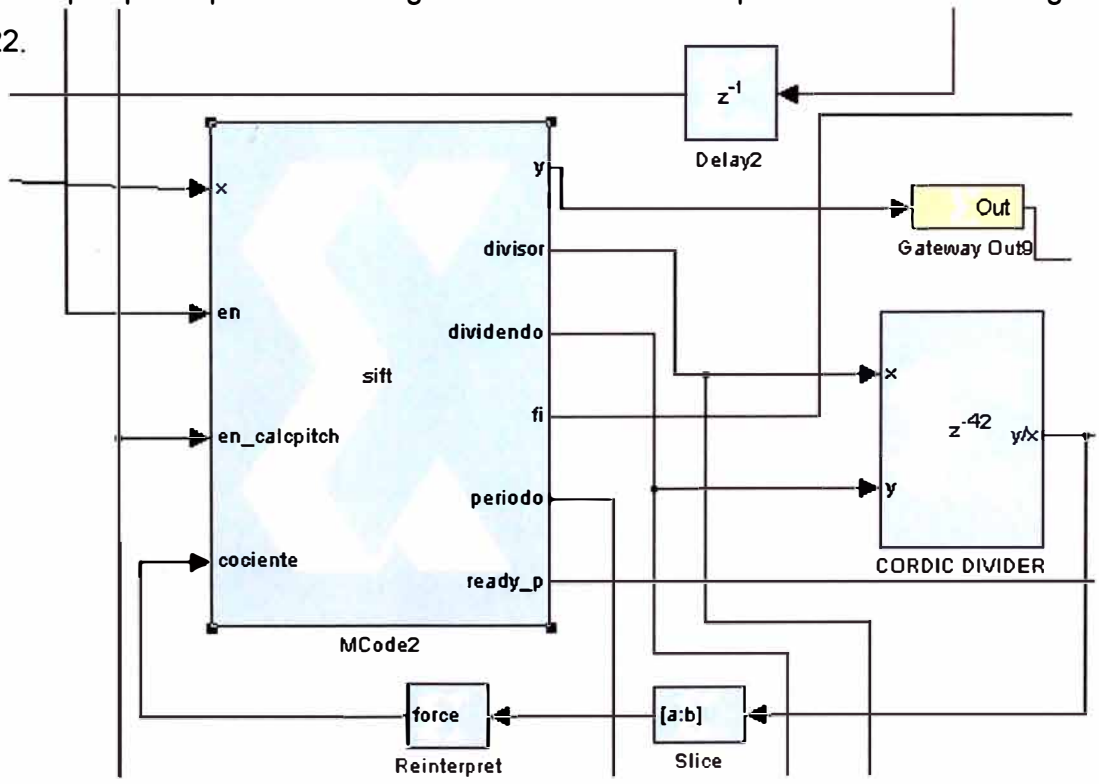


Fig. 6.22 Etapa que implementa el algoritmo SIFT.

Como se observa en la figura Fig. 6.22, se incluye un bloque Mcode1, un bloque CORDIC DIVIDER, un bloque Slice y un bloque Reinterpret. El bloque MCode aplica valores de tipo punto fijo de Xilinx como entradas para una función M. Dicha función M debe estar diseñada bajo los lineamientos, restricciones y comandos dados por Xilinx para el bloque MCode.

Este bloque provee una manera conveniente para implementar máquinas de estados finitos, lógica de control y sistemas de carga computacional variable. En la presente implementación, todo el bloque correspondiente al algoritmo SIFT se implementó como una máquina de estados. Esto se describe más adelante. El bloque Mcode está asociado a un archivo .m llamado "sift.m" tal como se muestra en la figura Fig. 6.23

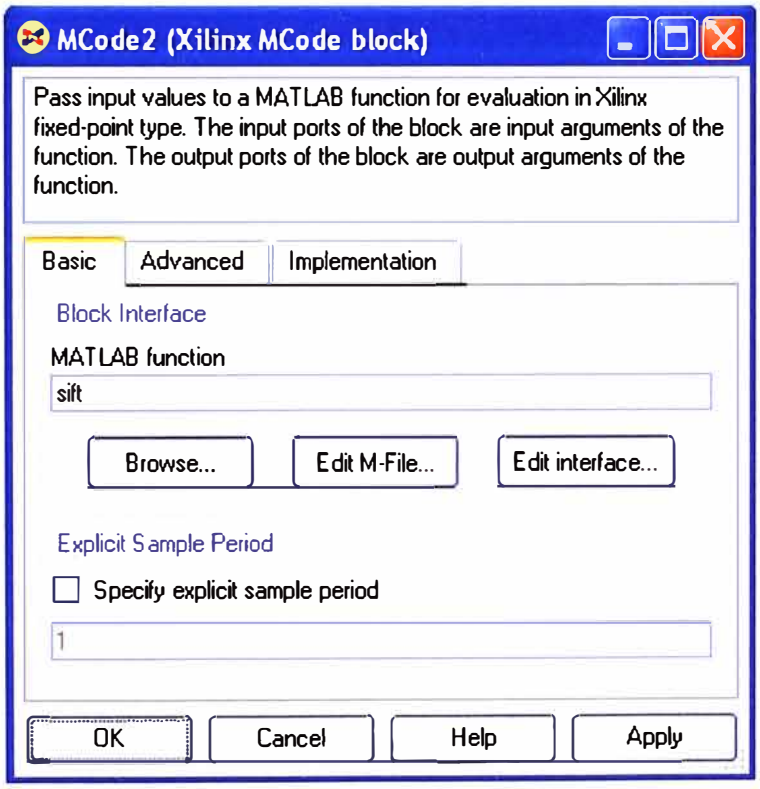


Fig. 6.23 Función .m asociada al bloque Mcode2.

La máquina de estados que implementa el algoritmo SIFT consta de 40 estados y su diagrama general se muestra en la Fig. 6.24 donde se puede apreciar la ruta que sigue la aplicación dentro de los estados. Dicha máquina de estados se describe a continuación. El código completo se encuentra en el CD en la ruta "..*System Generator*\ *sift.m*".

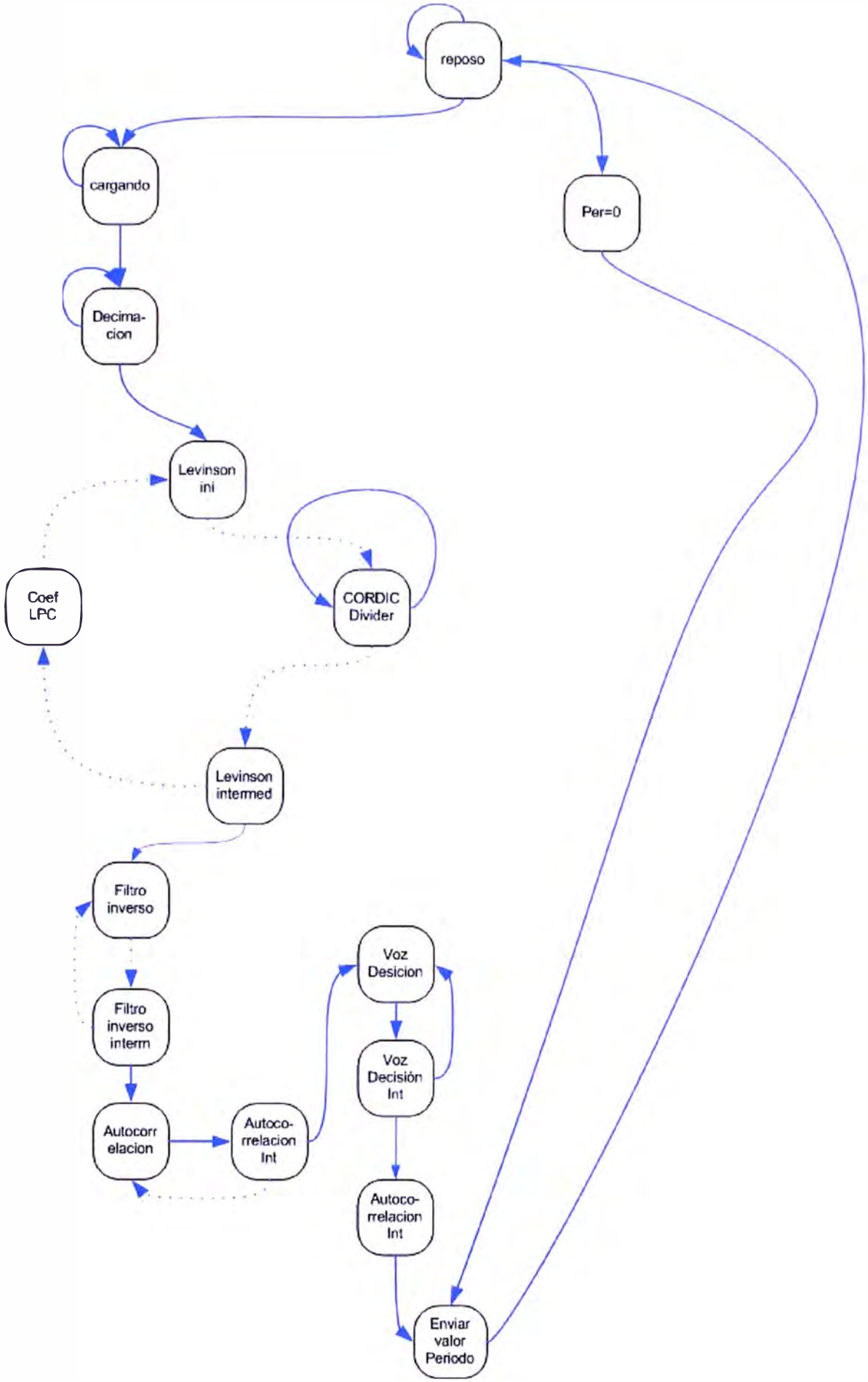


Fig. 6.24 Máquina de estados que implementa el algoritmo SIFT.

La máquina de estados mostrada en la Fig. 6.24 es un esquema simplificado ya que hay muchos estados intermedios que se representan por las líneas punteadas. Cada estado se ejecuta en un ciclo de reloj, que en nuestro caso es de Tsysclk = (1/24000000)*15. Todos los datos manejados por la máquina de estados están con el formato de Xilinx.

Las variables de punto fijo que se usan en la máquina de estados, se definen con las expresiones que se muestran a continuación:

Para definir una variable entera unidimensional sin signo y de 8 bits, se usa:

```
persistent flag, flag= xl_state(0,{xlUnsigned, 8, 0});
```

Para definir una variable fraccional unidimensional con signo, de 20 bits y con 16 bits de parte fraccional, se usa:

```
persistent a1, a1 = xl state(0, {xlSigned, 20, 16} );
```

Para definir una variable fraccional multidimensional (array) con signo, de 20 bits y con 16 bits de parte fraccional, se usa:

```
persistent a2, a2 = xl_state(zeros(1,11), {xlSigned, 20, 16} );
```

Para definir una variable booleana unidimensional, se usa:

```
persistent rper, rper= xl_state(false,{xlBoolean, 1, 0});
```

Para que el bloque MCode trabaje sin problemas, todas las variables involucradas deben estar definidas con la función xl_state, propia para dicho bloque Xilinx. Así mismo se debe tener en cuenta que solo se puede emplear un conjunto muy limitado de funciones propias del MATLAB. Por otro lado, Xilinx proporciona una serie de funciones para trabajar con valores de punto fijo, de entre dichas funciones podemos mencionar las siguientes:

Para dar formato o cambiar de formato: xfix().
Para saber el formato de una determinada variable: xl_arith(), xl_nbits(), xl_binpt()
Para tratamiento de bits: xl_or(), xl_and(), xl_xor(), xl_not, xl_lsh(), xl_rsh()

Para extraer partes de una determinada variable: xl_slice()

Para concatenar bits: xl_concat()

Para definir estados o variables persistentes: xl_state()

En el diseño de los estado se ha tenido en cuenta que el divisor cordico (Cordic Divider) tiene un retardo de 42 ciclos de reloj antes de arrojar un resultado válido. Esto es debido a que este divisor divide dos números de punto fijo de longitud arbitraria y arroja un resultado con la misma precisión. La configuración para el divisor córdico se muestra en la figura Fig. 6.25. Dicha configuración arroja un retardo de 42 ciclos de reloj.

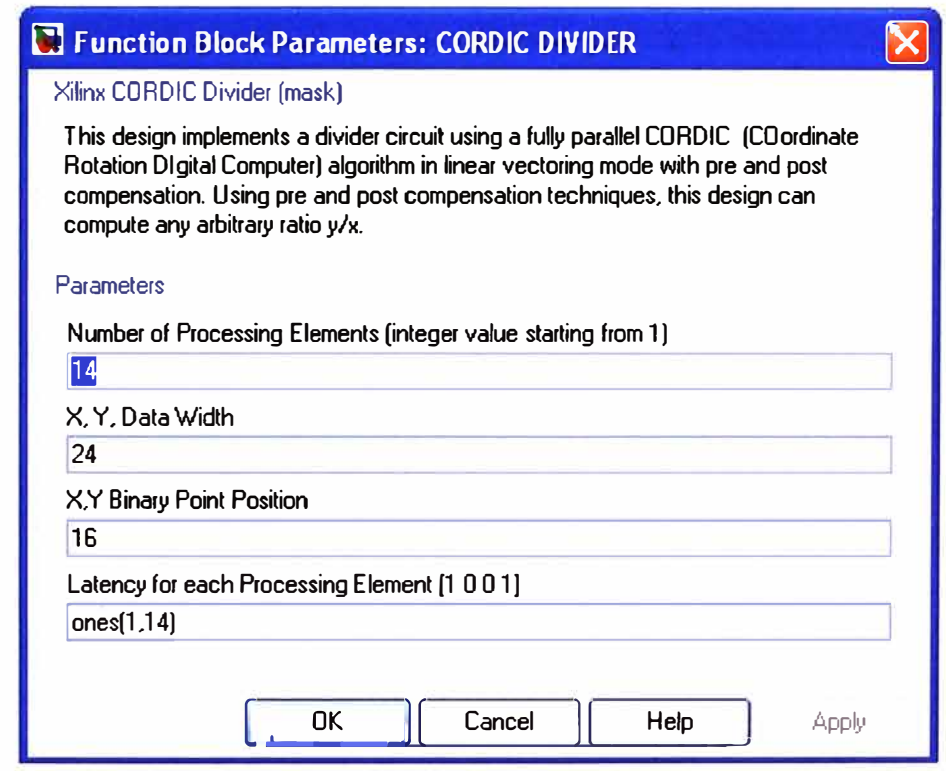


Fig. 6.25 Configuración del divisor córdico.

Como se puede apreciar en la figura Fig. 6.22 el bloque Mcode presenta 4 entradas y 6 salidas, las cuales se describen a continuación:

Entradas:

x: entrada de los datos procedentes del FIFO ubicado a la salida del bloque Filtro Pasabajo.

en: indica que los datos están listos en el FIFO ubicado luego del filtro pasabajo

en_calcpitch: esta señal de entrada indica que la energía calculada de la trama ha sobrepasado el umbral y que el calculo del pitch puede proceder.

- **cociente**: por esta señal de entrada ingresa el resultado de la división realizada por el divisor córdico (Cordic divider) luego de ser acondicionada por los bloques slice y reinterpret.

  Salidas:
- **divisor**: primera entrada al divisor córdico.
- **dividendo**: segunda entrada al divisor córdico.
- **ready_per:** indica a la siguiente etapa que el periodo ha sido calculado con éxito y esta disponible por el puerto periodo.
- **periodo:** arroja el periodo calculado, recordar que es 0 si la trama es "sorda".
- Las otras salidas se usan con fines de visualización de variables internas y comprobación del correcto funcionamiento de dicho bloque,

A continuación se muestran segmentos de código que ilustran la implementación realizada, el código completo puede verse en el CD bajo la ruta "..*System Generator*\ *sift.m*".

```
case cargando %estado

    if en == true
     w_y.push_front_pop_back(x);
     w_y_t.push_front_pop_back(x);
     state=int1;
     p=xfix(out_prec,0.25);
    else
     state=bridge1;
    end
      .
      .
      .
      .
case levinson4

      suma=1-ki(0)*ki(0);
      suma=suma*r1(10);
      E=xfix(r1_prec, suma);

      suma=r1(8)-a1(0)*r1(9);
      num=xfix(r1_prec, suma);

      div=num;
      divb=E;
      flag=0;
      state=levinson5;
       .
       .
```

```
      .
case levinson16

   f(flag)=xfix(fi_prec, 0);
   b(flag)=xfix(fi_prec, 0);
   bp(flag)=xfix(fi_prec, 0);
   flag=flag+1;
   state=levinson16;

   if (flag==xfix({xlUnsigned, 8, 0},5))
        state=levinson17;

   end
      .

      .

      .
case levinson20
    w_fi(i) = f(4);
    w_fi_t(i) = f(4);
    i=i+1;
    state=levinson17
    if (i==xfix({xlUnsigned, 8, 0},60))
        flag=0;
        state=levinson21;

    end
      .

      .

      .
case inter4

    q=xfix(r1_prec,I_V(i));

    if (i==0)
        a_f=xfix(af_prec,-3);
    elseif (i==1)
        a_f=xfix(af_prec,-2);
    elseif (i==2)
        a_f=xfix(af_prec,-1);
    elseif (i==3)
        a_f=xfix(af_prec,1);
    elseif (i==4)
        a_f=xfix(af_prec,2);
    elseif (i==5)
        a_f=xfix(af_prec,3);
    else
        a_f=xfix(af_prec,0);
    end

   sum=4*rpos +a_f;
   per= xfix(per_prec,sum);

   if (r2(40)==0)
    rval=xfix(rval_prec,1);
```

```
      .
case levinson16

   f(flag)=xfix(fi_prec, 0);
   b(flag)=xfix(fi_prec, 0);
   bp(flag)=xfix(fi_prec, 0);
   flag=flag+1;
   state=levinson16;

   if (flag==xfix({xlUnsigned, 8, 0},5))
        state=levinson17;

   end
      .

      .

      .
case levinson20
    w_fi(i) = f(4);
    w_fi_t(i) = f(4);
    i=i+1;
    state=levinson17
    if (i==xfix({xlUnsigned, 8, 0},60))
        flag=0;
        state=levinson21;

    end
      .

      .

      .
case inter4

    q=xfix(r1_prec,I_V(i));

    if (i==0)
        a_f=xfix(af_prec,-3);
    elseif (i==1)
        a_f=xfix(af_prec,-2);
    elseif (i==2)
        a_f=xfix(af_prec,-1);
    elseif (i==3)
        a_f=xfix(af_prec,1);
    elseif (i==4)
        a_f=xfix(af_prec,2);
    elseif (i==5)
        a_f=xfix(af_prec,3);
    else
        a_f=xfix(af_prec,0);
    end

   sum=4*rpos +a_f;
   per= xfix(per_prec,sum);

   if (r2(40)==0)
    rval=xfix(rval_prec,1);
```

```
else
 div=rmax;
 divb=r2(40);
 flag=0;
end

state=inter5;
```

A continuación se muestra los resultados de la salida del puerto "periodo" del bloque que implementa el algoritmo SIFT. Como se observa en la figura Fig. 6.26, en este caso se muestran los resultados de análisis sobre un segmento de voz compuesto por 25 tramas de análisis. Como se sabe, cada trama de análisis es de 160 muestras, por lo que la duración del segmento de análisis es de 25*160*(1/8000) = 0.5 segs. La figura Fig. 6.26 muestra también como es que el algoritmo SIFT realiza una especia de "tracking" a lo largo del segmento y arrojando como resultado el valor del periodo "pitch" para dicho segmento de voz y esto a su vez determina si dicho segmento es "sonoro" o "sordo".

Cuando el segmento es "sordo", el valor del periodo es de cero, tal como se puede apreciar en las primeras tramas de análisis. Por otro lado, cuando el segmento es "sonoro", dicho valor se refleja en el periodo. En el caso de la figura Fig. 6.26 el primer valor del periodo "pitch" es de 63, esto es 63 veces el tiempo de muestreo que equivale a 63*(1/8000) = 0,007875 segundos o una frecuencia "pitch" de 126.98 Hz.

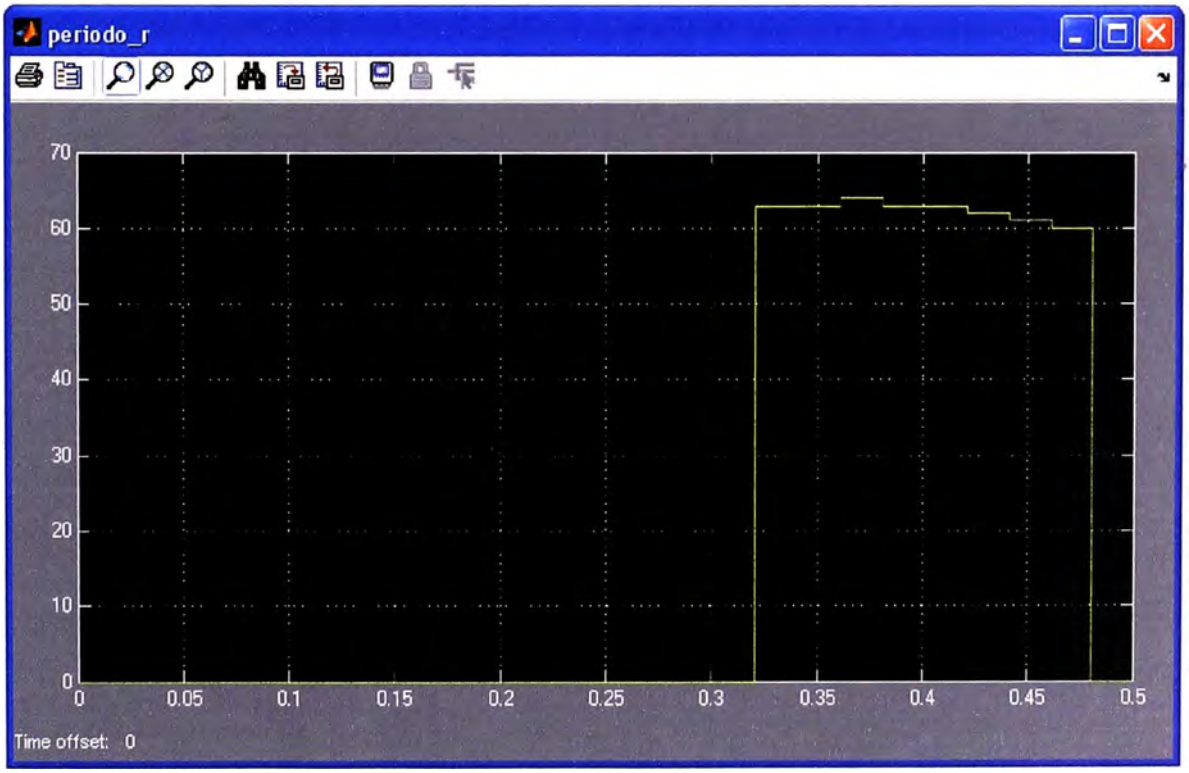


Fig. 6.26 Tracking del "pitch" para un segmento de análisis de duración 0.5 segs.

```
else
 div=rmax;
 divb=r2(40);
 flag=0;
end

state=inter5;
```

A continuación se muestra los resultados de la salida del puerto "periodo" del bloque que implementa el algoritmo SIFT. Como se observa en la figura Fig. 6.26, en este caso se muestran los resultados de análisis sobre un segmento de voz compuesto por 25 tramas de análisis. Como se sabe, cada trama de análisis es de 160 muestras, por lo que la duración del segmento de análisis es de 25*160*(1/8000) = 0.5 segs. La figura Fig. 6.26 muestra también como es que el algoritmo SIFT realiza una especia de "tracking" a lo largo del segmento y arrojando como resultado el valor del periodo "pitch" para dicho segmento de voz y esto a su vez determina si dicho segmento es "sonoro" o "sordo".

Cuando el segmento es "sordo", el valor del periodo es de cero, tal como se puede apreciar en las primeras tramas de análisis. Por otro lado, cuando el segmento es "sonoro", dicho valor se refleja en el periodo. En el caso de la figura Fig. 6.26 el primer valor del periodo "pitch" es de 63, esto es 63 veces el tiempo de muestreo que equivale a 63*(1/8000) = 0,007875 segundos o una frecuencia "pitch" de 126.98 Hz.

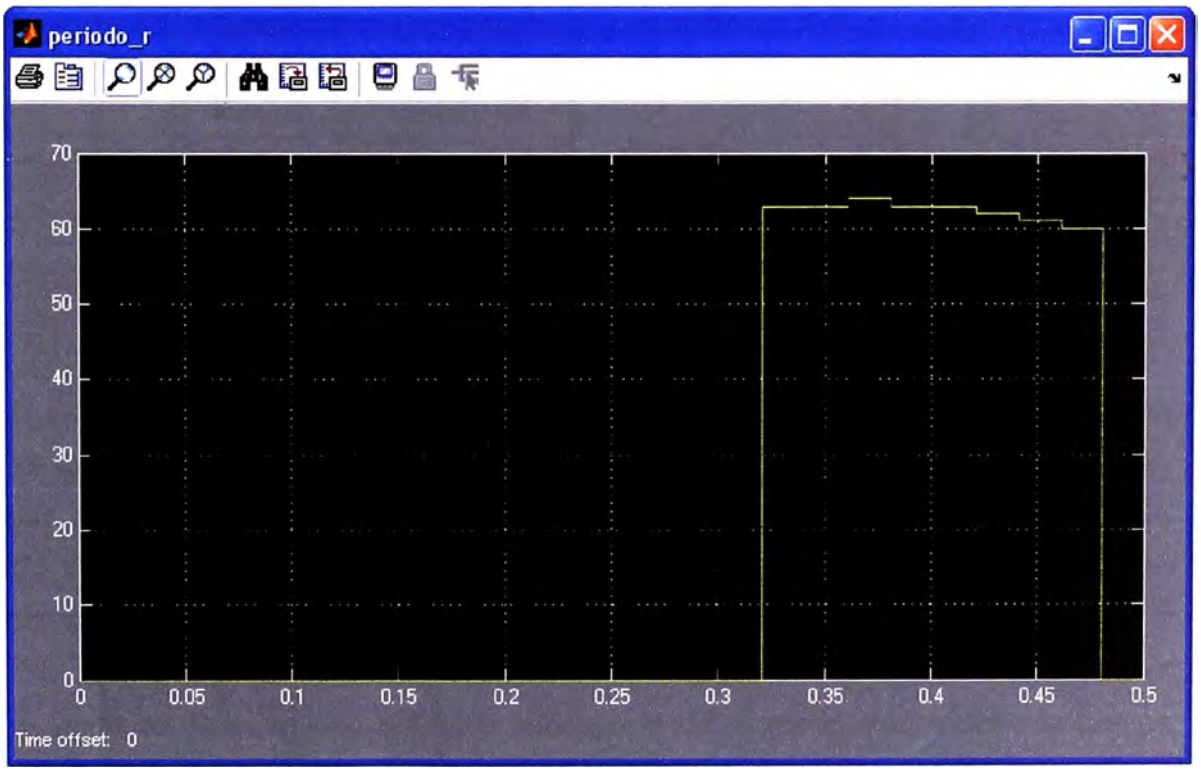


Fig. 6.26 Tracking del "pitch" para un segmento de análisis de duración 0.5 segs.

6.2.7 Bloque de calculo de la ganancia y de los coeficientes RC

Al igual que en el bloque Sift, el bloque que ejecuta el cálculo de la se implementa con un bloque Mcode, el cual, como se ha visto, ofrece una manera eficiente de implementar máquinas de estado finita. La figura Fig. 6.27 muestra la ubicación de este bloque dentro del diseño completo.

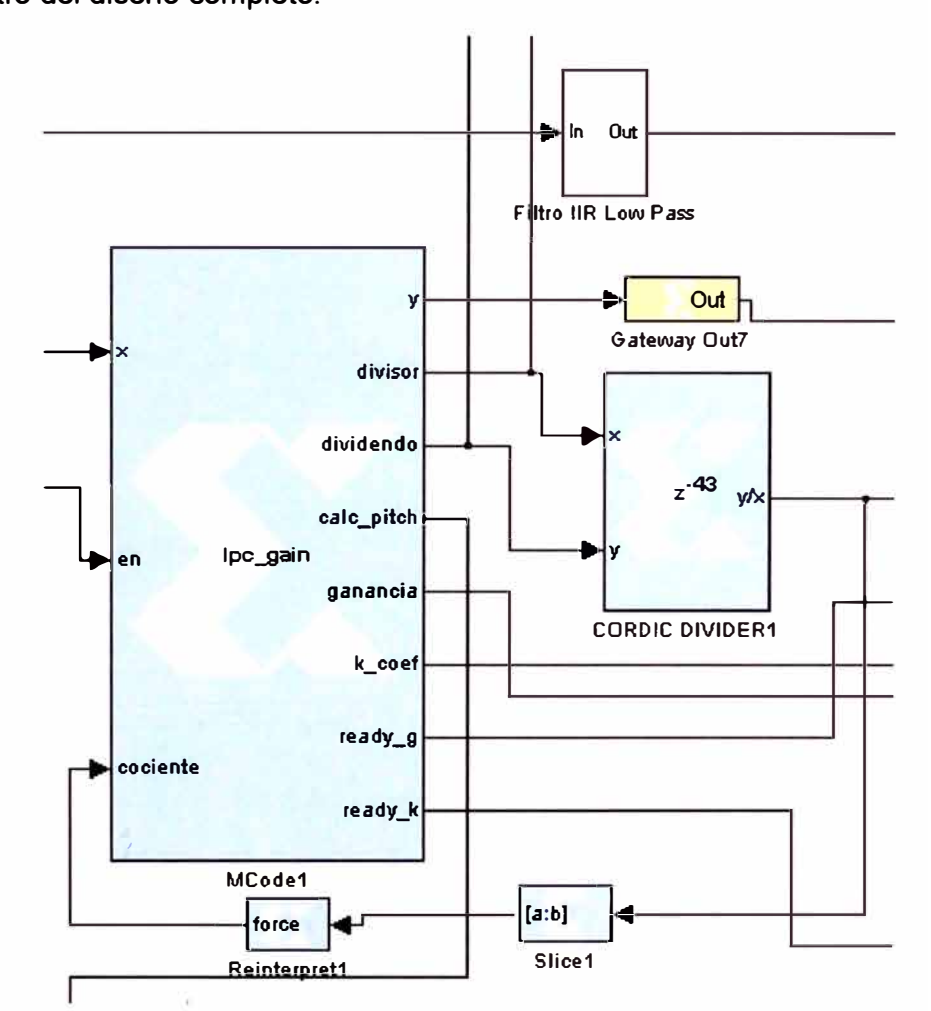


Fig. 6.27 Etapa que calcula los coeficientes RC y la ganancia.

Como se observa en la figura Fig. 6.27, además del bloque Mcode1, esta etapa incluye un divisor córdico, un bloque "slice" y un bloque "reinterpret". Estos 2 últimos bloques se usan para dar un correcto formato al resultado entregado por el bloque "Cordic Divider", de tal forma que la máquina de estado implementada en el bloque Mcode1 pueda trabajar a precisión máxima.

Como se sabe, un bloque Mcode tiene que estar asociado a un archivo .m que es el que implementa la función, en este caso, la función de máquina de estado. La función asociada es la lpc_gain.m tal como se puede apreciar en la figura Fig. 6.28.

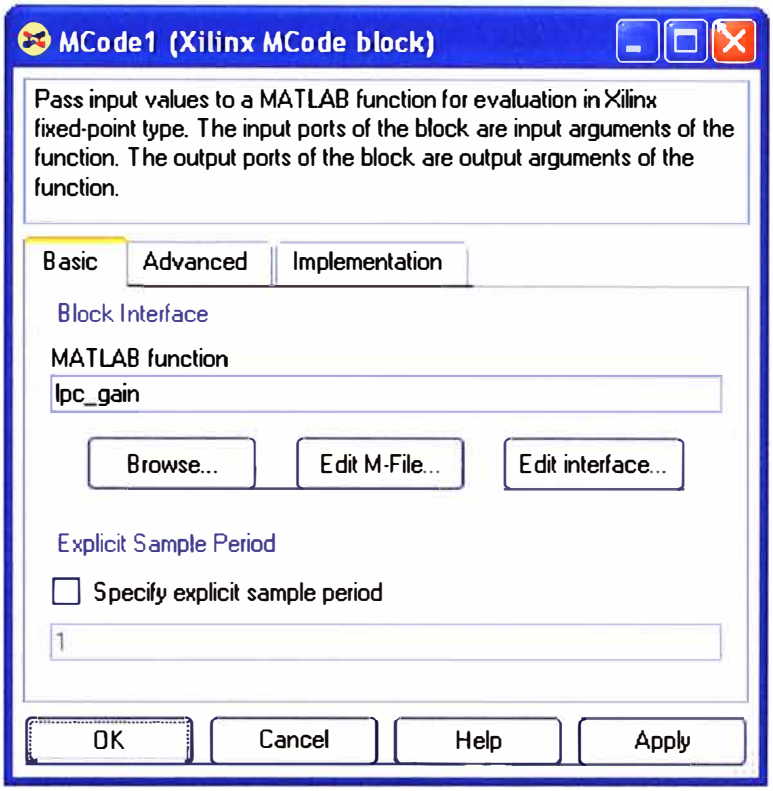


Fig. 6.28 Función .m asociada al bloque Mcode2.

La máquina de estados que implementa el calculo de la ganancia, los 10 coeficientes LPC así como también la cuantización de dichos valores, tiene 68 estados. Como se observa existen más estados que la máquina de estados que implementa el algoritmo Sift. Esto es debido a que en este caso se tienen que calcular 10 valores de coeficientes LPC, mientras que en el caso del algoritmo SIFT solo se necesitan calcular 4 coeficientes LPC. También, en este bloque es necesaria la implementación de un algoritmo de cálculo de raíz cuadrada de valores en punto fijo, cosa que no fue necesaria en el algoritmo SIFT.

Dicho segmento algorítmico para la raíz cuadrada requiere muchos estados los cuales sumados a los estados usados para el algoritmo de Levinson, autocorrelación y al cálculo de los 10 coeficientes LPC, arroja el número de 68 estados.

La figura Fig. 6.29 muestra el diagrama de estado general que implementa el bloque del cálculo de la ganancia y de los 10 coeficientes LPC. Como se mencionó, esta vez se tiene el algoritmo para la raíz cuadrada el cual requiere 19 estados.

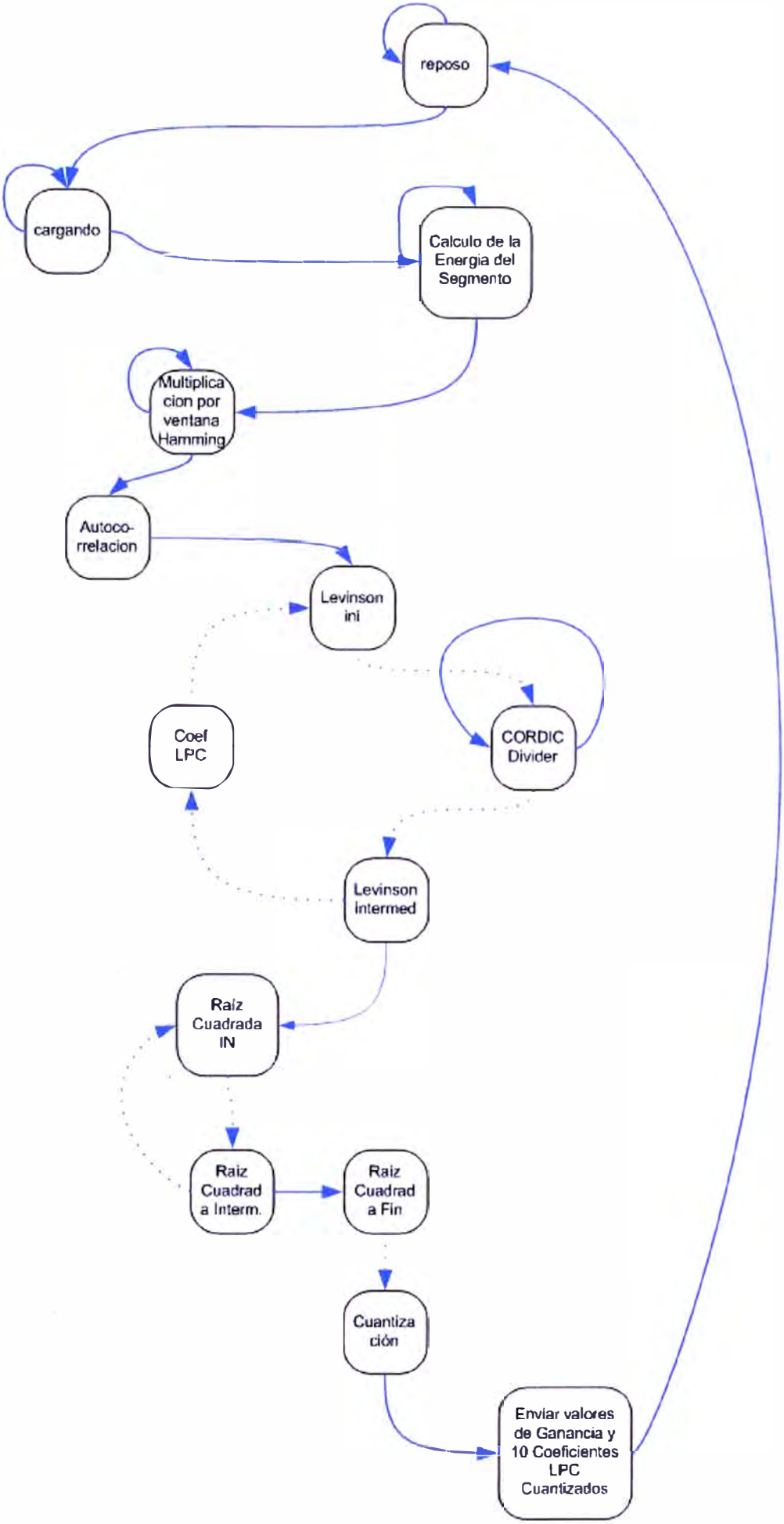


Fig. 6.29 Máquina de estados que implementa el calculo de la ganancia y Coef. LPC

Como se puede apreciar en la figura Fig. 6.27 el bloque Mcode1 presenta 3 entradas y 8 salidas, las cuales se describen a continuación:

Entradas:

x: entrada de los datos procedentes del bloque FIFO ubicado a la entrada del diseño.
en: indica que los datos del FIFO están listos para ser recibidos.
cociente: por esta señal de entrada ingresa el resultado de la división realizada por el divisor córdico (Cordic divider) luego de ser acondicionada por los bloques slice y reinterpret.

Salidas:

y: datos de salida para visualización.
divisor: primera entrada al divisor córdico.
dividendo: segunda entrada al divisor córdico.
calc_pitch: indica al bloque que implementa el algoritmo SIFT (Mcode2) que la trama de análisis tiene suficiente energía (superó el umbral) como para ameritar el cálculo vía el algoritmo SIFT.
ganancia: arroja el valor cuantizada de la ganancia.
k_coef: arroja los coeficientes LPC en la forma de coeficientes de reflexión.
ready_g: avisa al siguiente bloque que el cálculo de la ganancia ha culminado satisfactoriamente.
ready_k: avisa al siguiente bloque que los 10 coeficientes LPC (RC) han sido correctamente calculados y están listos para ser leídos.

Cabe resaltar que todos los bloques pertenecientes al diseño se ejecutan de manera paralela, es decir todos están activos y realizando una tarea en un determinado tiempo, ya sea calculando, esperando datos o en un bucle de reposo. Por lo cual, no hay un bloque inactivo en algún instante de tiempo.

La configuración del divisor córdico para este caso es similar al de la figura Fig. 6.25 solo que ahora presenta un retardo de 43 ciclos de reloj.

A continuación se muestran las salidas para los valores de ganancia y de los coeficientes LPC (RC).

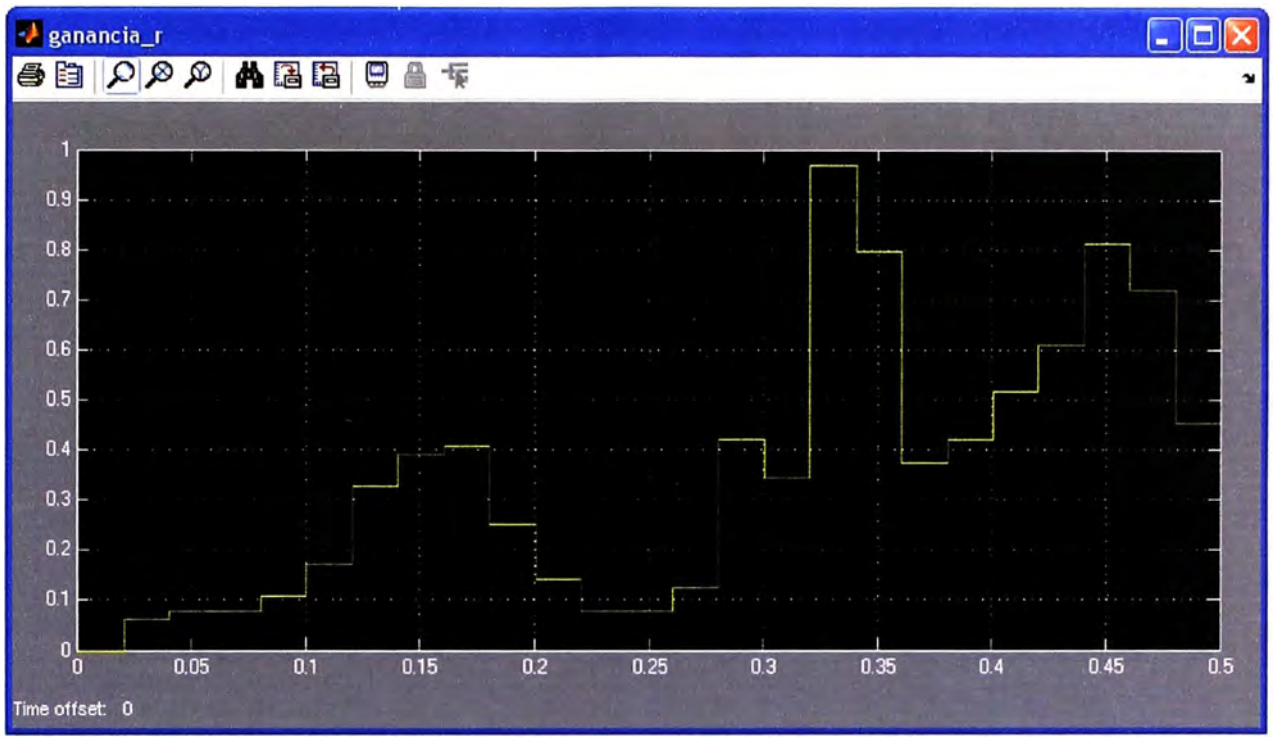


Fig. 6.30 Ganancia para 25 tramas (160 muestras / trama) consecutivas.

Como se observa en la figura Fig. 6.30, la ganancia presenta valores desde cercanos a cero hasta cercanos (o mayores) a uno. Dichos valores dependen de la amplitud y característica de la señal de entrada para dicho segmento de análisis.

Como se aprecia en las figuras Fig. 6.31 y Fig. 6.32, los valores de los coeficientes 10 LPC (RC) calculados se presentan al final de la trama de análisis casi de forma inmediata. Esto es obvio porque las partes de procesamiento (Sift, ganancia y coeficientes) se realizan a una velocidad 200 veces más rápidas que la velocidad de adquisición de las muestras.

También se puede apreciar en la figuras Fig. 6.31 y Fig. 6.32 que los valores de los coeficientes RC están limitados al intervalo [0, 1] lo que es correcto según la teoría, ya que garantiza estabilidad en el filtro de síntesis

El código completo se puede encontrar en el CD en la ruta "..*System Generator*\ *lpc_gain.m*".

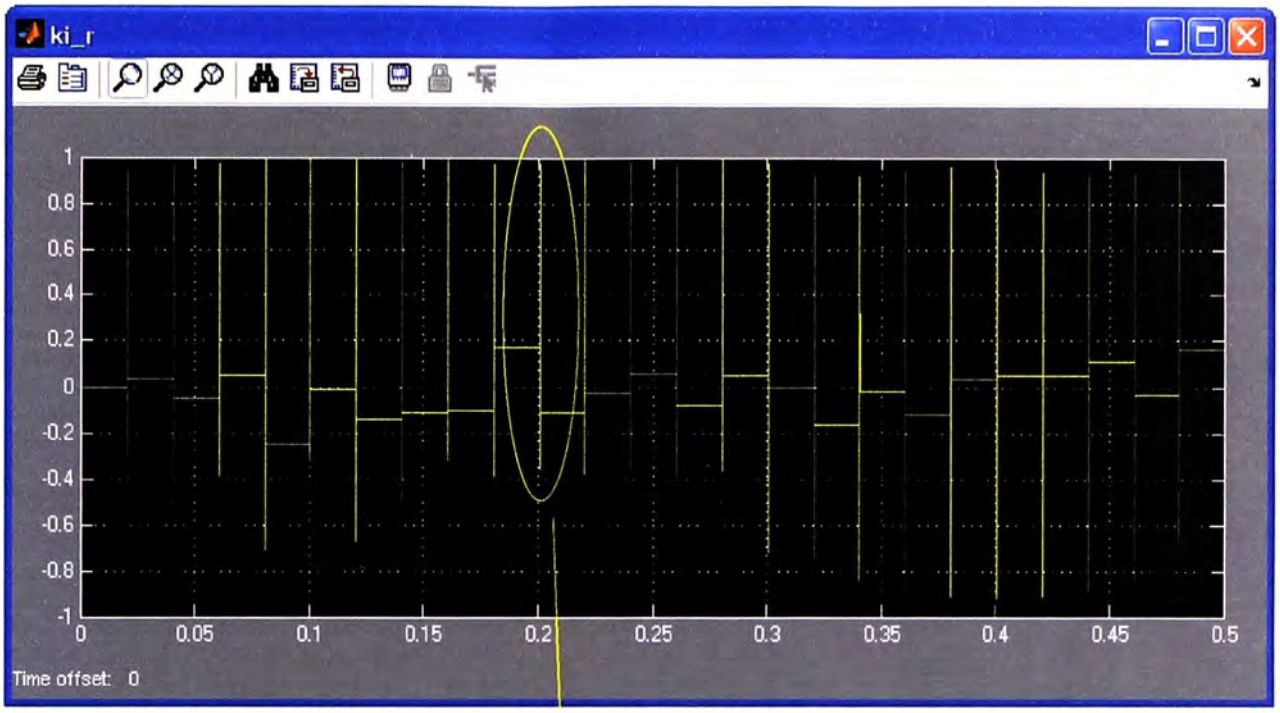


Fig. 6.31 Valores de los 10 coeficientes LPC (RC) al final de cada trama (25 tramas)

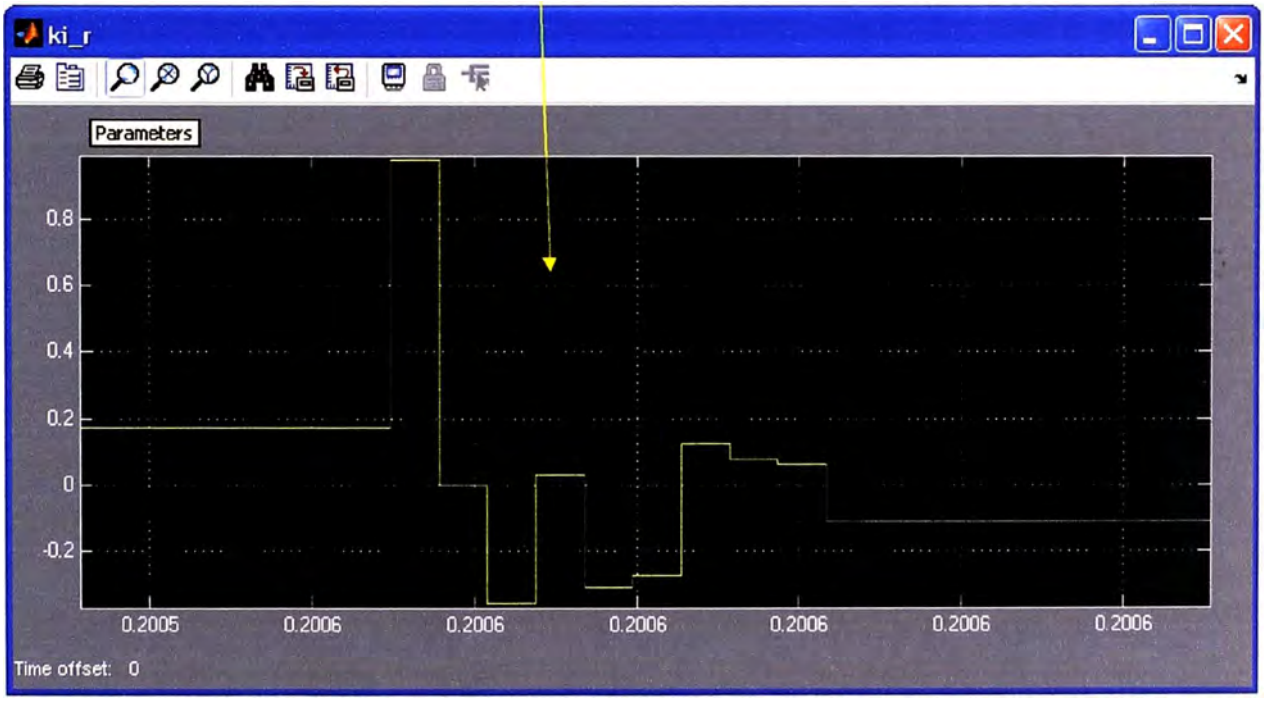


Fig. 6.32 Valores de los 10 coeficientes LPC (RC) ampliados para una trama.

6.2.8 Bloque decodificador

Al igual que los bloques que implementan el algoritmo SIFT, el calculo de la ganancia y el cálculo de los coeficientes LPC (RC), el bloque decodificador se implementa también como un bloque Mcode (Mcode3). Como se sabe, el bloque Mcode proporciona una manera fácil de implementar máquinas de estado finita. La ubicación del bloque dentro del diseño se muestra en la figura Fig. 6.33.

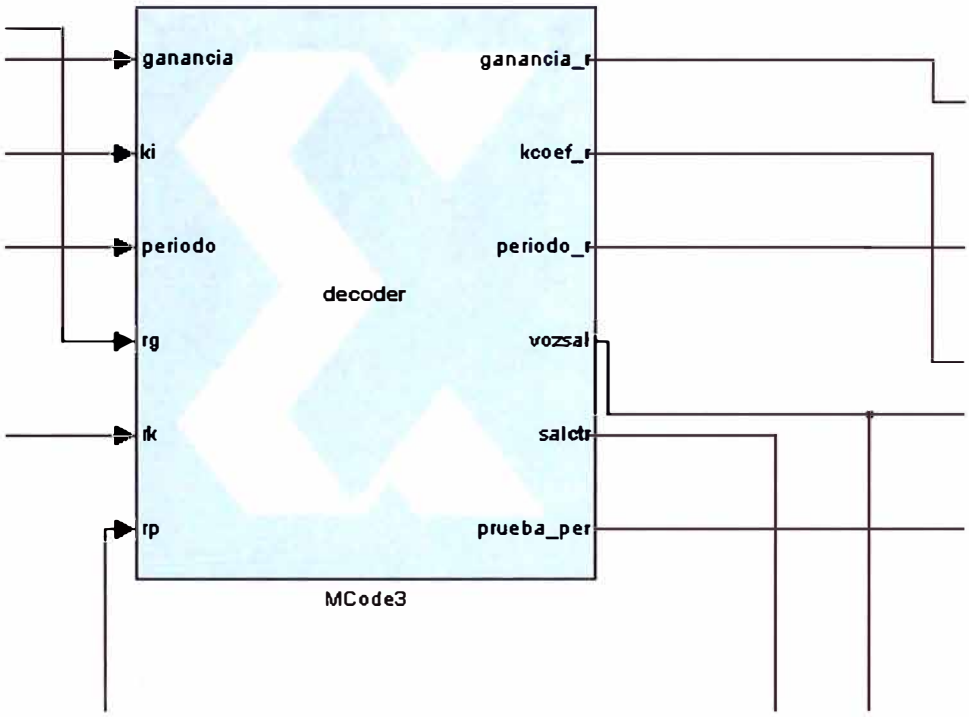


Fig. 6.33 Ubicación del bloque decodificador en el diseño completo.

El bloque decodificador no requiere de bloques externos ya que la máquina de estados no necesita realizar divisiones de punto fijo porque el algoritmo de decodificación no lo requiere, es por eso que no se necesita el uso de un divisor córdico externo ni de otros bloques para acondicionar externamente la señal. En este caso como las entradas que vienen de la parte del codificador (Sift, ganancia y coeficientes LPC) ya vienen correctamente formateadas y cuantizadas, el bloque decoder las recibe directamente.

Como se sabe, el bloque Mcode necesita una función .m asociada para la implementación de la máquina de estados, en este caso la función asociada es decoder.m tal como se puede apreciar en la figura Fig. 6.34.

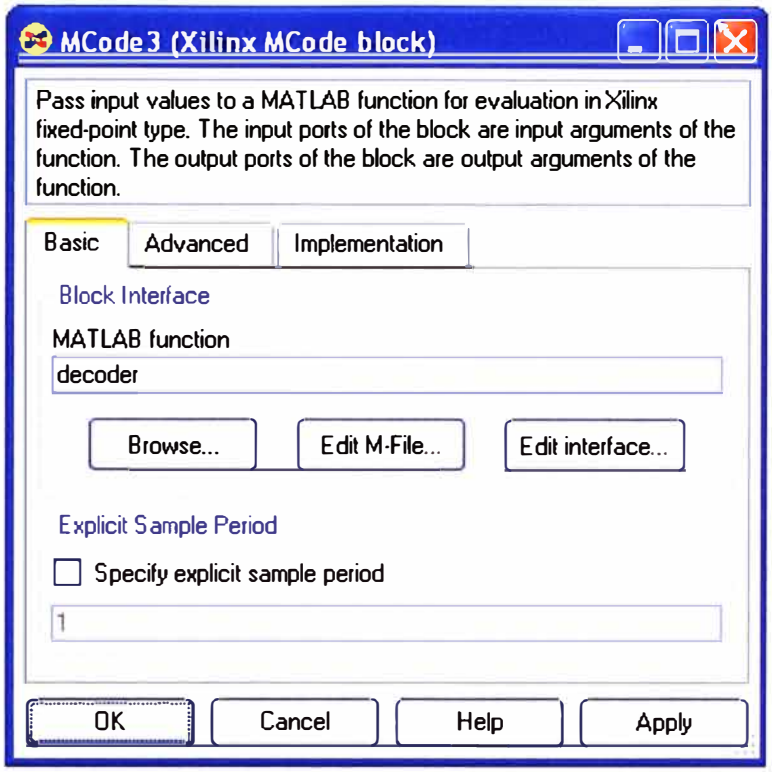


Fig. 6.34 Función .m asociada al bloque Mcode3

La máquina de estados que implementa el decodificador consta de 73 estados. En este caso, el algoritmo que implementa la parte de cálculo de la raíz cuadrada en punto fijo consta de 20 estados. Así mismo, la parte que corresponde al filtro todo polos consta de 37 estados el resto de estados están destinados a la carga de valores por los puertos de entrada, cambios de formatos y otras rutinas del decodificador (ajuste de ganancia y periodo). Cabe mencionar que la salida obtenida por el bloque decodificador trabaja a la frecuencia "up-sampled" por 200, por lo que al final se tiene que agregar un bloque de conversión que lleven los datos de la voz decodificada a una frecuencia de (1/8000). Esto se describirá mas adelante.

La figura Fig. 6.35 muestra la máquina de estados que implementa el bloque decodificador. Como se aprecia, en este caso tenemos estados para la raíz cuadrada, filtro *lattice* y bucles que se repiten 160 veces para generar las 160 muestras para la voz sintética.

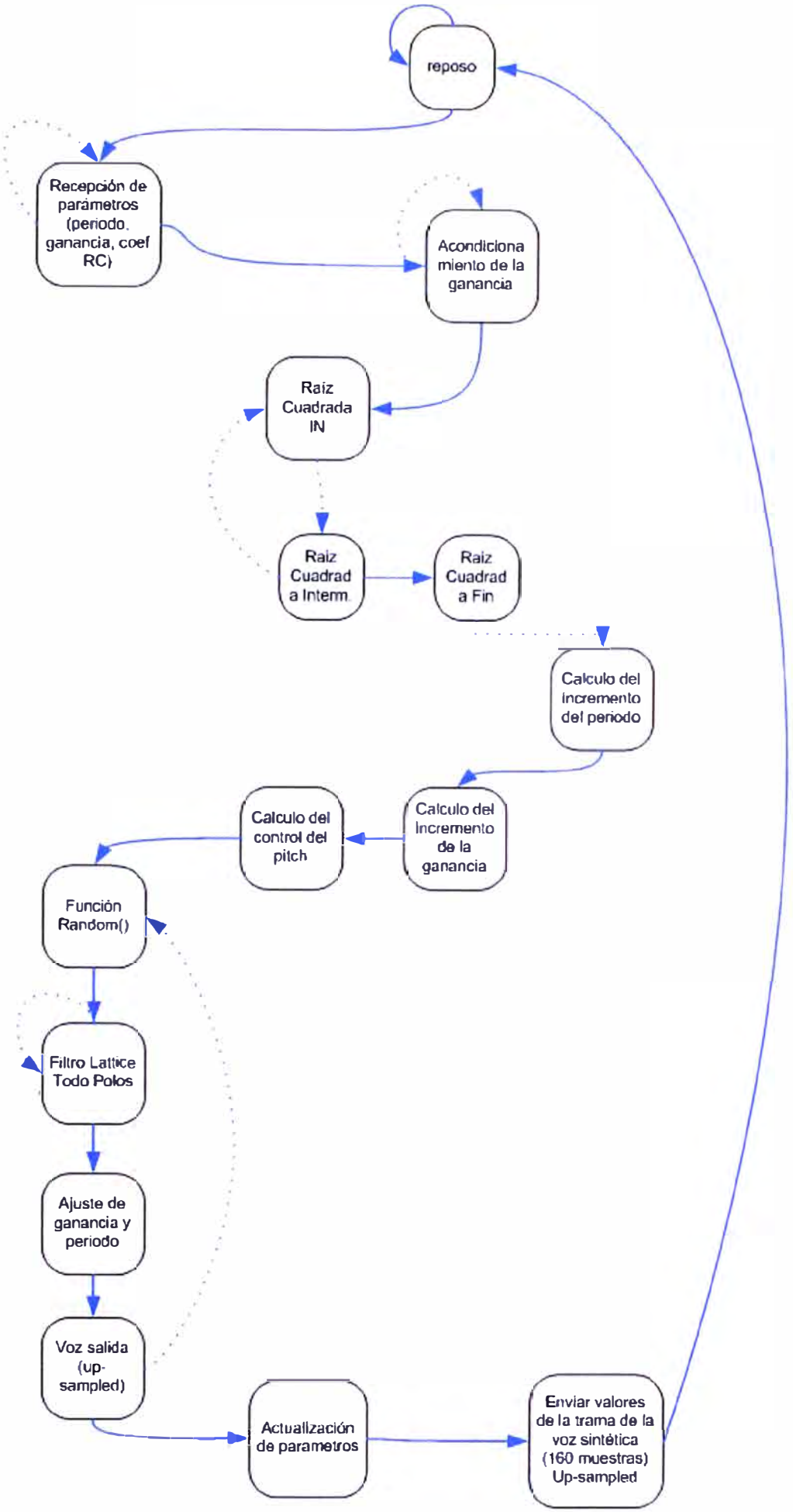


Fig. 6.35 Diagrama general de la máquina de estados para el decodificador

Como se puede apreciar en la figura Fig. 6.33, el bloque decodificador consta de 6 entradas y 6 salidas, que se describen a continuación:

Entradas:

ganancia: por este puerto se recibe la ganancia calculada por el bloque anterior, para una trama de análisis.
ki: por este puerto se reciben los 10 coeficientes RC(LPC) calculados por el bloque anterior, para una trama de análisis.
periodo: por este puerto de entrada se recibe el valor del periodo calculado por el bloque Sift, para una trama de análisis.
rg: indica que el valor de la ganancia está listo para ser leído.
rk: indica que los 10 valores RC(LPC) están listos para ser leídos.
rp: indica que el periodo está listo para ser leído

Salidas:

ganancia_r: ganancia recuperada.
kcoef_r: 10 coeficientes RC recuperados.
periodo_r: periodo recuperado.
voz_sal: arroja los 160 valores por trama de análisis de la voz sintética decodificada. Observar que estos valores están "up-sampled".
salctr: avisa a la siguiente etapa que los datos de la voz decodificada ya están listos para ser leídos.
prueba_per: puerto para pruebas.

Como se aprecia en las figuras Fig. 6.36 y 6.37, la salida de la voz sintética decodificada que proporciona el bloque decodificador está "up-sampled" es decir las muestras son entregadas a la frecuencia (1/8000)/200. Cuando se amplia el área de los datos, como en la figura Fig. 6.37 se pueden observar las muestras (160 muestras) correspondientes a la voz sintética decodificada de un segmento de análisis.

El código completo se puede encontrar en el CD en la ruta "..*System Generator*\ *decoder.m*".

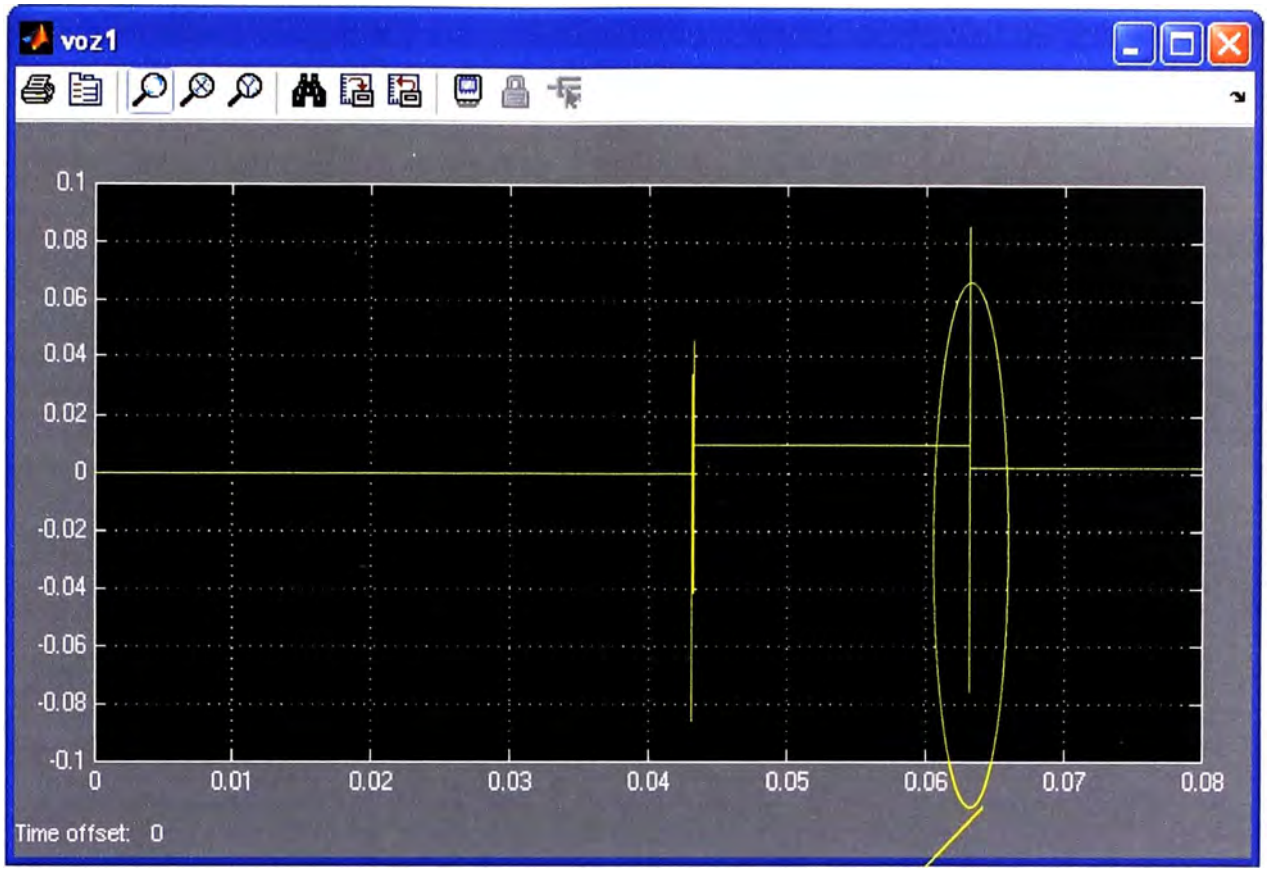


Fig. 6.36 Salida de la voz sintética para 2 tramas consecutivas (up-sampled)

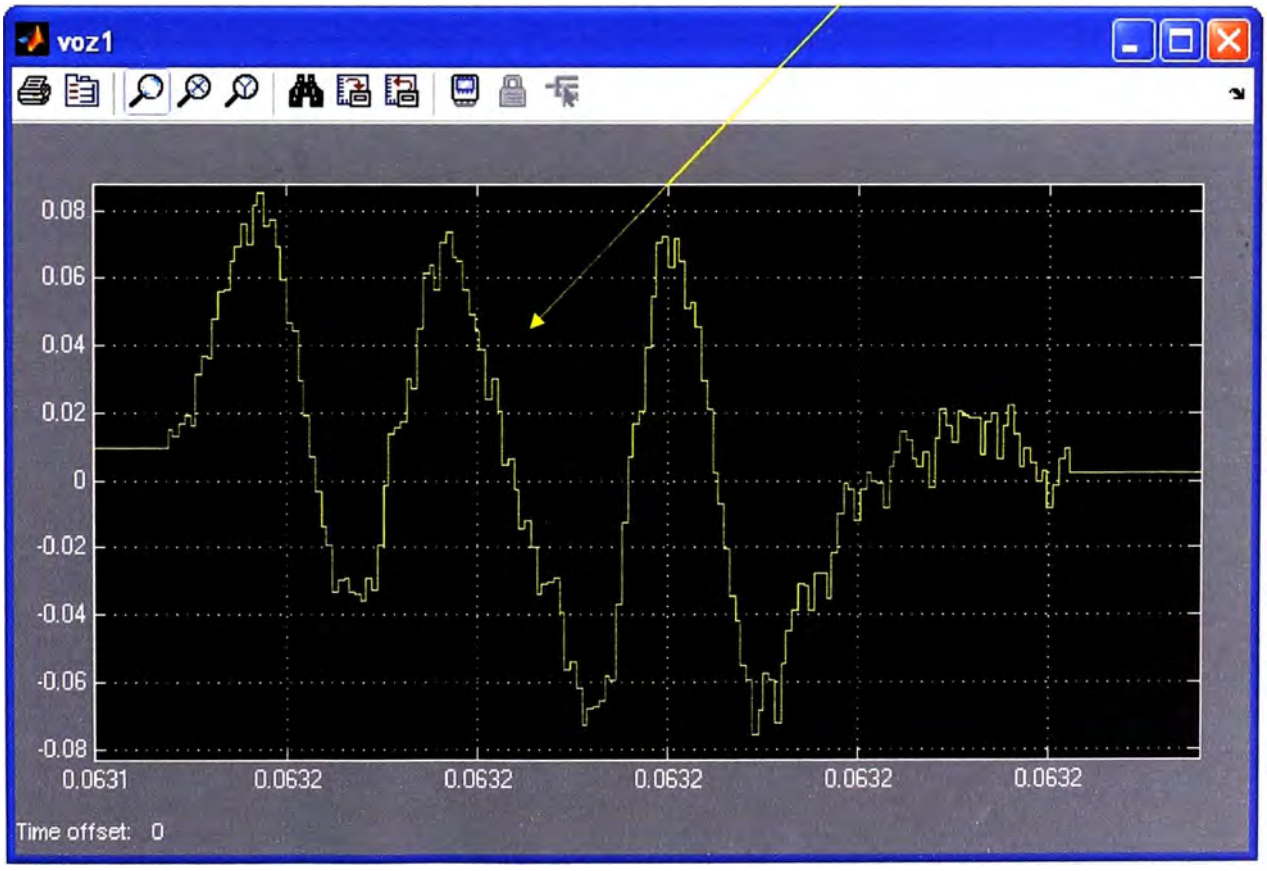


Fig. 6.37 Salida de la voz sintética (detalle)

6.2.9 Bloque de acondicionamiento de la voz sintética de salida

Como se ha visto en la sección anterior, los cálculos ejecutados por el bloque decodificador se realizan a una frecuencia de (1/8000)/200 para cada estado. Por lo que los datos de salida de la voz decodificada se dan a dicha frecuencia de reloj. Entonces, es necesario la inclusión de un bloque de salida que haga la conversión de las frecuencias y realice el "down-sampling" correspondiente de (1/8000)/200 a (1/8000) para de esta forma tener la voz sintética a la frecuencia de interés que es de 8000 Hz.

Los bloques que realizan el "down-samplig" y acondiciona la señal de salida se muestran en la figura Fig. 6.38.

Se observa que hay un bloque FIFO2 se usa para almacenar y entregar los datos de la voz sintética decodificada. También se observan además dos bloques Mcode (Mcode4 y Mcode5), un bloque "assert", un bloque "relational", un bloque "constant" y un bloque "down-sampling" que es el que al final realizará el cambio de frecuencia de (1/8000)/200 a (1/8000).

La funcionalidad de todo el conjunto de bloques de salida es el siguiente:

- Todos los datos que recibe el FIFO2 se realizan a la velocidad "up-sampled" es decir a (1/8000)/200. Esto esta controlado por los Mcode: Mcode4 y Mcode5 con funciones asociadas "salida.m" y "ctrl_salida.m" respectivamente.
- La lectura del FIFO2 esta controlada por los bloques Constant, Relational, Assert y por la señal procedente del bloque "Relational" de la figura Fig. 6.10.
- Finalmente el bloque realiza el "downsampling" por el valor de 200 está ubicado a la salida del FIFO2.

Finalmente, las figuras Fig. 6.39 y Fig. 6.40 muestran la entrada de la voz original (25 tramas de 160 muestras por trama a 16 bits) y la respectiva salida de la voz sintética decodificada correspondiente a todo el sistema (codificador mas decodificador). El detalle de todos los códigos implementados se puede encontrar en el CD en la ruta "..*System Generator*\". ***Más resultados se pueden encontrar en el Anexo A al final del presente documento.***

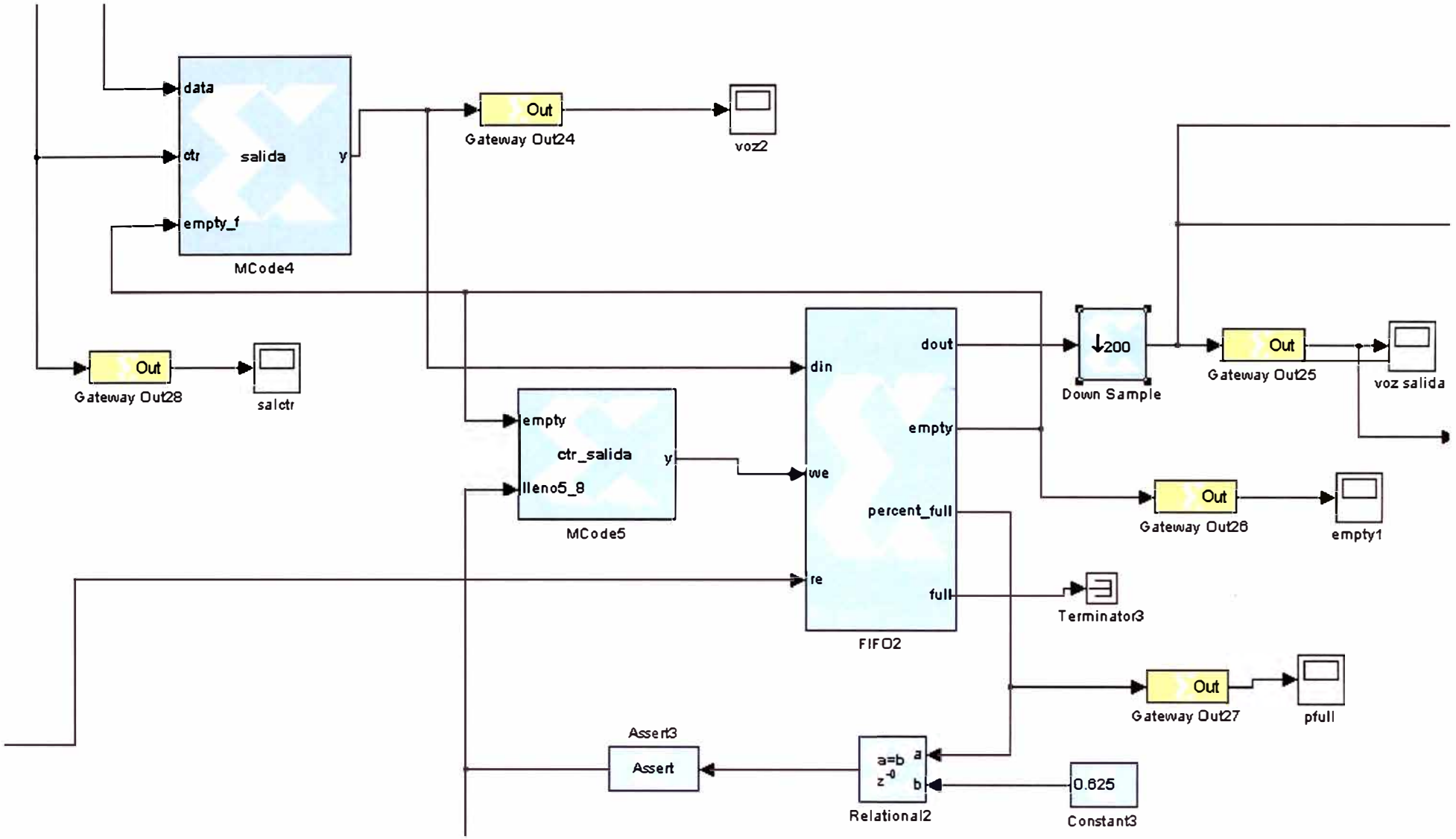


Fig. 6.38 Bloques para el acondicionamiento de la voz sintética de salida

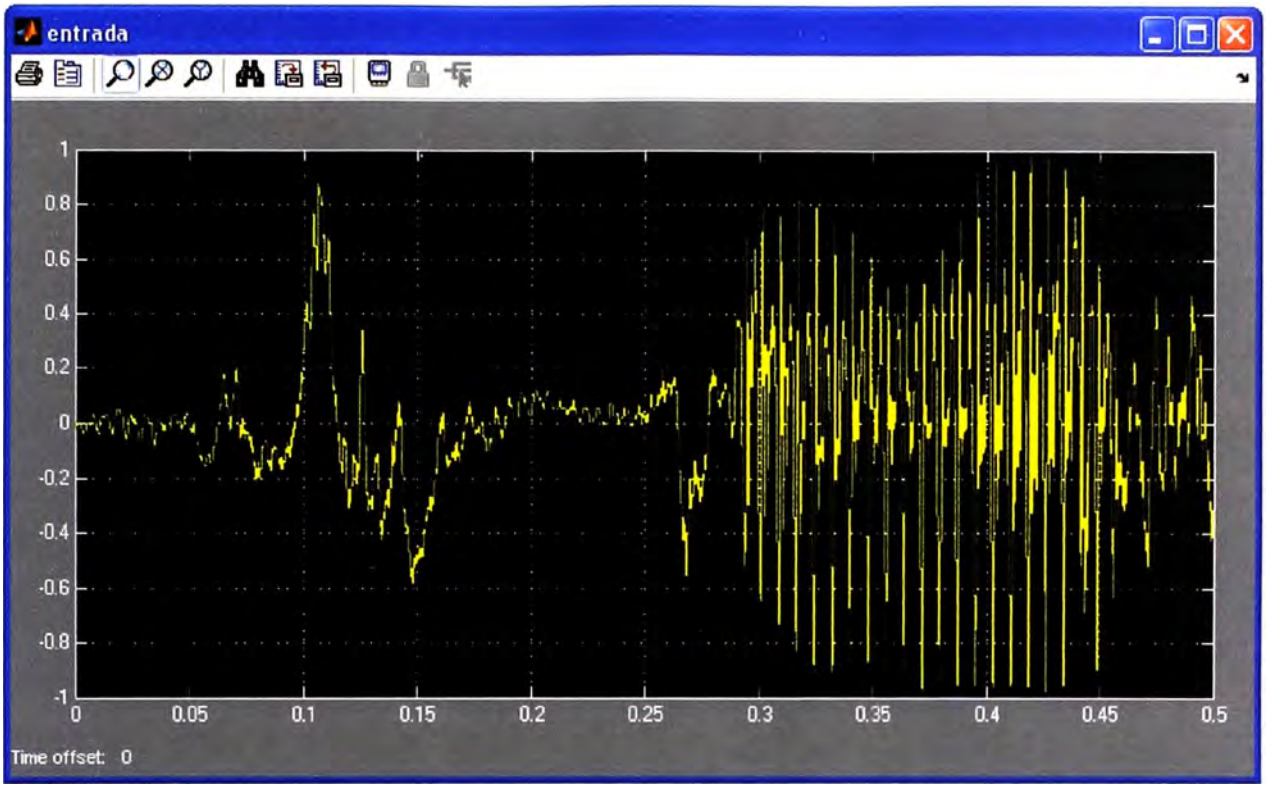


Fig. 6.39 Segmento de 0.5 segundos de la voz de entrada (25 tramas)

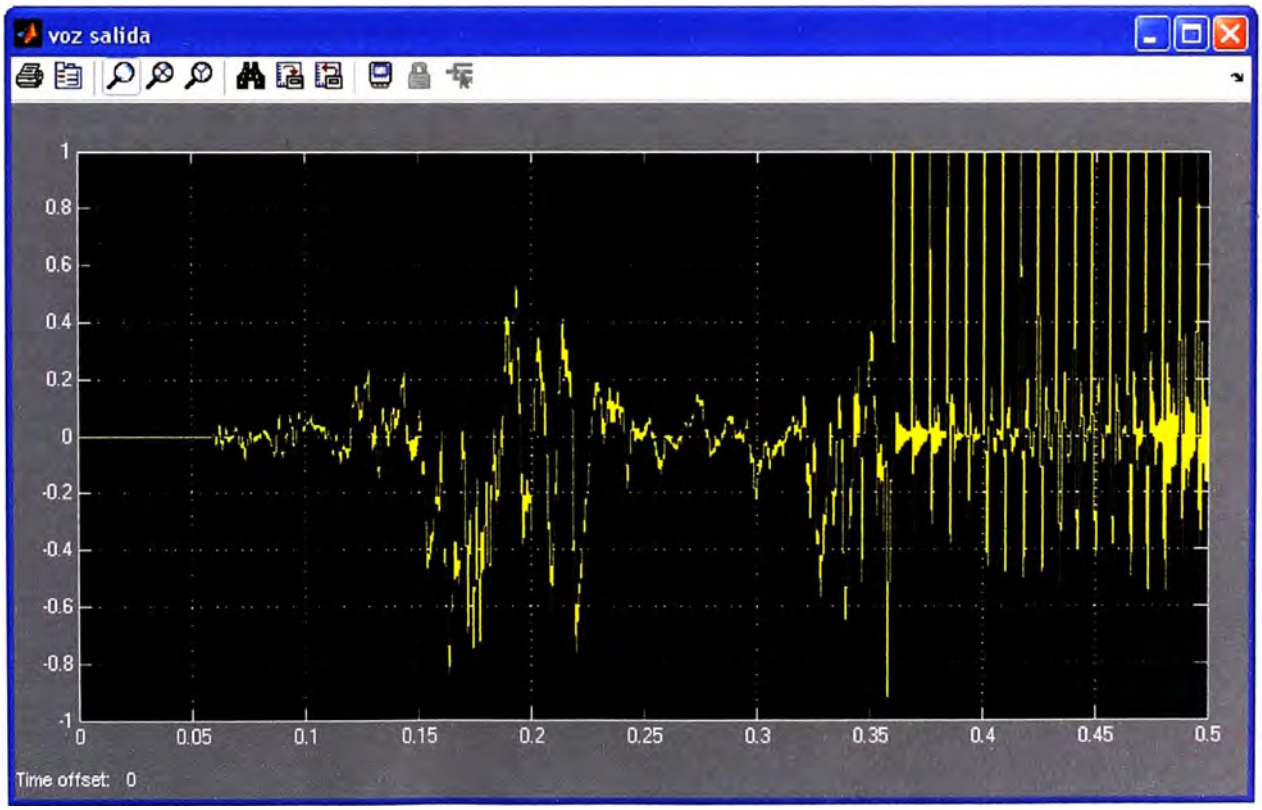


Fig. 6.40 Voz decodificada correspondiente a la voz de entrada de la Fig. 6.39

A continuación se presentan resultados complementarios para la parte de simulación del diseño descrito en el presente capítulo. Se pone especial énfasis en resultados de simulación concernientes a la parte del algoritmo SIFT, ya que este algoritmo es el corazón del codificador diseñado el cual depende en gran parte de un correcto "tracking" de la frecuencia fundamental de la voz. Así mismo, se muestran también comparaciones entre la señal de entrada y la señal de salida del decodificador implementado en la pltaforma de simulaciónSystem Generator para dispositivos FPGA.

La secuencia en la que se presentan los siguientes resultados de simulación es la siguiente: Segmento de secuencia original de la voz, "tracking" del pitch respectivo para dicho segmento de análisis, segmento correspondiente a la voz sintética decodificada para el segmento de análisis de entrada.

Cabe mencionar que la voz sintética decodificada obtenida por la simulación en la plataforma System Generator es prácticamente igual a la obtenida por el codificador implementado con algoritmo de punto flotante. Las secuencias de audio se pueden encontrar en el CD que viene junto con el presente documento en la ruta: ..\ *Voz sintética decodificada Simulacion System Generator*.

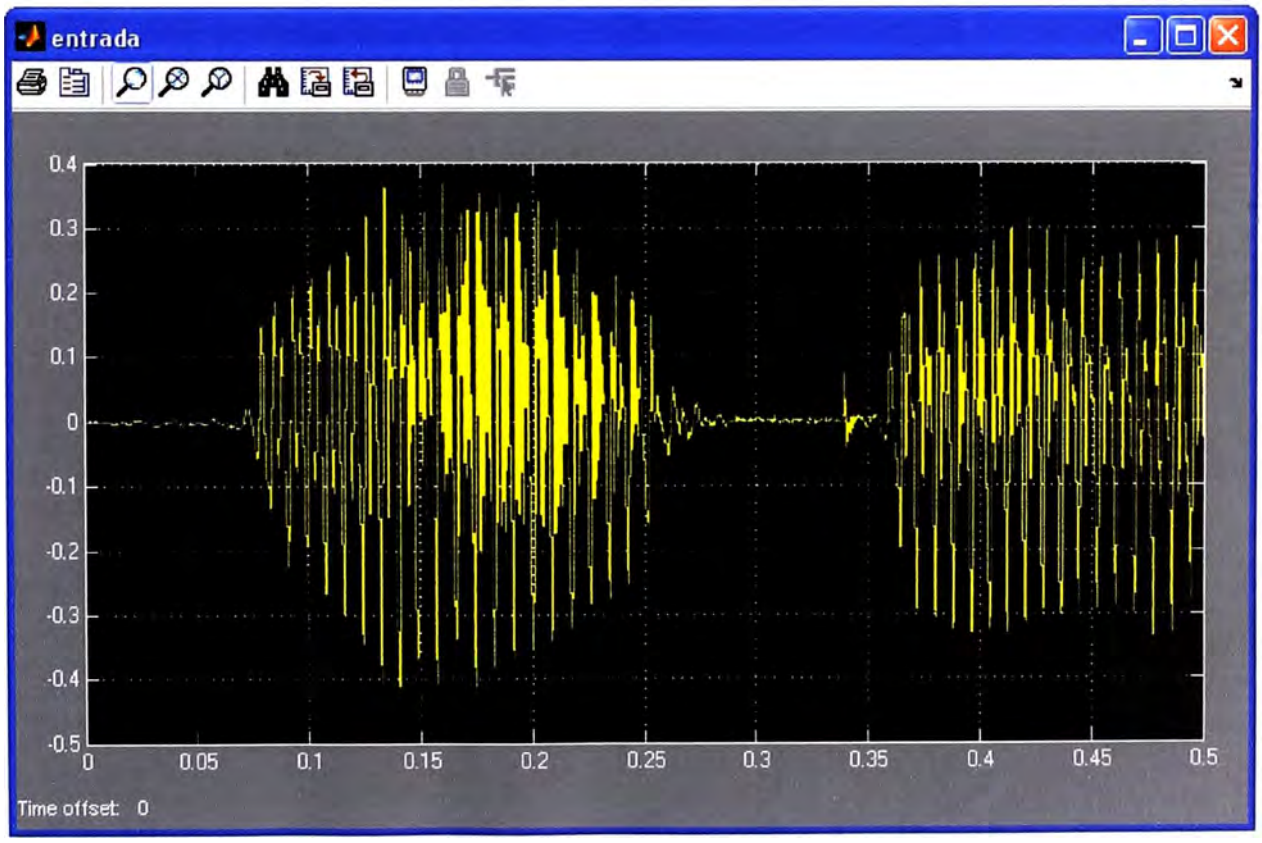


Fig. 6.41 Segmento de 0.5 segundos de la secuencia original "voz1_FPujaico.wav".

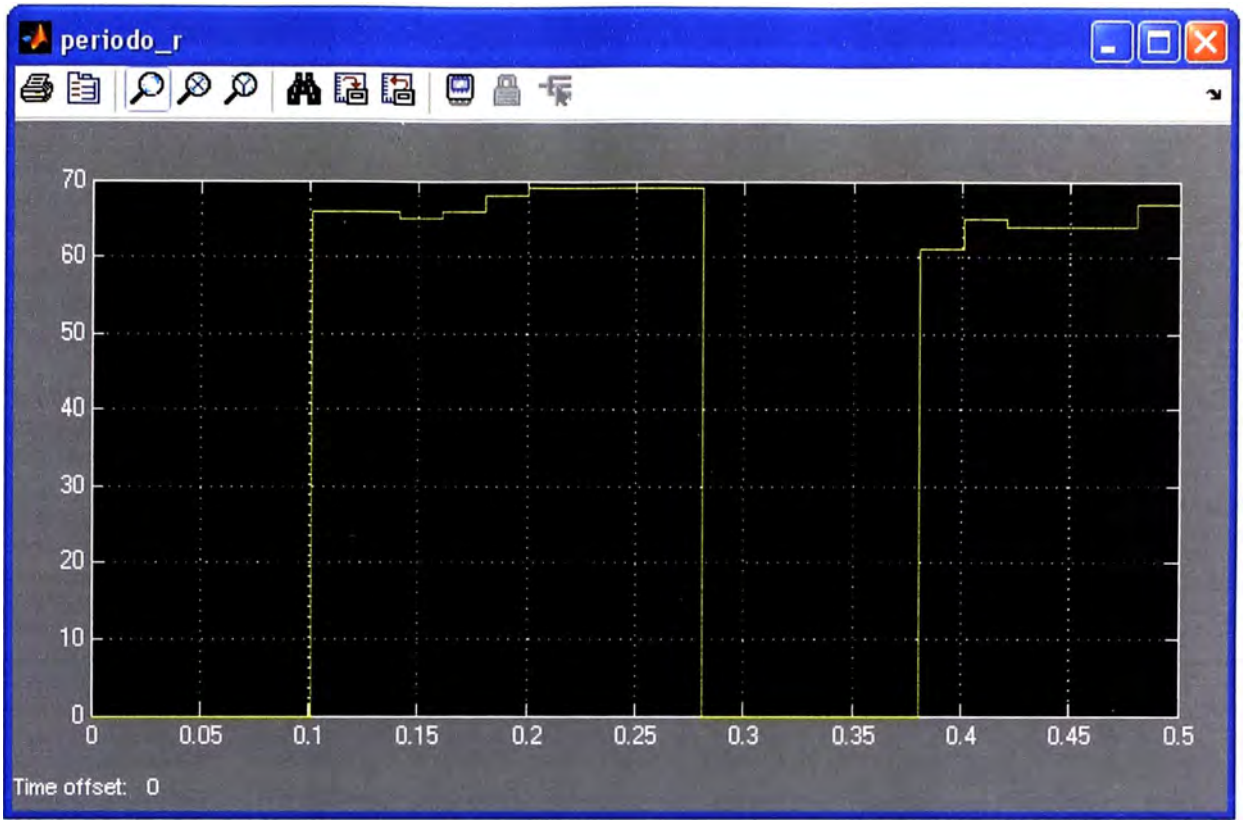


Fig. 6.42 "Tracking" del pitch para la secuencia de la Fig. 6.41

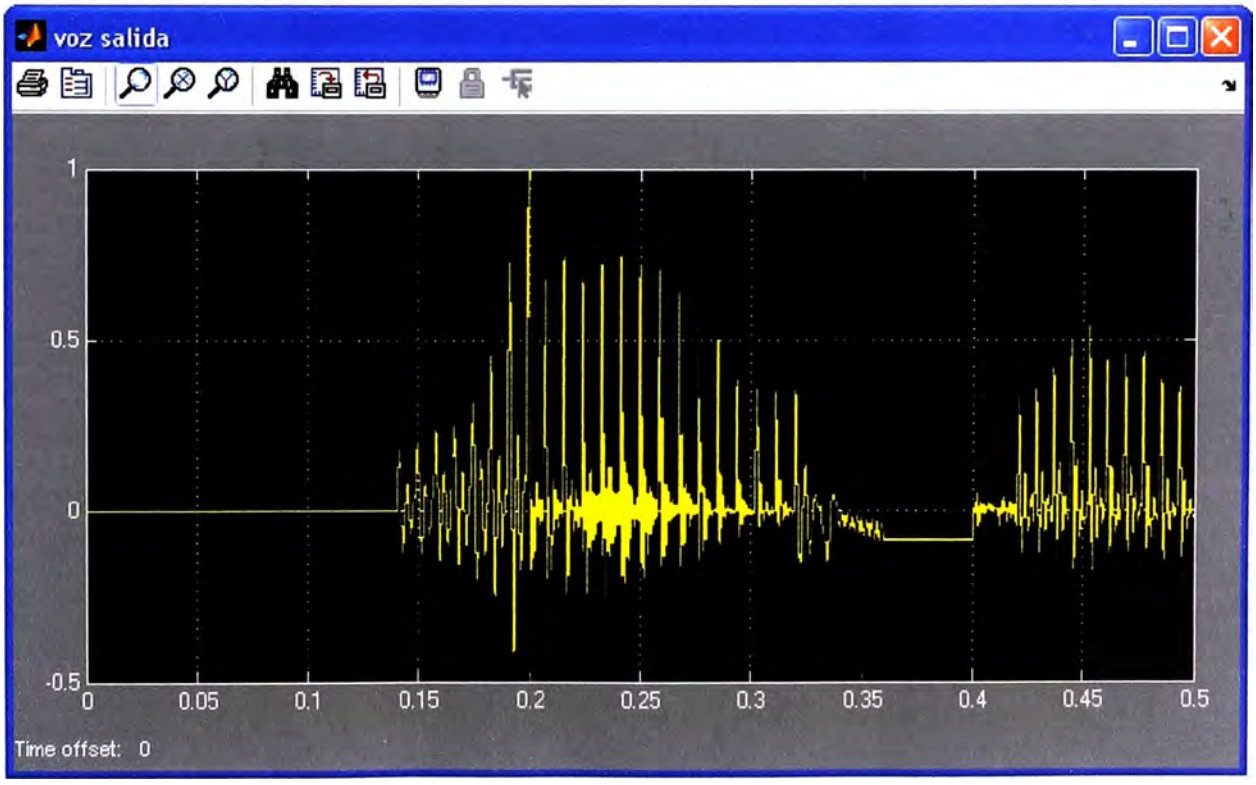


Fig. 6.43 Voz sintética decodificada para la secuencia de la Fig. 6.41

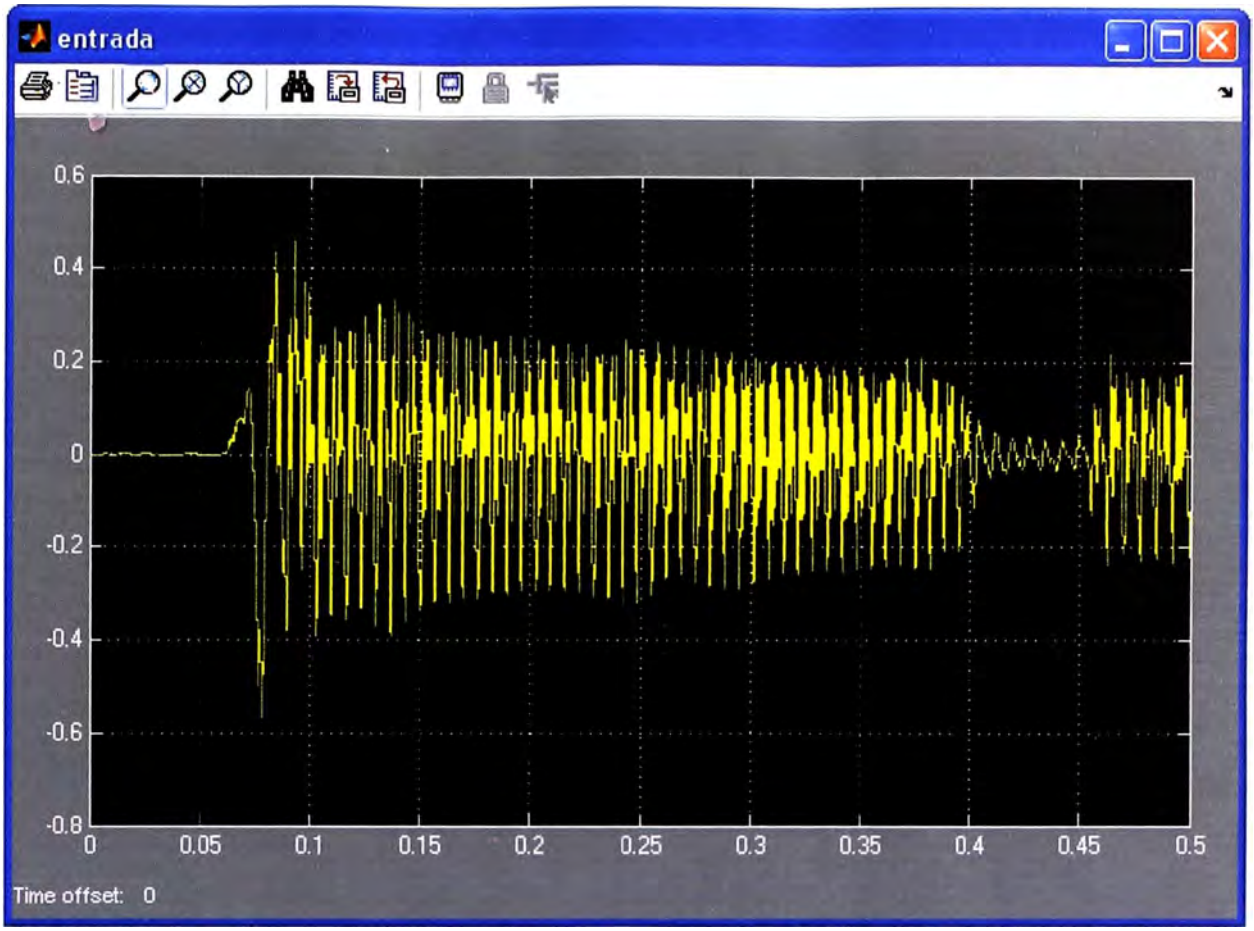


Fig. 6.44 Segmento de 0.5 segundos de la secuencia original "voz2_FPujaico.wav".

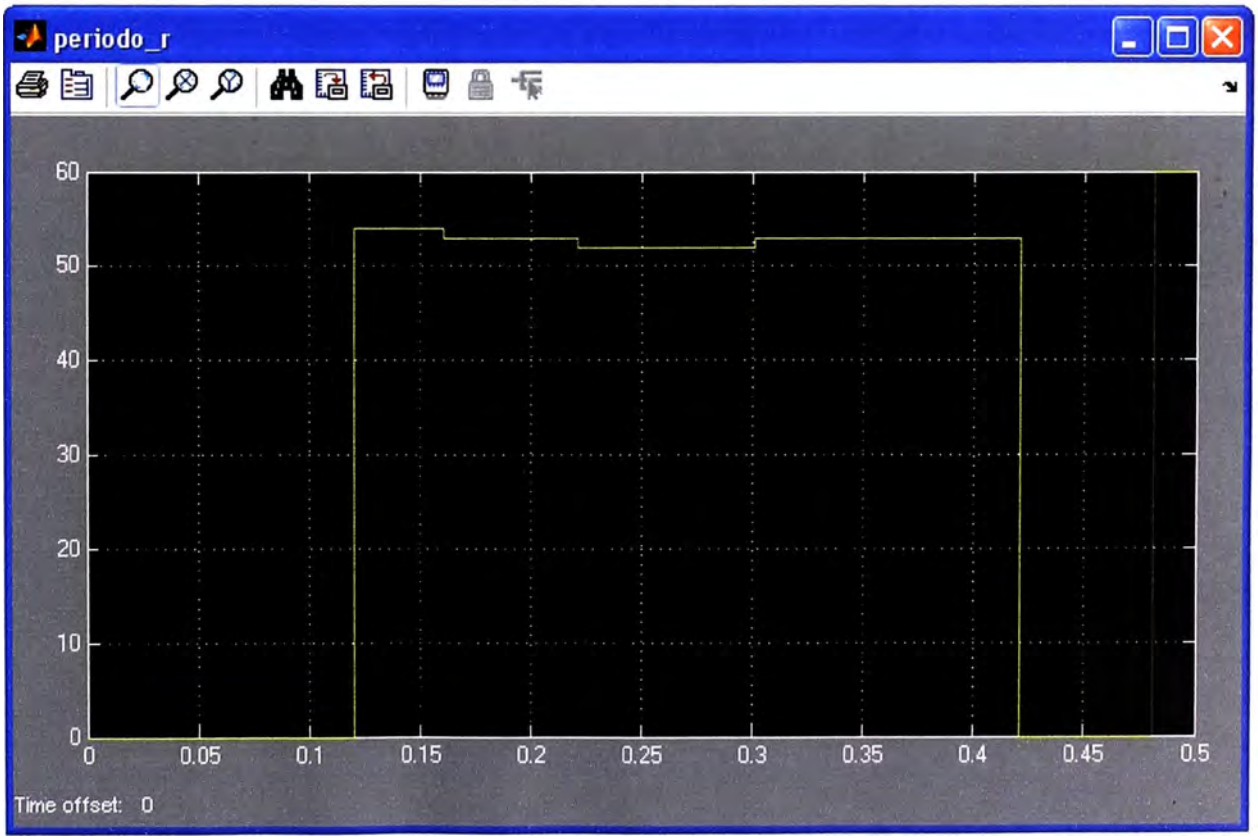


Fig. 6.45 "Tracking" del pitch para la secuencia de la Fig. 6.44

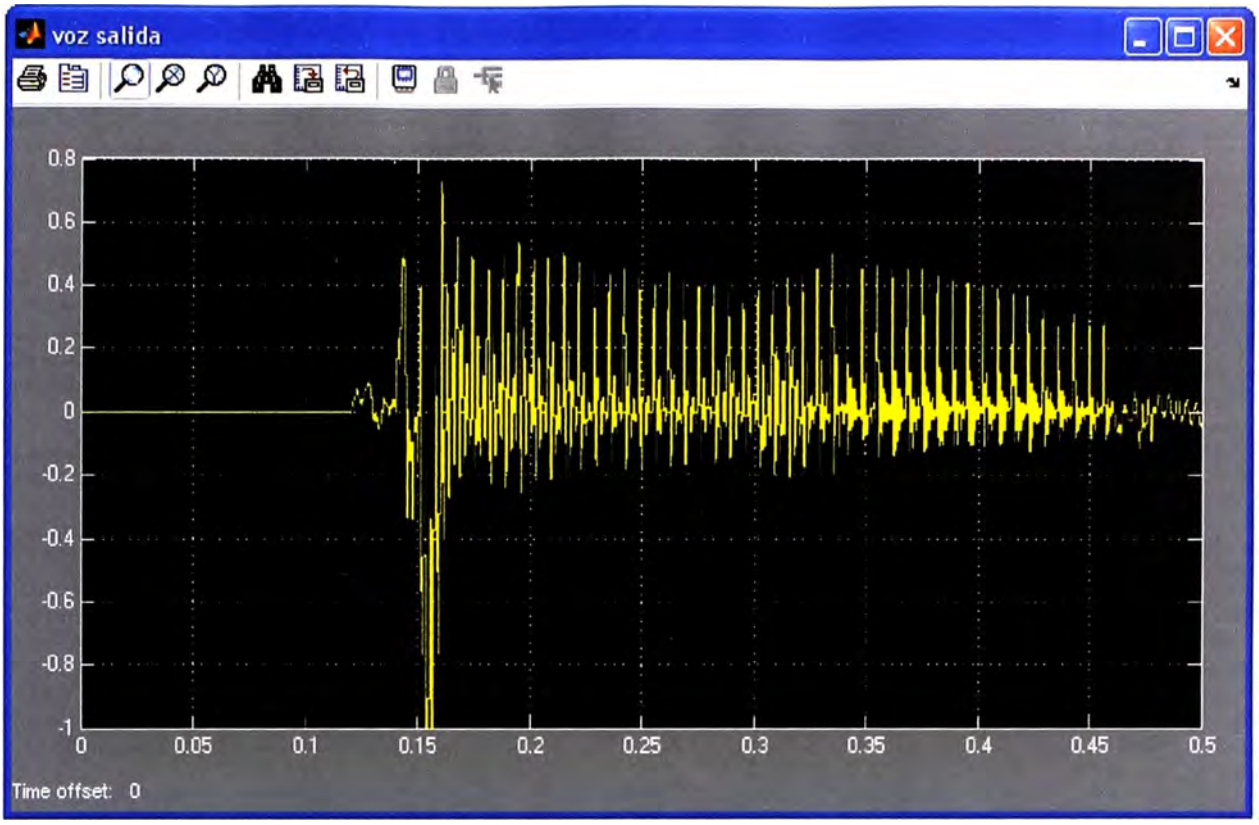


Fig. 6.46 Voz sintética decodificada para la secuencia de la Fig. 6.44

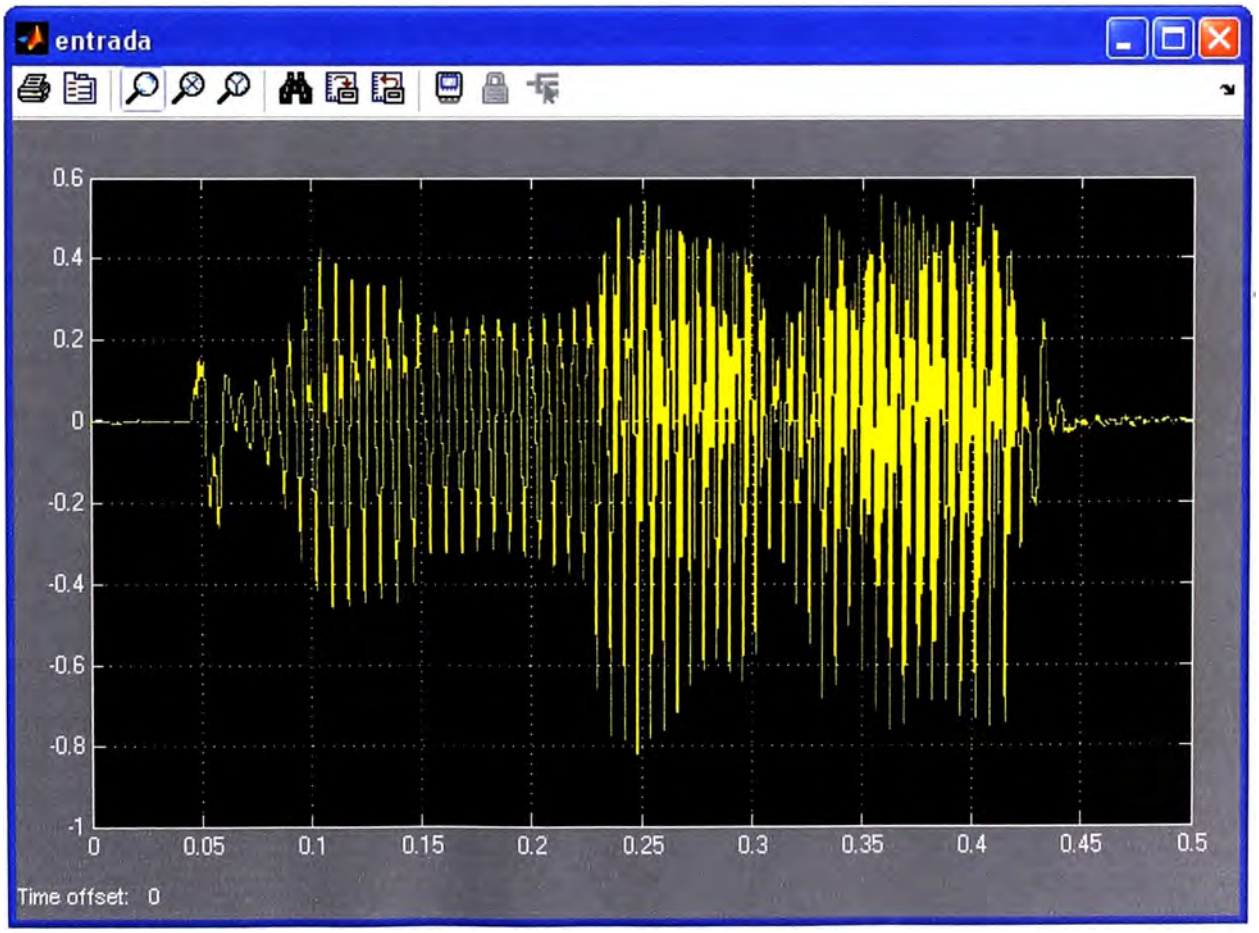


Fig. 6.47 Segmento de 0.5 segundos de la secuencia original "voz1_JHuaman.wav".

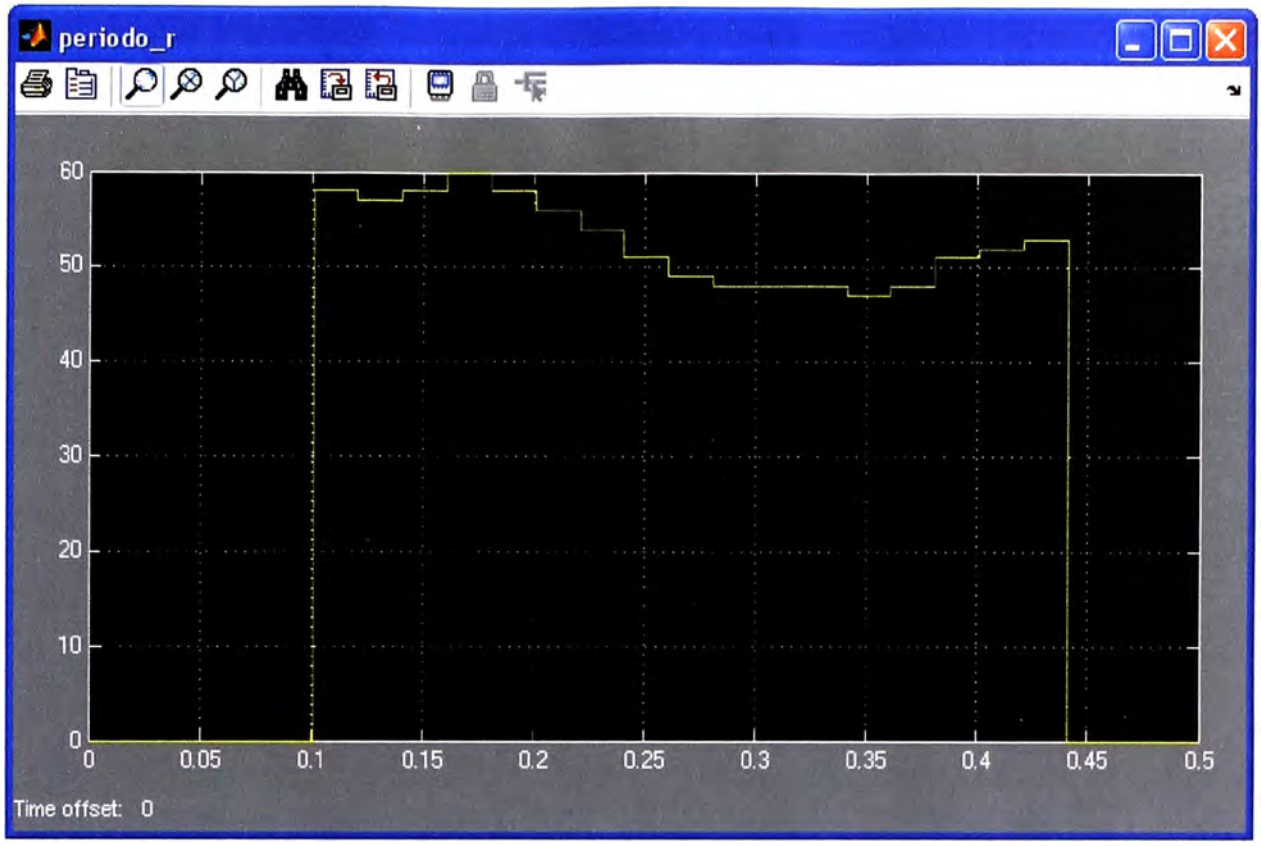


Fig. 6.48 "Tracking" del pitch para la secuencia de la Fig. 6.47

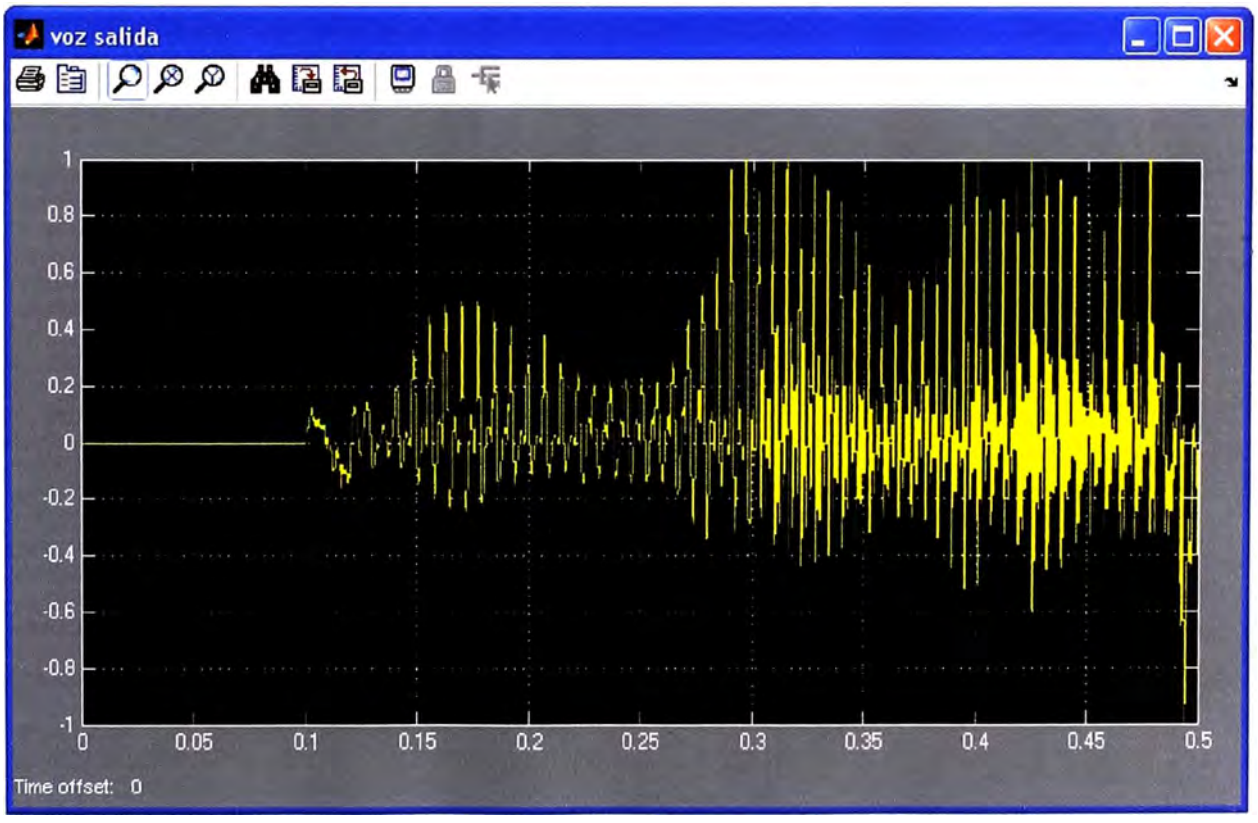


Fig. 6.49 Voz sintética decodificada para la secuencia de la Fig. 6.47

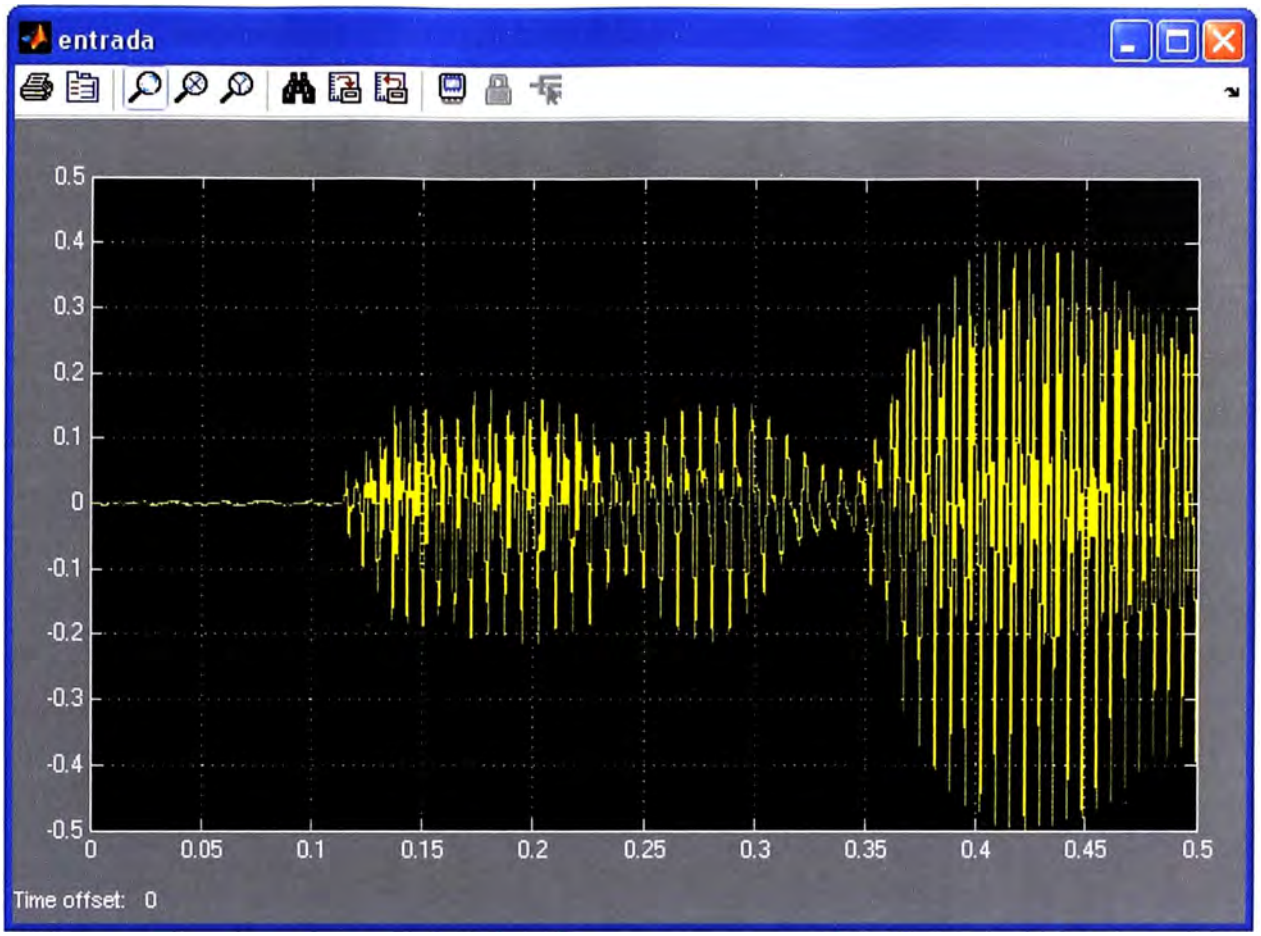


Fig. 6.50 Segmento de 0.5 segundos de la secuencia original "voz2_JHuaman.wav".

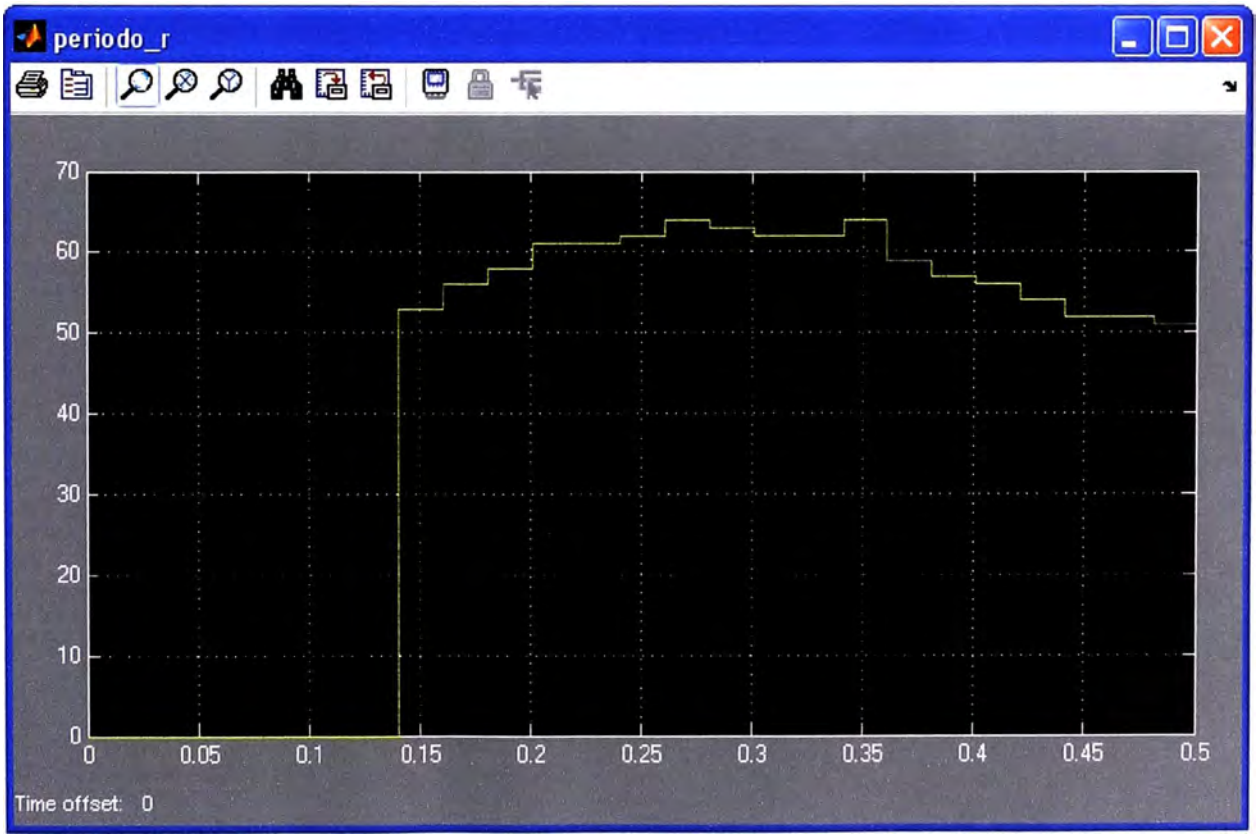


Fig. 6.51 "Tracking" del pitch para la secuencia de la Fig. 6.50

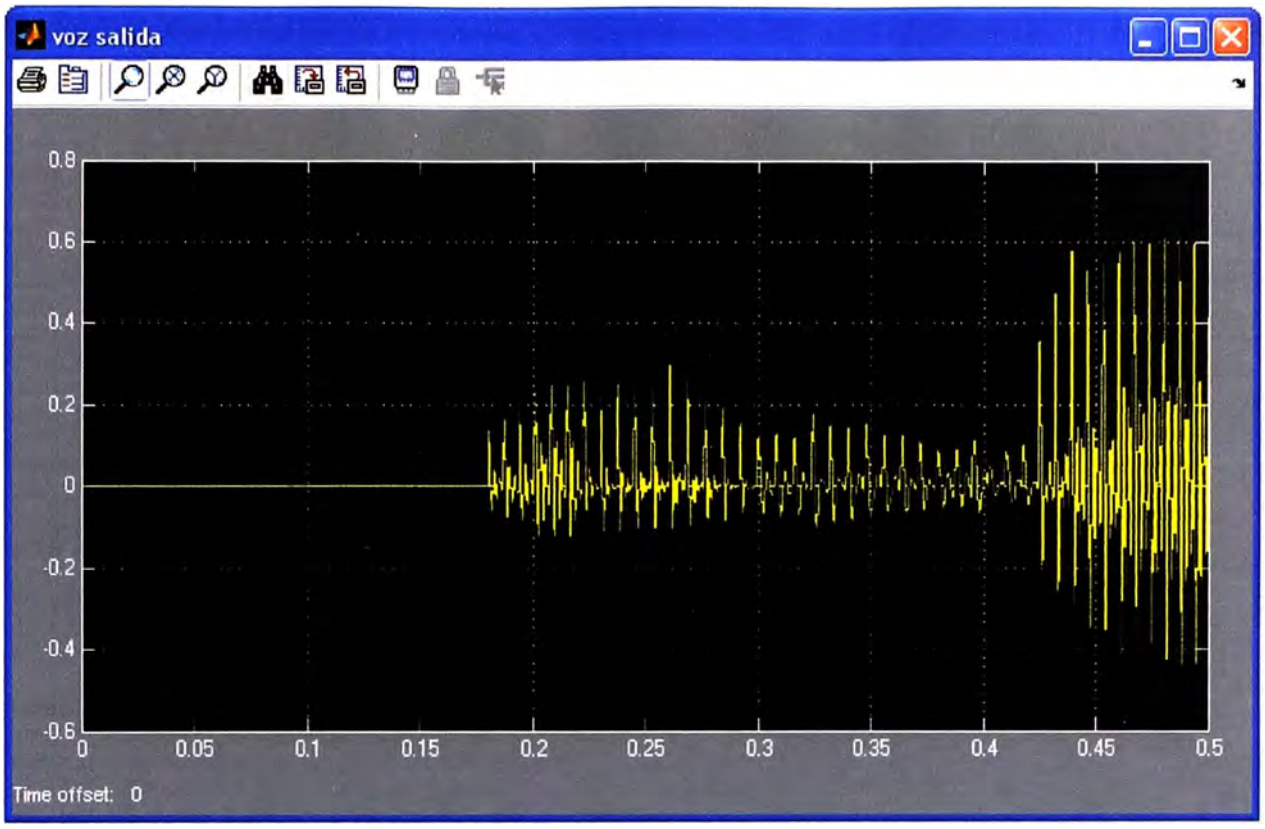


Fig. 6.52 Voz sintética decodificada para la secuencia de la Fig. 6.50

CAPITULO VII

DESCRIPCIÓN DEL DSP TMS320C6711 Y DEL FLUJO DE DESARROLLO DE SOFTWARE PARA DSP

El presente capítulo describe, en primer lugar, las características y prestaciones del DSP utilizado. Luego se pasa a describir los detalles de la arquitectura interna del DSP en mención, así como el flujo de diseño de un proyecto para DSP. Finalmente se realiza una comparación entre los modos de programación en punto fijo y punto flotante.

7.1 DSP TMS320C6711 Overview

Los Procesadores Digitales de Señales tales como los de la familia TMS320 de Texas Instruments vienen siendo usados en un amplio rango de aplicaciones de control, comunicaciones, procesamiento de imágenes, de voz, etc. El TMS320C6711 es un procesador digital de señales de punto flotante que es adecuado para algoritmos numéricamente intensivos.

Los DSP's de la familia TMS320C6X (C6X) de Texas Instruments son Dsp's de propósito especial con un tipo de arquitectura especializada y un set de instrucciones apropiado para procesamiento de señales. La arquitectura de un Dsp de la familia C6X es adecuado para algoritmos que requieren flexibilidad y eficiencia. Basado en una arquitectura VLIW (Very Long Instruction Word), los Dsp de la familia C6X son considerados los mas potentes Dsp's desarrollados por Texas Instruments. (42)

7.1.1 Tarjeta DSK C6711

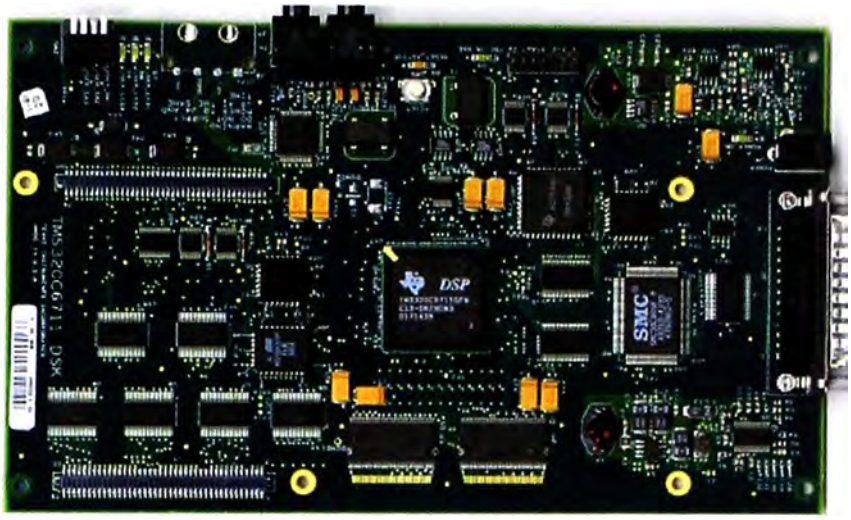


Fig. 7.1 Tarjeta DSK TMS320C6711 (42)

- **Hardware**
  - 150 MHz 'C6711 DSP
  - TI 16-bit A/D Converter ('AD535)
  - External Memory
    - 16M Bytes SDRAM
    - 128K Bytes Flash ROM
  - LED's
  - Daughter card expansion
  - Power Supply & Parallel Port Cable

- **Software**
  - Code Generation Tools

  (C Compiler, Assembler & Linker)
  - Code Composer Debugger
  - Example Programs & S/W Utilities
    - Power-on Self Test
    - Flash Utility Program
    - Board Confidence Test
    - Host access via DLL
    - Sample Program(s)

Las partes mas importantes de la tarjeta se describen en la Fig. 7.2:

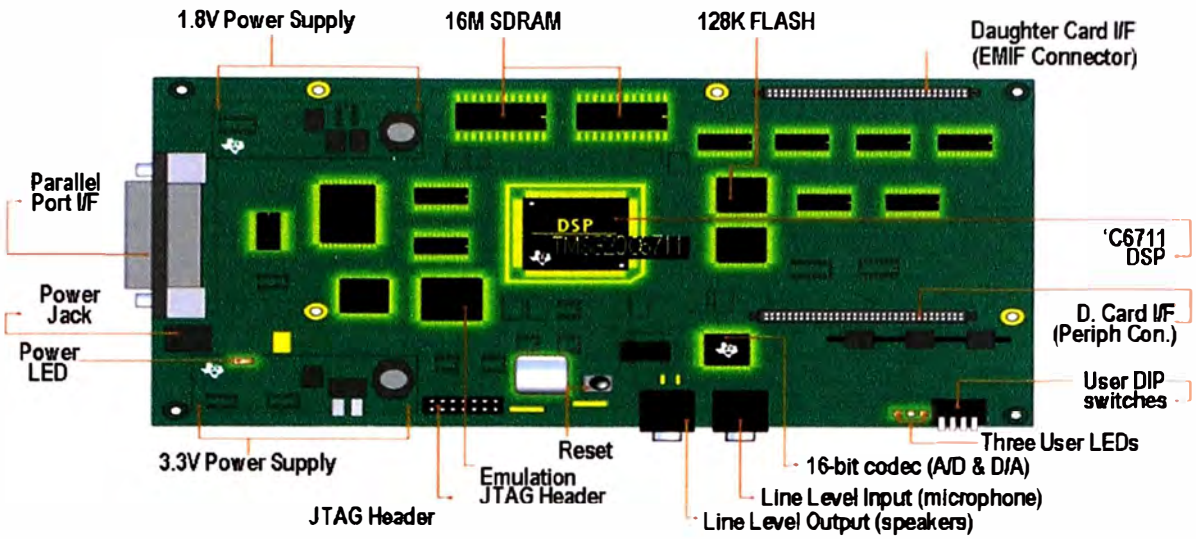


Fig. 7.2 Partes de la tarjeta DSK TMS320C6711 (42)

7.1.2 Mapa de Memoria

Se muestra en la Fig. 7.3.

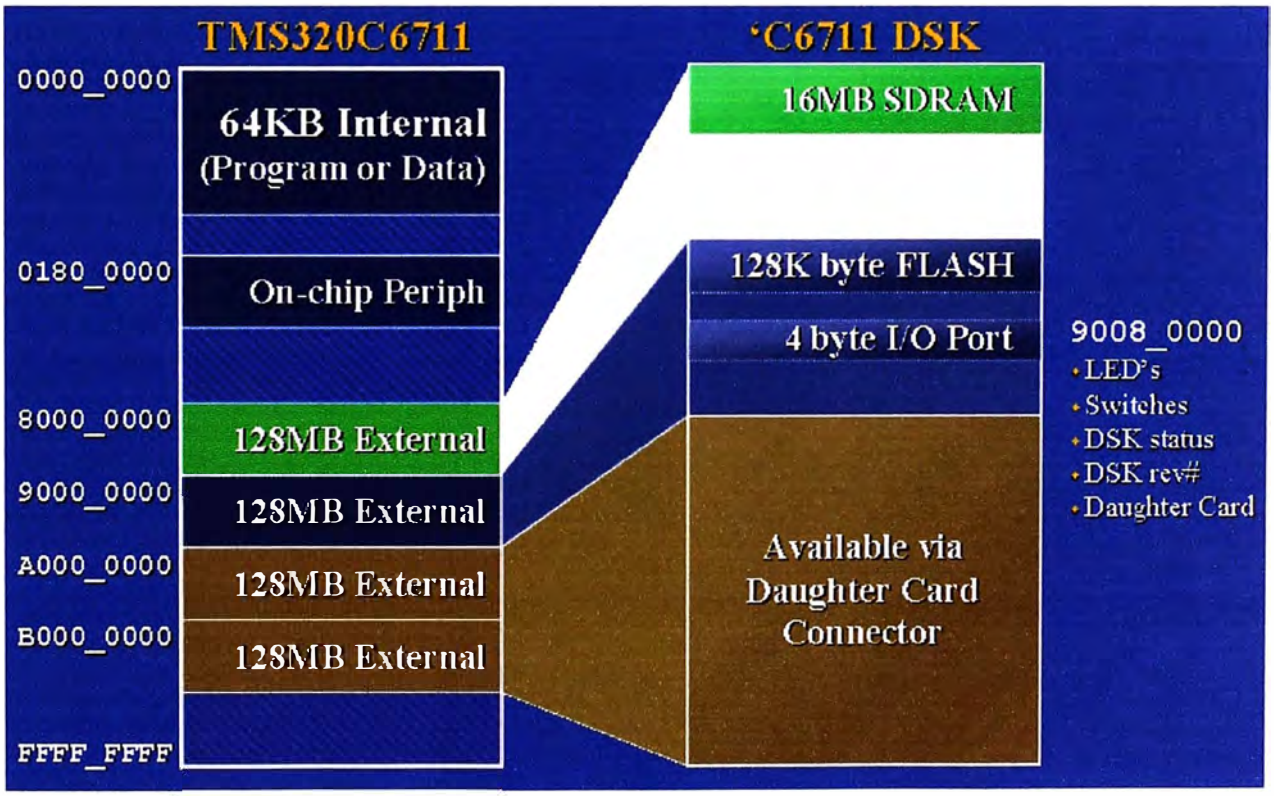


Fig. 7.3 Mapa de memoria del DSK TMS320C6711. (42)

7.1.3 Arquitectura de la familia TMS320C67X

El procesador C6711 esta basado en una arquitectura VLIW (Very Long Instruction Word). La memoria de programa interna está estructurada de tal forma que un total de 8 instrucciones pueden ser ejecutadas en cada ciclo de reloj. Por ejemplo con un clock rate de 150MHz, el C6711 es capaz de poner en "fetch" 8 instrucciones de 32 bits cada 1 / 150 MHz (o 6.6ns). El C6711 incluye una memoria interna de 72KB (64KB+4KB+4KB), 8 unidades funcionales (6 ALU's y 2 multiplicadores), un bus de dirección de 32 bits capaz de direccionar 4Gigabytes, y además 2 conjuntos de registros de 32 bits multipropósito.

La figura Fig. 7.4 ilustra el diagrama de bloques funcional del procesador C6711. Dos bancos de memoria independientes pueden ser accesados usando dos independientes buses. Memoria de programa, de datos y DMA (Direct Access Memory), cada uno tiene un bus separado, permitiendo al C6X ejecutar simultáneamente "program fetches", lectura y escritura de datos, y operaciones DMA. El espacio de memoria es Byte addressable. La memoria interna está organizada como memoria separada de programa y espacio de memoria de datos, contando con 2 puertos internos de 32 bits para accesar memoria interna.

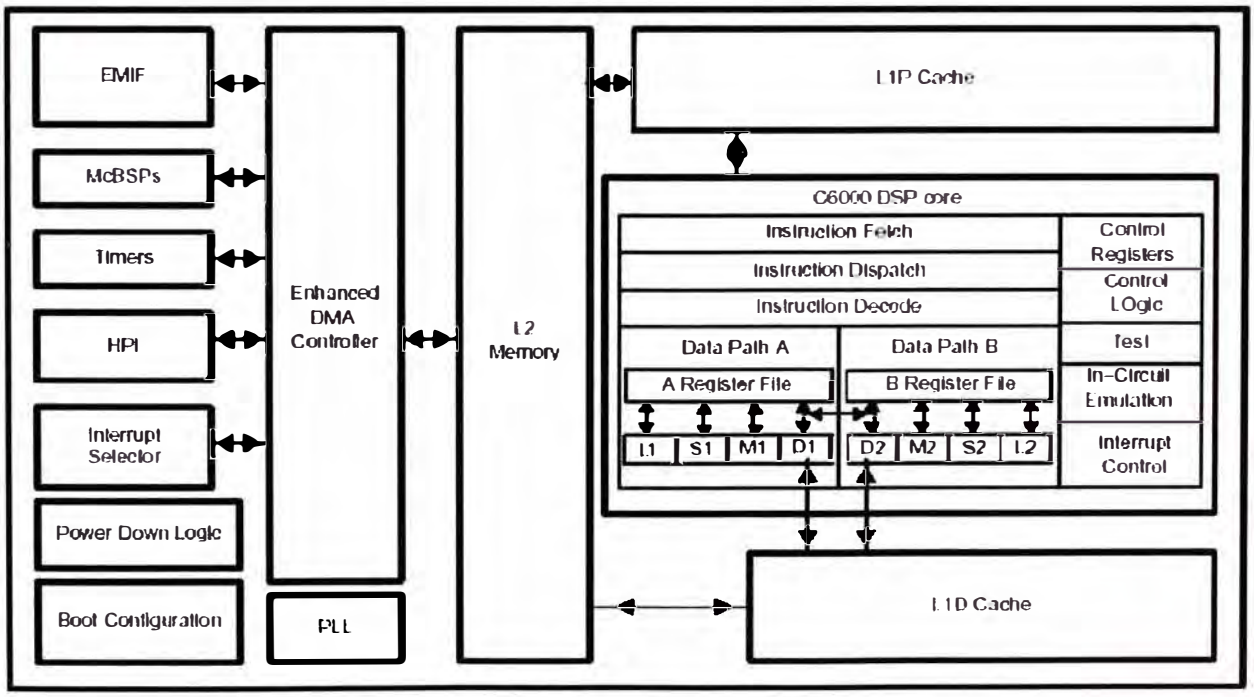


Fig. 7.4 Arquitectura interna de la familia TMS320C67X (44)

7.1.4 CPU de la familia TMS320C67X

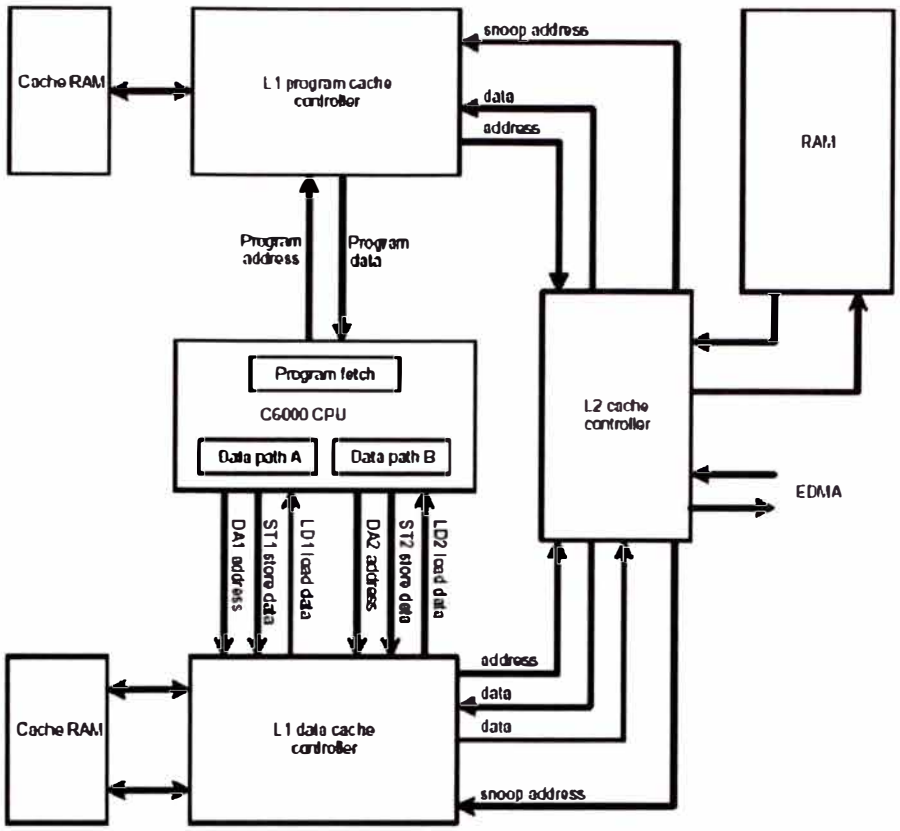


Fig. 7.5 Diagrama del CPU de la familia C67X (44)

El CPU consiste de 8 unidades funcionales dividido en dos "data paths" A y B como se muestra en la figura Fig. 7.5. Cada "data path" tiene cuatro unidades: (.M) para operaciones de multiplicación, (.L) para operaciones lógicas y aritméticas, (.S) para operaciones branch, manipulación de bits y operaciones aritméticas, (.D) para cargar/almacenar data y operaciones aritméticas. Las unidad .S y .L pueden ejecutar operaciones branch, aritméticas y lógicas. Todas las transferencias de datos emplean la unidad .D. Las operaciones aritméticas como SUB o ADD pueden ser ejecutadas por todas las unidades excepto .M.

.L y .S son dos ALU's de fixed/floating point aritmética. .D es un multiplicador de fixed/floating point.

Además, la arquitectura permite 2 "cross-paths" (1X, y 2X) los cuales hacen posible que las unidades de un "data path" accedan a un operando de 32 bits de un registro perteneciente a la otra "data path". Esto se ilustra en la Fig. 7.6 .

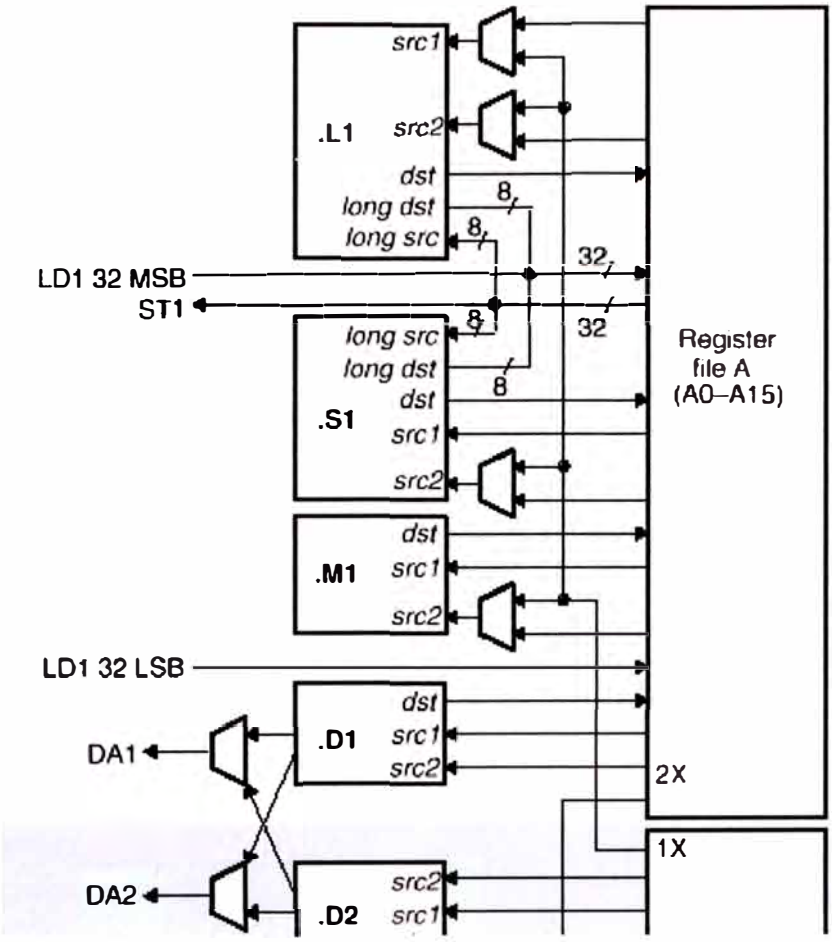


Fig. 7.6 Crosspaths para el CPU de la familia C67X (44)

7.1.5 Set de Instrucciones par la familia TMS320C6X

Se muestra en la Fig. 7.7.

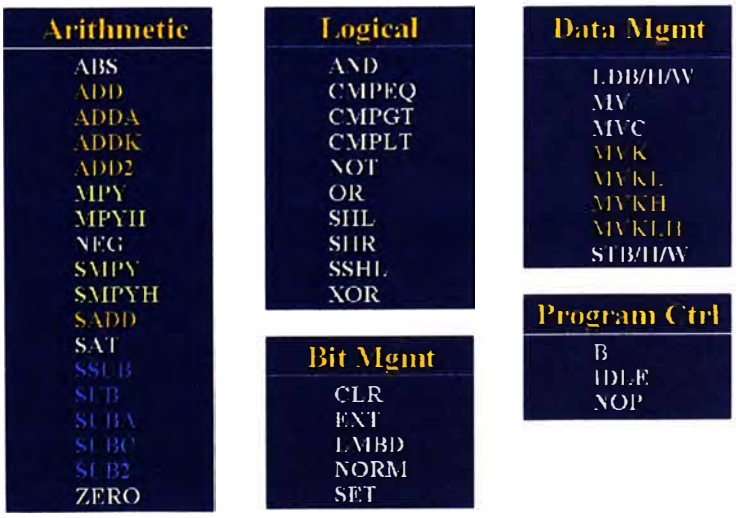


Fig. 7.7 Set de instrucciones para la familia C6X (42)

7.1.6 Periféricos de la familia TMS320C67X

Los periféricos se muestran en la Fig. 7.8 y se describen a continuación:

- **DMA Controller:** El DMA Controller transfiere datos entre rangos de direcciones en el mapa de memoria sin la intervención del CPU. El DMA controller tiene 4 programables canales y un quinto canal auxiliar.
- **EDMA Controller:** Ejecuta las mismas funciones que el DMA Controller. Tiene 16 canales programables y también un espacio RAM que le permite mantener múltiples configuraciones para futuras transferencias.
- **EMIF:** Soporta una interfaz "glueless" para varios tipos de dispositivos externos, incluyendo: SBSRAM (Synchronous Burst SRAM), SDRAM (Synchronous DRAM), Asynchronous devices, tales como SRAM, ROM y FIFO's; o también un memoria externa compartida.
- **McBSP:** El Multi-Channel Buffered Serial Port (McBSP) está basado en la interfaz estándar de puerto serial y puede almacenar "samples" seriales automáticamente con la ayuda del DMA/EDMA controller. También tiene capacidad multi-canal con los estándar de networking T1, E1 y MVIP. Ejecuta las siguientes funciones: Comunicación Full Duplex, Entramado y Clock independiente para Tx y Rx, registros de datos Double-buffered que permite un continuo flujo de datos.

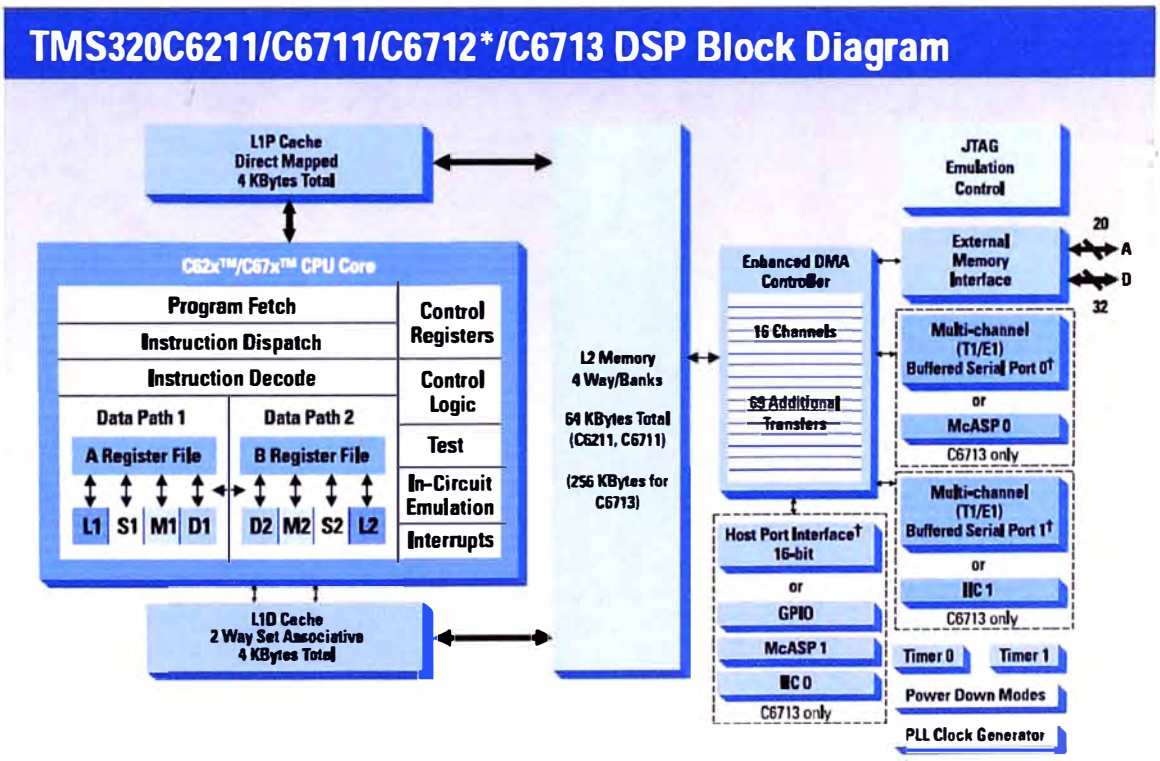


Fig. 7.8 Periféricos para la familia C67X (45)

7.1.7 Code Composer Studio

Es un completo ambiente de desarrollo integrado (IDE) que soporta todas las plataformas de DSP de Texas: TMS320C6000, Texas TMS320C5000, Texas TMS320C2000.

Características y beneficios:

- Ambiente de desarrollo que integra todas las herramientas dentro de una "easy-to-use" aplicación.
- Herramientas para análisis en Tiempo Real sin interferir en el programa del DSP.
- Compilador con herramientas de optimización de código orientado a tamaño y performance.
- Kernel escalable DSP/BIOS.
- Visualización de datos con posibilidad de múltiples formatos.
- Arquitectura open plug-in que te permite integrar herramientas especializadas de terceros.
- Real-Time Jtag escaneo.

El entorno de trabajo se muestra en la Fig. 7.9.

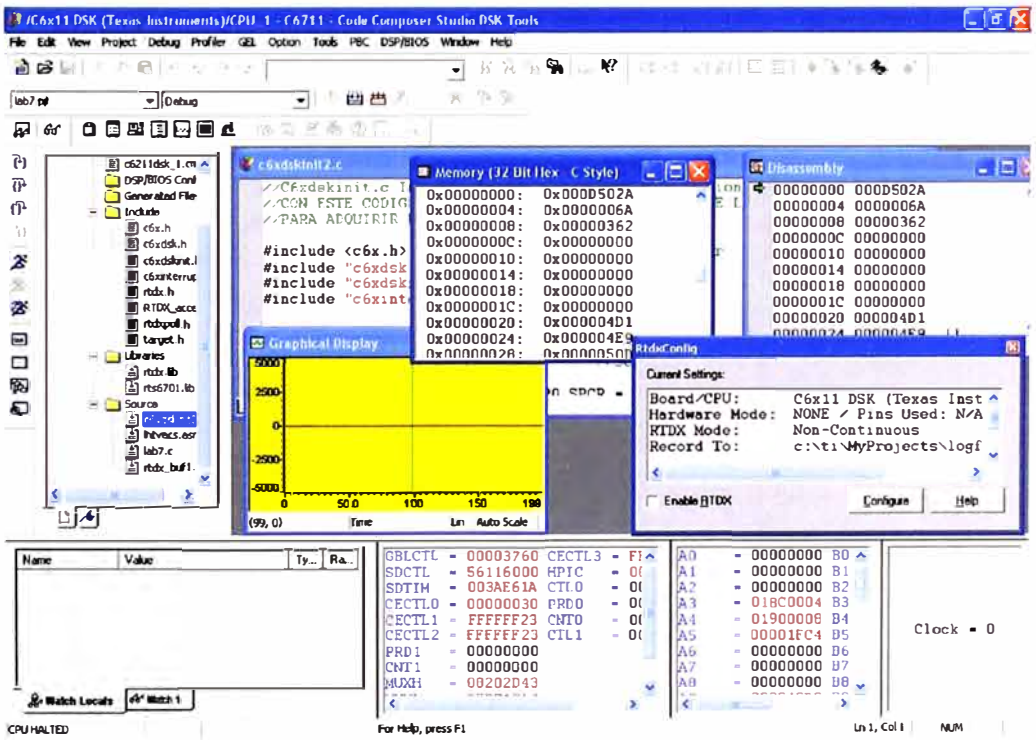


Fig. 7.9 Entorno de desarrollo Code Composer Studio (45)

7.2 Desarrollo del software en el Code Composer

De entre los componentes que nos ofrece el Code Composer para el desarrollo de software podemos mencionar las siguientes:

- TMS320C6000 code generation tools.
- Code Composer Studio Integrated Development Environment (IDE).
- DSP/BIOS plug-ins and API.
- RTDX plug-in, host interface, and API.

Estos componentes trabajan juntos tal como se muestra en la Fig 7.10.

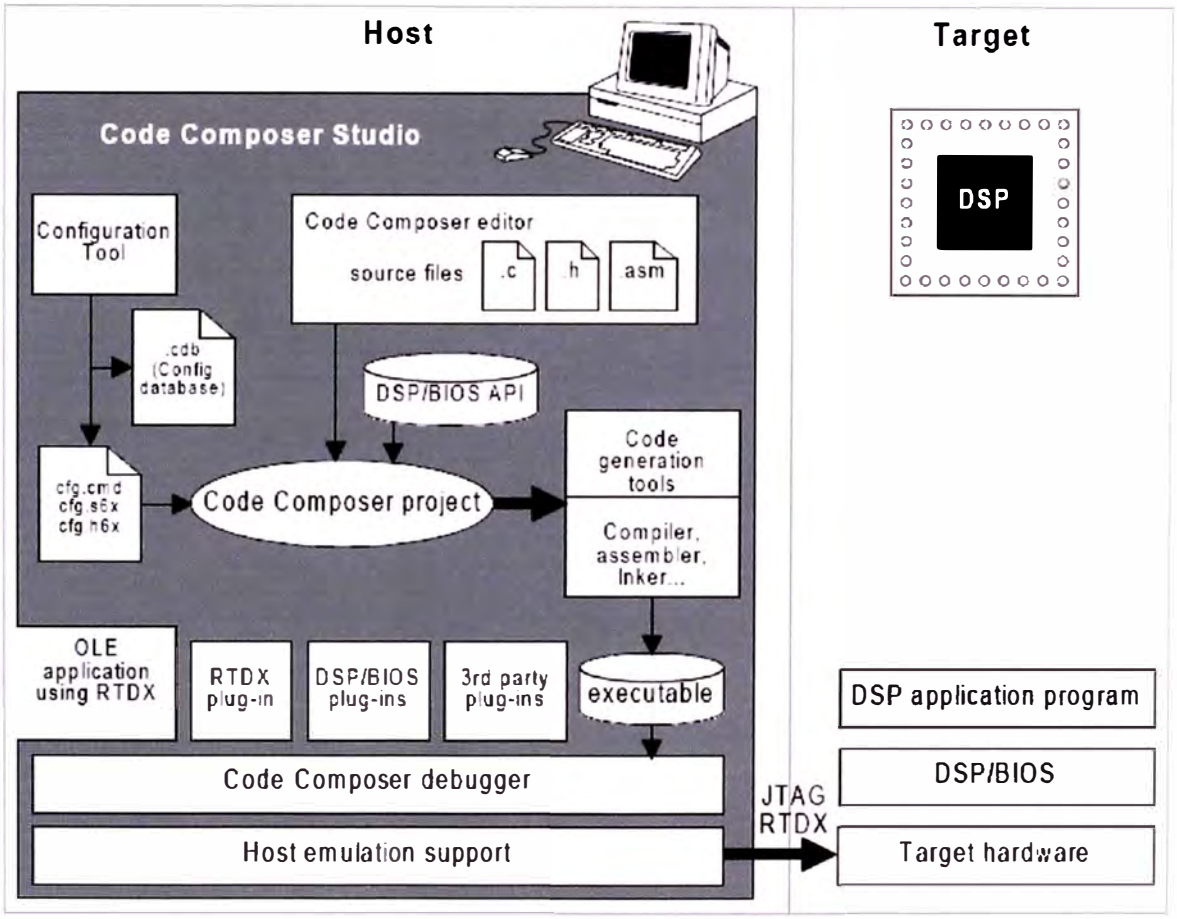


Fig. 7.10 Componentes en el desarrollo de Software en el Code Composer Studio (47)

El diagrama de flujo del desarrollo de software en el Code Composer para los DSP de la familia C6X se muestra en la Fig. 7.11.

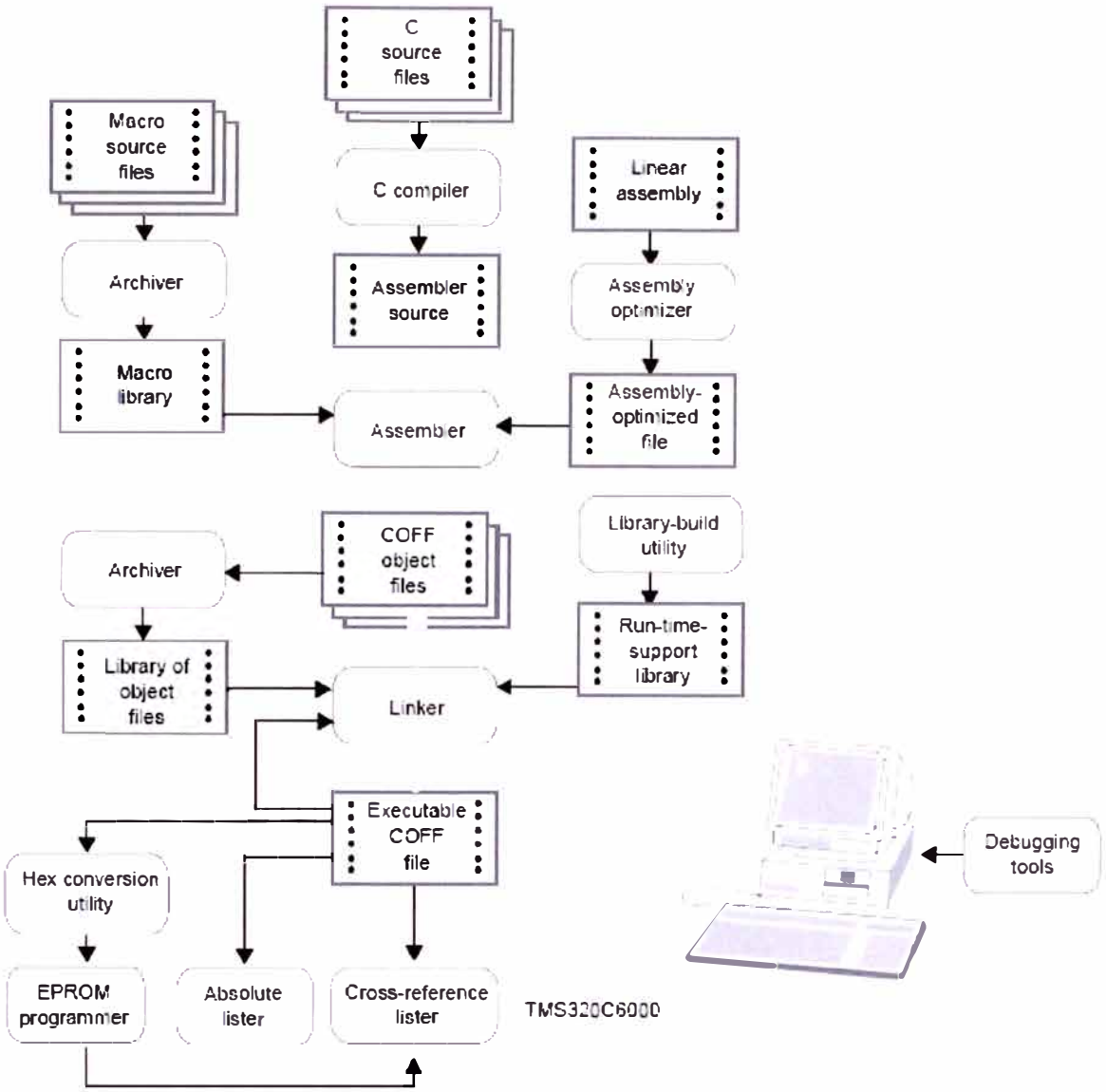


Fig. 7.11 Diagrama de Flujo para la generación de software en el Code Composer (47)

7.2.1 Creación del proyecto

Abrir el Code Composer Studio, Ir al Menú Project → New. Se abrirá la ventana mostrada en la Fig. 7.12, donde se introduce el nombre y la ruta del proyecto a crear.

El tipo de proyecto es un archivo ejecutable (.out) es decir es un programa ejecutable por el Target (DSP TMS320C67XX) el cual está totalmente compilado, ensamblado y enlazado. Al hacer clic en Finalizar, automáticamente se crea un archivo de proyecto lab1.pjt el cual guarda las rutas y todos los archivos relacionados al proyecto.

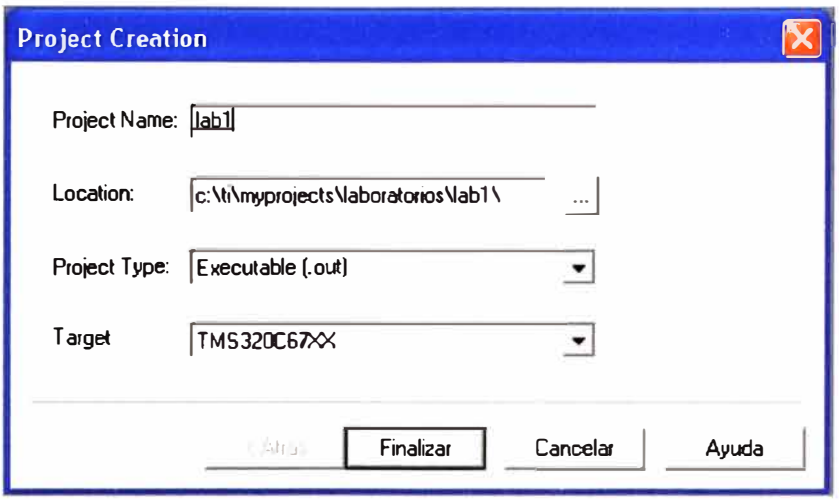


Fig. 7.12 Creación de un proyecto en el Code Composer Studio.

Notar que en la ventana Project View (Fig. 7.13), el único archivo agregado por default al entorno de trabajo (en GEL files) es el DSK6211_6711.gel : El cual es un archivo usado por el Code Composer en conjunto con el Target (DSKC6711 en nuestro caso), independientemente de cualquier proyecto y el cual permite al CCS hacer las siguientes funciones: quick test , reset del CPU, inicializar el EMIF (External Memory Interface) con lo cual el CCS se entera de la geometría y tamaño de la memoria, entre otras cosas.

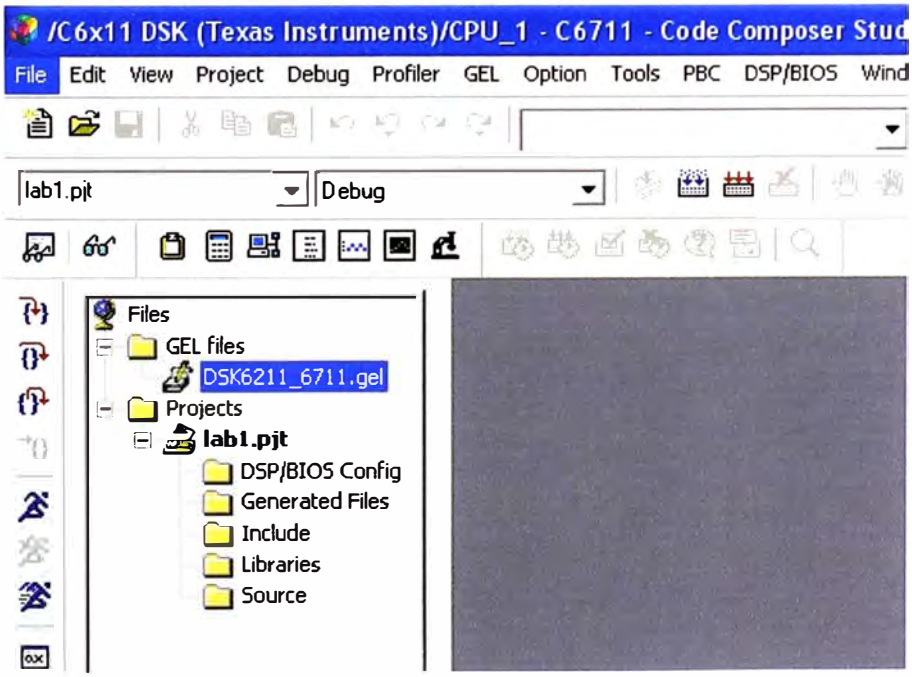


Fig. 7.13 Vista de la ventana Project.

7.2.2 Añadiendo archivos básicos de soporte y librerías.

Hay varios archivos, dados por Texas Instruments que deben ser añadidos a los proyectos. Dichos archivos contienen definiciones de registros, rutinas de inicialización de la tarjeta DSK y soporte de librerías pre-compiladas. La necesidad de agregar más o menos archivos de soporte y/o librerías depende del tipo de aplicación que vayamos a desarrollar. Para nuestro caso debemos agregar los siguientes archivos:

- **C6211dsk.cmd**: Este archivo es un archivo de tipo Linker Command Files, el cual es muy importante porque permite configurar la memoria del sistema mediante una distribución eficiente de las secciones (de código o data) dentro del mapa de memoria. Este archivo informa al enlazador cómo están organizados los vectores, la memoria interna, externa y la flash.
  Este archivo consta de éstas dos partes principales:

```
MEMORY
{
    VECS:  o=00000000h   l=00000200h          /* interrupt vectors */
    PMEM:  o=00000200h   l=0000FE00h   /* Internal RAM (L2) mem */
    BMEM:  o=80000000h   l=01000000h   /* CE0, SDRAM, 16 MBytes  */

}
```

Con ésta directiva que empieza con la palabra Memory, se delimitan las secciones de la memoria especificando exactamente el mapeo de todas las memorias físicas que contiene la tarjeta Dsk, memoria interna (L2 Cache), Memoria Externa (RAM, Flash, EEPROM), las cuales se traducen en los espacios de memoria VECS, PMEM y BMEM para éste caso.

```
SECTIONS
{
   .intvecs      >     VECS
   .text         >     PMEM
   mydata        >     PMEM
   mycode        >     BMEM
   .rtdx_text    >     BMEM
```

```
    .far          >     BMEM
    .stack        >     BMEM
    .bss          >     BMEM
    .cinit        >     BMEM
    .pinit        >     PMEM
    .cio          >     BMEM
    .const        >     BMEM
    .data         >     BMEM
    .rtdx_data    >     BMEM
    .switch       >     BMEM
    .sysmem       >     BMEM

}
```

Con la directiva Sections se distribuye el código escrito en las diferentes partes de la memoria. La sección .text generalmente contiene las primeras partes de un código extenso por lo que debe ir ubicada en una memoria "rápida", la cual en este caso es la PMEM (Cache de nivel 2 L2 Cache), el resto del código y variables adicionales van en la memoria externa BMEM (Banco de memoria SDRAM de 16Mb).

Observar que se han creado dos secciones llamadas "mydata" y "mycode", cada una de estas secciones nos ayuda a direccionar ya sea partes del código o variables a un determinado segmento de memoria (interna o externa).

- **rts6701.lib:** Es una actualización de la librería rts6700.lib (Run Time Soport Library) que viene incluido en el Code Composer Studio, la cual contiene código de soporte para nuestras aplicaciones y permite al Code Composer la depuración en Tiempo Real, es obligatorio incluir éste archivo en todo proyecto.

- **c6xdskinit.c:** En este archivo se incluyen las definiciones de las funciones que inicializan los periféricos de la tarjeta DSK, para tenerlo listo para la adquisición de los datos a través del Codec. Las rutinas que se incluyen en este archivo son las siguientes:

  void mcbsp0_init(): Para inicializar el puerto serial.
  void mcbsp0_write(int): Rutina de escritura hacia el Codec a través del puerto serial.
  int mcbsp0_read(): Rutina de lectura del Codec a través del puerto serial.

void TLC320AD535_Init(): Inicialización del Codec a través del puerto Serial.
void c6x_dsk_init(): Reseteo de interrupciones y configuración del EMIF.
void comm_poll(): Para hacer una lectura/escritura del Codec via Polling.
void comm_intr(): Dispara (triggering) todas las anteriores funciones.
int input_sample(): Rutina que hace uso de mcbsp0_read().
void output_sample(int): Rutina que hace uso de mcbsp0_write(int).

La ejecución de todas las funciones anteriores nos deja todo listo para trabajar con todos los periféricos disponibles de la tarjeta DSK. Es un archivo muy útil y modificable de acuerdo a cada necesidad., nos evitará escribir desde el comienzo todo el código para inicializar la tarjeta DSK.

- **intvecs1.asm:** Definición de las interrupciones en el DSP. Aquí se crea la IST (Interrupt Service Table). En este caso la única interrupción que vamos a necesitar es la interrupción producida por el Codec cada vez que genera una muestra de la señal de entrada, la cual la ubicamos por conveniencia como Interrupción 10.

Hasta el momento, nuestro entorno de trabajo luce tal como se muestra en la Fig. 7.14:

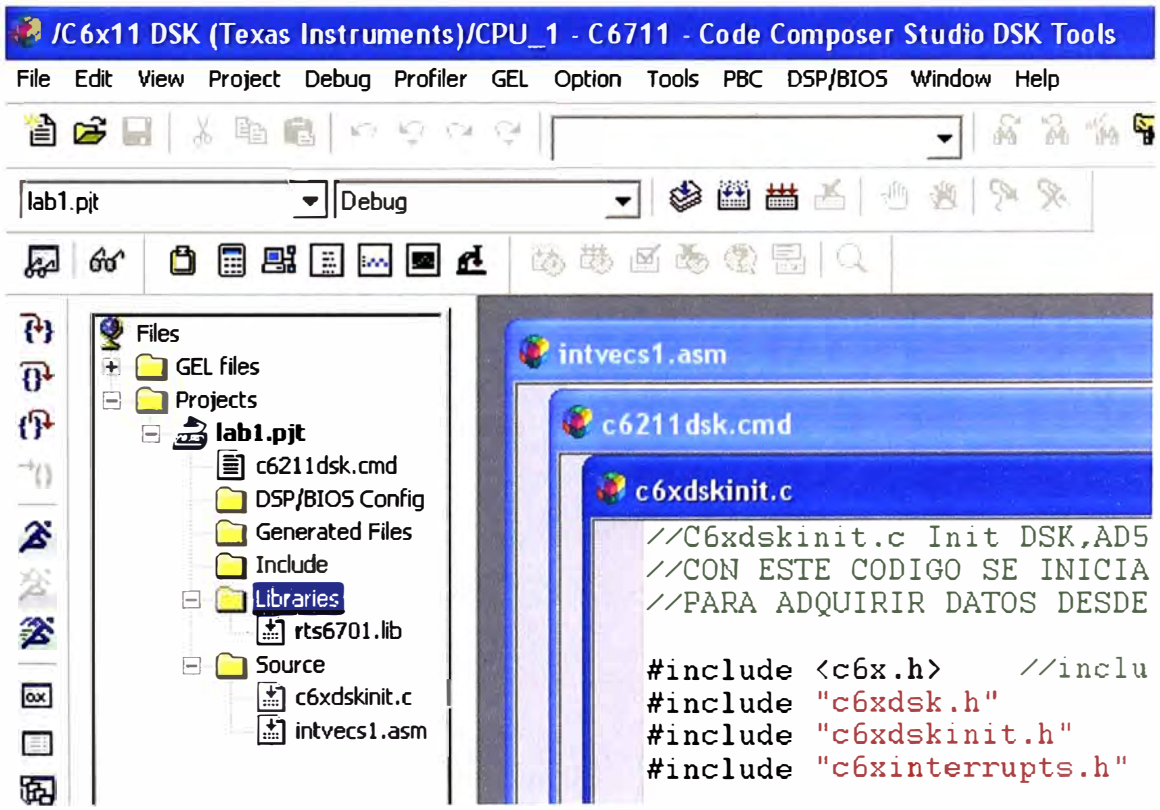


Fig. 7.14 Proyecto con los archivos de soporte añadidos.

Se pueden observar los 4 archivos que hemos agregado, pero notar también que la primera parte del archivo c6xdskinit.c hace referencia a 4 archivos de cabecera, los cuales aparecen después de cada directiva #include. Para que estos archivos sean visibles en el proyecto hacer lo siguiente:

Menu Project → Scan All Dependencies.

Con lo cual ya podemos ver los 4 archivos en la ventana Project View ubicados en la carpeta Include:

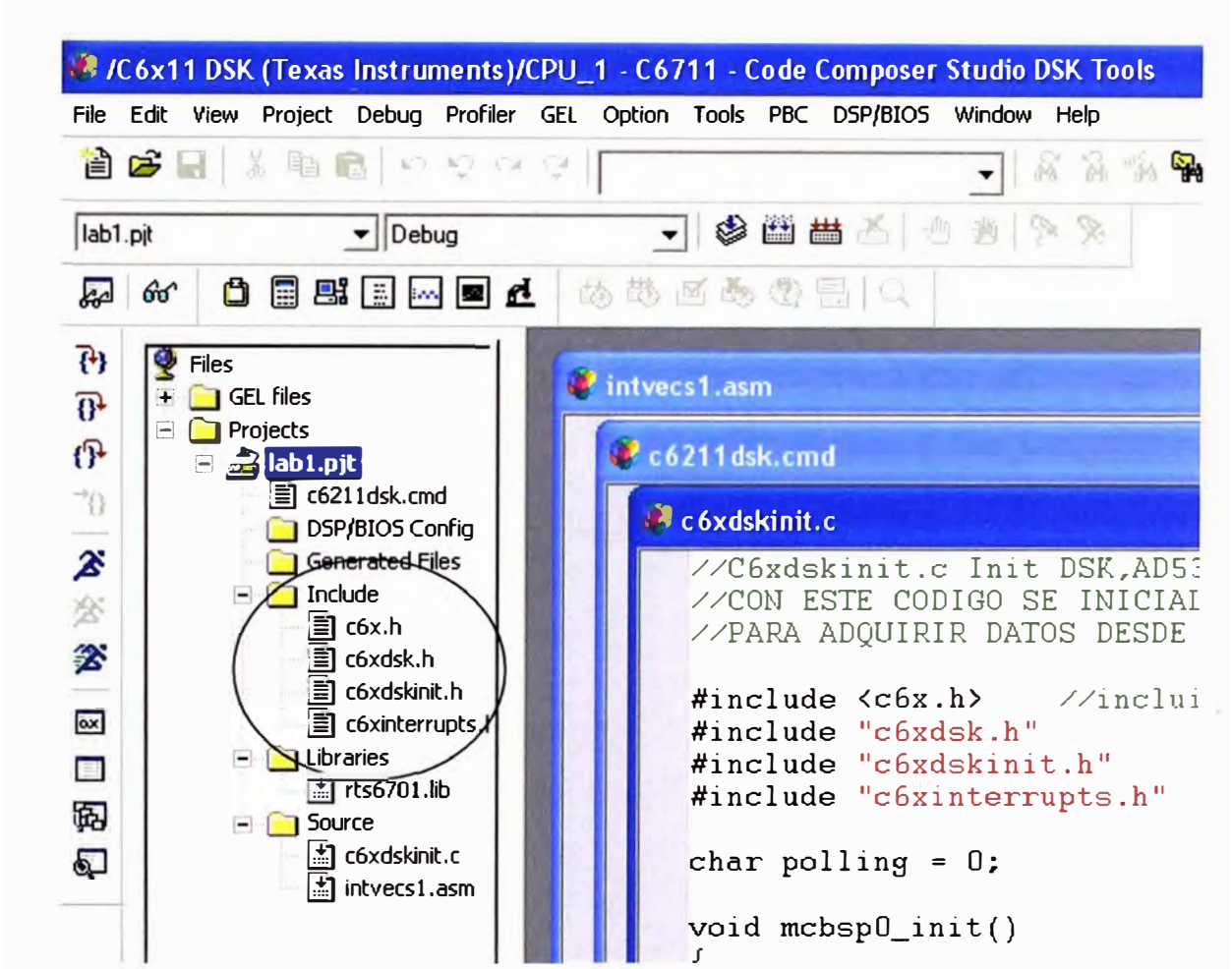


Fig. 7.15 Visualización de las dependencias en la ventana Project.

Con los archivos de soporte ya agregados, se procede a añadir los archivos de código fuente que ejecutarán las funciones del codificador. El decodificador se ha implementado como otro proyecto separado del codificador.

7.2.3 Compilando el proyecto

Lo siguiente es compilar el proyecto y obtener un archivo ejecutable que pueda ser ejecutado por el DSP. Para ello nos vamos al menú Project → Build. Si todo va bien, la ventana de salida "Output Window" aparecerá en la parte inferior indicándonos que la compilación se realizó con éxito y sin errores, tal como se muestra en la Fig. 7.16

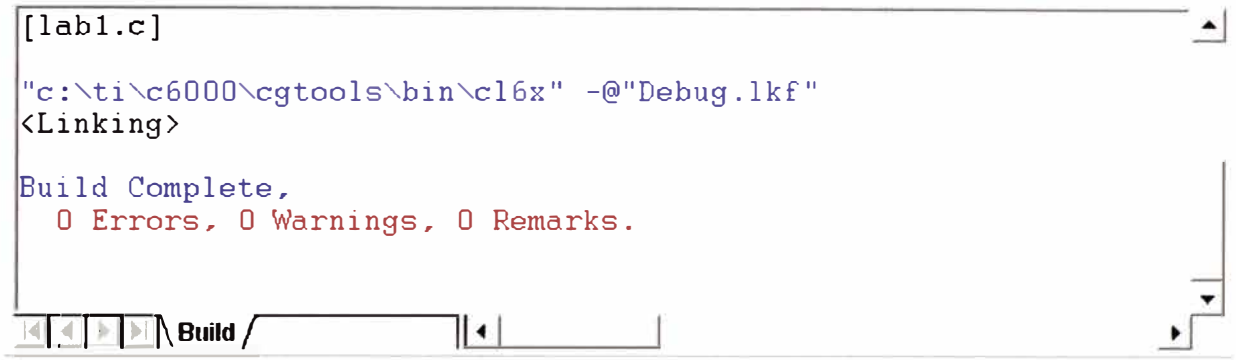


Fig. 7.16 Compilación exitosa del proyecto.

Cuando el CCS construye el archivo ejecutable, los archivos fuente y de cabecera codificados en lenguaje C son convertidos a código ensamblador a través del compilador. Luego, éste código ensamblado se convierte en un archivo con formato de archivo de objeto (COFF) que contiene las instrucciones del programa organizados en módulos. Finalmente el enlazador organiza esos módulos y la biblioteca en tiempo de ejecución (rts6701.lib) en posiciones de memoria para crear un archivo ejecutable que pueda ser descargado a la memoria del DSP. Cuando este ejecutable es cargado al DSK, las instrucciones de programa ensambladas, las variables globales y las bibliotecas en tiempo de ejecución son cargadas en las posiciones de memoria especificadas por el compilador.

7.2.4 Cargando y Ejecutando el programa en el DSP

Para cargar el programa a la memoria del Dsp, primero debemos resetear el CPU, nos vamos a **Debug→ Reset CPU**, una vez reseteado el CPU (CPU HALTED) nos vamos al menu **File→ Load Program** y escogemos el archivo lab1.out.

Una vez cargado el proyecto nos aseguramos que un micrófono o la salida de cualquier dispositivo de audio (walkman, discman) estén conectados en el Plug IN del DSK y el Plug Out lo conectamos a unos altavoces.

NOTA: Si se va a usar un micrófono, éste debe ser del tipo auto alimentado, ya que el Codec del DSK no se comporta como una tarjeta de sonido de una PC (la cual envía un phantom voltage que alimenta al micrófono).

Con todo lo anterior listo, nos vamos al menú **Debug → Run** y observe que en la parte inferior de la ventana se cambia el estado de "**CPU HALTED**" a "**CPU RUNNING**"

7.3 Descripción del modo de trabajo en Punto Fijo y Punto Flotante

En general, los procesadores DSP se pueden programar de dos formas: En punto Fijo y en punto Flotante (además de assembler). Cada uno presenta ciertas características y el escoger uno u otro modo de programación depende del tipo de proyecto que se va a desarrollar, tiempos, precisión, etc. El cuadro comparativo se muestra en la TABLA Nº 7.1.

TABLA Nº 7.1 Comparación entre la programación en punto fijo y punto flotante.

Característica	*Punto Fijo*	*Punto Flotante*
Presente en todos los DSP?	*Si (todos los DSP se pueden programar en punto Fijo)*	*No. (Se requiere que el DSP tenga módulos para operaciones de punto Flotante)*
Potencia Alcanzable	*Muy potente (Solo superable por assembler)*	*Potente, ya que las operaciones en punto flotante consumen mas ciclos de reloj que las operaciones en punto fijo.*
Dificultad de Programación	*Es más dificil de programar que en el caso de punto flotante, ya que se tiene que ajustar la precisión constantemente.*	*Relativamente fácil y directo. No requiere cuidado de la precisión porque no hay límite en el tamaño de las variables en bits.*
Tiempo de Ejecución	*Se ejecuta mas rápido que si estuviera implementado en punto flotante.*	*Requiere mas tiempo de ejecución ya que las operaciones en punto flotante requieren mas ciclos de reloj.*
Tamaño del Código	*En general es mas grande que en el caso de punto flotante.*	*Es reducido, sin embargo cuando se traslada a aseembler, puede resultar mas grande que en el caso de punto fijo.*
Tiempo de Desarrollo	*El tiempo de desarrollo de un proyecto en punto fijo es más grande que en el caso de punto flotante.*	*Tiempos de desarrollo relativamente mas cortos.*
Precisión de los resultados	*Depende de la pericia del programador.*	*Mucha precisión.*

CAPITULO VIII

DESCRIPCIÓN DEL CODIFICADOR IMPLEMENTADO EN EL DSP TMS320C6711

En el presente capítulo se muestran y explican las rutinas implementadas para el codificador y decodificador de voz en sus dos versiones: punto fijo de 16 bits y punto flotante. Así mismo, se muestran los resultados más importantes para cada etapa. Cabe mencionar que las imágenes del presente capítulo son tomadas directamente de la pantalla del Code Composer Studio, mediante el comando "print-page"

8.1 Señal de Entrada al Codificador

Para ambas versiones, la señal de entrada es la misma. La señal de voz de entrada al codificador es una señal estándar PCM de 16 bits muestreada a 8 KHz. La referencia para la tasa de compresión total del codificador se calcula teniendo como base la tasa entrante de la señal PCM la cual es:

16 bits/muestra * 8000 muestras /segundo = 128 000 bits / segundo

Por consiguiente, la tasa de bits de la señal entrante es 128 Kbps.

A continuación se muestra la señal entrante que ha sido cargada a la memoria del DSP, su configuración es:

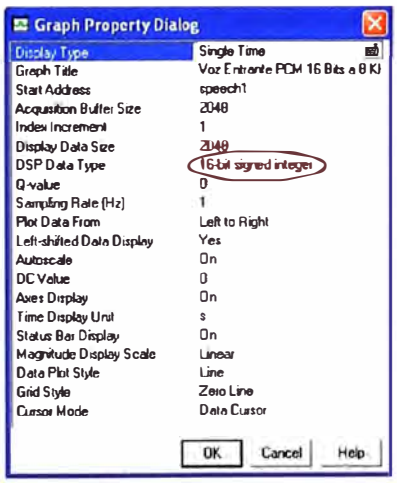


Fig. 8.1 Configuración para visualización de la señal entrante

La Fig. 8.2 nos muestra un segmento de aproximadamente un segundo de voz entrante (8160 muestras), como el límite de visualización de gráficos del Code Composer es de 2048, se ha tenido que partir en cuatro ventanas,

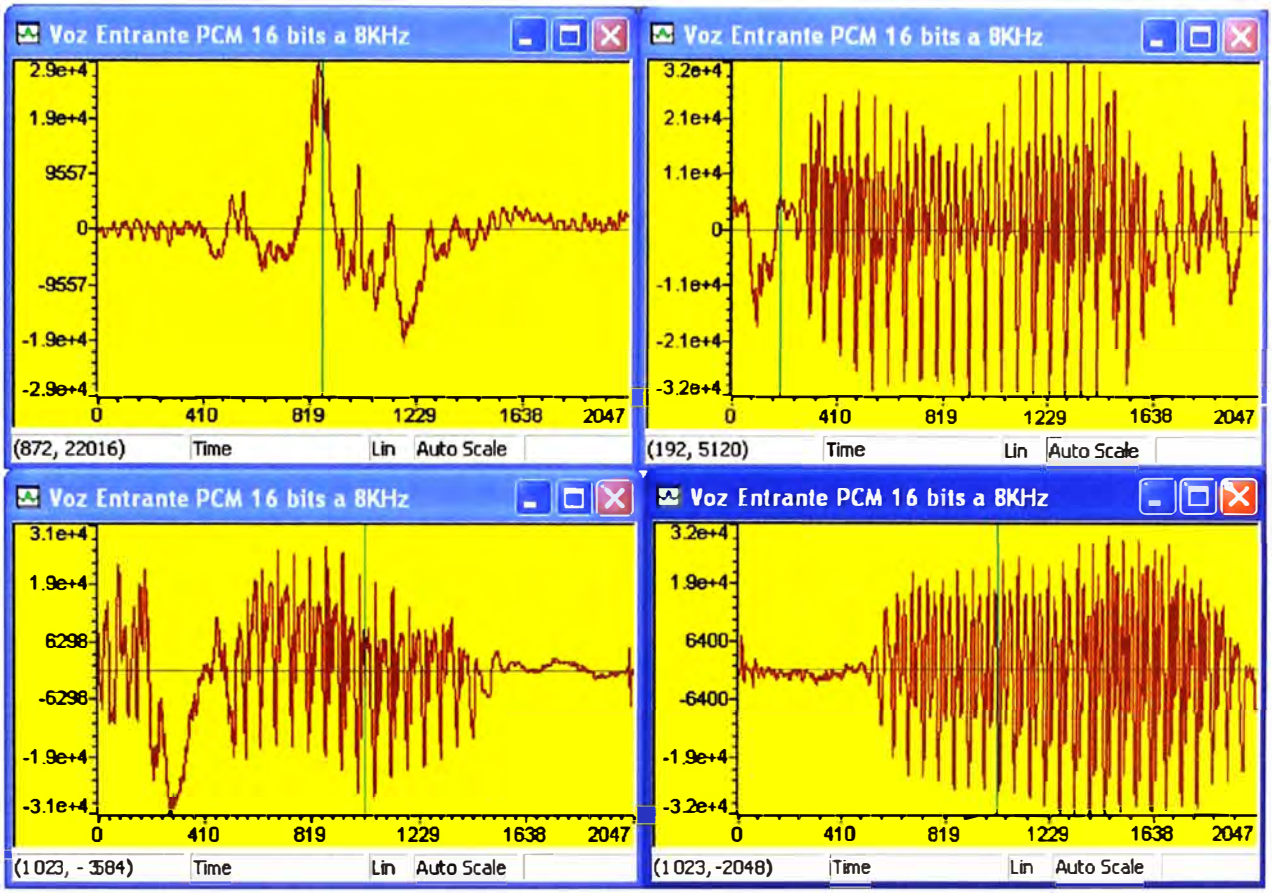


Fig. 8.2 Señal de voz entrante PCM 16 bits a 8 KHz (1 segundo / 8160 muestras de voz aprox.)

8.2 Ventana de Hamming

La ventana de Hamming es un componente principal en el codificador implementado, en la fase inicial del algoritmo se usa principalmente para la estimación de la autocorrelación (Ver ecuaciones (3.18) y (3.19)). La ventana de Hamming es de longitud N = 240 que corresponde al tamaño de la trama de análisis.

En la figura Fig. 8.3 se muestra la ventana de Hamming para la versión punto fijo (Q2.14), y punto flotante. Observar las escalas en el eje vertical.

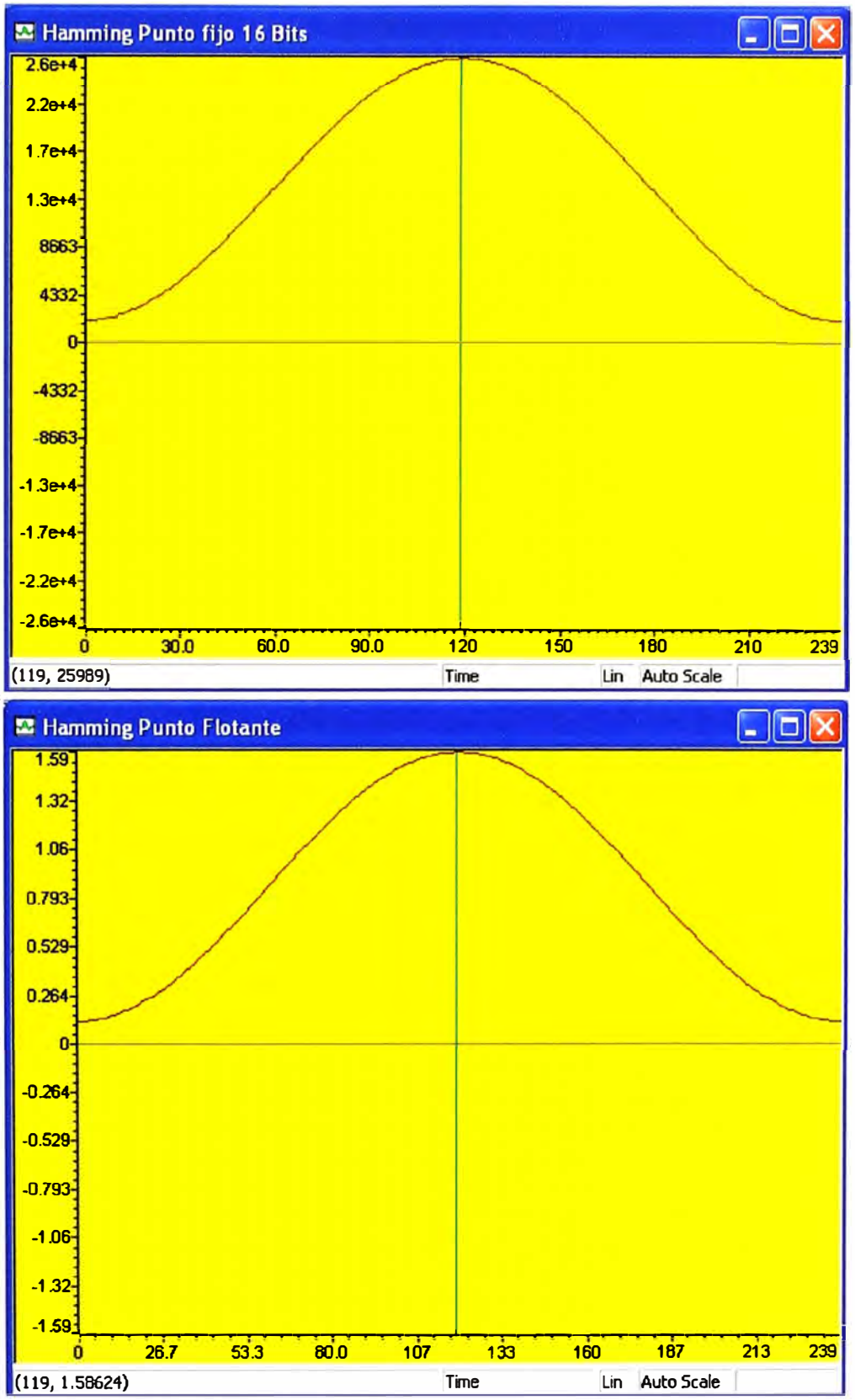


Fig. 8.2 Ventana de Hamming para punto fijo (arriba) y punto flotante (abajo)

8.3 Esquema general del Algoritmo para la versión en Punto Fijo

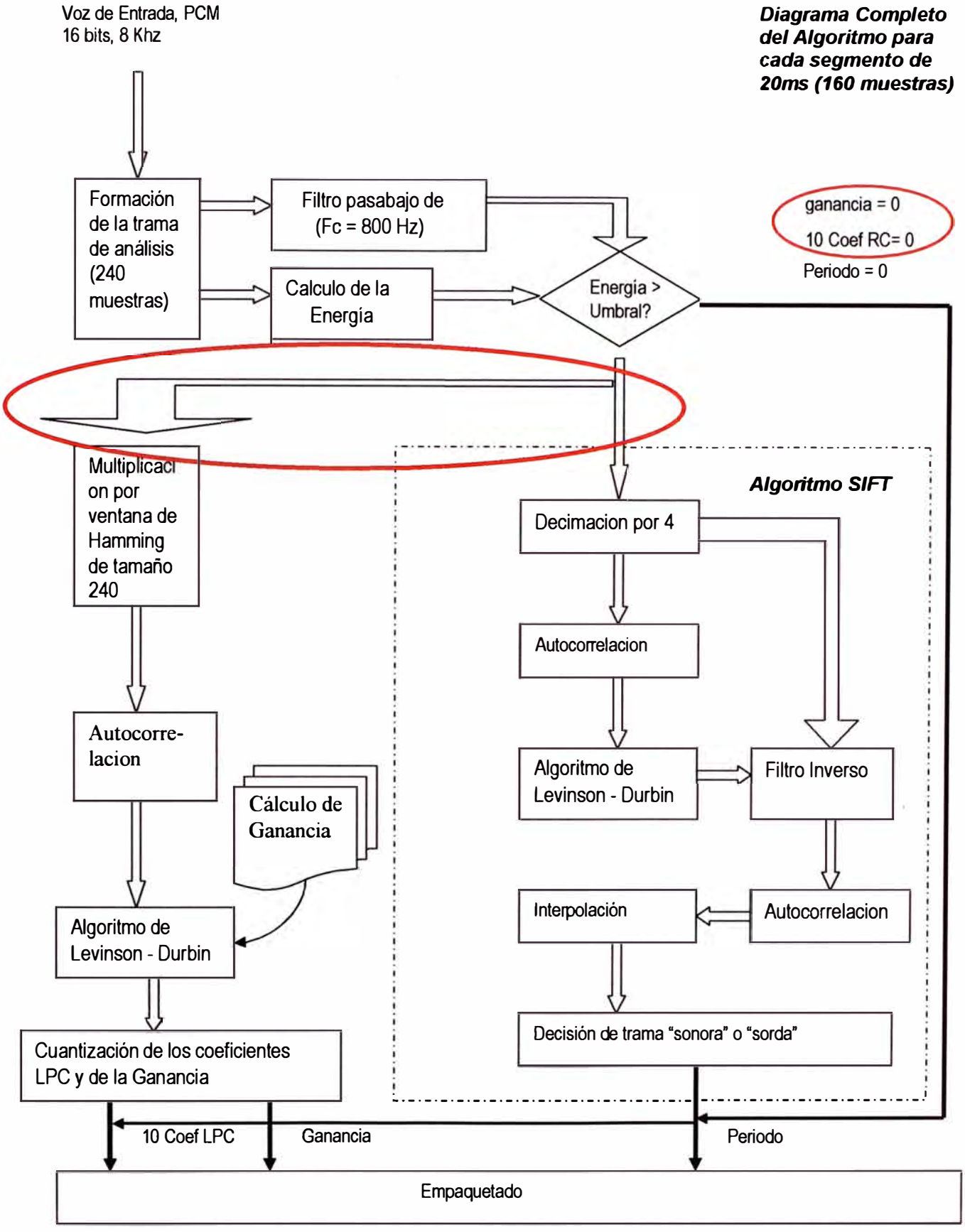


Fig. 8.3 Diagrama general del codifcador implementado en Punto Fijo.

La figura Fig. 6.1 del capítulo 6 corresponde al esquema del codificador que se implementó en la plataforma System Generator. Dicho esquema es el mismo que emplea la versión en punto flotante. Por otro lado, la figura Fig. 8.3 nos muestra la variante para el esquema del codificador en el caso de la versión para punto fijo.

En la figura Fig. 8.3 se puede apreciar que las diferencias están señaladas mediante un círculo de color rojo, y se describe a continuación:

- El algoritmo de la versión en punto fijo de 16 bits tiene manejo de precisión limitada (a 16 bits), es por eso que no se puede manipular de manera normal valores muy pequeños que escapen fuera del rango ofrecido por el formato de 16 bits. Por lo tanto es necesario que el segmento de análisis tenga un nivel de energía mínimo de tal forma que garantice que sus valores se encuentren del rango manejable por los algoritmos de punto fijo. Esto se visualiza claramente en la figura Fig. 8.3 donde se aprecia que la mayoría de los bloques de cálculo (periodo pitch, ganancia y coeficientes RC) son accesados solamente si la energía sobrepasa un umbral. Caso contrario, se setean todos los valores a cero, es decir, periodo pitch igual a cero, ganancia igual a cero y todos los coeficientes RC a cero.

- En el caso de la versión en punto flotante, la cual se muestra en la figura Fig. 6.1, el bloque de cálculo de la ganancia y los coeficientes RC siempre es accesado sin importar del nivel de energía de la trama de entrada. Esto es debido a que las operaciones en punto flotante no tienen problemas con la precisión y pueden manejar valores tan pequeños como 10^{-10} sin ningún problema.

- Adicionalmente, cada parte o bloque del algoritmo de punto fijo maneja sus propias variables a una determinada precisión, teniendo de esta manera varios formatos a lo largo de los bloques que conforman la versión en punto fijo. Esto es así, porque se trata que cada bloque ofrezca máxima precisión al siguiente bloque. La precisión con que trabaja cada bloque del algoritmo en punto fijo se denota mediante la notación *Qx.y*, la cual significa que la variable está con precisión de "*x*" bits para la parte entera y con "*y*" bits para la parte decimal, con un total de "*x + y*" bits.

8.4 Filtro Pasabajo

El filtro pasabajo con frecuencia de corte de 800Hz, se muestra en la ecuación 6.1 del capítulo 6 y su respuesta en frecuencia se muestra en la figura Fig. 6.20. Para el caso de la versión en punto fijo, los coeficientes están con formato Q1.15 y se muestran en la figura Fig. 8.4.

Name	Value	Type	Radix
fa_fixed16	0x00005A98	short[6]	hex
[0]	25335	short	dec
[1]	-25989	short	dec
[2]	6779	short	dec
[3]	-16052	short	dec
[4]	25989	short	dec
[5]	9937	short	dec

Watch Locals | Watch 1

Fig. 8.4 Coeficientes en formato punto fijo Q1.15 para el filtro pasabajo.

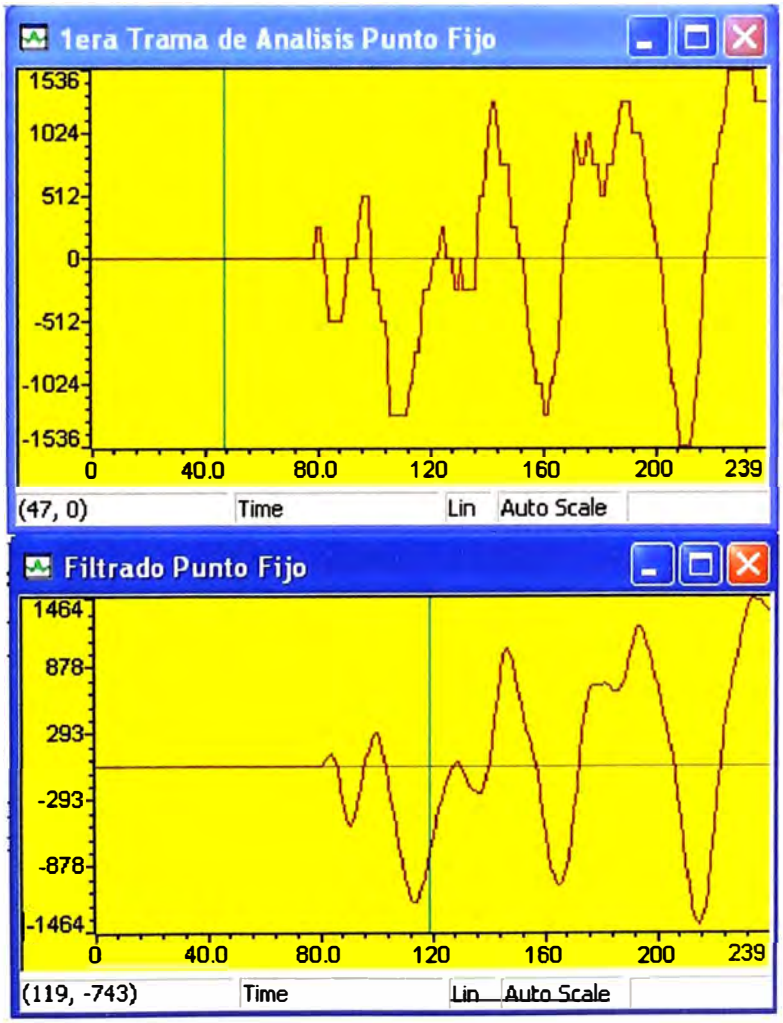


Fig. 8.5 Entrada en punto fijo y filtrado en punto fijo en formato Q1.15

Para el caso de la versión en punto flotante, las salidas se muestran en la figura Fig. 8.6.

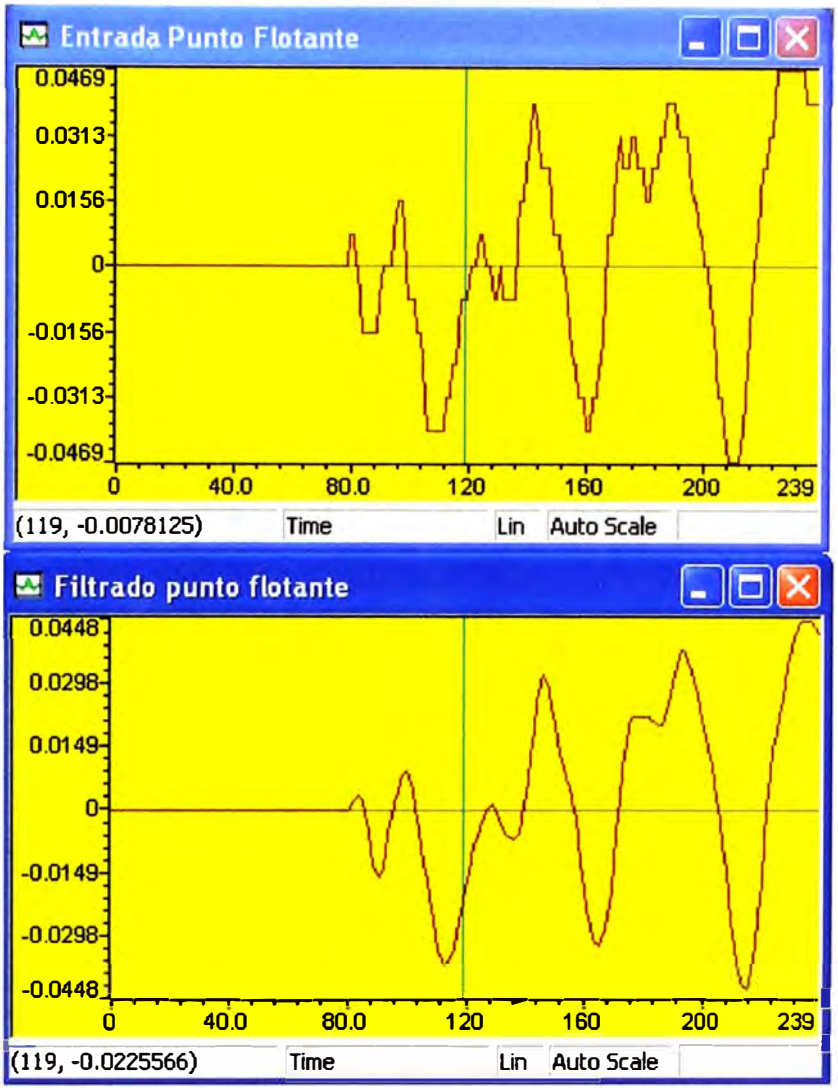


Fig. 8.6 Entrada en punto flotante y filtrado en punto flotante

Los segmentos de código correspondientes a las implementaciones tanto en punto fijo como en punto flotante se encuentran en el CD en la ruta "*..\DSP*".

Allí se puede observar claramente que, en general, las rutinas en punto fijo tienen más líneas de código que las rutinas en punto flotante y esto es debido al cuidado que se debe tener con la precisión para cada etapa.

8.5 Calculo de la energía del segmento de análisis

El cálculo de la energía del segmento de análisis es fundamental sobretodo para el algoritmo en punto flotante, ya que garantiza que los niveles de la señal se encuentran dentro de los rangos manejables por las rutinas en punto fijo. La energía se calcula de la trama de análisis (240 muestras). A continuación se muestran las rutinas tanto en punto fijo como en punto flotante que calculan la energía.

```
/*calculo de la energia punto flotante*/

            energia=0;
            for(i=0;i<BUFLEN;i++)
                  energia+=w_s[i]*w_s[i];


/*calculo de la energia punto fijo*/

            energia=0;
            for(i=0;i<BUFLEN;i++)
                  energia+=(fixmul32(in[i],in[i]))>>6; //en Q8.24
```

Los valores de energía calculados se muestran en la figura Fig. 8.7 para las dos versiones.

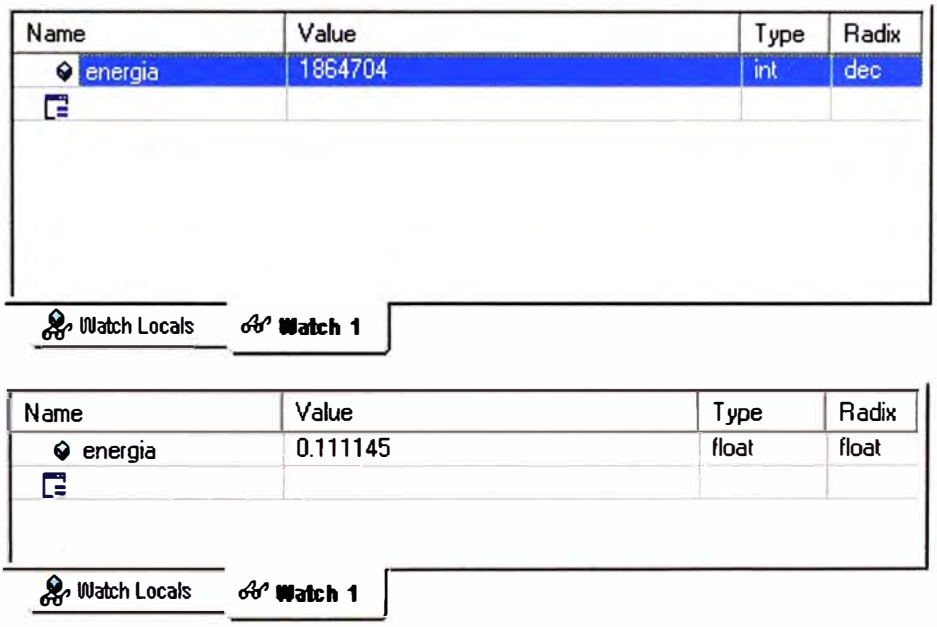


Fig. 8.7. Energía del segmento punto fijo (arriba), punto flotante (abajo)

El formato del valor de la energía en punto fijo es Q8.24, con lo cual se demuestra fácilmente que los valores mostrados en la figura Fig. 8.7 son equivalentes.

$$0.111145 = 1864704 / (2^{24})$$

Como se observa en los segmentos de código mostrados, para el caso de punto fijo hay que emplear una rutina dedicada de multiplicación de valores de punto fijo y además el comando shift (>>) para desplazar a nivel de bits.

El umbral usado para la energía es de 0.1 (1677721 en formato Q8.24) en la versión en punto fijo, mientras la versión en punto flotante puede tener un umbral indistintamente pequeño. En este caso se ha superado el umbral y proceden el resto de bloques.

8.6 Algoritmo Sift: Decimación por 4

Tal como se describió en el capítulo 4, el siguiente bloque del algoritmo Sift realiza una decimación (down-sample) por el valor de 4 al segmento de análisis filtrado (de 240 muestras). El segmento de análisis filtrado se muestra en las figuras Fig. 8.5 y 8.6 para los dos casos. Las secuencias que han sido decimadas se muestran en las figuras Fig. 8.8 y 8.9. Se puede apreciar que ahora las secuencias tienen 60 muestras (el segmento original de análisis tiene 240 muestras)

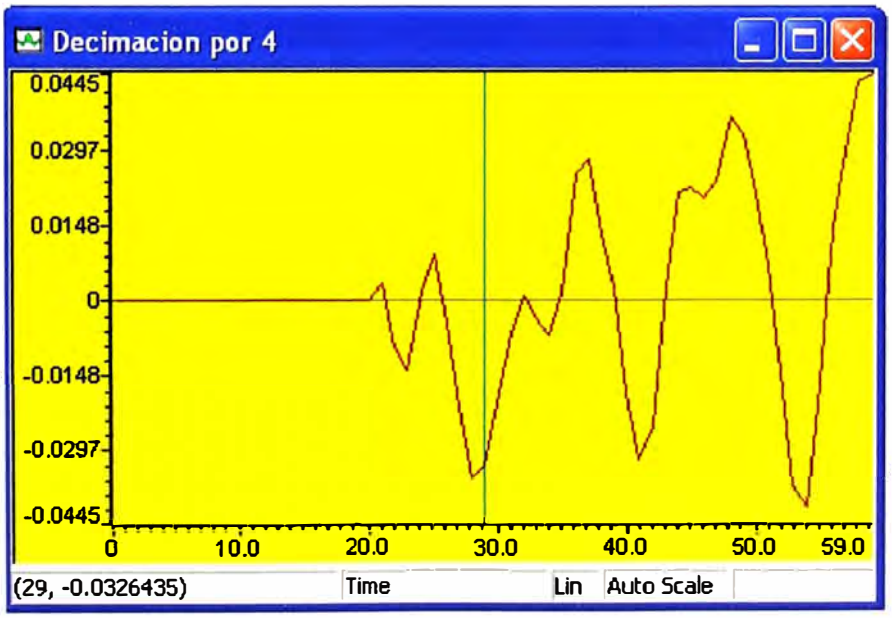


Fig. 8.8. Señal diezmada por 4, punto flotante

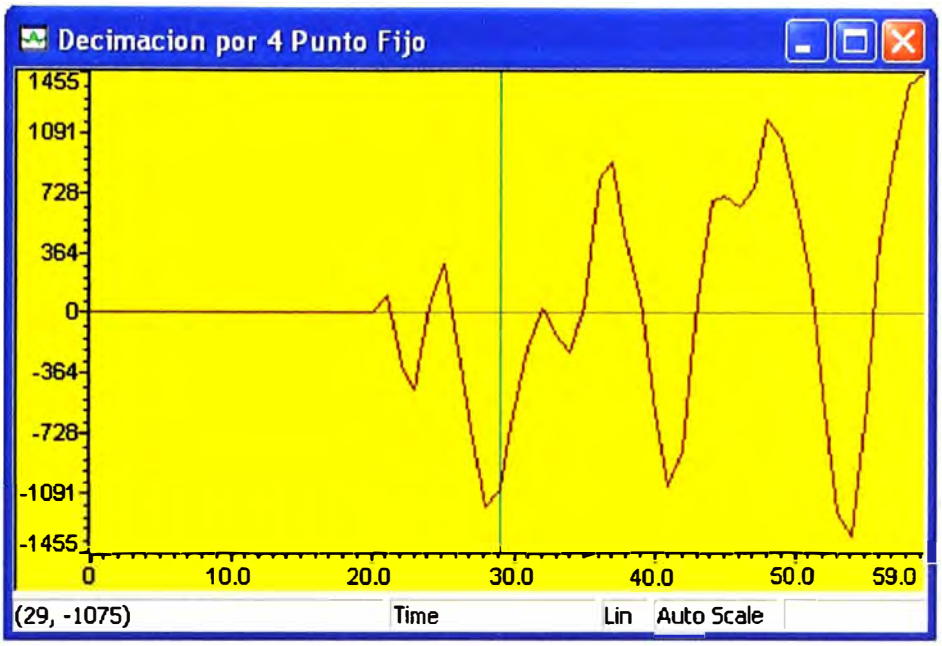


Fig. 8.9. Señal diezmada por 4, punto fijo.

8.7 Algoritmo Sift: Cálculo de la autocorrelación de la señal diezmada

A continuación se tiene el bloque de cálculo de los coeficientes de autocorrelación. Dichos coeficientes son necesarios para el posterior cálculo de los coeficientes RC que constituirán el filtro inverso.

En punto flotante, el algoritmo no es muy complicado, y se muestra a continuación.

```
/*calculo de la Autocorrelación en punto flotante*/

      j=0;

      for (k=0; k <= PITCHORDER+6 ; k++) {
        r1[k]=0;
        for (i=0; i < BUFLEN/DOWN-j; i++) {

            r1[k]=r1[k]+deci[i]*deci[i+k];

            }
        j++;
      }
```

El esquema se complica en punto fijo debido a la precisión limitada. Para el caso de punto fijo se creó la función :

```
void fixed_autocorrelacion(fixed16 *w, fixed16 *r3, int *entero, int
rango, int flag)
```

Dicha función calcula automáticamente la precisión con la que debe operar los coeficientes de autocorrelación y devuelve dichos valores calculados.

Por ejemplo, la siguiente línea de código recibe como entrada la señal decimada (deci_fixed16), además recibe como entrada la longitud de la señal de entrada (BUFLEN/DOWN) y un flan (1). Las salidas son los valores de la autocorrelación en el array r1_fixed16 y la precisión en la variable "preci".

```
fixed_autocorrelacion(deci_fixed16, r1_fixed16, &preci, BUFLEN/DOWN,1 );
```

El código que implementa la función `fixed_autocorrelacion`, se puede encontrar en el CD en la ruta "*..\DSP\pc_fixed\main.c*". A continuación se muestra un diagrama de flujo del algoritmo que implementa dicha función.

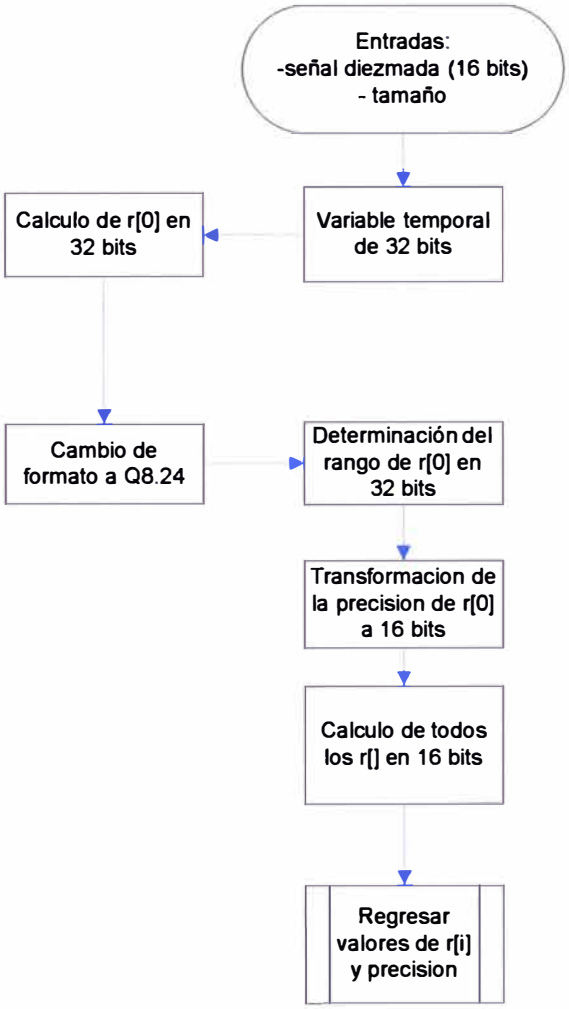
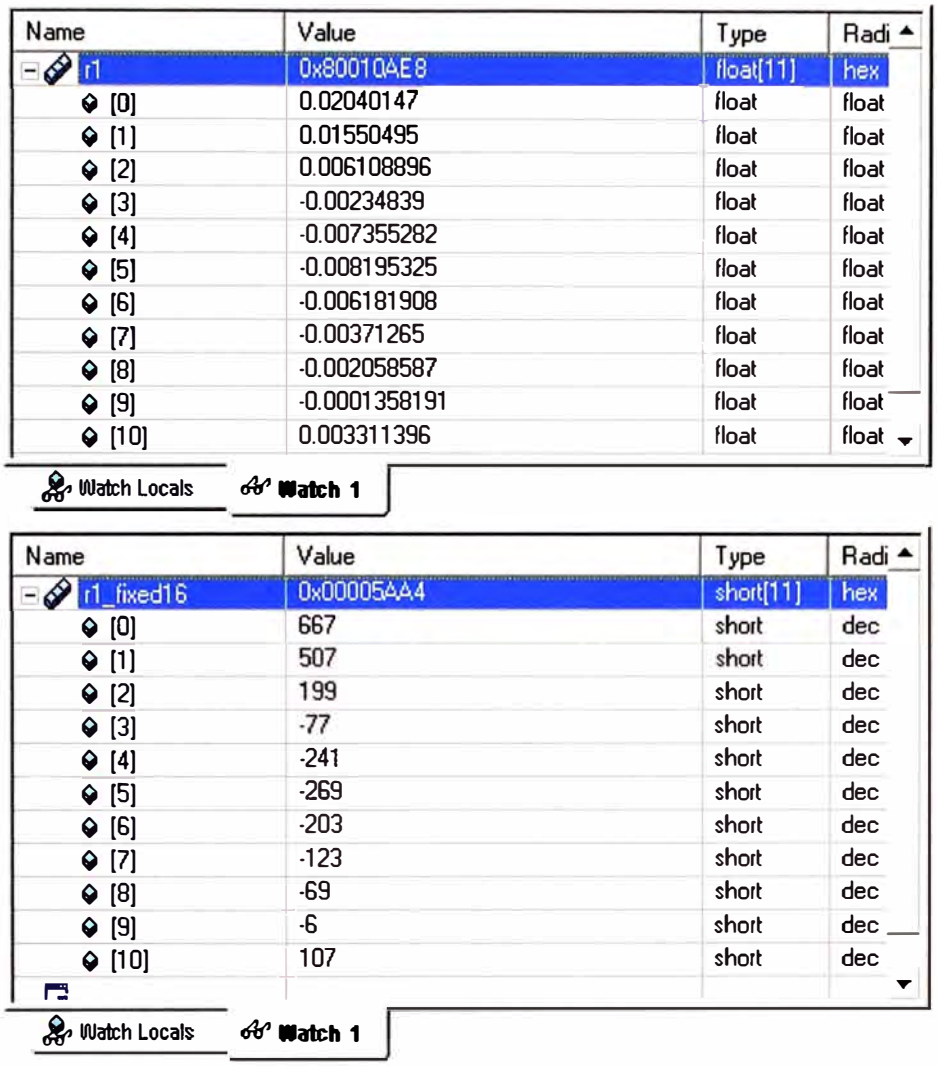


Fig. 8.10. Diagrama del Algoritmo para la autocorrelación en punto fijo.

La figura Fig. 8.11 muestra los respectivos valores calculados para los coeficientes de autocorrelación para la señal diezmada.



Name	Value	Type	Radi
r1	0x80010AE8	float[11]	hex
[0]	0.02040147	float	float
[1]	0.01550495	float	float
[2]	0.006108896	float	float
[3]	-0.00234839	float	float
[4]	-0.007355282	float	float
[5]	-0.008195325	float	float
[6]	-0.006181908	float	float
[7]	-0.00371265	float	float
[8]	-0.002058587	float	float
[9]	-0.0001358191	float	float
[10]	0.003311396	float	float

Watch Locals | Watch 1

Name	Value	Type	Radi
r1_fixed16	0x00005AA4	short[11]	hex
[0]	667	short	dec
[1]	507	short	dec
[2]	199	short	dec
[3]	-77	short	dec
[4]	-241	short	dec
[5]	-269	short	dec
[6]	-203	short	dec
[7]	-123	short	dec
[8]	-69	short	dec
[9]	-6	short	dec
[10]	107	short	dec

Watch Locals | Watch 1

Fig. 8.11. Coeficientes de autocorrelación, punto flotante (arriba) y punto fijo (abajo)

Para los valores en punto fijo mostrados en la figura Fig. 8.11, la precisión resultante es Q1.15., y esto puede ser fácilmente verificable.

Por ejemplo para el primer coeficiente r1[0] = 0.02040147 en punto flotante, su respectivo valor en punto fijo con precisión Q1.15 es 667,

$$r1\_fixed16[0] = 667 / (2^{15}) = 0.020355224609375$$

8.8 Algoritmo Sift: Algoritmo de Levinson para el cálculo de los coeficientes RC

El siguiente bloque es el cálculo de los 4 coeficientes RC que conformarán el filtro inverso en el bloque posterior. Para ello se recurre al algoritmo de Levinson, el cual se describe en la sección 3.8 del capítulo 3. Las ecuaciones que gobiernan el algoritmo de Levinson se dan en (3.31), (3.32), (3.33) y (3.34). Las ecuaciones para la conversión de los coeficientes LPC a coeficientes RC se dan en (3.37) y (3.38).

El algoritmo implementado en punto flotante se rige directamente por las ecuaciones (3.31) a (3.34).

Para el caso de punto flotante, la función usada para implementar el algoritmo de Levinson es:

```
lpc_durbin(r1, PITCHORDER, ki, &G);
```

Para el caso de la versión en punto fijo, el algoritmo es mucho más complejo debido al cuidado que debe tenerse con la precisión.

```
durbinfixed16(r1 fixed16, PITCHORDER, preci, ki fixed16, &G fixed16);
```

La figura Fig. 8.12 muestra el diagrama de flujo para el algoritmo de la versión en punto fijo. El código completo puede encontrarse CD en la ruta "..*DSP\lpc_fixed\main.c*". Como se observa, el algoritmo necesita como datos de ingreso además de los coeficientes de autocorrelación r1_fixed16[i], la precisión con la que entran dichos coeficientes. Dicha precisión es calculada en el bloque anterior, tal como se vio. Adicionalmente, el algoritmo hace un llamado a una función externa para calcular la raíz cuadrada de números de punto fijo de 32 bits. Dicha función es la siguiente:

```
fixed32 fixsqrt32_2(fixed32 x, int precision)
```

La función para el calculo de la raíz cuadrada necesita como entrada la precisión del valor al que se le calculará la raíz cuadrada. Dicha precisión es obtenida dentro de la misma rutina del algoritmo de Levinson.

El código completo para el algoritmo de la raíz cuadrada de 32 bits en punto fijo se puede encontrar en el CD en la ruta "..*DSP\lpc_fixed\main.c*".

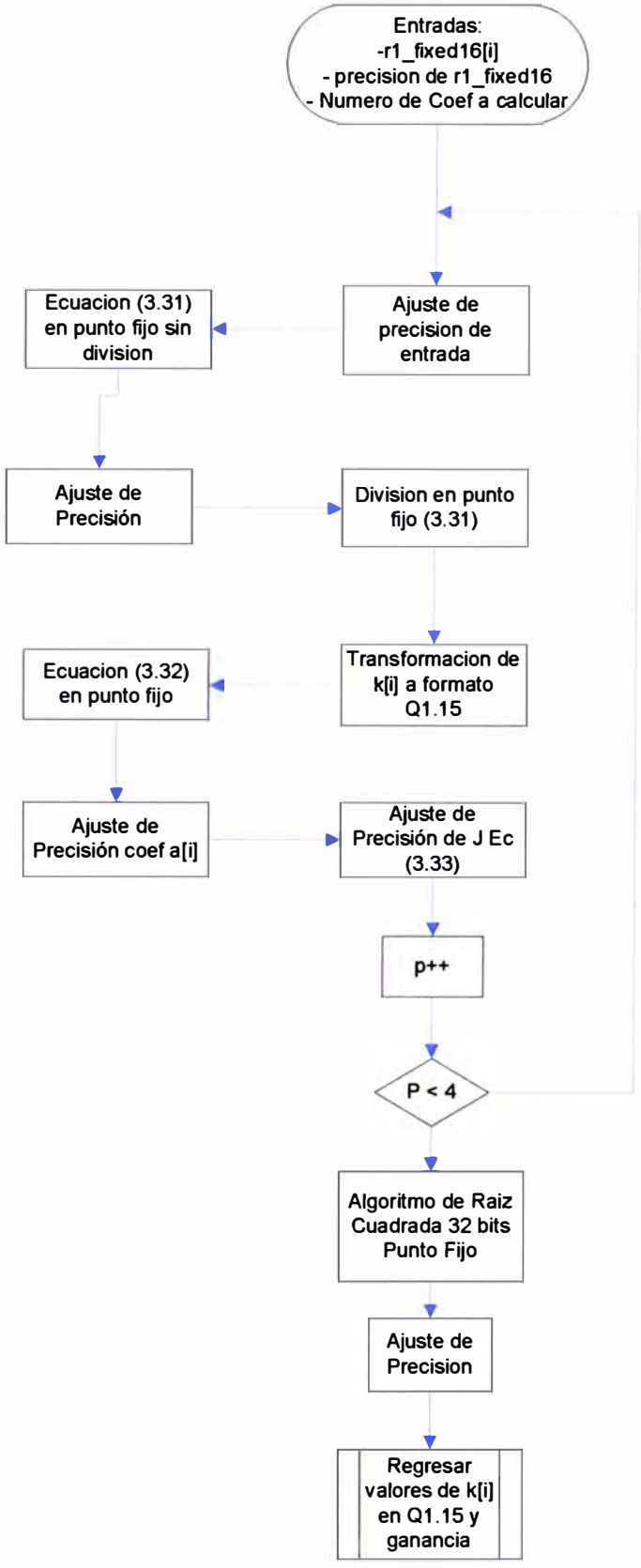
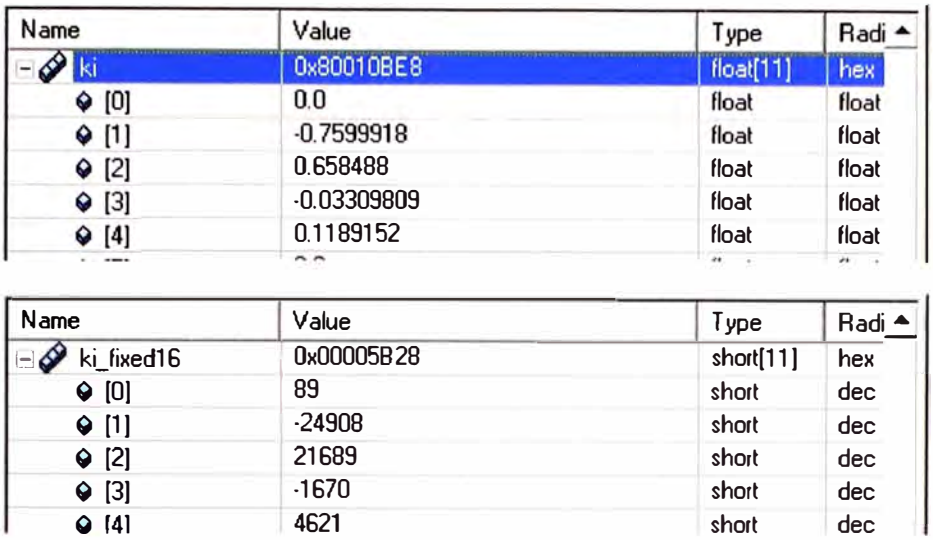


Fig. 8.12. Diagrama del Algoritmo de Levinson para el cálculo de los coeficientes RC.

La figura Fig. 8.13 muestra los valores de los 4 coeficientes de reflexión obtenidos por el algoritmo de Levinson para ambas versiones.



Name	Value	Type	Radi
ki	0x80010BE8	float[11]	hex
[0]	0.0	float	float
[1]	-0.7599918	float	float
[2]	0.658488	float	float
[3]	-0.03309809	float	float
[4]	0.1189152	float	float

Name	Value	Type	Radi
ki_fixed16	0x00005B28	short[11]	hex
[0]	89	short	dec
[1]	-24908	short	dec
[2]	21689	short	dec
[3]	-1670	short	dec
[4]	4621	short	dec

Fig. 8.13. Coeficientes de reflexión RC (ki[]) p. flotante (arriba), p. fijo (abajo)

La equivalencia de los valores obtenidos en la figura Fig. 8.13 puede ser comprobada fácilmente. Se sabe que los coeficientes obtenidos del algoritmo en punto fijo están con precisión Q1.15., por lo tanto, la conversión a valores reales se muestra a continuación:

TABLA Nº 8.1 Valores de los coeficientes RC obtenidos

Coeficiente	Valor Punto Flotante	Valor Punto Fijo
K[1]	-0.7599918	$-24908/(2^{15}) = 0.7601318359$
K[2]	0.658488	$21689/(2^{15}) = 0.66189575195$
K[3]	-0.03309809	$-1670/(2^{15}) = 0.0509643554$
K[4]	0.1189152	$4621/(2^{15}) = 0.141021728515$

Como se puede observar en la TABLA Nº 8.1 los valores obtenidos por el algoritmo en punto fijo se aproxima a los valores reales (punto flotante). Si bien es cierto hay un determinado error presente, dicho error no afecta de forma considerable la voz sintética decodificada tal como se verá mas adelante.

8.9 Algoritmo Sift: Salida del filtro inverso

Una vez calculados los 4 coeficientes RC, el siguiente bloque realiza el filtrado de la señal diezmada a través del filtro todo ceros conformado por los 4 coeficientes RC. La estructura del filtro es del tipo Lattice y se muestra en la figura Fig. 8.14

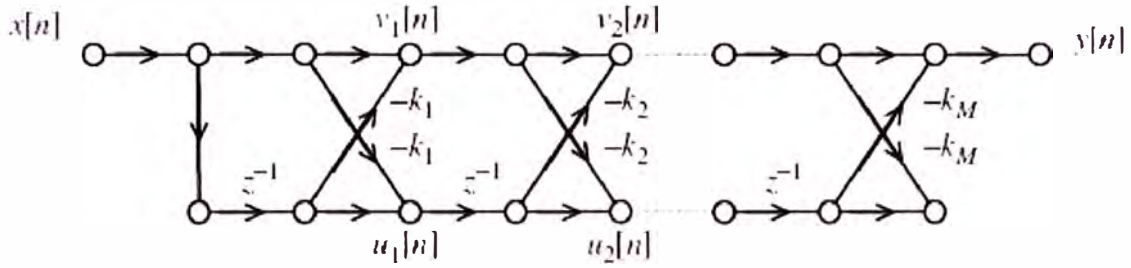


Fig. 8.14 Estructura Lattice para un filtro todo ceros (4)

La rutina que implementa el algoritmo para el filtro Lattice en punto flotante se muestra a continuación:

```
void lpc_inverse_filter(float *w, float *k)
{
    int i, j;
    float b[PITCHORDER + 1], bp[PITCHORDER + 1], f[PITCHORDER + 1];

    for (i = 0; i <= PITCHORDER; i++)
        b[i] = f[i] = bp[i] = 0.0;

    for (i = 0; i < BUFLEN / DOWN; i++) {
        f[0] = b[0] = w[i];
        for (j = 1; j <= PITCHORDER; j++) {
            f[j] = f[j - 1] + k[j] * bp[j - 1];
            b[j] = k[j] * f[j - 1] + bp[j - 1];
            bp[j - 1] = b[j - 1];

        }

            w[i] = f[PITCHORDER];
    }
    }
```

La versión en punto fijo se puede encontrar en el CD en la ruta "*..\DSP\lpc_fixed\main.c*", ya que es más extensa que su contraparte en punto flotante. Las salidas del filtro inverso se muestran en la figura Fig. 8.15.

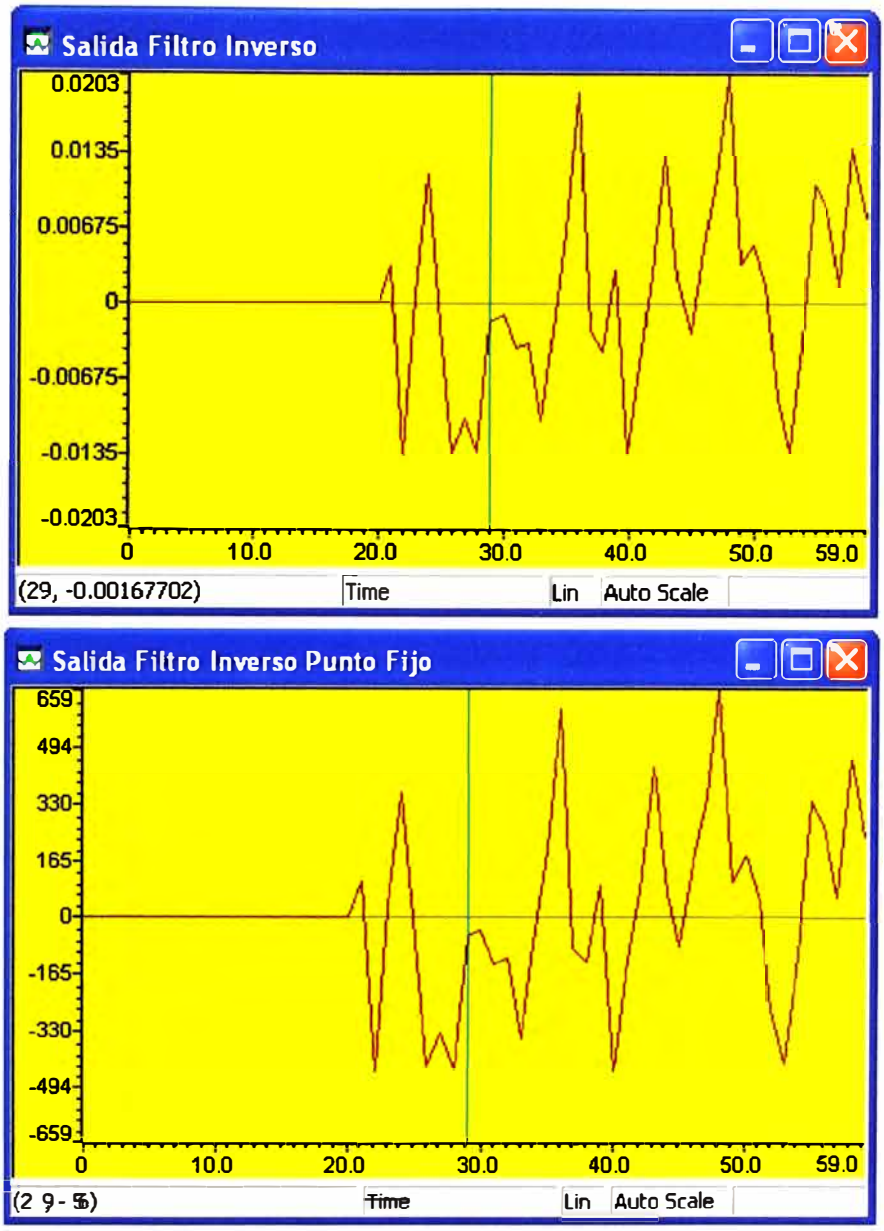


Fig. 8.15 Salida del filtro inverso para ambas versiones (entradas en Fig 8.8 y 8.9)

8.10 Algoritmo Sift: Autocorrelación de la salida del filtro inverso

Esta parte del algoritmo Sift calcula 40 valores de la autocorrelación de la señal de salida del filtro inverso. Las funciones usadas son las mismas que se usan en la sección 8.7 tanto para punto fijo como para punto flotante. Los resultados se pueden visualizar en las figuras Fig. 8.16 y 8.17.

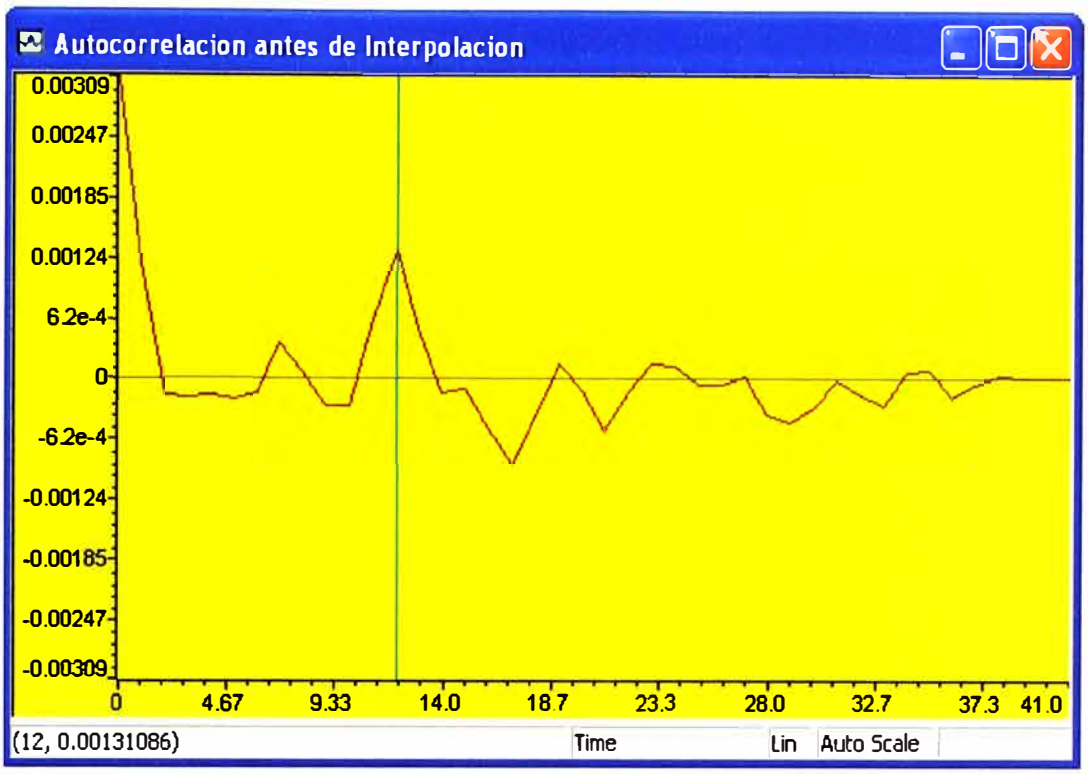


Fig. 8.16 Valores de la Autocorrelación de la señal diezmada, punto flotante

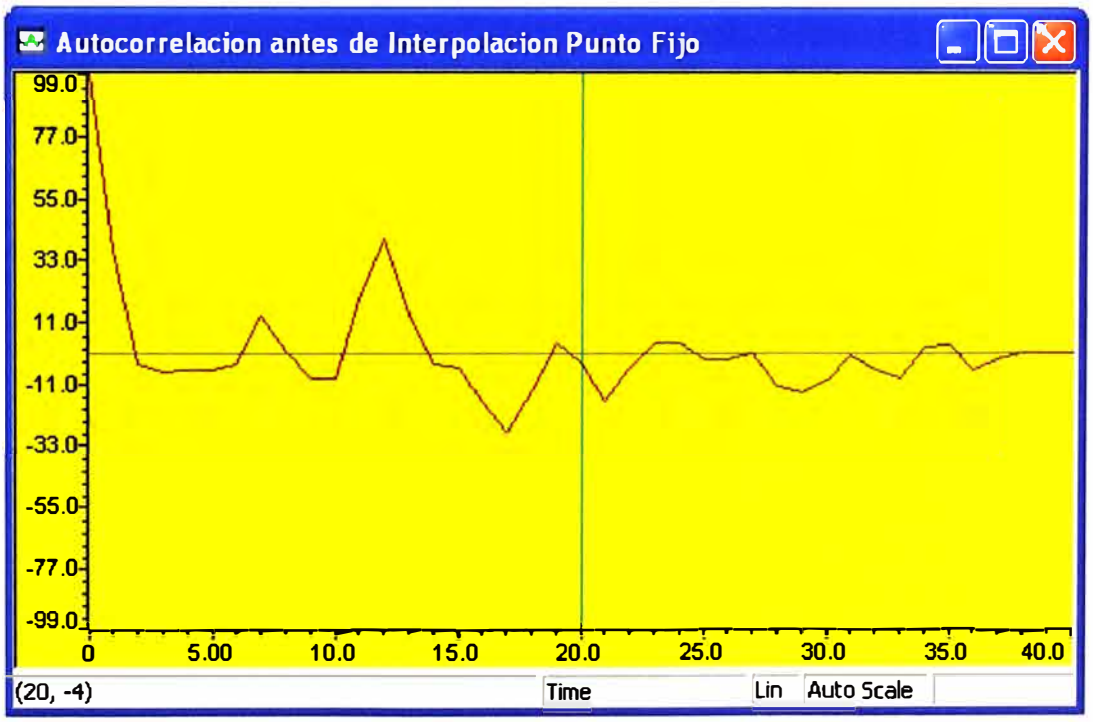


Fig. 8.17 Valores de la Autocorrelación de la señal diezmada, punto fijo

8.11 Algoritmo Sift: Determinación del valor pico e interpolación

Como se ha visto, el algoritmo Sift realiza una interpolación de 6 valores, 3 a la izquierda del valor pico y 3 a la derecha del valor pico, tal como se muestra en la figura Fig. 4.4 y que se repite en la figura Fig. 8.18

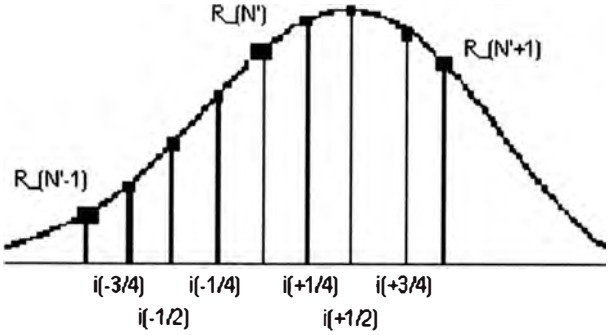


Fig. 8.18 Interpolación de valores alrededor del valor pico.

Esta etapa también se encarga de etiquetar un segmento como "sonoro" o "sordo", el algoritmo SIFT trabaja con el valor pico interpolado pero normalizado con respecto a r_0, es decir *r2(max) / r2[0]* = *rval*, la descripción del criterio es el siguiente:

- Si *rval* > 0.4 el segmento j es clasificado como "sonoro". Pero si el segmento "j-1" fue "sordo" y el segmento "j-2" fue "sonoro" entonces el segmento "j-1" se reetiqueta como "sonoro".

- Si *rval* < 0.4, y los segmentos "j-1" y "j-2" no están etiquetados como "sonoros" entonces el segmento j se clasifica como "sordo". Caso contrario se verifica si *rval* >0.3, si esto es cierto entonces el segmento j es etiquetado como "sonoro".

El diagrama de flujo de la determinación del periodo pitch se muestra en la figura Fig. 8.19. Hay que recordar que un valor de periodo pitch igual a cero significa que dicho segmento es un segmento "sordo". Los correspondientes códigos que implementan esta parte del algoritmo se pueden encontrar en el CD en la ruta "..*\DSP\lpc_fixed\main.c*" y "..*\DSP\lpc_floating\main.c*".

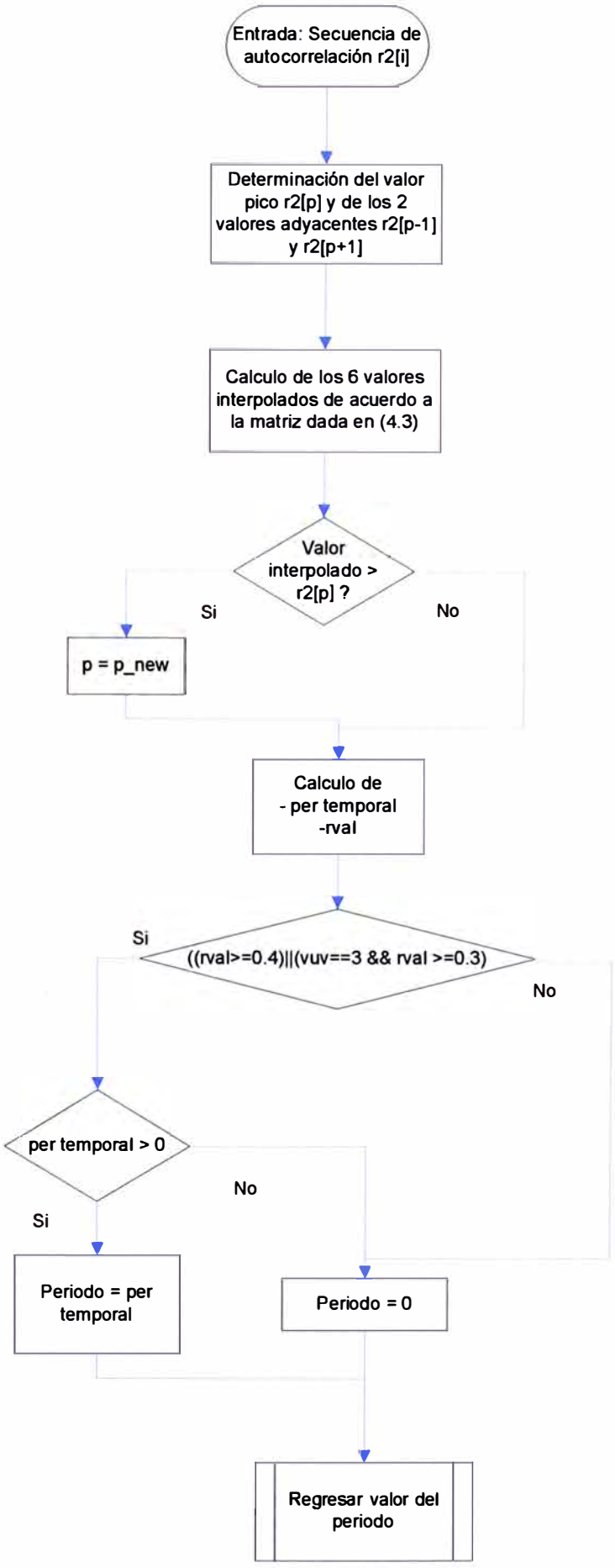


Fig. 8.19 Diagrama de flujo para la determinación del periodo pitch

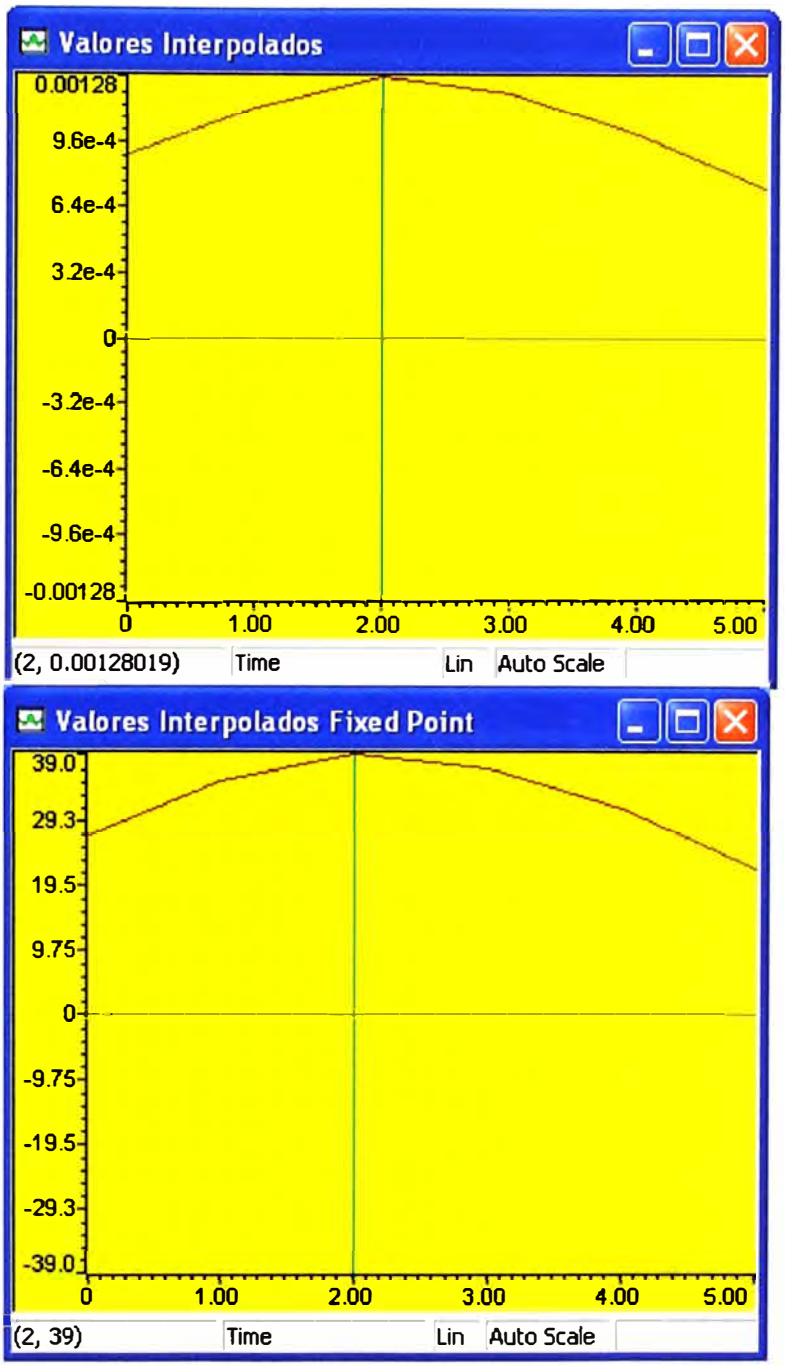


Fig. 8.20 Los 6 valores interpolados p. flotante (arriba), p. fijo (abajo)

Los valores interpolados para las dos versiones se muestran en la figura Fig. 8.20. De las figuras Fig. 8.16 y 8.17 se puede observar que el valor pico de la secuencia de autocorrelación ocurre para p=12. Como los valores interpolados no son mayores que el valor pico, se mantiene p =12 y como *rval* se encuentra dentro de los límites definidos por el algoritmo, el valor final del periodo es de 12x4=48. El valor de *rval* y el valor final del periodo se muestran en la figura Fig. 8.21 para ambas versiones.

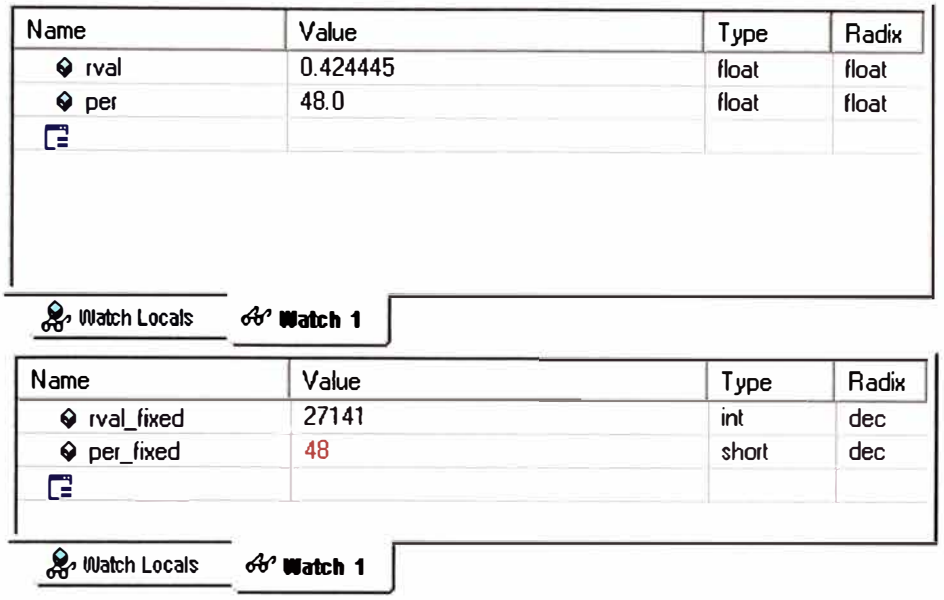


Fig. 8.21 Valores de *rval* y el periodo final para ambas versiones.

Como se observa en la figura Fig. 8.21, se puede realizar la equivalencia entre los valores en punto fijo y en punto flotante.

TABLA Nº 8.2. Valores de *rval* y *per* para ambas versiones

Variable	Punto Flotante	Punto Fijo
rval	0.424445	$27141/(2^{16})= 0.41413879394$
per	48	48

Como se observa en la TABLA Nº 8.2, el valor de rval para el caso de punto fijo es muy cercano al valor real (punto flotante).

La cuantización del valor del periodo a enviar se realiza con 8 bits. Es decir el valor del periodo a empaquetar debe estar dentro del rango [0 255]. En el caso de los valores de la tabla 8.2 el valor del periodo es de 48, lo cual traducido a segundos es 48*(1/8000) y traducido a Hz es 8000/48 = 166.66 Hz.

8.12 Coeficientes RC y Ganancia: Enventanado Hamming

De acuerdo a los diagramas de las figuras Fig. 6.1 y Fig. 8.3 el siguiente bloque debe realizar un enventanado al segmento de análisis de entrada (240 muestras).

Para el caso de punto flotante, el algoritmo que ejecuta el enventanado es sencillo y se muestra a continuación.

```
/****Enventanado floating point****/

      for(i=0;i<BUFLEN;i++)
      {w_w[i]=w_h[i]*w_s[i];
      }
```

Para el caso de punto fijo tenemos las siguientes líneas de código.

```
/***Enventanado Fixed point***/

for(i=0;i<BUFLEN;i++)
{
w_w_fixed16[i]=(fixed16)( (fixmul32(w_h_fixed16[i],in[i])) >>15 );
/*resultado en Q2.14*/
```

Las salidas de la señal enventanada se muestran en las figuras Fig. 8.22 y 8.23.

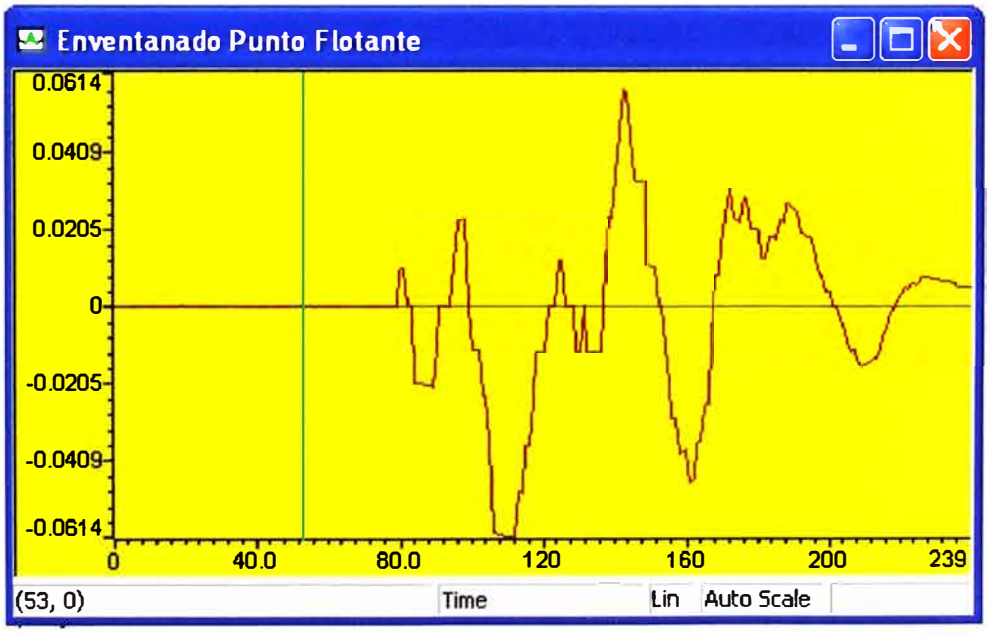


Fig. 8.22 Señal enventanada punto flotante.

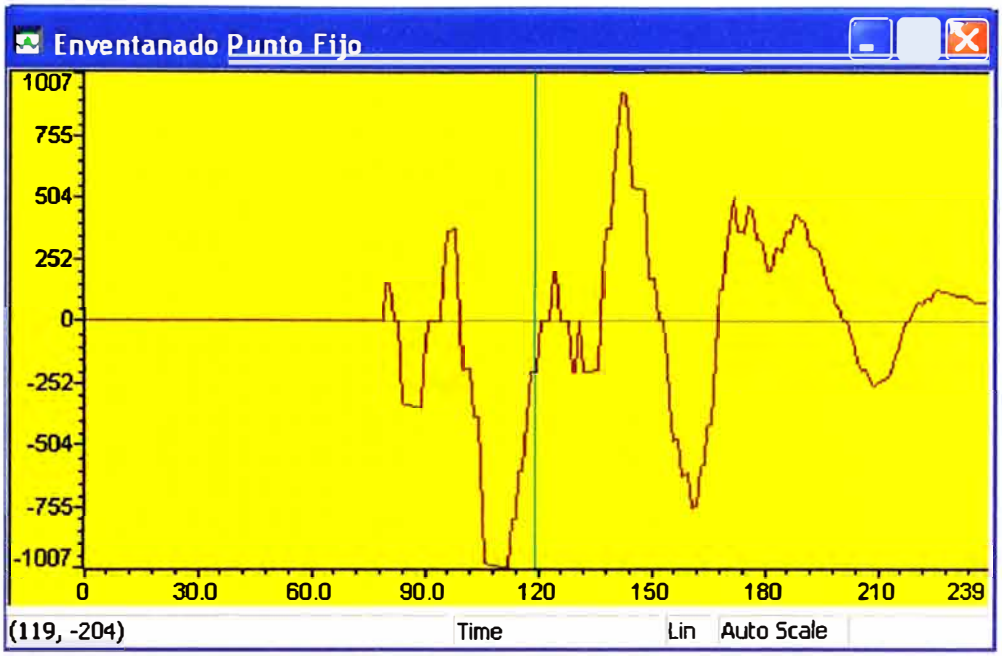


Fig. 8.23 Señal enventanada punto fijo

8.13 Coeficientes RC y Ganancia: Autocorrelación de Señal Enventanada

Este segmento del algoritmo realiza el cálculo de los 11 primeros valores de la autocorrelación de la señal enventanada, con el fin de usarlos después en el cálculo de los 10 coeficientes RC y la ganancia.

Para el caso del algoritmo en punto flotante, se tiene el siguiente código:

```
j=0;

for (k=0; k < LPC_FILTORDER+1 ; k++) {
  r3[k]=0;
  for (i=0; i < BUFLEN-j; i++) {

      r3[k]=r3[k]+w_w[i]*w_w[i+k];

  }
  j++;

}
```

Para el caso de la versión en punto fijo, la función usada es la misma que se usa en la sección 8.7, el código se pueden encontrar en el CD en la ruta "*..\DSP\lpc_fixed\main.c*". Los resultados se pueden visualizar en las figura Fig. 8.24.

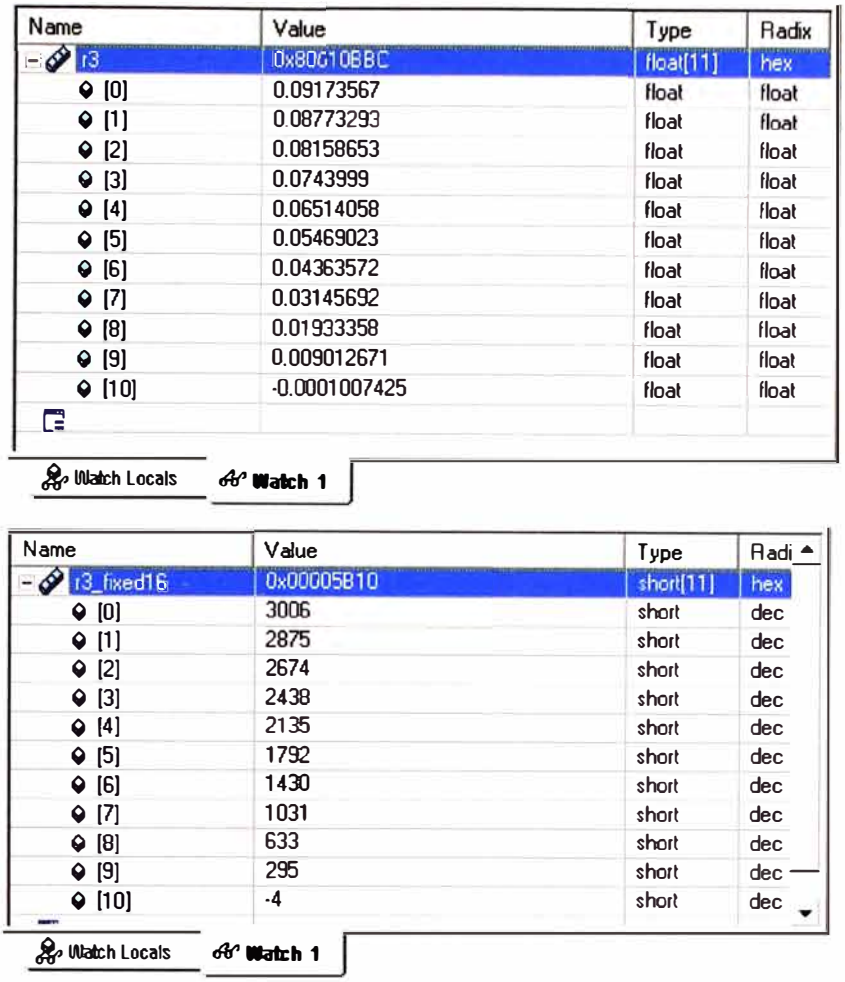


Fig. 8.24 Valores de la autocorrelación de la señal enventanada, p. flotante (arriba), p. fijo (abajo).

TABLA Nº 8.3. Valores de la autocorrelación de la señal enventanada (flotante y fijo)

Variable	Punto Flotante	Punto Fijo
r3[0]	0.09173567	$3006/(2^{15}) = 0.09173583984$
r3[1]	0.08773293	$2875/(2^{15}) = 0.087738037109$
r3[2]	0.08158653	$2674/(2^{15}) = 0.081604003906$
r3[3]	0.0743999	$2438/(2^{15}) = 0.074401855468$
r3[4]	0.06514058	$2135/(2^{15}) = 0.065155029296$
r3[5]	0.05469023	$1792/(2^{15}) = 0.0546875$
r3[6]	0.04363572	$1430/(2^{15}) = 0.043640136718$
r3[7]	0.03145692	$1031/(2^{15}) = 0.031463623046$
r3[8]	0.01933358	$633/(2^{15}) = 0.0193176269531$
r3[9]	0.009012671	$295/(2^{15}) = 0.0090026855468$
r3[10]	-0.0001007425	$-4/(2^{15}) = 0.0001220703125$

8.14 Coeficientes RC y Ganancia: Algoritmo de Levinson

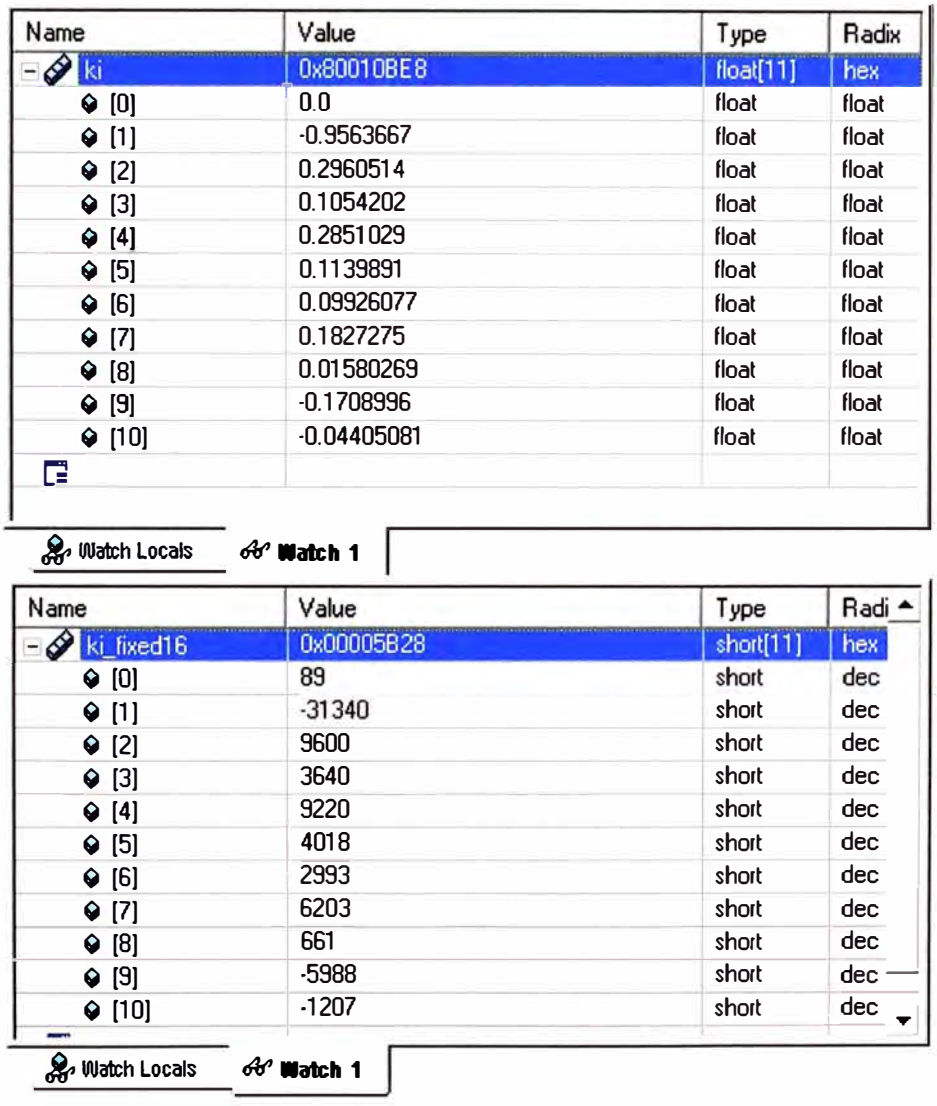
Este segmento del algoritmo se rige por las mismas rutinas (punto fijo y flotante) que se vieron en la sección 8.8. La única diferencia es que en este caso se calculan 10 coeficientes RC y no 4 como fue en el caso del algoritmo SIFT.

Para punto flotante:

```
lpc_durbin(r3, LPC_FILTORDER, ki, &G_temp);
```

Para punto fijo:

```
durbinfixed16(r3_fixed16, LPC_FILTORDER, preci, ki_fixed16, &G_temp_fixed16);
```



Name	Value	Type	Radix
ki	0x80010BE8	float[11]	hex
[0]	0.0	float	float
[1]	-0.9563667	float	float
[2]	0.2960514	float	float
[3]	0.1054202	float	float
[4]	0.2851029	float	float
[5]	0.1139891	float	float
[6]	0.09926077	float	float
[7]	0.1827275	float	float
[8]	0.01580269	float	float
[9]	-0.1708996	float	float
[10]	-0.04405081	float	float

Watch Locals | Watch 1

Name	Value	Type	Radi
ki_fixed16	0x00005B28	short[11]	hex
[0]	89	short	dec
[1]	-31340	short	dec
[2]	9600	short	dec
[3]	3640	short	dec
[4]	9220	short	dec
[5]	4018	short	dec
[6]	2993	short	dec
[7]	6203	short	dec
[8]	661	short	dec
[9]	-5988	short	dec
[10]	-1207	short	dec

Watch Locals | Watch 1

Fig. 8.25 Coeficientes RC calculados, p. flotante (arriba), p. fijo (abajo)

Como en casos anteriores, la equivalencia entre los valores obtenidos por el algoritmo en punto fijo puede comprobarse fácilmente, esto se muestra en la TABLA Nº 8.4.

TABLA Nº 8.4. Valores de los 10 coeficientes RC (flotante y fijo)

Coeficiente	Punto Flotante	Punto Fijo
ki[0]	-0.9563667	$-31340/(2^{15}) = 0.956420898$
ki[1]	0.2960514	$9600/(2^{15}) = 0.29296875$
ki[2]	0.1054202	$3640/(2^{15}) = 0.111083984375$
ki[3]	0.2851029	$9220/(2^{15}) = 0.28137207031$
ki[4]	0.1139891	$4018/(2^{15}) = 0.122619628906$
ki[5]	0.09926077	$2993/(2^{15}) = 0.091339111328$
ki[6]	0.1827275	$6203/(2^{15}) = 0.189300537109$
ki[7]	0.01580269	$661/(2^{15}) = 0.0201721191406$
ki[8]	-0.1708996	$-5988/(2^{15}) = -0.18273925781$
ki[9]	-0.04405081	$-1207/(2^{15}) = -0.0368347167$

Los valores de la ganancia calculados por el algoritmo de Levinson se muestran en la figura Fig. 8.26. El formato de salida de la ganancia en punto fijo es Q2.14.

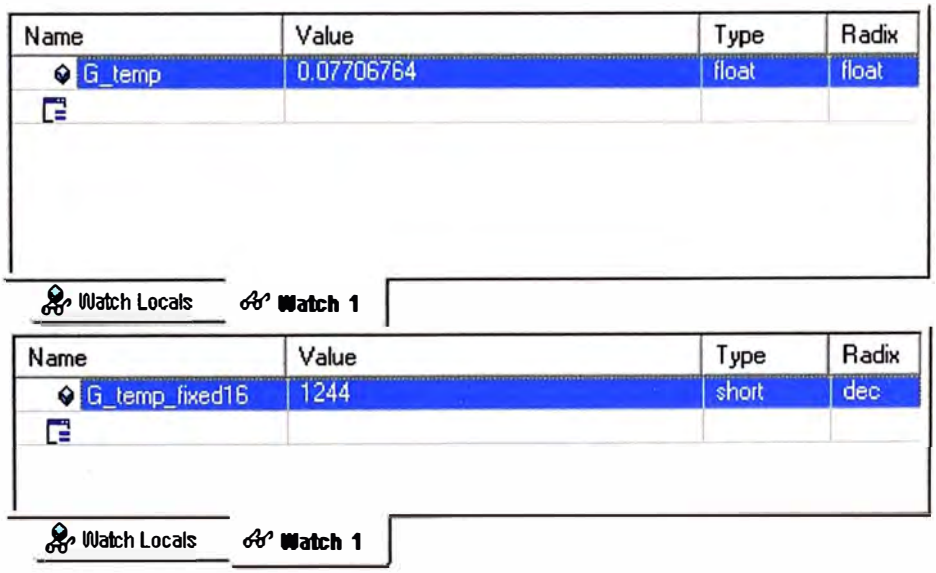


Fig. 8.26 Valores de la Ganancia (flotante y fijo)

La equivalencia en punto fijo es: $1244/(2^{14}) = 0.075927734375$ el cual es un valor muy cercano al valor real dado por la rutina en punto flotante (ver figura Fig. 8.26).

8.15 Empaquetamiento

TABLA Nº 8.5 Resumen de Distribución de los bits del codificador

Empaquetamiento de bits

	Coeficientes RC	Ganancia	Periodo Pitch
Actualización	Cada Frame de 20 ms (160 muestras)	Cada Frame de 20 ms (160 muestras)	Cada Frame de 20 ms (160 muestras)
Coeficientes Involucrados	10 Coeficientes ***RC***	1 valor de *la* ***Ganancia***	1 valor del **periodo** pitch
Tipo de Análisis	Lazo Abierto, ventana de Hamming, autocorrelacion, algoritmo de Levinson,	Lazo Abierto, ventana de Hamming, autocorrelacion, algoritmo de Levinson,	Algoritmo SIFT
Asignación de Bits	Cada coeficiente RC se cuantiza con 8 bits	La ganancia de cuantiza con 8 bits	El periodo se cuantiza con 8 bits
Total de Bits por Frame	80	8	8
Tasa de Bits	4000 bps	400 bps	400 bps

Como se aprecia en la TABLA Nº 8.5, la presente implementación usa 8 bits para la cuantización de todos los parámetros involucrados, lo que se traduce en una tasa de bits de 4800 bps.

TABLA Nº 8.6 Distribución de los bits en el stream-bit de salida del codificador.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
RC[1]								*RC[2]*								*RC[3]*							
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
RC[4]								*RC[5]*								*RC[6]*							
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72
RC[7]								*RC[8]*								*RC[9]*							
73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96
RC[10]								*Ganancia*								*Periodo Pitch*							

8.16 Decodificador

El esquema del decodificador se muestra en la figura Fig. 6.2 del capítulo 6. El esquema para el decodificador es empleado en ambas versiones (tanto punto flotante como punto fijo). El diagrama de flujo del algoritmo se muestra en la figura Fig. 8.28.

Como se observa en la figura Fig. 8.28, luego de recuperar los valores de todos los parámetros enviados por el codificador (periodo pitch, ganancia y los 10 coeficientes RC), el decodificador realiza un ajuste de la ganancia recibida. Dicho ajuste depende del valor del periodo recibido, y funciona del siguiente modo: cuando la trama a decodificar es "sorda" (per = 0) la ganancia recibida es reducida por un factor. Cuando la trama recibida es "sonora", la ganancia es modificada de acuerdo al valor del periodo recibido. Esto se ilustra en el diagrama de flujo de la figura Fig. 8.27.

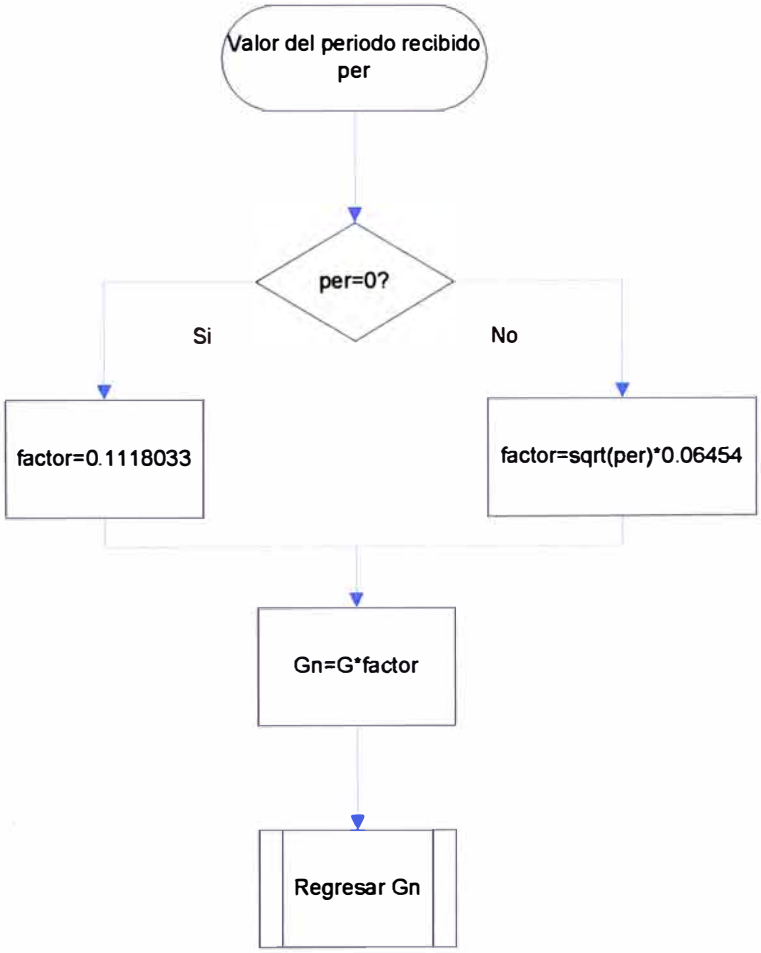


Fig. 8.27 Diagrama de flujo para el ajuste de la ganancia

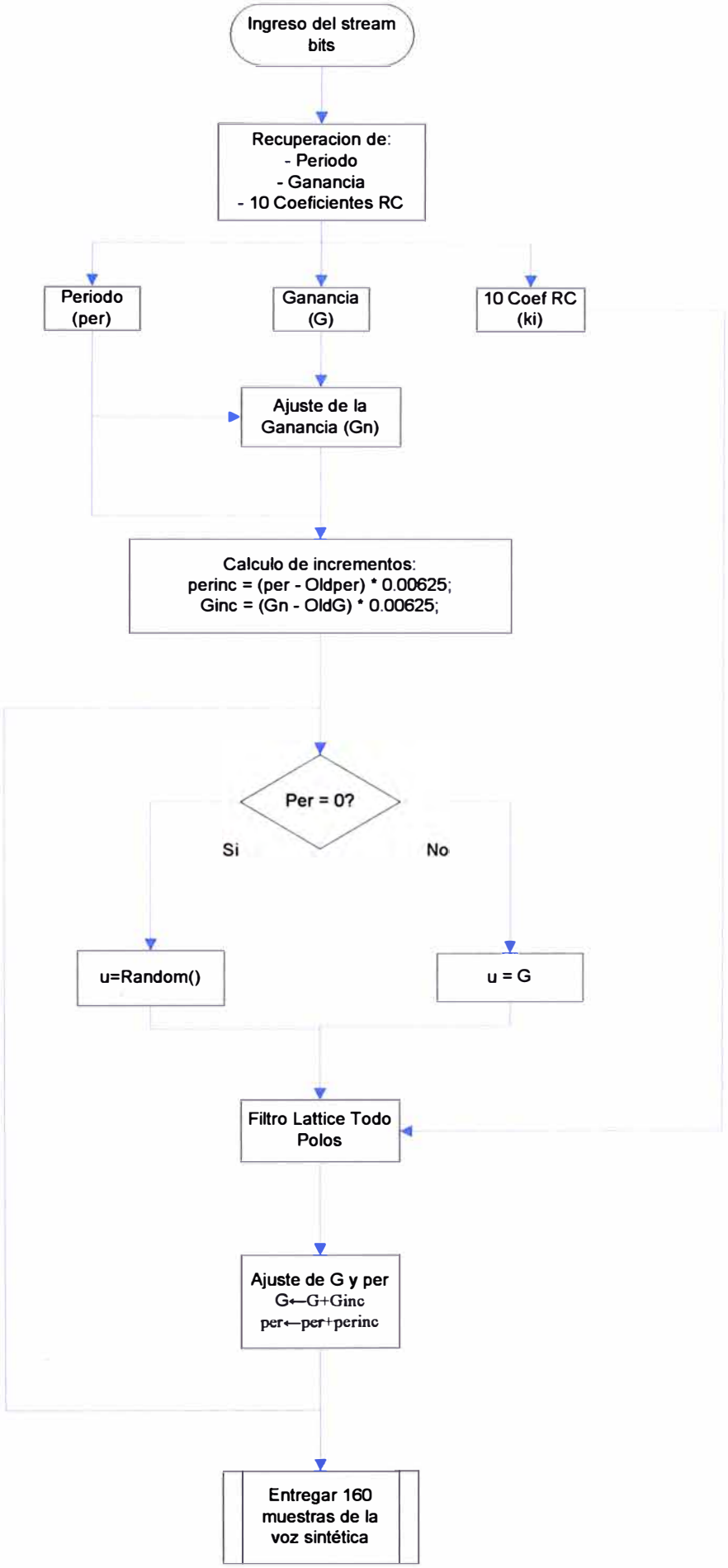


Fig. 8.28 Diagrama del algoritmo para el decodificador

Se observa también en la figura Fig. 8.28, que se calculan dos valores de incremento tanto para el periodo como para la ganancia, dichos valores imprimen pequeños incrementos y actúan en el calculo de cada muestra de las 160 que corresponden a la trama de la voz sintética decodificada. El resultado es un "smoothing" en la voz sintética decodificada ya que se emplean valores dinámicos tanto para el periodo pitch como para la ganancia. El código completo para el decodificador para ambas versiones se puede encontrar en el CD en las rutas "*..\DSP\pc_fixed\main.c*" y "*..\DSP\pc_floating\main.c*".

A continuación se muestran varias salidas de la voz sintética decodificada para distintas secuencias de entrada para los codificadores tanto en punto fijo como en punto flotante.

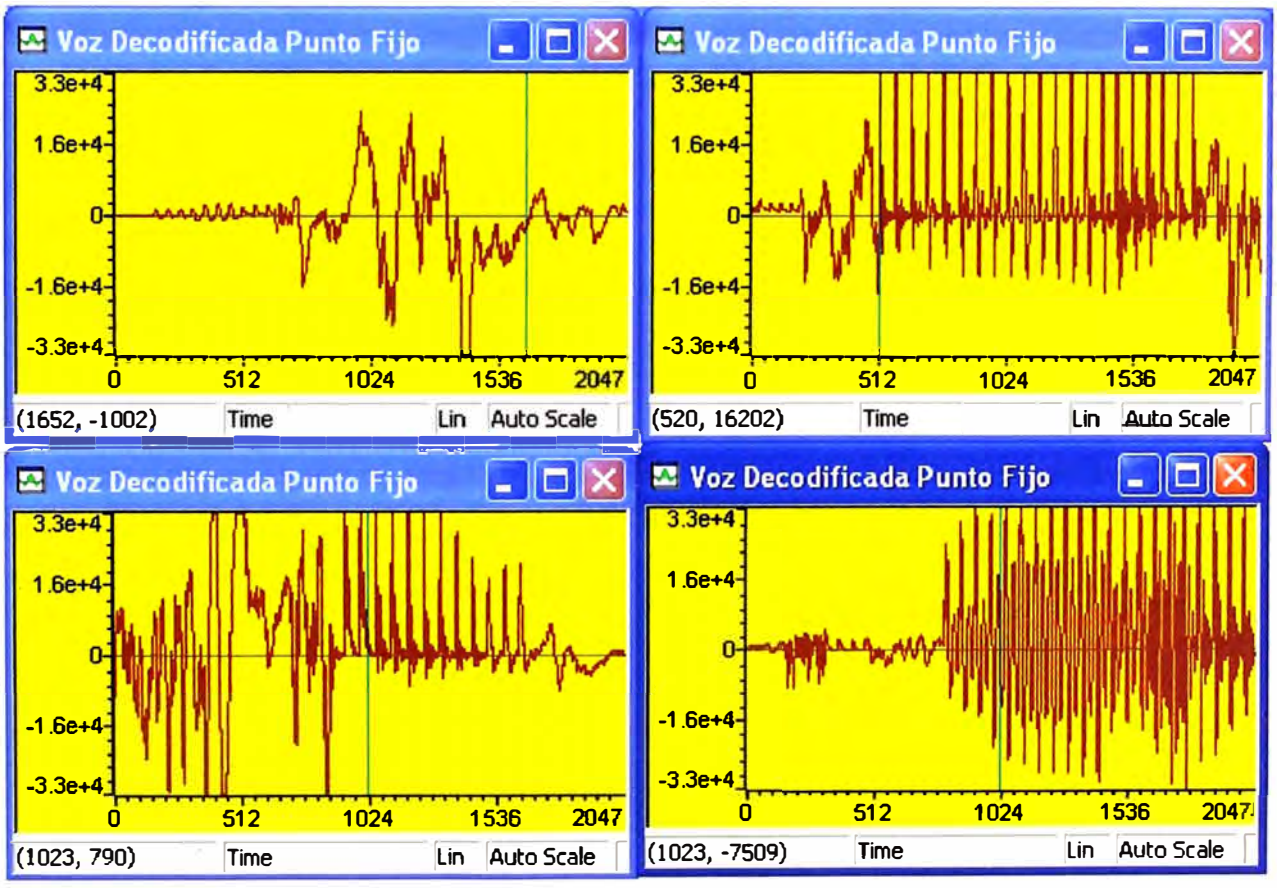


Fig. 8.29 Voz sintética (codificador en punto fijo) para la señal de entrada de la Fig. 8.2

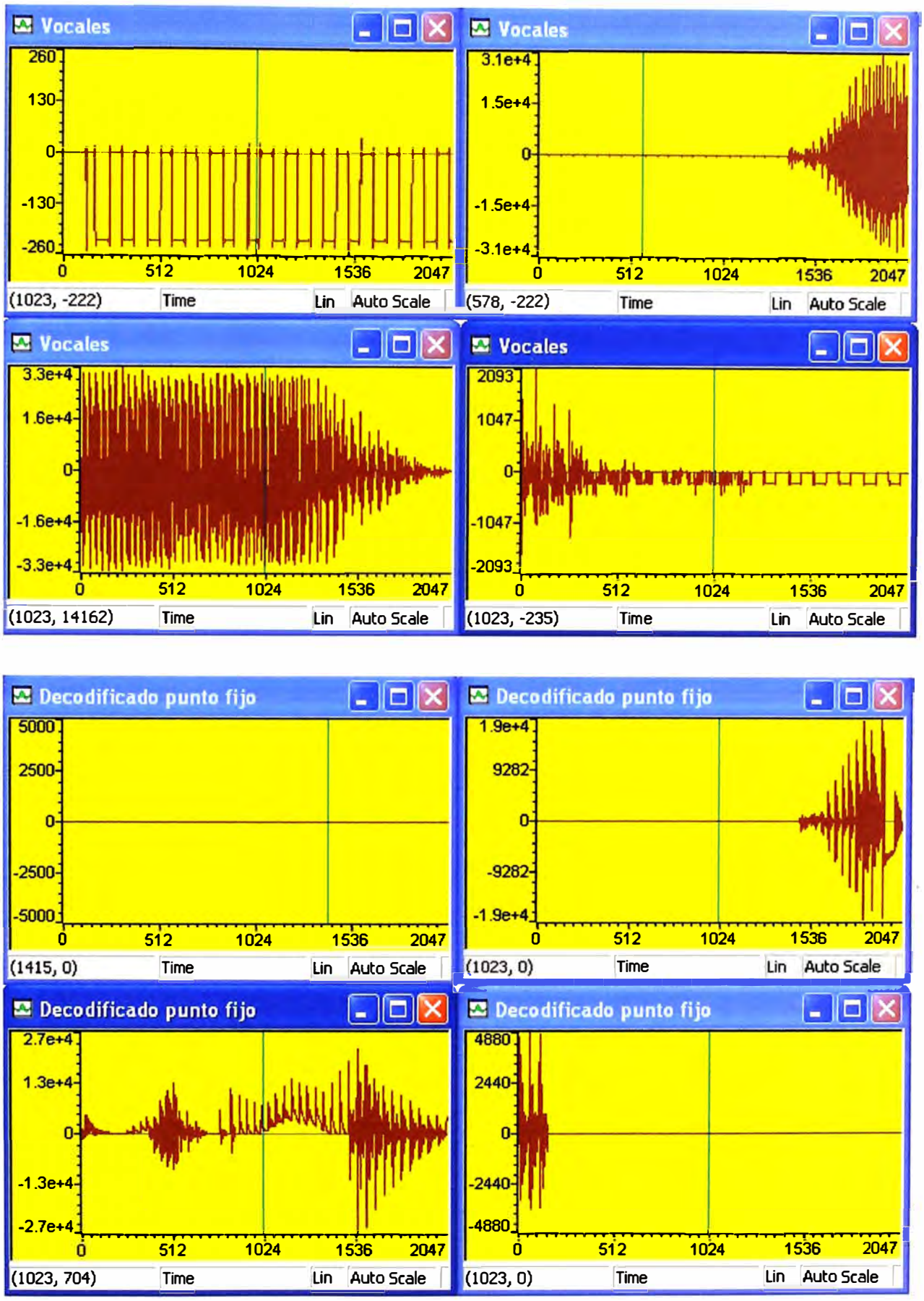


Fig. 8.30 Voz de entrada (arriba) y voz sintética para el codificador en punto fijo (abajo)

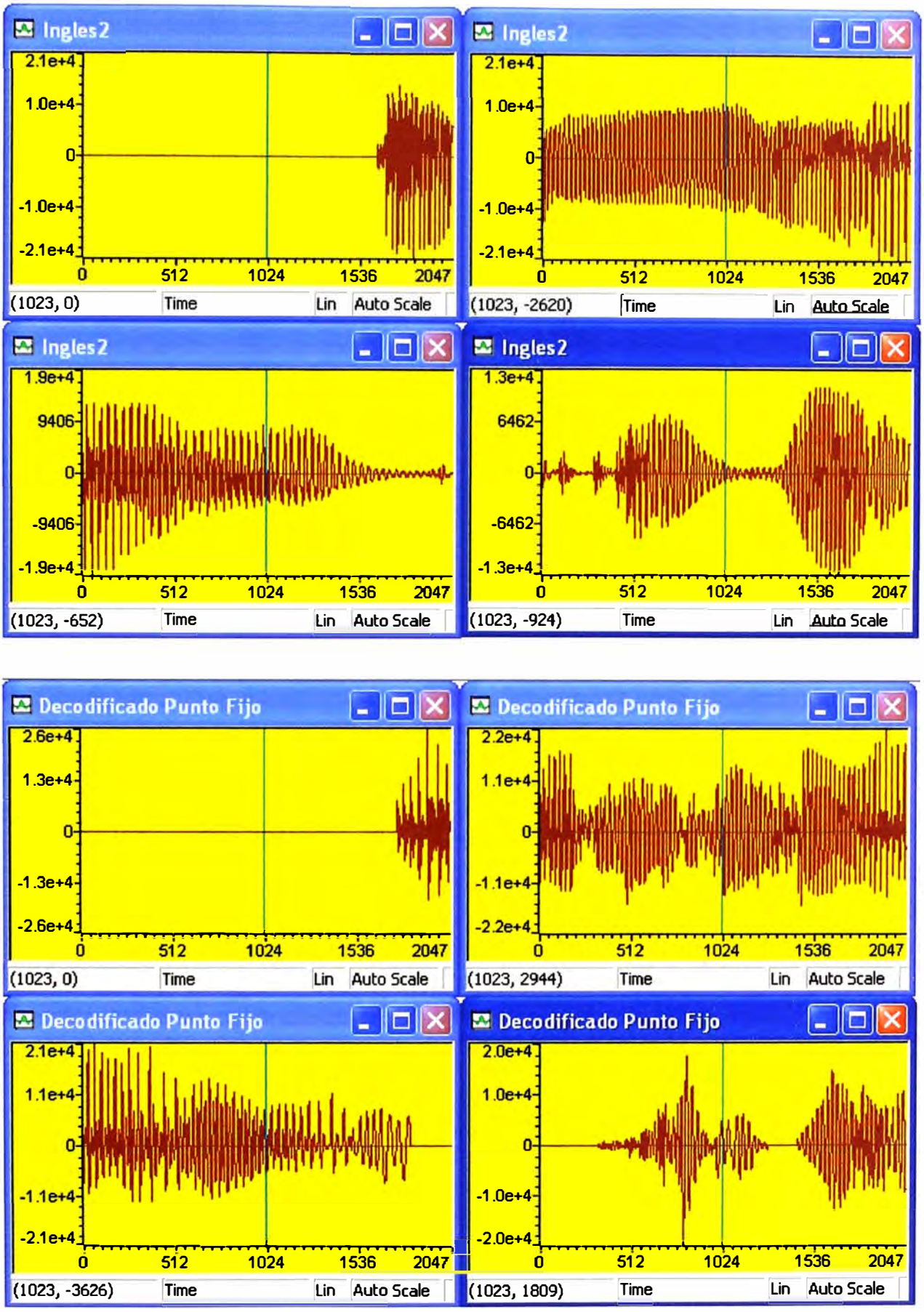


Fig. 8.31 Voz de entrada (arriba) y voz sintética para el codificador en punto fijo (abajo)

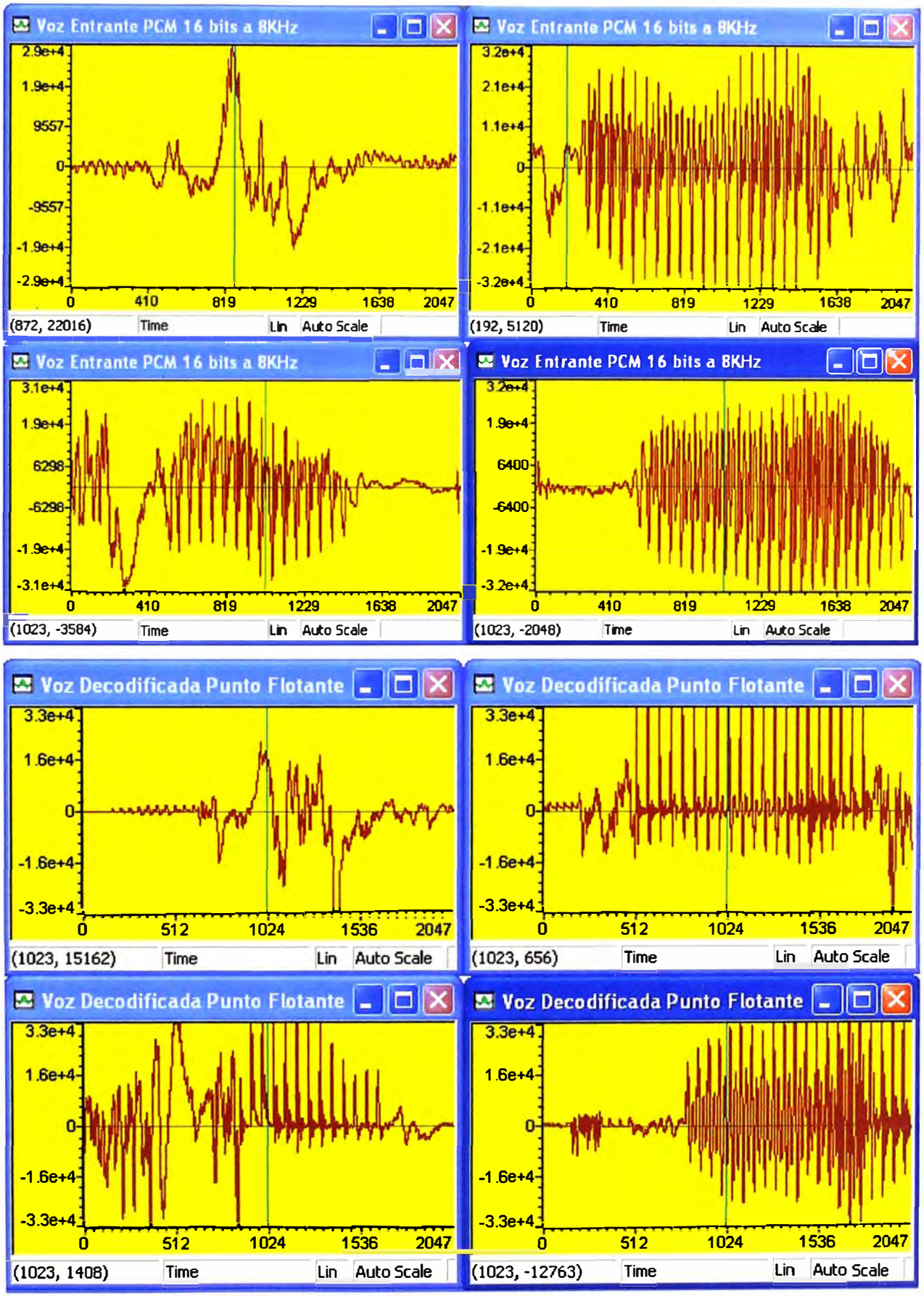


Fig. 8.32 Voz de entrada (arriba) y voz sintética para el codificador en punto flotante (abajo)

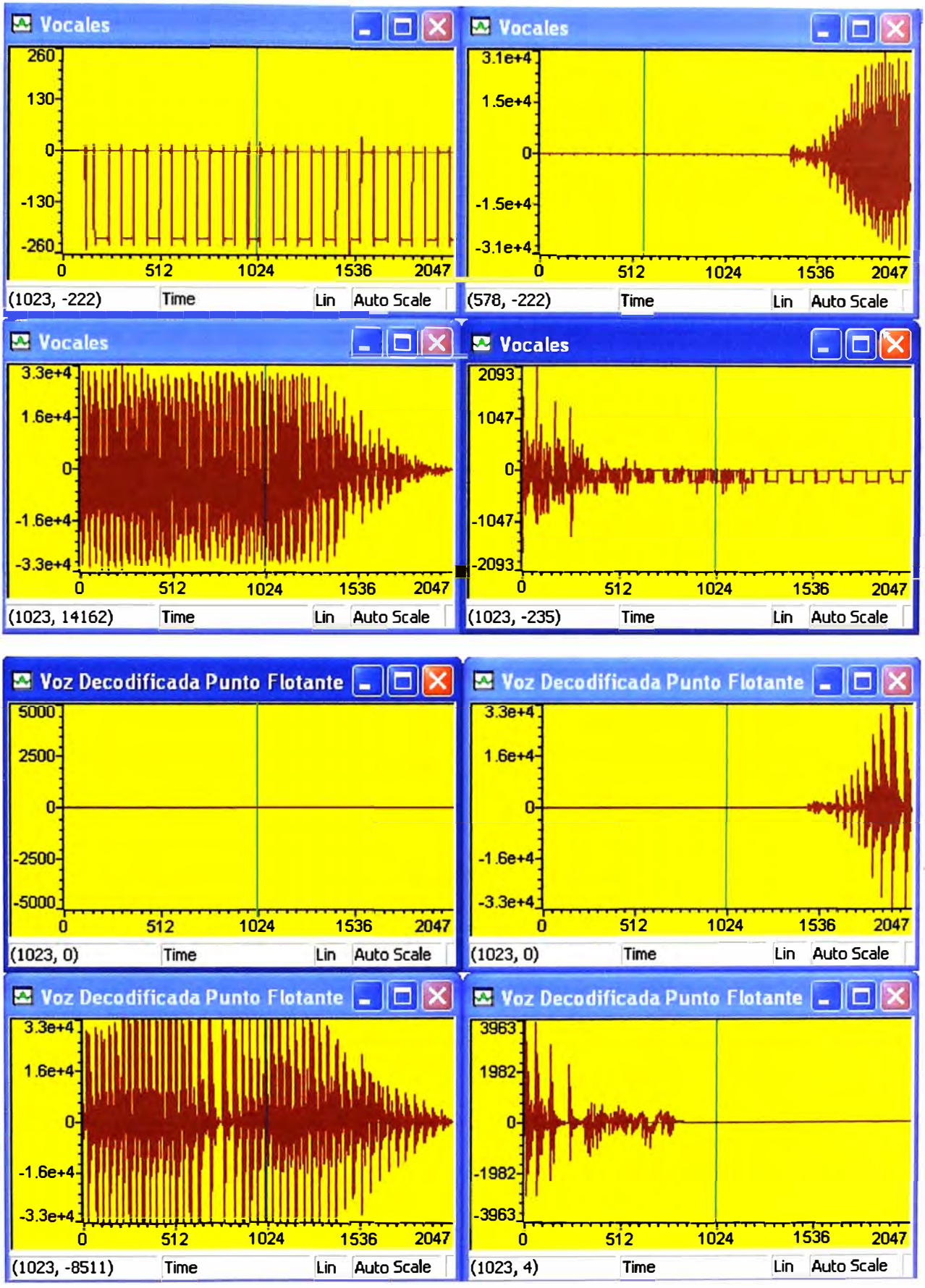


Fig. 8.33 Voz de entrada (arriba) y voz sintética para el codificador en punto flotante (abajo)

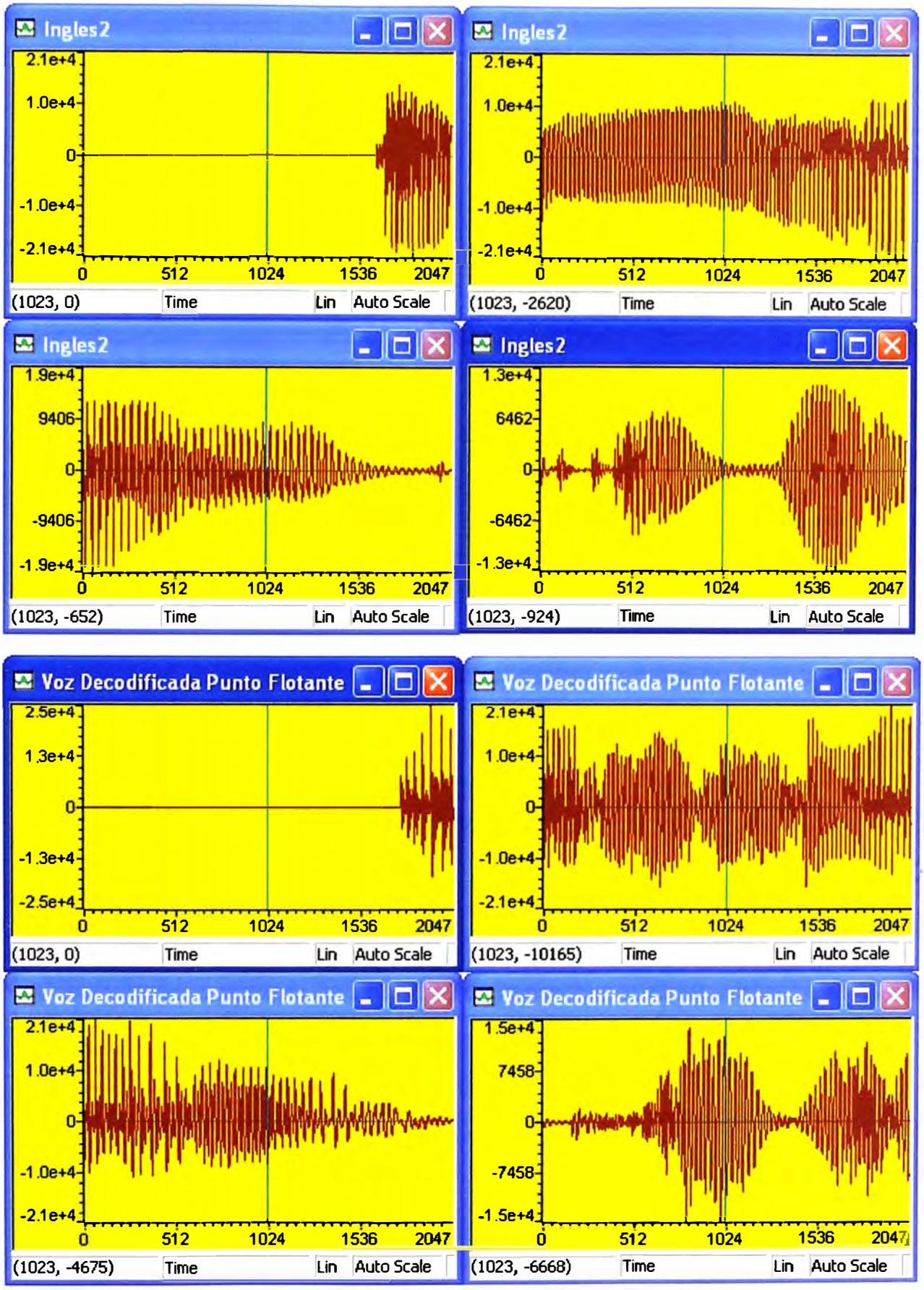


Fig. 8.33 Voz de entrada (arriba) y voz sintética para el codificador en punto flotante (abajo)

8.17 Tiempo de ejecución del algoritmo y Optimización del código para óptimo desempeño en el DSP

El compilador, "builder", "linker" y otras herramientas del Code Composer, nos ofrecen una serie de alternativas para depurar nuestras aplicaciones y optimizarlas de acuerdo a nuestras necesidades. En todo proceso de optimización es necesario hacer un balance entre el tamaño del código, velocidad de ejecución, nivel de paralelismo, anular unas funciones, mantener otras e ir ajustando. En este caso se ha tratado de optimizar la velocidad de ejecución del código. En el caso del código implementado, los pasos para la optimización del algoritmo fueron los siguientes:

8.17.1 Reacomodo de funciones críticas a la IRAM

El reacomodo de las funciones críticas a la IRAM tiene como objetivo hacer que los segmentos de código que deben ser ejecutados con mas rapidez se carguen a la memoria que es más rápida, en este caso es la IRAM (ver Fig. 7.7). Dicho reacomodo se logra con la directiva:

```
#pragma CODE_SECTION(funcion,seccion)
```

Luego del reacomodo de las funciones críticas a la IRAM, el tiempo de ejecución presenta una primera reducción.

8.17.2 Reubicación de variables críticas a la IRAM.

El siguiente ajuste es el direccionamiento de las variables que son usadas por las funciones críticas, a la memoria IRAM, por medio de la directiva

```
#pragma DATA_SECTION (variable, seccion);
```

Por ejemplo para el algoritmo en punto fijo se usaron las directivas que se muestran a continuación.

```
#pragma DATA_SECTION (in, "mydata");
#pragma DATA_SECTION (w_y_fixed16, "mydata");
#pragma DATA_SECTION (w_h_fixed16, "mydata");
#pragma DATA_SECTION (w_w_fixed16, "mydata");
#pragma DATA_SECTION (deci_fixed16, "mydata");
```

Luego de reubicar las variables críticas a la IRAM se obtiene el segundo incremento de velocidad para el tiempo de ejecución del algoritmo.

8.17.3 Opciones del compilador para optimización

Para las optimizaciones anteriores, no se emplearon las opciones de optimización del compilador. Dichas opciones se describen en la presente sección y al final se presentan los resultados que resultan luego de aplicar dichas opciones de optimización. Para mas información sobre optimización del código, referirse al documento spru 187: "TMS320C6000 Optimizing Compiler user's Guide". Las opciones de compilación se muestran en las figuras Fig. 8.34 y Fig. 8.35.

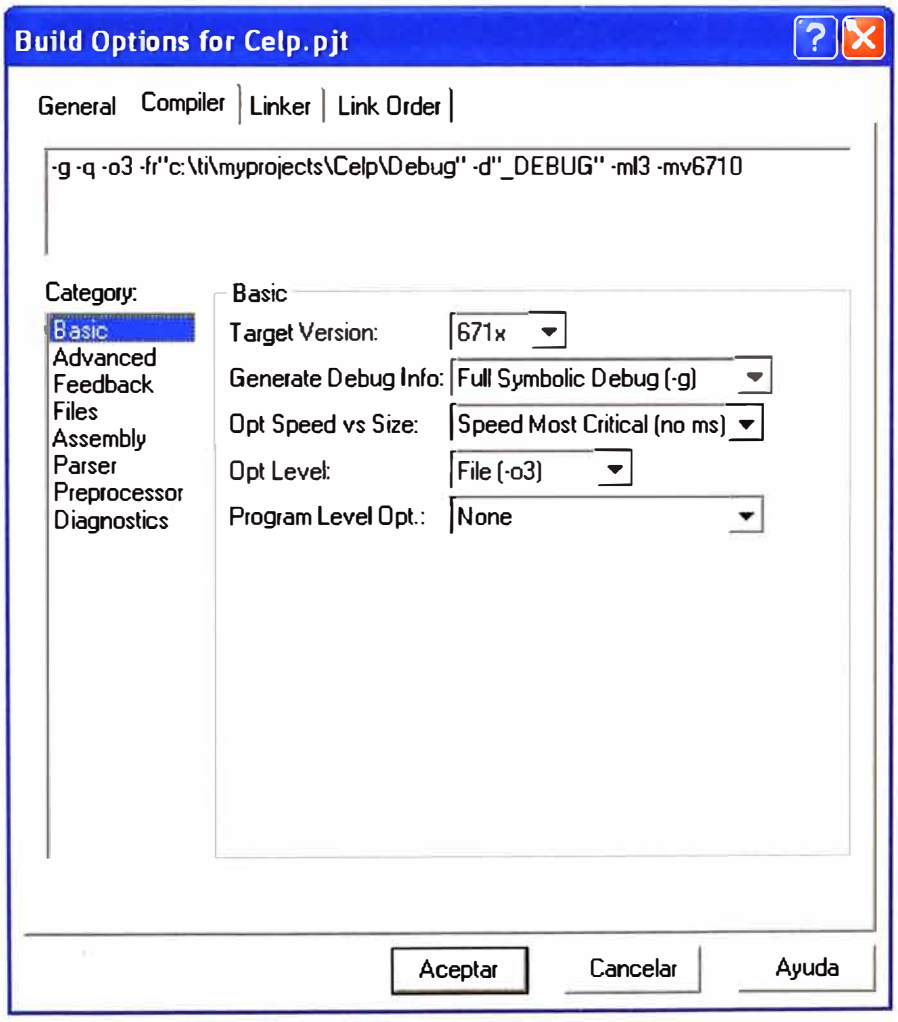


Fig. 8.34 Opciones de compilación. Básicas.

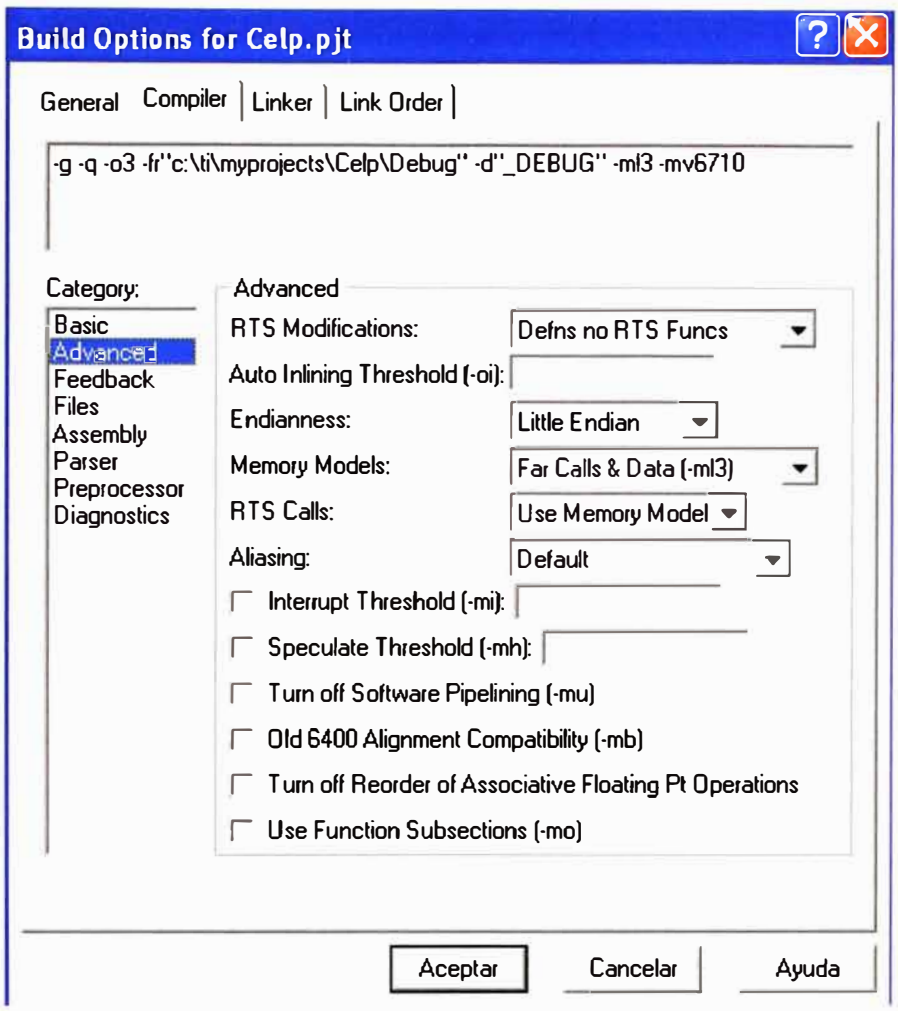


Fig. 8.35 Opciones de compilación. Avanzadas

Las opciones de compilación empleadas fueron las siguientes:

- ***Opt Speed vs Size.*** Se configuró a "*Speed Most Critical (no ms)*". Con esta configuración se orientó la prioridad hacia la velocidad de ejecución sacrificando el tamaño del código. De esta forma el compilador optimiza el código en velocidad.

- ***Opt Level.*** Se configuró a ***"-o3"***. En esta configuración el optimizador ejecuta lo siguiente:

Ejecuta simplificación control-flow-graph, Ubica variables en registros, Ejecuta rotación en los bucles, Elimina código no usado, Simplifica expresiones y expresiones, Ejecuta propagación local, Retira asignaciones no usadas, Elimina expresiones locales comunes, Ejecuta paralelismo en software, Ejecuta optimización de bucles, Elimina subexpresiones globales comunes, Ejecuta unrolling en los bucles, Retira todas las

funciones que nunca son llamadas, Reordena las declaraciones de funciones de tal forma que los atributos de las funciones a llamadas son conocidas cuando el llamante es optimizado, etc.

- ***Generate Debug Info.*** Configurado en "none". Normalmente cuando se desarrolla un software se necesita depurar el mismo. Para ello el Code Composer inserta múltiples etiquetas y comandos adicionales al código de la aplicación para lograr la depuración. Para evaluar la performance máxima del código se desactiva el debuger con la opción "none".

- ***Memory Models.*** Configurado en "Far Calls & Data". Cuando se va a implementar un programa grande y son muchas las variables, arreglos y matrices que se van a emplear, es de esperar que la memoria IRAM no alcanzará para ingresar dichos códigos y variables. Para ello se tiene que recurrir al modelo de memoria de llamadas lejanas a funciones y datos. De esta forma se puede aprovechar todo el mapa de memoria. La desventaja es que se añaden cabeceras a las instrucciones los cuales consumen más ciclos de reloj.

Luego de aplicar las opciones de optimización mostradas, se llegan a los valores resumidos en la figura Fig. 8.36, siendo el promedio de ejecución del código (codificador más decodificador) de ***5 ms***.

Como se puede apreciar, hay una mejora sustancial con relación al valor inicial de tiempo de ejecución (codificación más decodificación) el cual era de aproximadamente 400 ms. Con esto se ha demostrado que el proceso de optimización es un proceso de refinamiento de código. Es obvio que se puede seguir optimizando aún más para mejorar el tiempo de ejecución. Si se quiere hacer una reducción aún mayor en el tiempo de ejecución, lo siguiente es pasar a assembler las rutinas críticas.

	Algoritmo Punto Fijo	Algoritmo Punto Flotante
Tiempo de ejecución para codificar/decodificar 8160 muestras (1.02 segundos)	**158.2522 ms**	**168.1171 ms**

Fig. 8.36 Tiempo de ejecución de los algoritmos.

CAPITULO IX

PRUEBA DE CALIDAD DEL CODIFICADOR IMPLEMENTADO

En el presente capítulo se presentan los resultados de las pruebas subjetivas del codificador implementado evaluando la voz sintética decodificada mediante el test MOS.

9.1 Test MOS (Mean Opinion Score)

El test MOS consiste en una evaluación subjetiva de la calidad de síntesis de voz de un sistema. Fue normalizado por el comité Consultivo Internacional de Telefonía y Telegrafía (CCITT) a principios de los años 80 y se ha utilizado principalmente para medir la calidad en sistemas de comunicación celular digital.

El test consiste en realizar una encuesta de opinión a un conjunto de individuos de prueba los cuales deben evaluar una grabación de voz según la TABLA Nº 9.1.

TABLA Nº 9.1 Nivel de calificación en el test MOS

NOTA	CALIDAD	ESFUERZO DE ESCUCHA	DEGRADACION
5	Excelente	Posible relajación completa, no requiere ningún esfuerzo.	Inaudible
4	Buena	Atención necesaria, no se requiere esfuerzo apreciable	Audible pero no molesta
3	Aceptable	Se necesita esfuerzo moderado	Ligeramente molesta
2	Mediocre	Se necesita esfuerzo considerable	Molesta
1	Mala	Cualquier esfuerzo no permite comprender	Muy Molesta

En el presente trabajo, se escogió un universo de 10 personas (no especialistas en voz) a quienes se les hizo escuchar 10 conjuntos de secuencias de voz de diferentes

personas. Cada conjunto estaba compuesto por una secuencia original, una secuencia decodificada con el algoritmo de punto flotante y una secuencia decodificada con el algoritmo de punto fijo.

Las 10 secuencias originales que fueron grabadas en el CIDFIEE UNI, son archivos .wav de 16 bits a 8 KHz y son las siguientes:

Secuencia "voz1_DArgandona.wav". Se muestra en la Fig. 9.1.
Secuencia "voz1_FPujaico.wav". Se muestra en la Fig. 9.2.
Secuencia "voz1_JdelCarpio.wav". Se muestra en la Fig. 9.3.
Secuencia "voz1_JHuaman.wav". Se muestra en la Fig. 9.4.
Secuencia "voz2_DArgandona.wav". Se muestra en la Fig. 9.5.
Secuencia "voz2_FPujaico.wav". Se muestra en la Fig. 9.6.
Secuencia "voz2_JdelCarpio.wav". Se muestra en la Fig. 9.7.
Secuencia "voz2_JHuaman.wav". Se muestra en la Fig. 9.8.

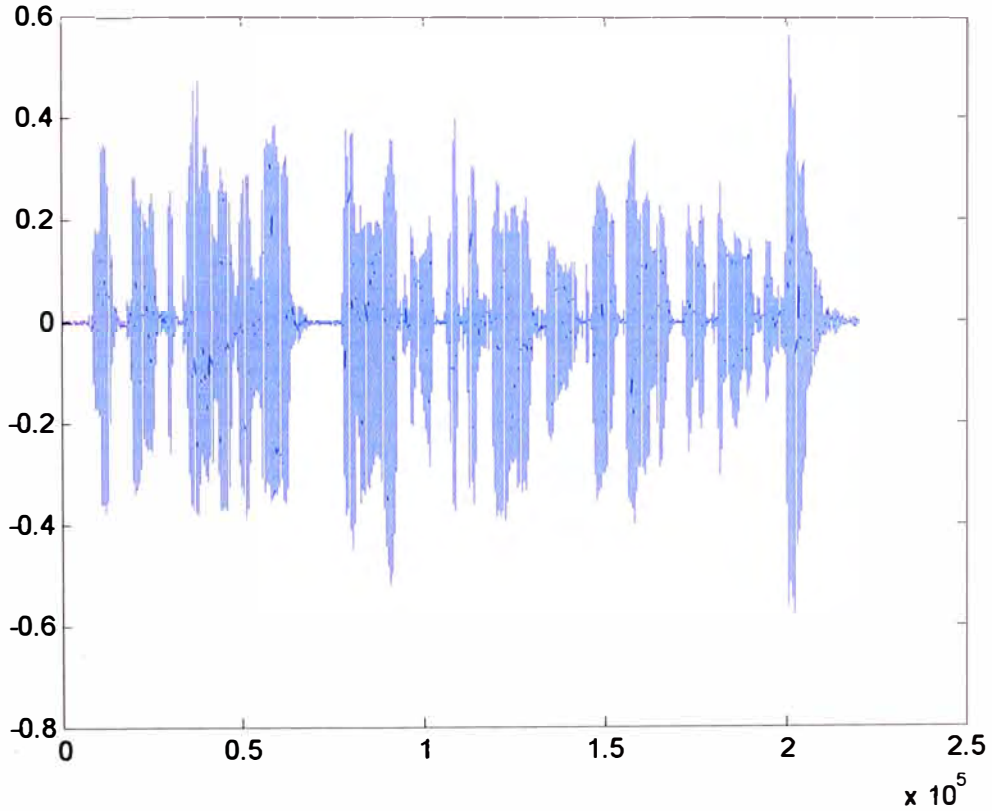


Fig. 9.1 Secuencia de voz original "voz1_DArgandona.wav": "*Los codificadores de voz tienen muchas aplicaciones en el campo de las telecomunicaciones.*"

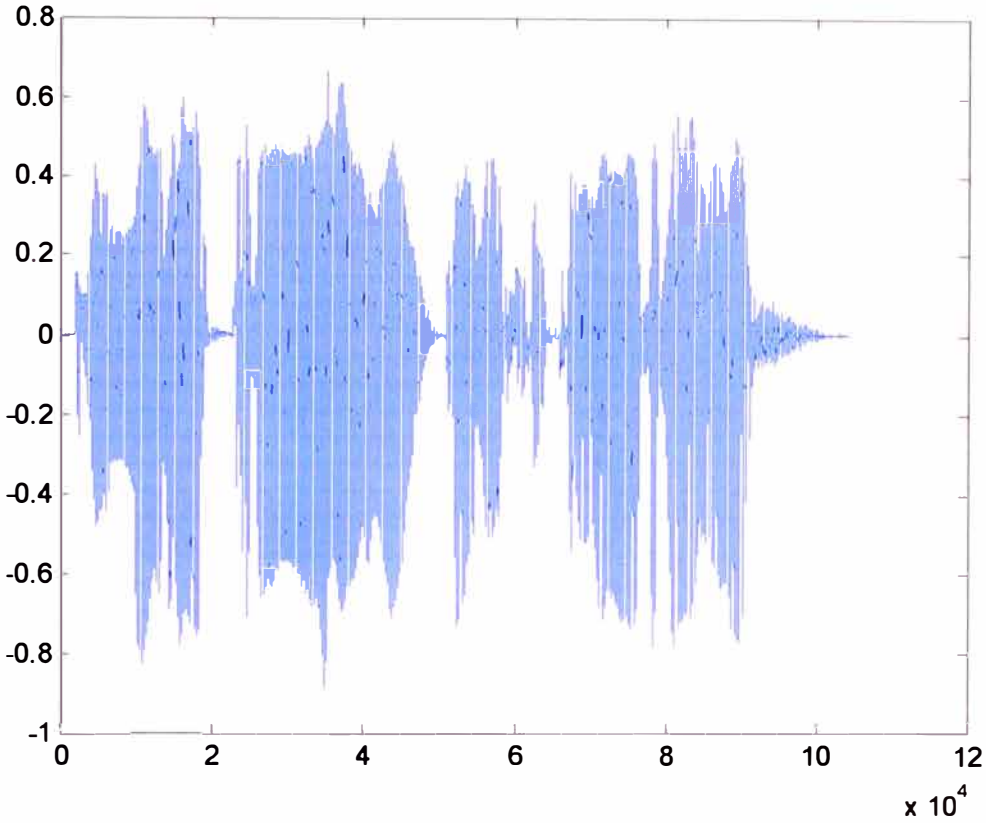


Fig. 9.2 Secuencia de voz original "voz1_FPujaico.wav": "*Primera prueba del codificador de voz*"

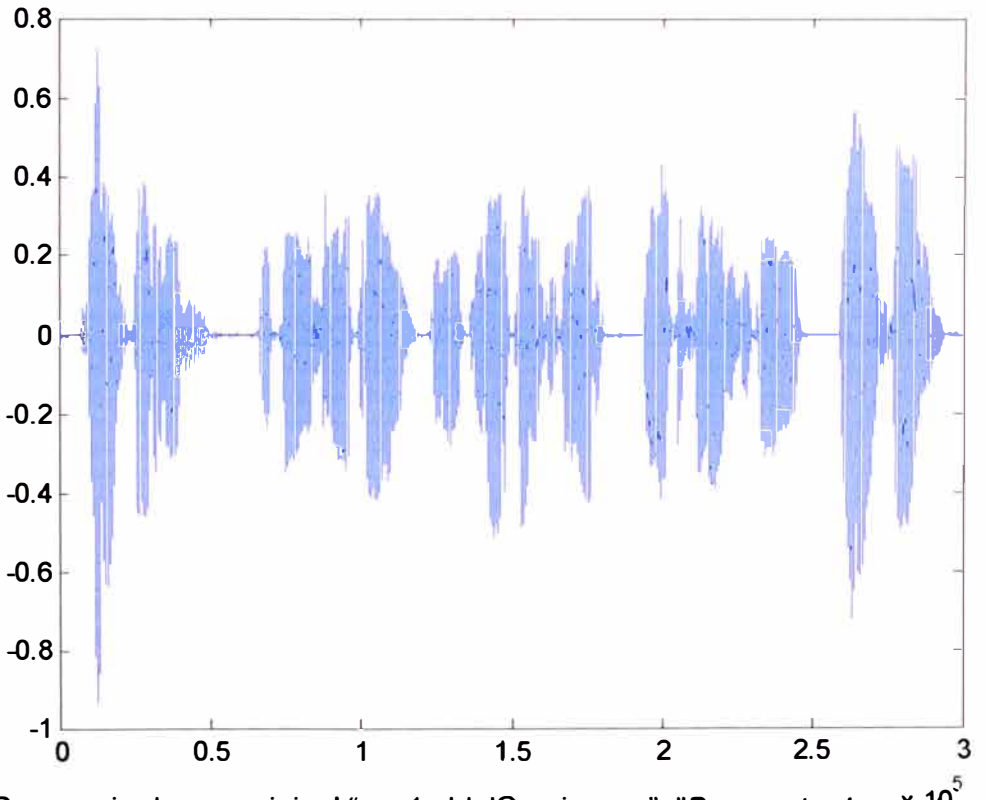


Fig. 9.3 Secuencia de voz original "voz1_JdelCarpio.wav": "*Buenas tardes, esta es una prueba con el primer codificador Lpc modificado Uni-Fiee*"

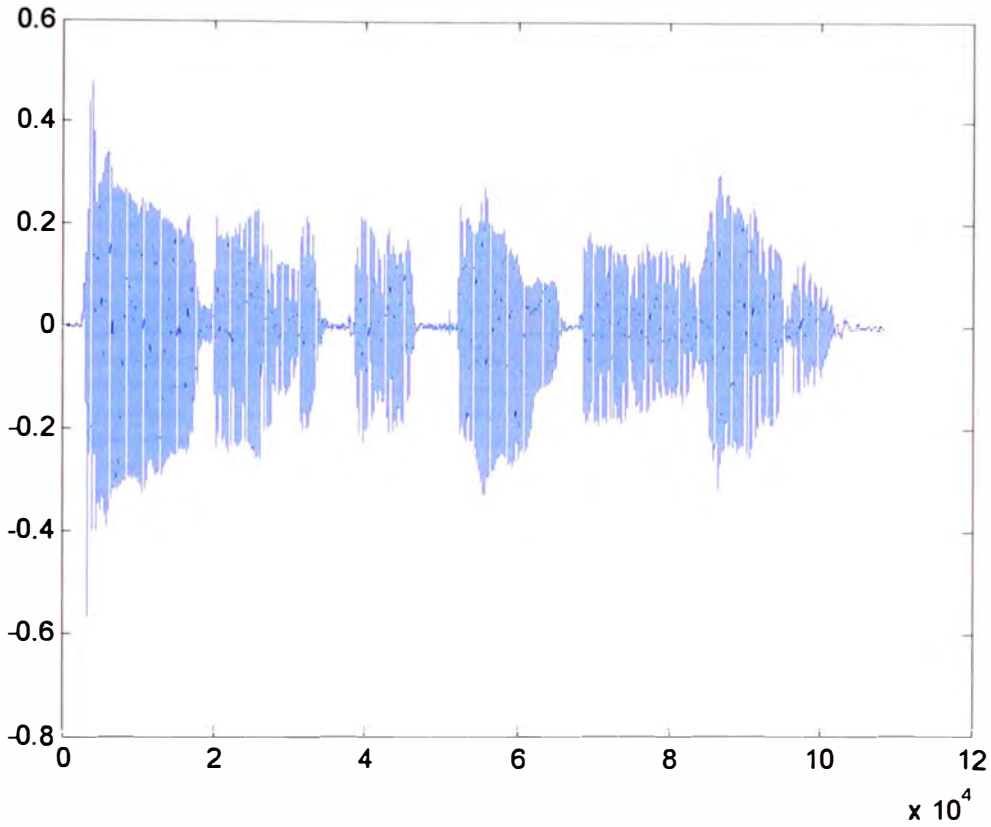


Fig. 9.4 Secuencia de voz original "voz1_JHuaman.wav": "*Prueba de investigación para Huaura*"

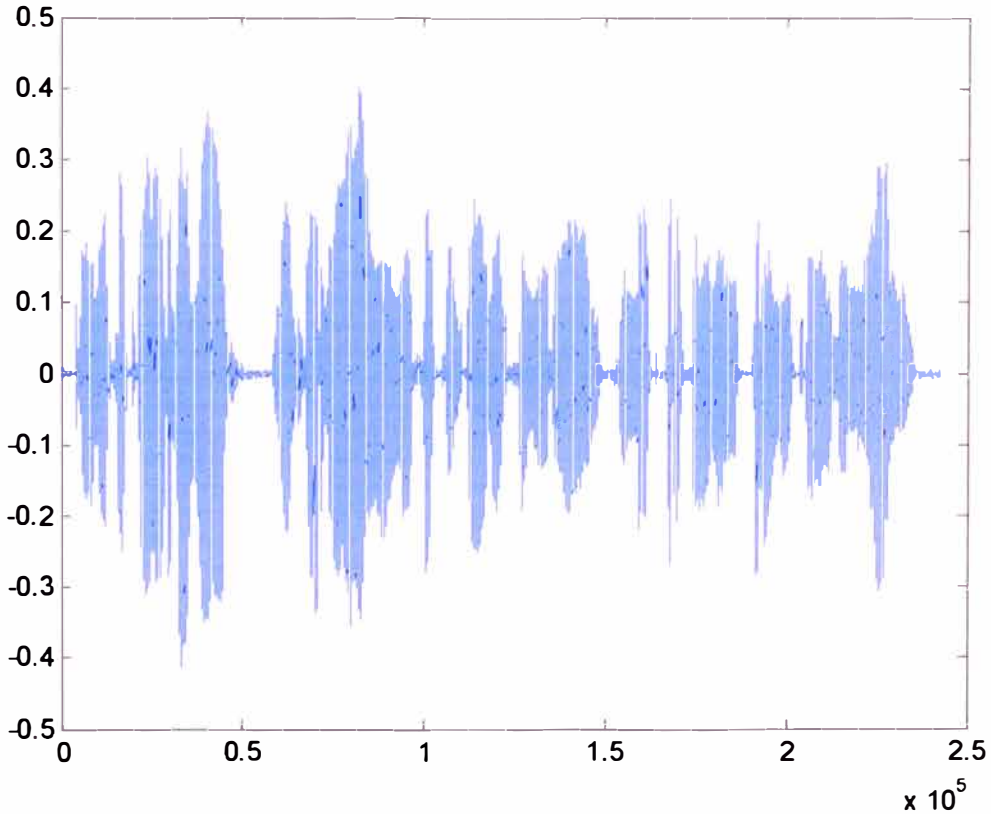


Fig. 9.5 Secuencia de voz original "voz2_DArgandona.wav": "*Codificador de voz desarrollado en la facultad de Ingeniería Eléctrica y Electrónica de la UNI*"

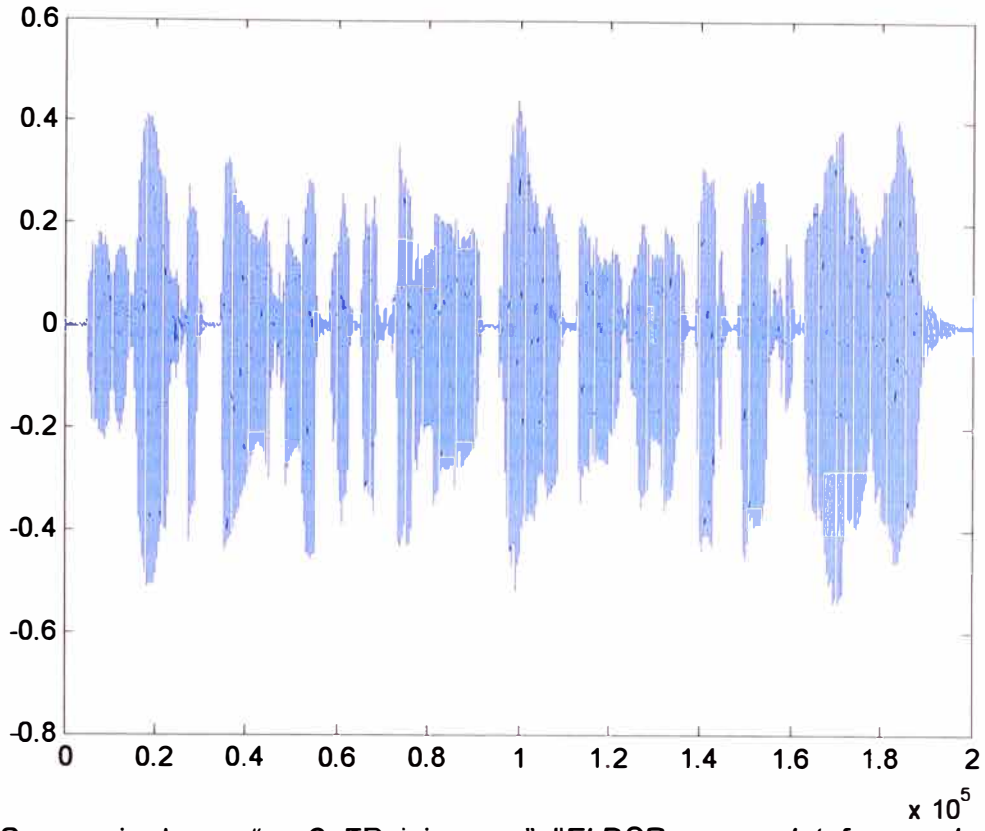


Fig. 9.6 Secuencia de voz "voz2_FPujaico.wav": "*El DSP es una plataforma adecuada para implementar codificadores de voz*"

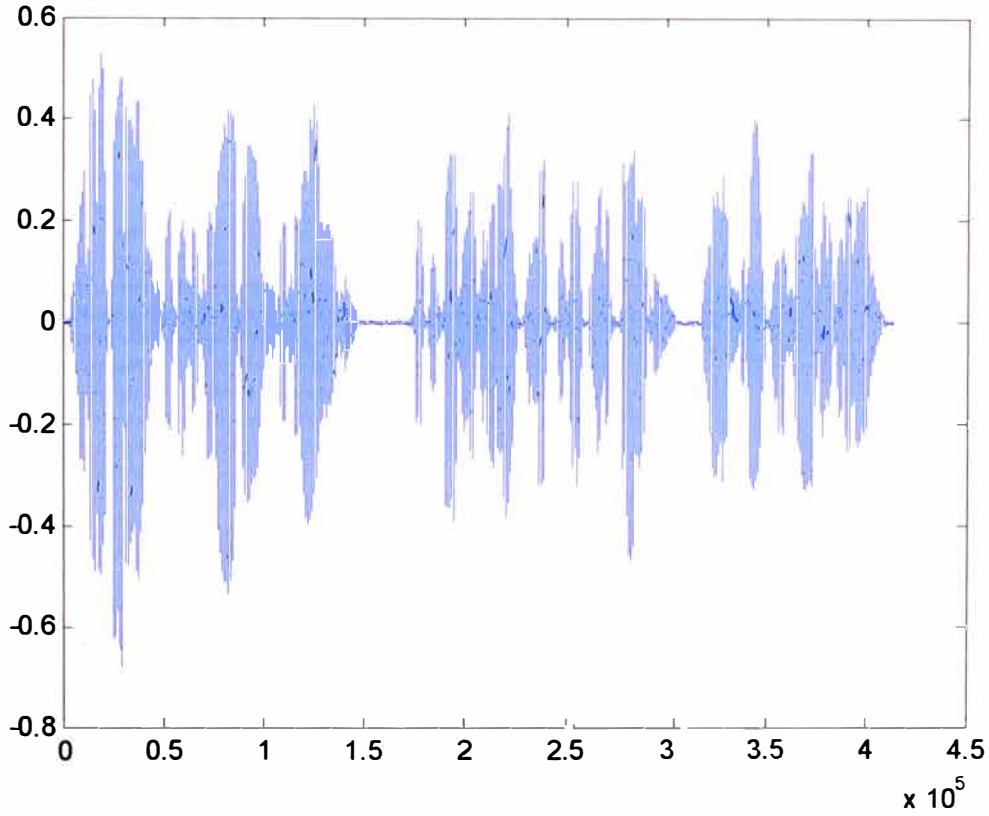


Fig. 9.7 Secuencia de voz "voz2_JdelCarpio.wav": "*Laboratorio de procesamiento de señales, Facultad de Ingeniería Eléctrica y Electrónica, Universidad Nacional de Ingeniería*"

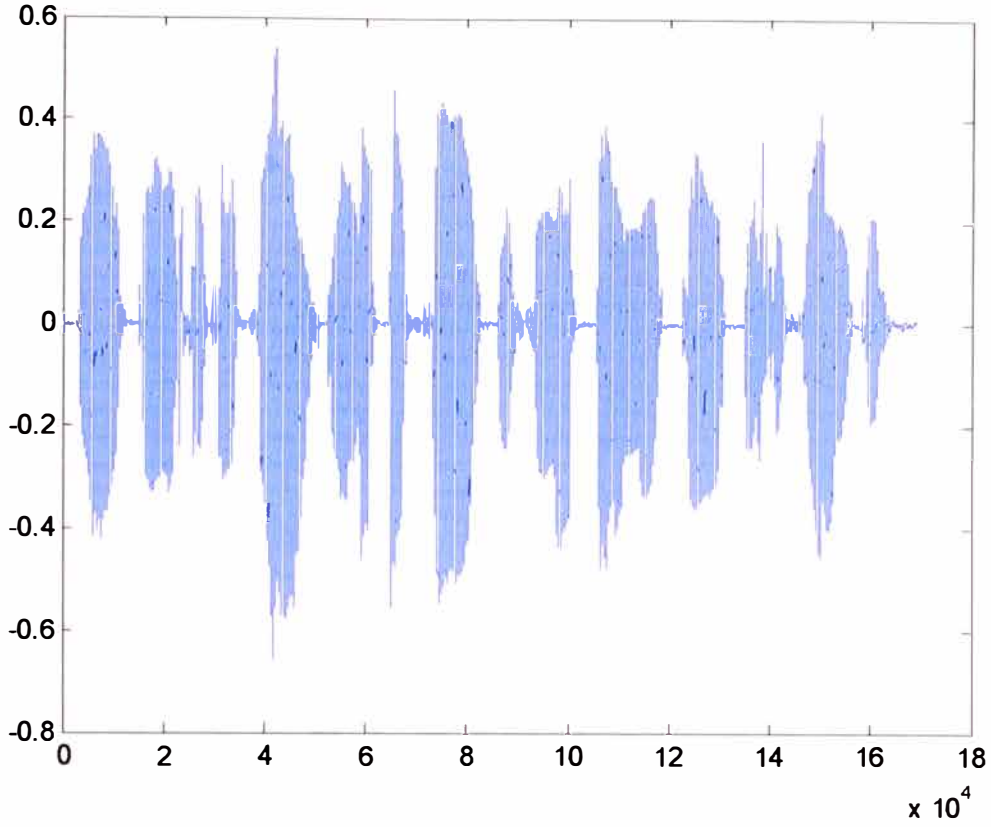


Fig. 9.8 Secuencia de voz "voz2_JHuaman.wav": "*La codificación LPC es una técnica eficiente*"

9.2 Resultados del Test MOS

Los resultados de las pruebas se muestran a continuación. Cada individuo calificó 2 secuencias de voz sintéticas decodificadas (versión punto fijo y punto flotante) comparándolas con la secuencia original (nota 5 en la escala).

TABLA Nº 9.2. Test MOS para la secuencia "voz1_DArgandona.wav"

		Tipo de secuencia	
Secuencia	**Individuo**	**Algoritmo punto Fijo**	**Algoritmo punto Flotante**
"voz1_DArgandona.wav"	1	3,00	4,00
	2	3,00	3,00
	3	3,00	3,00
	4	3,00	4,00
	5	3,30	3,90
	6	3,00	3,90
	7	3,20	3,80
	8	3,00	4,00
	9	3,50	3,80
	10	3,00	4,00
	Promedio	3,1	3,74

TABLA Nº 9.3 Test MOS para la secuencia "voz1_FPujaico.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz1_FPujaico.wav"	1	4,00	4,00
	2	4,00	4,00
	3	3,50	3,50
	4	4,00	4,00
	5	3,50	3,80
	6	3,80	3,80
	7	4,00	4,40
	8	3,80	4,00
	9	4,00	4,00
	10	3,60	4,00
	Promedio	3,82	3,95

TABLA Nº 9.4. Test MOS para la secuencia "voz1_JdelCarpio.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz1_JdelCarpio.wav"	1	3,50	3,20
	2	3,00	3,00
	3	3,00	3,50
	4	3,50	4,00
	5	3,00	3,80
	6	3,10	3,50
	7	3,00	4,00
	8	2,90	3,80
	9	2,80	3,60
	10	3,00	3,50
	Promedio	3,08	3,59

TABLA Nº 9.5. Test MOS para la secuencia "voz1_JHuaman.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz1_JHuaman.wav"	1	4,50	4,50
	2	4,00	4,00
	3	4,20	4,20
	4	4,50	4,50
	5	4,00	4,00
	6	4,40	3,80
	7	4,20	3,90
	8	3,80	4,00
	9	4,00	4,20
	10	3,60	4,40
	Promedio	4,12	4,15

TABLA Nº 9.6. Test MOS para la secuencia "voz2_DArgandona.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz2_DArgandona.wav"	1	3,80	4,00
	2	4,00	3,90
	3	3,80	3,50
	4	3,80	4,00
	5	3,70	3,50
	6	4,00	4,20
	7	3,60	4,00
	8	3,80	4,00
	9	3,80	3,90
	10	4,00	3,80
	Promedio	3,83	3,88

TABLA Nº 9.7. Test MOS para la secuencia "voz2_FPujaico.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz2_FPujaico.wav"	1	3,20	4,00
	2	4,00	3,90
	3	3,50	3,80
	4	3,20	4,00
	5	3,70	3,50
	6	4,00	4,00
	7	3,60	3,80
	8	3,70	3,80
	9	3,90	4,00
	10	3,60	3,90
	Promedio	3,64	3,87

TABLA Nº 9.8. Test MOS para la secuencia "voz2_JdelCarpio.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz2_JdelCarpio.wav"	1	3,20	3,60
	2	3,50	3,50
	3	3,80	3,80
	4	3,20	3,60
	5	3,70	3,50
	6	3,60	3,90
	7	3,50	4,00
	8	3,40	3,80
	9	3,50	3,80
	10	3,60	3,60
	Promedio	3,5	3,71

TABLA Nº 9.9. Test MOS para la secuencia "voz2_JHuaman.wav"

Secuencia	Individuo	Tipo de secuencia	
		Algoritmo punto Fijo	Algoritmo punto Flotante
"voz2_JHuaman.wav"	1	4,00	4,20
	2	3,20	3,20
	3	4,20	4,20
	4	4,00	4,20
	5	4,00	4,00
	6	3,80	4,00
	7	3,60	3,80
	8	3,80	3,80
	9	4,00	4,00
	10	3,60	4,00
	Promedio	3,82	3,94

CONCLUSIONES

A lo largo del presente trabajo se han abarcado varios temas, desde fundamentos teóricos del procesamiento de voz, estudio de varias técnicas para codificadores de voz, múltiples esquemas algorítmicos, optimización de código, de programación, pruebas de calidad de la voz, entre otros temas.

Las conclusiones que se derivan del presente trabajo son las siguientes:

- La codificación de la voz a bajas tasas de bits y con una buena calidad de la voz sintética decodificada, es posible empleando los esquemas y algoritmos apropiados.

- Existen diversos esquemas y técnicas estandarizadas que codifican la voz en un amplio rango de valores de la tasa de bits y ofrecen a la vez, diversos grados de calidad de la voz decodificada. (Capítulo 1)

- Los canales de banda angosta imponen una restricción fuerte en cuanto a la tasa de bits permitida para los codificadores de voz, lo que nos obliga a centramos en el desarrollo de codificadores para bajas tasas de bits (Capítulo 1).

- Entre las técnicas más ampliamente difundidas para codificación a bajas tasas de bits están las basadas en el modelo LPC-10, la cual ofrece una calidad de voz moderada y por ello ha sido ampliamente usada desde los años 80 (capítulo 3).

- Se ha comprobado que el algoritmo SIFT para el cálculo óptimo del periodo "pitch" para una trama de voz es un algoritmo eficiente y que conlleva a resultados satisfactorios (Capítulos 4, 6 y 8).

- Las plataformas disponibles para desarrollo de codificadores de voz en hardware son variadas (DSP, FPGA, Microprocesadores, ASIC, etc.) presentando cada una

de ellas ventajas y desventajas, pero ofreciendo a la vez versatilidad al diseñador (Capítulo 5).

- Se ha comprobado que con herramientas computacionales adecuadas como el System Generator de Xilinx, se pueden simular diseños con "precisión de bit y de clock" tal como se vio en el capítulo 6. La precisión de la herramienta empleada es igual a la que se obtendría trabajando en el hardware real de un dispositivo FPGA (Xilinx), por lo que la simulación realizada es un paso importante para posteriores trabajos en dispositivos FPGA de Xilinx. (Capítulo 5 y 6).

- Se consiguió implementar de manera exitosa el codificador en el hardware DSP TMS320C6711 de Texas Instruments en dos versiones: punto fijo de 16 bits y punto flotante. La versión en punto fijo requirió muchas consideraciones adicionales para el tratamiento de las rutinas y variables debido a que se trabajó con un ancho ajustado de 16 bits. (Capítulo 8).

- Uno de los objetivos de la presente implementación fue la de minimizar el tiempo de ejecución del código (ya que el sistema destino era un DSP), empleando algoritmos adecuados que optimizan diversos bloques, entre ellos los más críticos: algoritmo SIFT, cálculo de la ganancia y coeficientes RC (Capítulo 8).

- La codificación en DSP requiere una serie de consideraciones especiales que dependen de la memoria física disponible, los tiempos de acceso, tipos de llamada a funciones, opciones de compilación, etc. Como se vio en el Capítulo 8, el proceso de optimización del código es un proceso escalonado y el nivel de optimización que se puede alcanzar depende del nivel al que lleguemos programando el código, si se quiere optimizar al máximo, todo el código se deberá realizar en assembler puro, lo cual muchas veces es impráctico (Capitulo 8).

- En el presente trabajo se logró un nivel de optimización considerable en lo que respecta a la velocidad de ejecución del código, lo cual se logró por medio de la optimización del código para el DSP y usando las opciones correctas del compilador. (Capítulo 8).

- La calidad de la voz sintética decodificada tiene un MOS promedio que se sitúa entre 3 y 4, esto dependiendo de la secuencia de voz a codificar y de la versión

del algoritmo (punto fijo o flotante). En la sección 8.2 se ha comprobado mediante las pruebas del test MOS que en general la calidad de la voz producto del algoritmo en punto flotante es superior a la calidad de la voz producto de la versión en punto fijo. Esto era de esperarse ya que la versión en punto fijo de 16 bits es una versión muy ajustada, que podría implementarse sin ningún problema en otros hardware destino de menos prestaciones que un DSP, tal como un microcontrolador de 16 bits.

- Finalmente, el código implementado se ha realizado de forma modular de modo que las futuras mejoras pueden ser realizadas solo a los bloques de código sin afectar el resto del código.

BIBLIOGRAFÍA

[1] Rabiner, L; Schafer, R; "Digital Processing of Speech Signals", 1978, Prentice Hall

[2] Xuendong Huang, Alex Acero, Hon, "Spoken Language Processing – A Guide to Theory, Algorithm and System Development" 2001, Prentice Hall

[3] Randy Goldberg, Lance Rick, "A Practical Handbook of Speech Coders" 2000, CRC

[4] Wai Chu, "Speech Coding Algorithms, Foundation and Evolution of Standardized Coders" 2003, Jhon Wiley Sons

[5] Wu Chou, Biing Hwang Juang, "Pattern Recognition in Speech and Language Processing", 2003, CRC Press

[6] Tatham Mark, Morton Katherine, "Developments in Speech Synthesis", 2005, Wiley and Sons

[7] Levinson Stephen, "Mathematical Models for Speech Technology", 2005, Wiley and Sons

[8] Jurafsky, D; Martin, J.; "Speech and Language Processing", 2000, Prentice Hall

[9] Divenyi, Pierre; "Speech Separation by Humans and Machines", 2005, Kluver Academic Publishers

[10] Héctor Kaschel C.1 Francisco Watkins1 Enrique San Juan U.,"Compresión de Voz mediante Técnicas Digitales para el Procesamiento de Señales y Aplicación de Formatos de Compresión de Imágenes", Rev. Fac. Ing. - Univ.Tarapacá, vol. 13 Nº 3, 2005

[11] Javier Alejandro Bustos Jiménez, "Estudio de Sistemas de compresión de voz digital orientado a telefonía celular", Universidad de Chile, 2002

[12] Sara Grassi, "Optimized Implementation of Speech Processing Algorithm", 1998

[13] F. Itakura, "Line Spectrum Representation of Linear Predictive Coefficients of Speech Signals", J. Acoust. Soc. Amer., vol. 57, S35, 1975

[14] J. Picone and G. Doddington, "A Phonetic Vocoder", Proc. IEEE Acoust., Speech Signal Processing, pp. 580-583, 1989

[15] Faiza Anees, Mariam Shakeel, Shaista Nazir, "Implementation of Acoustic Echo Cancellation in Matlab and TI's TMS320 C6711", COMSATS INSTITUTE OF INFORMATION AND TECHNOLOGY, 2004

[16] Islam, Tammana; "Interpolation of Linear Prediction Coefficients for Speech Coding", Dept. of Electrical Engineering, McGill University, Canada

[17] Texas Instruments, "Implementing Vocoder y HF algorithms in DSP", 1995, Texas Isntruments.

[18] Tremain T., Kemp, David, "Evaluation of low rate speech code for HF" US Department of Defense, 1993

[19] LeBlanc W., Bhattacharya B.,"Efficient Search and Design Procedures for Robust Multi Stage VQ of LPC Parameters for 4kbps Speech coding" IEEE Transactions on Speech and Audio Processing, VOL 1, Nº 4, 1993.

[20] Tremain T., Campbell J.,"A comparison of US Goverment Standard Voice Coders", US Department of Defense, 1989.

[21] Crochiere R., Rabiner L. "Interpolation and Decimation of Digital Signals - A tutorial review", Proceedings of the IEEE, Vol 69, Nº3, 1981

[22] Jhon Makhoul, "Linear Prediction - A Tutorial review", Proceedings of the IEEE, Vol 63, Nº 4, 1975.

[23] P. Vaidyanathan, "Multirate Digital Filters Filter Banks Polyphase Networks and Applications: A Tutorial", Proceedings of the IEEE, Vol 78, Nº 1, 1990

[24] Juan Bello, Laurent Daudet "Onset Tutorial", IEEE Transactions on Speech and Audio Processing Vol 13, Nº 5, 2005

[25] Andreas Spanias, "Speech Coding: A Tutorial Review", Proceedings of the IEEE, Vol 82, Nº 10, 1994

[26] Campbell J, Tremain T, "Voiced / Unvoiced classification of speech with applications to the US government LPC-10E algorithm", Department of Defense, 1986

[27] Jhon Markel, "The Sift algorithm for Fundamental Frequency Estimation", IEEE Transactions on Audio and Electroacoustics, Vol 20, Nº 5, 1972

[28] Thomas E. Tremain, "The Government Standard Linear Predictive Coding Algorithm: LPC-10," Speech Technology Magazine, April 1982, p. 40-49..

[29] Kroon & Atal, "Pitch Predictors with High Temporal Resolution," ICASSP '90, S12.6

[30] Kroon, Peter and Bishnu Atal, "On Improving the Performance of Pitch Predictions in Speech Coding Systems," IEEE Speech Coding Workshop, September 1989.

[31] FPGA XILINX WorkShop "FPGA Design Flow, Basic FPGA Architecture". Xilinx University Program, Xilinx, 2003.

[32] FPGA XILINX WorkShop "DSP Design Flow, Introduction". Xilinx University Program, Xilinx, 2003.

[33] "Xilinx University Program Virtex-II Pro Development System, Hardware Reference Manual", Xilinx, 2005