Curso Teórico-Práctico en Procesamiento Digital de Señales, enfocado al audio

Luis MARTÍNEZ-FERRÍN, Alejandro ROSSO-PÉREZ

Trabajo de grado para optar al título de Ingeniero Electrónico

Director
Jorge Mario Valencia Upegui
Master of Arts in Music Technology - Ingeniero Electrónico

Universidad Pontificia Bolivariana
Escuela de Ingenierías
Facultad de Ingeniería Eléctrica y Electrónica
Programa de Ingeniería Electrónica
Medellín
2015

Dedicatoria

A mis padres por la vida; a la vida por mis padres.

Luis.

A mis abuelos, que con su esfuerzo e incondicional cuidado, construyeron un mundo de la nada.

Alejandro.

Contenido

Lista de Figuras

Glosario

ADC: Siglas en inglés para Conversor Análogo Digital. Dispositivo que se encarga de muestrear y cuantizar una señal analógica, generalmente de voltaje.

Algoritmo: Conjunto de instrucciones organizadas secuencialmente, que permiten resolver un problema determinado generalmente de manera óptima.

Aliasing: Efecto indeseado en una señal muestreada, que ocurre cuando la frecuencia de muestreo es inferior a la frecuencia de Nyquist. Debido a esto, frecuencias altas aparecen como frecuencias de menor valor, impidiendo reconstruir la señal original.

Ancho de Banda: Es la porción del espectro en la que se encuentra contenida la energía de la señal.

Bit reversing: Algoritmo de permutación del orden de los bits de un valor binario, muy útil en la implementación de la FFT y otros algoritmos de DSP.

Bypass: En procesamiento de señales es el término que se usa para designar aquel proceso en el que las señales no se modifican.

Convolución: Operación matemática que se realiza entre dos señales y de gran utilidad en el procesamiento para calcular la salida de sistemas LTI.

Correlación: Producto interior que se define entre dos señales y sirve para conocer matemáticamente que tan parecidas son.

Cuantización: Proceso mediante el cual se discretiza una señal en términos de su amplitud.

DAC: Siglas en inglés para Conversor Digital Análogo. Dispositivo que se encarga de convertir una palabra binaria en un valor de voltaje.

DARAM: Siglas en inglés para RAM de Acceso Dual.

DFT: Siglas en inglés para Transformada Discreta de Fourier.

Distorsión: Es la deformación que sufre una señal al pasar por un sistema

DMA: Sigas en inglés para Acceso Directo a Memoria.

DSP: Puede referirse al área de estudio del Procesamiento Digital de Señales o al hardware digital especializado para realizar las tareas de dicha área de estudio (Procesador Digital de Señales).

DTFT: Siglas en inglés para Transformada de Fourier de Tiempo Discreto.

Endianness: Es la forma en la que se organizan en memoria los bytes de un dato en relación a su peso. Estos pueden ser normalmente *big enian* o *little endian.*

Espectro: Contenido de frecuencias de una señal en relación a su amplitud y fase.

FFT: Siglas en inglés para Transformada Rápida de Fourier. Algoritmo computacional que permite calcular de manera rápida y eficiente la DFT de una señal.

Filtro: Operador matemático que se aplica sobre una señal o secuencia numérica, que permite modificar su espectro.

Frecuencia de Nyquist: Frecuencia mínima a la que debe muestrearse una señal para que esta pueda ser reconstruida sin pérdida de información a partir de las muestras tomadas.

Guard bits: Bits de extensión para la prevención de *overflow* en las operaciones de las unidades MAC de un DSP

Latencia: Es el tiempo que tarda un sistema en producir una salida luego de ser excitado con cierta entrada

LTI: Siglas en inglés para Lineal e Invariante en el Tiempo.

LUT: Siglas en inglés para *Lookup table*. Es un arreglo de datos calculados y almacenados en memoria previamente a la ejecución de un programa, usados para sustituir la computación de estos en tiempo de ejecución

MAC: Unidad fundamental en un procesador digital de señales encargada de realizar operaciones de multiplicación y acumulación.

MIPS: Siglas en inglés para Mega Instrucciones por Segundo. Es una medida de la capacidad de cómputo de un procesador.

Muestreo: Proceso mediante el cual se discretiza una señal en el tiempo.

Rango dinámico: Es el margen que hay entre el nivel máximo de señal posible sin distorsión y el ruido de fondo, medido en decibeles.

Retardo: Desplazamiento causal de una señal o muestra discreta en una unidad de tiempo discreto

SARAM: Siglas en inglés para RAM de Acceso Simple.

STFT: Siglas en inglés para Transformada de Fourier de Tiempo Corto.

Resumen

Con este trabajo de grado se buscan desarrollar de manera clara y ordenada los conocimientos teóricos y prácticos fundamentales que sirvan de guía y apoyo a la docencia, para la creación de un curso en procesamiento digital de señales enfocado al audio. Para el desarrollo de las prácticas se usa el kit de desarrollo TMS320C5535 eZdsp™ de Texas Instruments y el entorno de desarrollo Code Composer Studio.

Palabras clave: DSP, procesamiento de señales, audio, programación, sistemas de tiempo discreto.

Abstract

The idea of this work is to gather the theoretical and practical knowledge required to teach an introductory course in digital signal processing, with an emphasis on audio applications. To develop the practices it is used the TMS320C5525 eZdsp™ USB Kit from Texas Instruments together with the Code Composer Studio IDE.

Keywords: DSP, signal processing, audio, programming, discrete time systems.

INTRODUCCIÓN

El procesamiento digital de señales es una técnica de manipulación matemática de la información usada para modificar, extraer o sintetizar las características importantes de una señal mediante el uso de hardware y software. En la actualidad, los diferentes avances tecnológicos han llevado a la implementación masiva de estas técnicas en diversos campos y aplicaciones como radares y GPS para navegación, celulares inteligentes, reproductores multimedia, cámaras digitales, televisores, etc. Y en el audio digital para: grabación, producción, síntesis, mezcla, creación de efectos de sonido, análisis de ondas de audio, entre otras.

En el ámbito académico se dictan alrededor del mundo diversos cursos sobre el tema, los cuales utilizan como parte del aprendizaje módulos de desarrollo de DSP y software matemáticos como Matlab®. Algunas de las universidades que incluyen estos cursos en su plan de estudio son: Universidad de Oxford, Universidad de Cambridge, Universidad de Stanford, Colegio de Ingeniería de Miami, Universidad de Griffith, Universidad de Sydney y Universidad de Birmingham. También se destacan un gran número de grupos de investigación en estas academias, que estudian las aplicaciones del procesamiento digital en el audio, la bioingeniería, la sismología, la biométrica, el aprendizaje de máquina, la meteorología, el reconocimiento de imágenes, video y voz.

Debido a esto, en este proyecto de grado se estudian los diferentes aspectos teóricos que influyen en la construcción de un curso teórico-práctico en procesamiento digital de señales enfocado al audio que pueda ser tomado como parte de la Bolsa de Optativas desde diferentes programas de pregrado de la Escuela de Ingenierías de la Universidad Pontificia Bolivariana, y que además esté incluido en el portafolio de Formación Continua de esta. Se espera que este trabajo sirva como material bibliográfico de apoyo a la docencia.

En la sección 1 se describe el módulo de desarrollo escogido y los criterios considerados para su elección. En las secciones 2 y 3 se exponen las bases teóricas necesarias para el desarrollo del curso. En la sección 4 se revisan los aspectos fundamentales para el manejo del hardware y el software. En las secciones de la 5 a la 11 se encuentran los temas escogidos para desarrollar el curso. Finalmente en la sección 12 se tratan otros aspectos relevantes para la docencia.

1. HARDWARE Y SOFTWARE ADQUIRIDOS

Para que este proyecto resulte en un curso completo y viable, es primordial escoger un equipo que permita una enseñanza óptima que además sea cautivadora para el estudiante. A partir de allí lo primero es determinar los requisitos mínimos que deben cumplir el DSP y el kit de desarrollo de acuerdo a los alcances de la materia, dentro de un costo razonable. Entre estos están:

- Procesador de 16 bits de punto fijo o 32 bits de punto flotante. Esta resolución garantiza en procesamiento de audio un rango dinámico amplio y un ruido de cuantización bajo
- Códec de audio estéreo (ADC y DAC) con frecuencia mínima de muestreo de 44.1 kHz que cuente con una entrada y una salida analógica
- 90000 palabras de memoria RAM (*Random Access Memory*). Esta capacidad permite almacenar suficientes muestras de audio para realizar las prácticas propuestas
- 100 MIPS para poder realizar todos los algoritmos de procesamiento de audio sin afectar los ciclos de lectura y escritura del códec de audio

Además de las características mencionadas, es importante considerar la calidad de la documentación disponible y el soporte brindado en los sitios web y foros de los fabricantes. Esto garantiza un respaldo adicional que permita depurar eficientemente los problemas que puedan presentarse durante el desarrollo del curso.

Entre los diversos módulos que se evaluaron están:

- Raspberry Pi B+
- Arduino Due
- Altera DSP Development Kit, Cyclone II
- Analog Devices ADAU1452
- Analog Devices ADSP-2189M EZ-KIT
- Texas Instruments TMS320VC5510 DSP Starter Kit
- Texas Instruments C5535 eZdsp USB Stick Development Kit

En la Tabla 1 se encuentran comparados los requerimientos mínimos de los módulos evaluados. En esta tabla se han omitido algunos de los elementos ya listados debido a que no están diseñadas completamente para lo que se necesita en el desarrollo del curso.

Debido a la relación costo/capacidades descritas anteriormente, se ha escogido el kit de desarrollo de Texas Instruments C5535 eZdsp™ para el desarrollo de este curso. Dicho módulo (Figura 1) cumple satisfactoriamente los requerimientos mínimos mencionados, además de estar respaldado por foros activos,

librerías y una muy buena documentación dispuesta abiertamente por el fabricante.

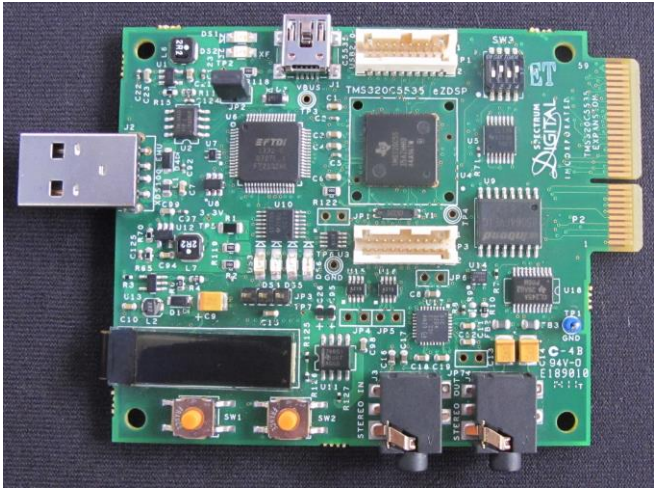


Figura 1. Módulo de desarrollo seleccionado

Este kit de desarrollo de Texas Instruments cuenta con el DSP TMS320C5535 de punto fijo de 16 bits, con capacidad de 320KB de RAM, 2 unidades MAC y una velocidad de procesamiento de hasta 240 MIPS. En la tarjeta se encuentran además:

- Un códec de audio estéreo AIC3204 de 16 bits y con una frecuencia de muestreo de hasta 192KHz, conectado a dos *jacks* estéreos de salida y de entrada
- Una memoria flash SPI de 8 MB
- Un conector para micro SD
- Emulador JTAG para USB, XDS100
- 5 leds de diferentes colores
- Un *display* OLED de 96*16 pixeles
- Dos pulsadores
- Capacidad de expansión para interfaz inalámbrica
- Periféricos de entrada y salida: I^2C, I^2S, SPI, UART

Aparte de esto, el kit C5535 eZdsp™ viene con el software de desarrollo Code Composer Studio (CCS) de Texas Instruments, basado en Eclipse y tiene capacidad para soportar los lenguajes: C, C++ y *assembly*. Este IDE funciona en los sistemas operativos Windows y Linux.

En la Figura 2 se puede observar el entorno usual de trabajo del Code Composer Studio.

1.1. Arquitectura del DSP TMS320C5535

La familia de DSP C55xx lanzados en el 2011 por Texas Instruments, son dispositivos de muy buena acogida en el mercado del procesamiento digital ya que pese a su bajo costo poseen un gran desempeño y un consumo energético reducido.

Tabla 1. Comparativo de los kits de desarrollo

Kit de desarrollo	Número de bits	MIPS	Unidades *MAC*	RAM (kB)	ROM (MB)	ADC/ DAC	Frecuencia máxima de muestreo (kHz)	I/O analógicas	Precio aprox. (USD)
Altera Cyclone II	24	-	-	8704	4	Sí	96	2 entradas y una salida estéreo	199
Analog Devices ADAU1452	32	294	4	160	128 (kB)	Sí	192	2 entradas y 4 salidas estéreo	200
Analog Devices ADSP-2189M EZ-KIT	16	75	1	4	256 (kB)	Sí	64	1 entrada y 1 salida estéreo	300
Texas Instruments TMS320VC5510	24	400	2	8192	756 (kB)	Sí	96	2 entradas y 2 salidas estéreo	295
Texas Instruments C5535 eZdsp	16	240	2	320	8	Sí	192	1 entrada y 1 salida estéreo	99

Como se observa en la Figura 3, la CPU de este DSP cuenta con una estructura de bus interna compuesta por un bus de programa, 3 buses de lectura de datos: dos de 16 bits y uno de 32 bits, dos buses de escritura de datos de 16 bits y otros buses adicionales de uso específico para periféricos y DMA. Con ellos el procesador puede realizar hasta 4 lecturas y 2 escrituras de 16 bits en un solo ciclo de reloj. Además, el núcleo cuenta con dos unidades MAC, cada una capaz de realizar multiplicaciones de 17x17 bits y una suma de 32 bits en un solo ciclo, y dos unidades ALU (*Arithmetic Logic Unit*) de 40 y 16 bits (Texas Instruments Incorporated, 2011).

Aparte de esto, el sistema está construido de forma tal que cada uno de los cuatro controladores DMA permita realizar transferencias de datos de hasta 32 bits en paralelo e independientes a la actividad de la CPU. El DSP C5535, cuenta con una memoria interna de 320 KB dividida en un bloque DARAM y otro SARAM, de 64 KB y 256 KB respectivamente.

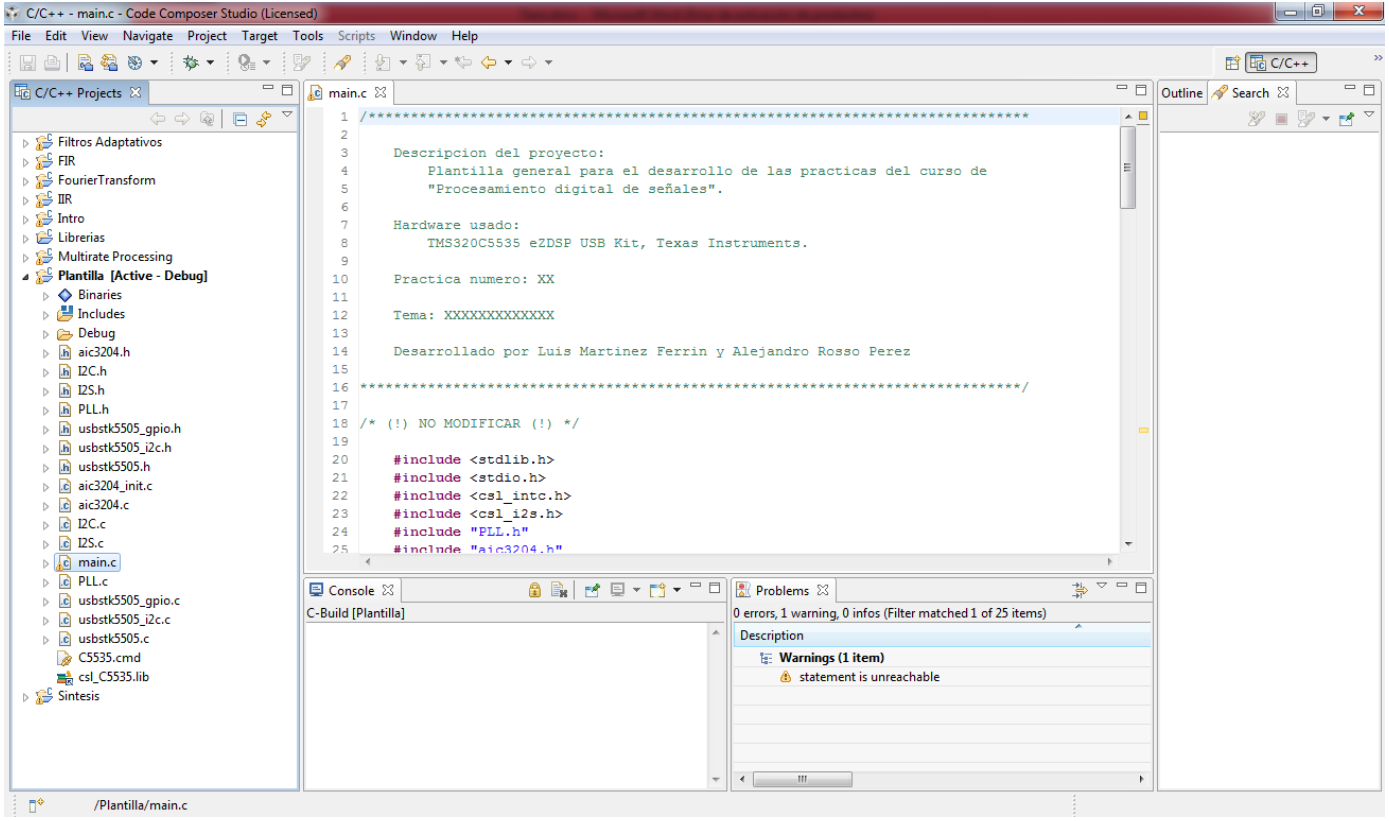


Figura 2. Entorno de desarrollo

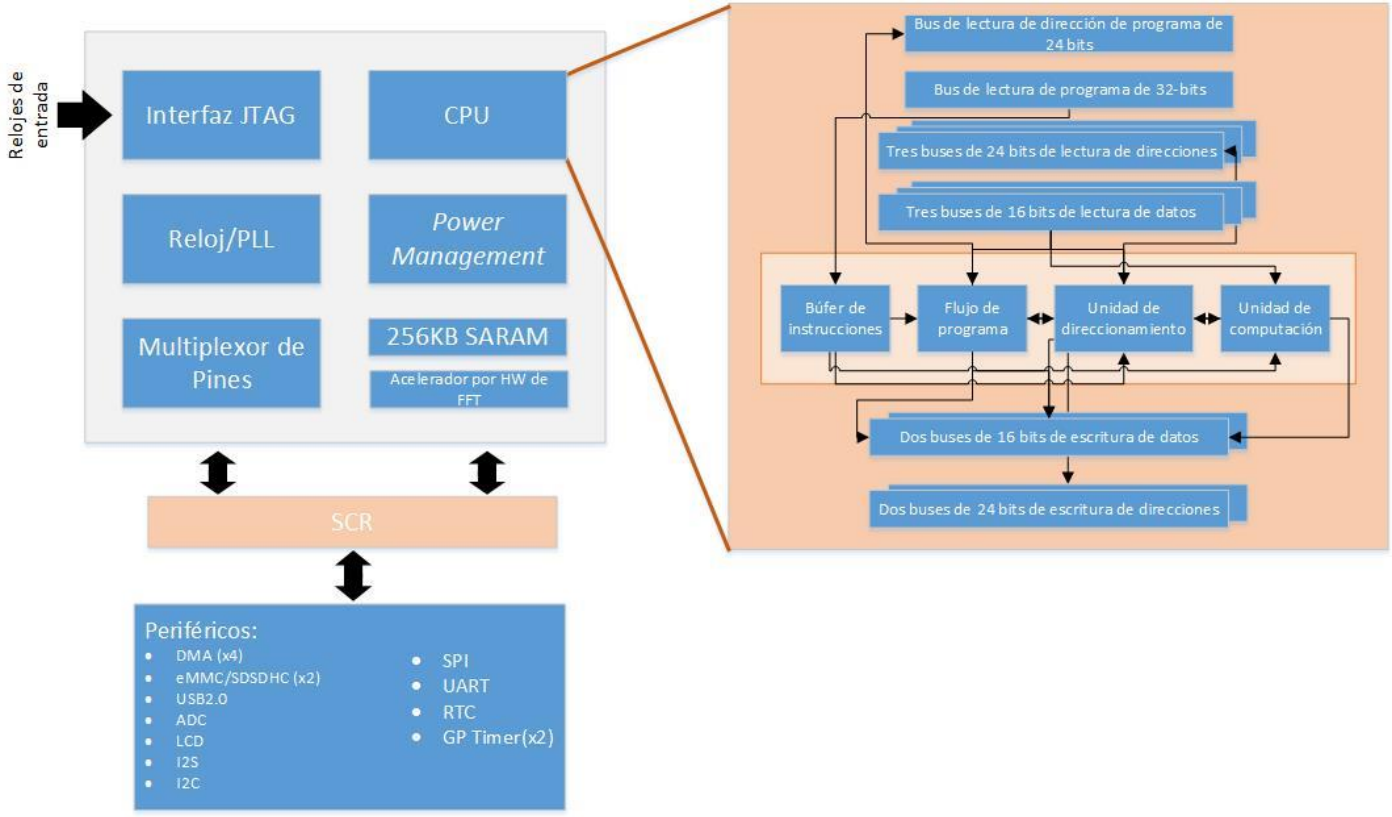


Figura 3. Arquitectura del DSP adquirido

2. BASES TEÓRICAS: SISTEMAS Y SEÑALES DE TIEMPO DISCRETO

Para facilitar el entendimiento de los conceptos que se van a desarrollar en las siguientes secciones es importante revisar primero las bases teóricas que permitan comprender el mundo digital del procesamiento de señales.

2.1. Señales básicas

En la Tabla 2 se relacionan las señales básicas usadas en sistemas y señales con la representación usada en este documento.

Tabla 2. Señales básicas

Señal	Representación
Impulso unitario	$\delta[k]$
Escalón unitario	$\mu[k]$
Rampa unitaria	$r[k]$
Senoidal	$sen[\omega k]$
Cosenoidal	$cos[\omega k]$
Exponencial	$e^{\frac{k}{\tau}}$

2.2. Teorema del muestreo

Lo primero que se debe considerar cuando se trabaja con señales digitales es que estas son representaciones numéricas de señales analógicas, que se obtienen al muestrear estas últimas. Dicho muestreo debe realizarse sobre una señal con ancho de banda finito para garantizar que el procesamiento no genere pérdida de información y debe ser como mínimo a una frecuencia dos veces mayor (frecuencia de Nyquist) a la frecuencia máxima contenida en la señal original.

Dicha afirmación es conocida como el teorema del muestreo y se resume en la inecuación

$$F_S \geq 2 * F_{MAX} .$$

2.3. Cuantización

Para que una señal pueda ser entendida por un hardware digital -como el DSP que se usará en este trabajo- es necesario que sea discreta tanto en el tiempo (muestreo) como en la amplitud (cuantización). Esto último consiste en la asignación de valores binarios a los diferentes niveles de amplitud de la señal para que puedan ser fácilmente procesados.

La cuantización es llevada a cabo por los ADC y el número de bits b usados en la conversión afecta directamente el rango dinámico RD_{ADC} de esta, según la relación

$$RD_{ADC} = 20 * \log_{10}(2^b) \approx 6 * b \; dB .$$

Además, el proceso de cuantización añade un ruido a la señal -conocido como ruido de cuantización- que afecta la relación señal a ruido de la siguiente manera:

$$SN_{ADC} = (1.76 + RD_{ADC}) \; dB .$$

2.4. *Ecuaciones en diferencias*

Las operaciones básicas que se realizan sobre las señales de tiempo discreto son: la suma, la multiplicación por constantes y el retardo. A partir de estas se construyen ecuaciones que representan matemáticamente sistemas lineales, como algunos de los filtros y efectos que se trabajan más adelante.

Todas las ecuaciones en diferencias tienen la forma

$$y[k] = \sum_{n=0}^{N-1} b_n * x[k-n] - \sum_{m=1}^{M-1} a_m * y[k-m],$$

en la que los coeficientes a_n y b_n representan constantes en los sistemas LTI, o funciones en términos de k en sistemas más complejos. Y $x[k]$ y $y[k]$ representan secuencias discretas de entrada y salida, respectivamente.

Ejemplo 1. En la Figura 4 se observa una señal $x[k] = \{ 0, 2, 1, 0, 2, 1, 0 \}$, sobre la cual se aplica la ecuación en diferencias

$$y[k] = x[k] - 2 * x[k-1].$$

La señal resultante $y[k]$ se observa en la Figura 5.

2.5. *Función de transferencia*

Otra forma de representar sistemas LTI de tiempo discreto es mediante su función de transferencia. Esta permite conocer la respuesta en frecuencia de los sistemas de una forma explícita que puede obtenerse directamente de la ecuación en diferencias (aceptando que $z^{-n} * x[k] = x[k-n]$) de la siguiente manera:

$$H(z) = \frac{\sum_{n=0}^{N-1} b_n * z^{-n}}{1 + \sum_{m=1}^{M-1} a_m * z^{-m}}.$$

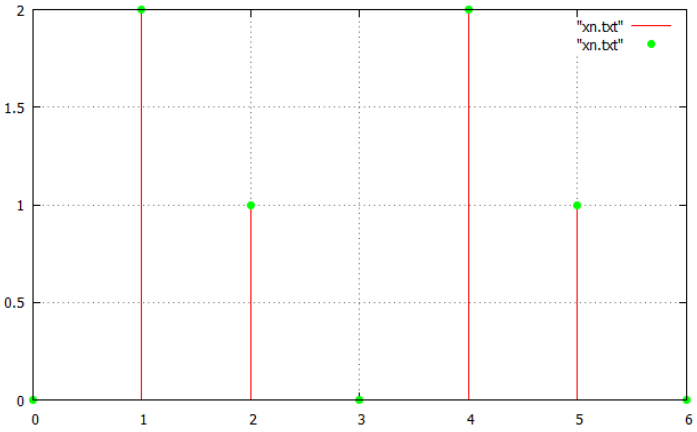


Figura 4. Gráfica de la señal $x[k]$ para el Ejemplo 1

2.6. *Variables de estado*

Las características de un sistema también pueden ser entendidas como las interacciones que se dan al interior de este, en relación a las entradas y salidas del mismo. Dichas relaciones se pueden representar mediante las variables de estado. Esto permite modelar y analizar sistemas LTI y también aquellos sistemas que son no-lineales y variantes en el tiempo.

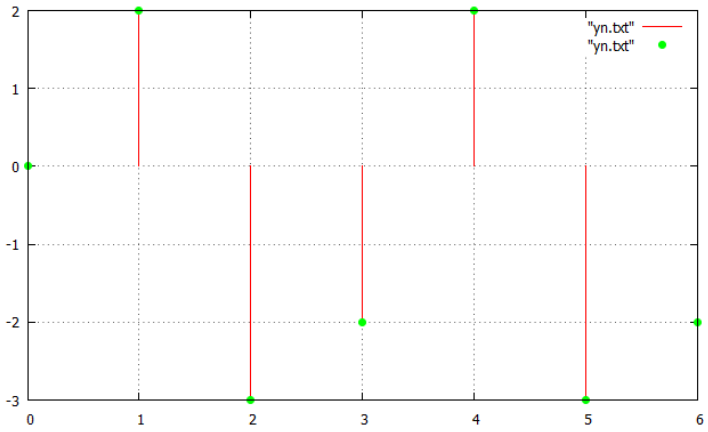


Figura 5. Gráfica de la señal $y[k]$ para el Ejemplo 1

2.7. *Diagramas de bloques*

Finalmente, para tener una representación gráfica y facilitar el entendimiento de los procesos internos de un sistema (como realimentaciones, retardos, etc.) y como estos modifican las señales, se tienen los diagramas de bloques.

En la Figura 6 se pueden observar los elementos más importantes de dichos diagramas, como son: sumadores, multiplicadores, retardos, funciones de transferencia y líneas de flujo. Además la variable $\dot{v}[k]$ representa una variable de estado del sistema.

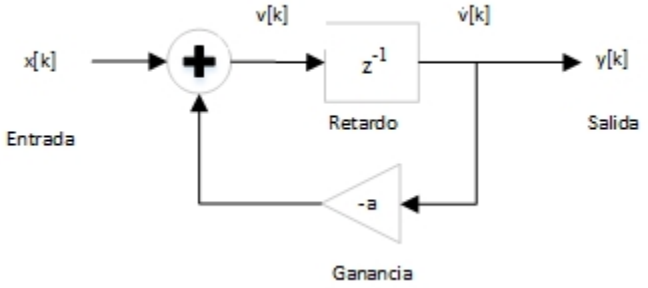


Figura 6. Diagrama de bloques ilustrativo

2.8. *Transformada discreta de Fourier*

La transformada de Fourier es una herramienta matemática que permite analizar una señal en términos de su contenido de frecuencias. Para el caso de las señales de tiempo discreto esta herramienta recibe el nombre de "transformada discreta" y es aplicada sobre señales periódicas o de duración finita (Lathi, 1998).

La ecuación que describe el comportamiento espectral de una señal $x[k]$ con longitud N, en términos de dicha transformada se da como

$$X[n] = \sum_{k=0}^{N-1} x[k] * e^{-\frac{2\pi nk}{N}},$$

donde la función $X[n]$ representa la característica espectral de la señal y es de igual longitud que esta.

De manera análoga, una señal puede reconstruirse a partir de su espectro usando la expresión

$$x[k] = \frac{1}{N} \sum_{n=0}^{N-1} X[n] * e^{\frac{2\pi nk}{N}} .$$

Esta ecuación se conoce como transformada discreta de Fourier inversa o IDFT por sus siglas en inglés.

2.9. Transformada rápida de Fourier

La FFT es una manera de calcular eficientemente la DFT o la DFT inversa de una señal. Aunque existen diversos algoritmos para distintos casos de acuerdo a la longitud de la señal, uno de los más conocidos para implementar la FFT (Cooley-Tukey), requiere que la longitud de la señal sea una potencia natural de 2. Así, se puede obtener esta transformada al separar la señal original en otras más pequeñas –de menor cantidad de muestras-, cuya DFT se puede computar más rápido.

Las ventajas computacionales de este algoritmo se evidencian en la disminución del número de operaciones requeridas para obtener la DFT. Como se muestra en la Figura 7, dicho número disminuye (para una longitud $N > 2$) de N^2 operaciones a $N \log_2 N$. Se puede inferir fácilmente que para una longitud de solo 32 muestras el algoritmo de Cooley-Tukey logra un factor de ahorro del 640%, aproximadamente.

2.10. Convolución

Teóricamente, la convolución es un operador matemático que se aplica a dos señales $f[k]$ y $g[k]$ y que se obtiene como

$$l[k] = \sum_{n=-\infty}^{\infty} f[n] * g[k-n] .$$

Simbólicamente se expresa como $l[k] = f[k] \star g[k]$ y es usada para describir de cierta forma las similitudes entre las señales relacionadas –una de ellas estando invertida en el tiempo-.

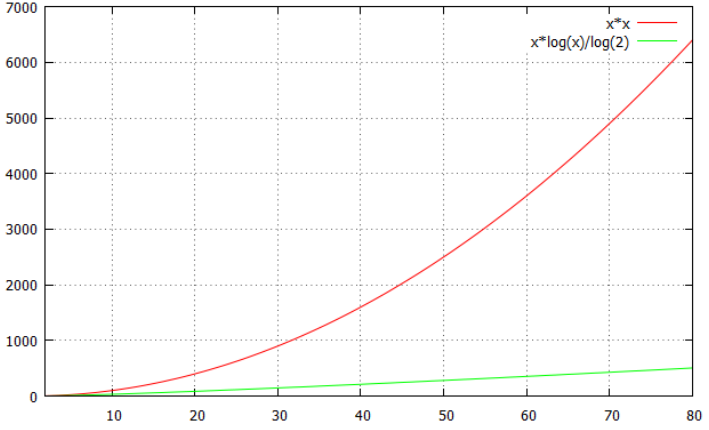


Figura 7. Costo computacional del algoritmo Cooley-Tukey

En el procesamiento digital de señales el uso de este operador es de gran importancia ya que permite conocer a partir de la respuesta al impulso de un sistema $h[k]$, la salida ante cualquier señal $f[k]$, como la convolución entre estas dos señales. O simplemente $f[k] \star h[k]$ (Proakis & Manolakis, 1996).

Este concepto puede ser aplicado al procesamiento de audio, bien sea para implementar un filtro FIR o para agregar el efecto de ambiencia de un espacio determinado a una pista de audio.

Aparte de esto, muchas veces la convolución entre señales de gran longitud puede resultar en un elevado costo computacional, lo que puede generar retardos indeseados en el procesamiento de tiempo real. Para superar este inconveniente se puede aprovechar una propiedad muy importante de la transformada de Fourier que asocia la convolución en el dominio del tiempo con una multiplicación en el dominio de la frecuencia y viceversa. De la siguiente forma:

$$f[k] \star g[k] \leftrightarrow \mathcal{F}\{f[k]\} * \mathcal{F}\{g[k]\}$$

y

$$f[k] * g[k] \leftrightarrow \frac{1}{2\pi}\mathcal{F}\{f[k]\} \star \mathcal{F}\{g[k]\} .$$

3. BASES TEÓRICAS: LENGUAJE C PARA EL PROCESAMIENTO DIGITAL DE SEÑALES

Como ya se mencionó en la primera sección de este trabajo, la programación del hardware adquirido se realizará usando el lenguaje de programación C. Se ha escogido este lenguaje –y no otros de los que soporta el software de desarrollo- ya que su sintaxis es fácil de comprender y a la vez permite generar un código suficientemente eficiente para hacer procesamiento en tiempo real gracias a las características de bajo nivel que posee. Además, C es uno de los lenguajes de programación más empleado en ciencias e ingenierías, quizá, debido a su facilidad de portabilidad a diferentes plataformas (Embree, 1995).

3.1. Palabras reservadas

En C se tienen ciertas palabras que representan operaciones o funciones propias del lenguaje y que no pueden ser usadas por el programador para fines diferentes a los ya establecidos. Estas son: *auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while*.

3.2. Tipos de datos

Para poder desarrollar un programa en C y en general en cualquier lenguaje de programación, es necesario tener elementos en los cuales se pueda almacenar información y sobre los cuales se realicen operaciones. Cada uno de los tipos de datos y la respectiva cantidad de información que pueden almacenar, se encuentran detallados en la Tabla 3.

3.3. Casting

El operador *casting*, sirve para convertir un tipo de dato en otro diferente. Así se aprovecha de mejor manera la memoria y se evitan problemas de compatibilidad entre diferentes variables al realizar operaciones. Es necesario tener cuidado con este operador al relacionar elementos de diferente tamaño ya que esto puede provocar perdida de información.

Tabla 3. Tipos de datos en C

Tipo de dato	Tamaño en bits	Rango signado	Rango no-signado
char	8	-128, 127	0, 255
short	16	-32768, 32767	0, 65535
int	16	-32768, 32767	0, 65535
long	32	±2.1474e9	0, 4.2949e9
float	32	±1.0e±38	-
double	64	±1.0e±306	-

La sintaxis de este operador es la siguiente:

```
VariableNueva =
    (NuevoTipoDeDato)VariableOriginal;
```

Ejemplo 2. Las siguientes líneas muestran el uso básico del operador *casting* en C:

```
short x;
long y = 100;
x = (short) y;
```

3.4. Estructuras

Muchas veces, al desarrollar programas de mayor complejidad es necesario crear elementos nuevos que puedan almacenar más información y que puedan ser accedidos de una manera simple y organizada. Para esto existen en C elementos llamados estructuras, cuya sintaxis es:

```
struct NombreDeLaEstructura
{
      tipo1 variable1;
      tipo2 variable2;
      ...
      tipoN variableN;
};
```

Agregar este código en la cabecera de un programa, permite que la variable tipo estructura se pueda usar desde cualquier función, como:

```
struct NombreDeLaEstructura

      NombreDeLaVariable;
```

A veces esta declaración resulta un poco engorrosa. Por ello se prefiere el uso de la siguiente sintaxis que permite simplificar un poco las cosas:

```
typedef struct
{
      tipo1 variable1;
      tipo2 variable2;
      ...
      tipoN variableN;
}NuevoTipoDeDato;
```

Y la declaración de la variable se reduce a:

```
NuevoTipoDeDato NombreDeLaVariable;
```

3.5. Punteros

Un puntero es un tipo de variable que sirve para almacenar la dirección de memoria de otra variable o de algún segmento del código. Estos elementos son muy importantes para acceder a los espacios de memoria de una manera eficiente y ordenada y son de gran utilidad en la implementación de buffers en el procesamiento digital de señales.

Así como las variables tienen un tipo de dato asociado, los punteros también deben ser declarados dependiendo del tipo de

dato que se quiera referenciar. Estos se declaran de la siguiente manera:

```
TipoDeDato variable1, variable2;
TipoDeDato*  puntero1;  //Declaración  del
                        //puntero1
TipoDeDato  *puntero2;  //Declaración  del
                        //puntero2
puntero1 = &variable1; //Operador
                       //dirección
variable2 = *puntero1; //Operador
                       //indirección
```

De las líneas anteriores es importante explicar los siguientes conceptos:

- Para poder usar un puntero hay que definirlo primero como tal. Para hacer esto se usa el caracter `*` precedido del tipo de dato o antepuesto al nombre del puntero
- El operador dirección `&`, sirve para obtener la dirección de memoria en la que se halla la variable que precede
- El operador indirección `*`, sirve para devolver el valor que se encuentra almacenado en la variable a la cual apunta un puntero

3.6. Arreglos

Un arreglo es un tipo de variable que puede almacenar varios elementos del mismo tipo. Es decir, un bloque de memoria en el que los elementos son contiguos y de igual tamaño.

Estos se definen e inicializan de la siguiente manera:

```
TipoDeDato arreglo1[]=
        {elemento1,elemento2,…};
TipoDeDato  arreglo2[n];  //  'n'  es  un
// número natural que indica el número de
// elementos del arreglo
arreglo2 [0] = elemento1;
arreglo2 [1] = elemento2;
…
arreglo2 [n-1] = elemento_n;
```

Los punteros y los arreglos están íntimamente relacionados ya que un arreglo es en realidad un puntero que señala la primera posición del bloque de elementos que contiene. Así:

```
int x, *y;
x  =  arreglo1[0];  //  Almacena  en  'x'  el
                   //del valor  elemento 1
y  =  arreglo1;    // Almacena  en  'y'  la
//dirección   del   primer   elemento   del
//arreglo
```

3.7. Elementos para el control de flujo del programa

En C existen sentencias que permiten modificar el comportamiento y orden de ejecución de un programa de acuerdo al resultado de una condición lógica.

La sentencia IF-ELSE, permite ejecutar un bloque de instrucciones si se evalúa como verdadera una condición lógica -*if*-, o por el contrario ejecutar otro bloque si no se cumple dicha condición -*else*-. También es posible evaluar tantas condiciones como se desee; cada una con un bloque diferente asociado -*else if*-. Esta sentencia es excluyente, es decir, si se cumple alguna condición se ejecutará el código respectivo y las demás condiciones no se evaluarán.

La sintaxis de esta sentencia es:

```
if(condición1)
{
        // Bloque 1
}
else if(condición2)
{
        //Bloque 2
}
…
else
{
        //Bloque n
}
```

La sentencia SWITCH, permite seleccionar las instrucciones a ejecutar según el valor que tome una variable.

Su sintaxis es:

```
switch (variable)
{
case valor1:
        // Instrucción_1
        break;
case valor2:
        // Instrucción_2
        break;
…
default:
       // Instrucción_n
        break;
}
```

Al implementar esta sentencia es importante tener en cuenta que la variable y los valores que esta pueda tomar sean del mismo tipo. En caso de no coincidir con ninguno, el programa entrará a ejecutar las instrucciones que se encuentren después de la expresión *default*. Además, cada bloque de instrucciones debe terminar con *break*. De lo contrario se ejecutarán las instrucciones de los casos siguientes.

La sentencia WHILE, permite ejecutar cíclicamente un bloque de instrucciones siempre que se cumpla cierta condición.

La sintaxis de esta sentencia es:

```
while (condición)
{
	// Instrucciones
}
```

Una variación muy usual de este ciclo incluye la sentencia *DO* y permite ejecutar las instrucciones de la misma forma que el bloque *while*, pero garantizando que estas se realicen al menos una vez.

```
do
{
	// Instrucciones
}while(condición);
```

Este tipo de sentencias deben implementarse con cuidado ya que pueden generar *loops* infinitos en el programa.

La sentencia FOR, permite crear ciclos iterativos siempre y cuando se cumpla una condición lógica.

La sintaxis de esta sentencia es la siguiente:

```
for (instrucción1;condición;instrucción2)
{
	// Instrucciones
}
```

La `instrucción1` se ejecuta únicamente al iniciar el ciclo, mientras que la `instrucción2` se ejecuta en cada iteración después de realizar el bloque de código dentro del ciclo, siempre y cuando se valide como verdadera la condición lógica.

Estas últimas sentencias de carácter cíclico permiten el uso de las instrucciones *break* y *continue*, y permiten forzar la salida del ciclo o una nueva iteración del mismo, respectivamente.

La sentencia *for* suele usarse como indica el siguiente ejemplo.

Ejemplo 3: El siguiente bloque de código se ejecuta 10 veces y permite inicializar un arreglo con diferentes valores cada vez. Cuando finaliza la ejecución de dicho código, el arreglo almacena los valores 0, 5, 10, 15,…, 45, y la variable `i` vale 10.

```
int i, arreglo[10];
for (i = 0; i < 10; i++)
{
	arreglo[i] = i*5;
}
```

3.8. Memoria dinámica

En algunos programas puede ocurrir que al momento de compilar no se pueda conocer la cantidad de memoria requerida para realizar alguna tarea, sino que esta dependa de algún dato externo que se obtenga durante la ejecución del mismo.

Para estas situaciones el lenguaje C cuenta con funciones que permiten reservar o eliminar memoria en tiempo de ejecución.

Malloc, calloc y realloc. Estas tres funciones permiten reservar y manipular bloques de memoria del sistema en tiempo de ejecución. Respectivamente sirven para:

- Reservar un bloque de memoria de un tamaño específico:
  `malloc (TamañoEnBytes);`
- Reservar un bloque de memoria de un tamaño específico, pero inicializado en '0':
  `calloc (NúmeroDeElementos, TamañoEnBytes);`
- Modificar el tamaño de un bloque ya reservado:
  ```
  realloc (PunteroDelBloque,
          NuevoTamañoEnBytes);
  ```

Para complementar el uso de estas funciones, la instrucción *free* permite liberar un espacio de memoria previamente reservado con alguna de las funciones ya mostradas. Así:

```
free(PunteroDelBloque);
```

Ejemplo 4. El siguiente ejemplo muestra el uso de las funciones usadas para reservar y eliminar memoria dinámica.

```
int tamaño = 10;
float *arreglo, *arreglo2;
arreglo = (float*)
malloc(tamaño*sizeof(float));
arreglo = (float*) realloc(arreglo,
              2*tamaño*sizeof(float));
arreglo2 = (float*) calloc(3*tamaño,
              sizeof(float));
free(arreglo);
free(arreglo2);
```

Al ejecutar el código anterior (exceptuado las últimas dos líneas) se tiene reservado un espacio en memoria que permite almacenar 20 elementos tipo *float* en la variable `arreglo`. Además se tiene la variable `arreglo2` de 30 posiciones llena con ceros. Finalmente, al ejecutar el código por completo la memoria que se tenía reservada para los arreglos se liberará y podrá utilizarse para almacenar otros elementos.

3.9. Directivas del compilador

Las directivas en C son todas aquellas instrucciones que se escriben generalmente en la cabecera del código. Es decir, entre el comienzo del archivo y la función *main*. Estas permiten escribir el código de forma ordenada, facilitando la legibilidad, reutilización y compilación del código, sobre todo cuando se tienen programas muy extensos o que contienen diferentes archivos fuente.

Las instrucciones que se usan con este fin se distinguen fácilmente del resto del código ya que son precedidas por el caracter '#'.

Include. Esta directiva sirve para incluir al código elementos externos como archivos de cabecera de librerías (archivos ".h") u otros archivos fuente (archivos ".c").

El efecto de esta función en el código es sustituir la línea en la que aparecen por el contenido del archivo que se está indicando.

Define. Esta directiva permite definir constantes o macros al comienzo de un código que mejoran el entendimiento y organización del mismo.

Pragma. Esta directiva permite dar información adicional al compilador sobre secciones particulares del código. Esto ayuda a optimizar el código objeto generado aprovechando ciertas características de la plataforma. El uso de esta directiva debe ser muy cuidadoso ya que afecta considerablemente la portabilidad del código (Texas Instruments Incorporated, 2011).

Otras. Además de las directivas ya mencionadas existen otras declaraciones –menos usuales- que permiten desarrollar sentencias más complejas en esta parte del código. Estas son: `#ifdef`, `#ifndef`, `#endif`, `#undef`, `#else` y `#elif`.

Ejemplo 5. A continuación se presenta el contenido de 2 archivos que tienen código en C. Luego de ellos se encuentran: el código equivalente al utilizar la directiva `#include` y el código equivalente a utilizar la directiva `#define`.

- Archivo "funcion.h":

```
#define MAX(a, b)  (a >= b) ? a : b
#define Constante 100
```

- Archivo "ejemplo.c"

```
#include "funcion.h"
int main (void)
{
       int x = 10, y = 5,  z;
       z  = MAX(x, y); // z vale 10
       return Constante;
//El programa devuelve "100"
}
```

- Resultado 1:

```
#define MAX(a, b)  (a >= b) ? a : b
#define Constante 100
int main (void)
{
       int x = 10, y = 5,  z;
       z  = MAX(x, y); // z vale 10
       return Constante;
//El programa devuelve "100"
}
```

- Resultado 2:

```
int main(void)
{
int x = 10, y = 5,  z;
z  =  (x >= y) ? x : y; // z vale 10
return 100;
//El programa devuelve "100"
}
```

Ejemplo 6. La directiva `#pragma MUST_ITERATE`, propia del compilador del DSP adquirido (incluido en el CCS), permite suministrar información extra sobre las ejecuciones de un bucle específico. A continuación se muestra un posible uso de dicha directiva.

```
int i, j = 0;
#pragma MUST_ITERATE (1)
for(i=0; i < 5; i++)
{
       j += i;
}
```

En este caso, la directiva indica al compilador que el ciclo se va a ejecutar al menos una vez.

Para mayor información sobre el uso de esta y otras directivas "PRAGMA", se sugiere revisar el manual "*TMS320C55x DSP Programmer's Guide"* de Texas Instruments (Texas Instruments Incorporated, 2001).

4. CONSIDERACIONES GENERALES PARA EL DESARROLLO DE LAS PRÁCTICAS

Para poder utilizar el kit de desarrollo TMS320C5535 eZdsp™ es necesario tener en cuenta los aspectos básicos que se listan a continuación.

4.1. Instalación del Code Composer Studio

El CCS es el entorno de desarrollo que permite escribir, compilar, programar y depurar los proyectos que se quieran realizar en los DSP de Texas Instruments. Por esto, es fundamental su instalación para poder desarrollar cualquier programa. El CCS se puede encontrar en el DVD incluido en el kit de desarrollo, o puede ser descargado gratuitamente (en su más reciente versión) de la página de Texas Instruments.

Para instalar este programa, se ejecuta el archivo de instalación y se selecciona el paquete "Platinum Edition" dejando las demás opciones por defecto. Una vez instalado el CCS y al ejecutar por primera vez el programa se debe suministrar una dirección para guardar los proyectos. Se sugiere la ubicación "C:\CCS_WorkSpace". En este directorio se deben copiar las carpetas "inc", "src" y "lib" presentes en la dirección "Codigos\ArchivosComunes" del DVD anexo. Además, antes de usar el programa se requiere activar el software mediante una licencia que podrá ser generada a través de la opción "Activate a License/Use Free Limited License".

Finalmente se debe indicar al depurador el tipo de dispositivo que se va a programar. Para configurar dicha característica se debe escoger la opción "New Target Configuration File" del menú "Target". Una ventana cómo la mostrada en la Figura 8 aparecerá y en ella se deberá marcar la opción presente e indicar un nuevo nombre para la configuración (se sugiere: C5535DSP).

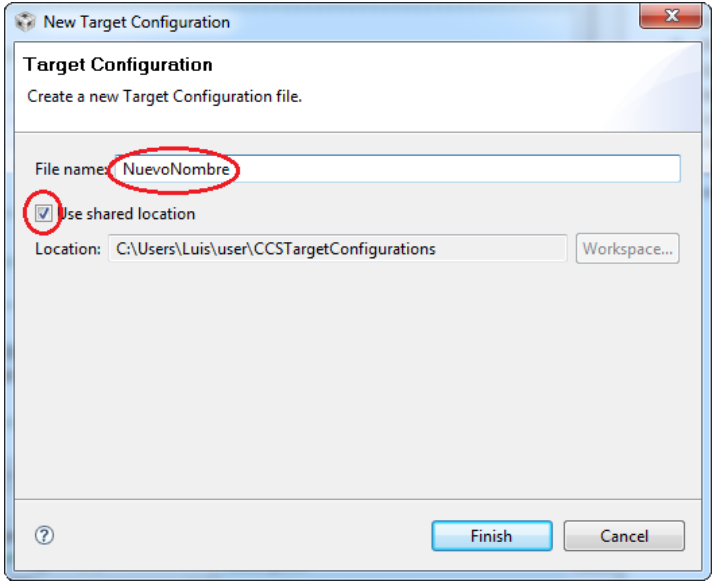


Figura 8. Configuración de un dispositivo nuevo

Luego de hacer clic en el botón “Finish”, una ventana se abrirá en el panel principal del CCS. En esta se deben llenar las opciones: “Connection” con la opción “Texas Instruments XDS100v2 USB Emulator” y “Device” con la opción “EZDSP”. Posteriormente se debe hacer clic en el botón “Save”, cómo se muestra en la Figura 9.

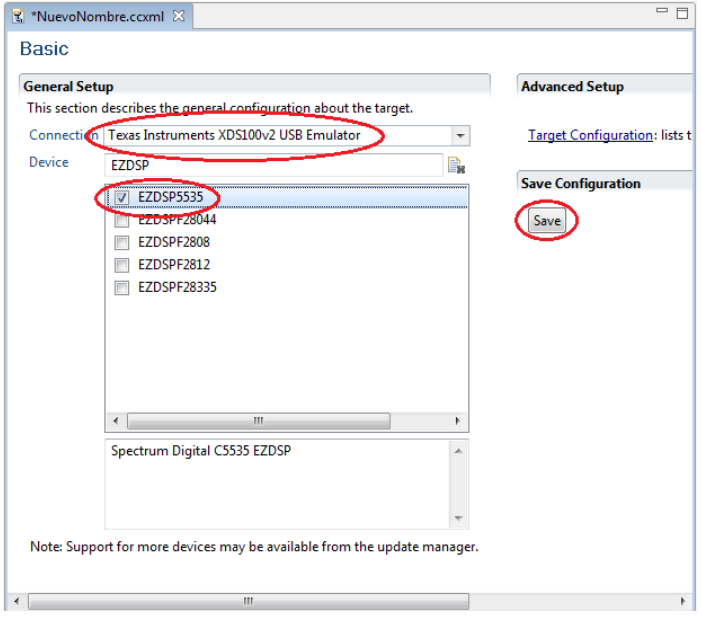


Figura 9. Configuración del archivo ".ccxml"

4.2. *Proyecto nuevo*

Una vez abierto el CCS es posible crear un proyecto nuevo a través del menú contextual “File/New/CCS Project”. Luego de dar el nombre al proyecto se elige en la lista desplegable de “Project Type” la opción C5000, dejando las demás opciones por defecto. Se pueden omitir las demás páginas de este asistente, exceptuando la titulada “Project Settings”. Esta debe ser configurada como se muestra en la Figura 10.

Luego de que se ha creado el proyecto se debe acceder a las propiedades del mismo para modificar las opciones de la pestaña “Runtime Model Options”, tal cómo se muestra en la Figura 11. En esta misma ventana en la opción “Include Options” se debe añadir la carpeta “inc”, contenida en el directorio del *workspace*, tal como se muestra en la Figura 12.

Una vez se han actualizado dichas propiedades se deben añadir los archivos “.lib”, relacionados con las librerías necesarias en cada proyecto. Para esto se debe hacer clic derecho en el proyecto y al seleccionar la opción “Link Files”, se debe agregar la librería CSL (*Chip Support Library*), presente en el archivo “csl_C5535.lib”. Eventualmente, algunos proyectos requerirán además la librería “55xdsph_r3.lib”. Estos archivos se pueden encontrar en la carpeta “lib” del *workspace*.

Finalmente, para facilitar el proceso de aprendizaje se ha creado una plantilla que permite escribir los códigos de los algoritmos sin preocuparse por la configuración previa del hardware (frecuencia

del reloj, I2S, I2C, frecuencia del códec, entre otros). Dichos archivos pueden encontrarse en el directorio "Codigos/PlantillaDSP" del DVD anexo y su contenido debe ser copiado al nuevo proyecto.

Para ejecutar un programa se debe hacer clic, primero en la opción "Debug Active Project" de la pestaña "Target" y posteriormente en la ventana del depurador en el ícono de *play*.

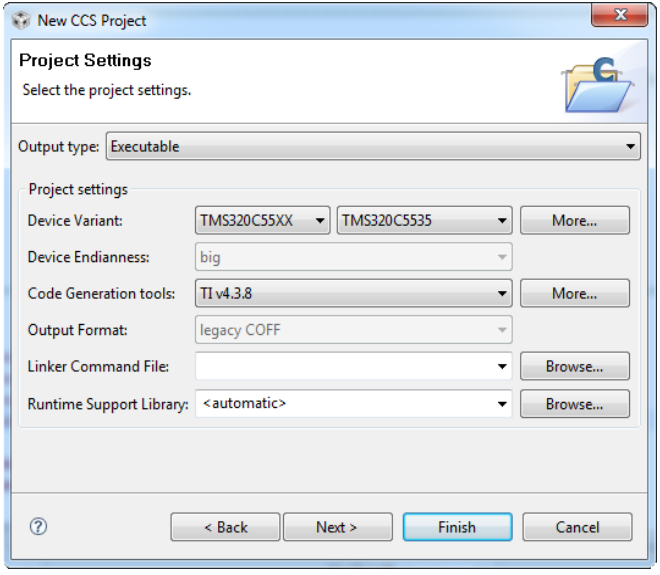


Figura 10. Configuración de un proyecto nuevo

Al ir desarrollando nuevas prácticas los proyectos se pueden crear a partir de esta plantilla modificando el archivo "main.c" y añadiendo los archivos adicionales ".h" y ".c" y librerías que sean requeridas.

Las instrucciones mencionadas deberían permitir con los archivos de la plantilla, la ejecución de un programa de prueba que usa los audífonos y el micrófono del kit permitiendo a modo de *bypass* reproducir en la salida la misma señal de entrada.

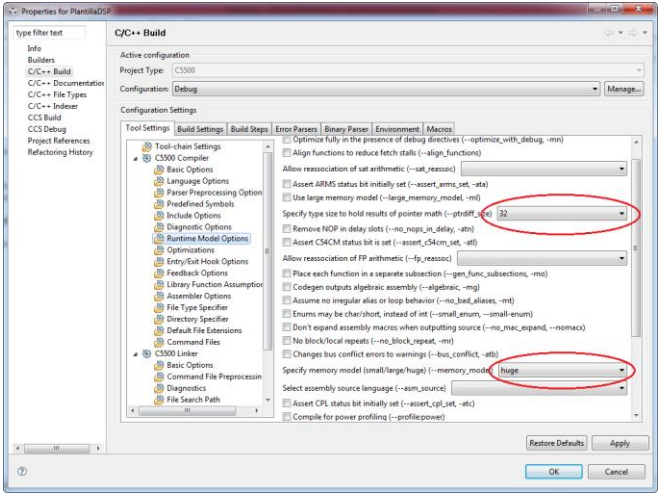


Figura 11. Configuración de las propiedades del proyecto

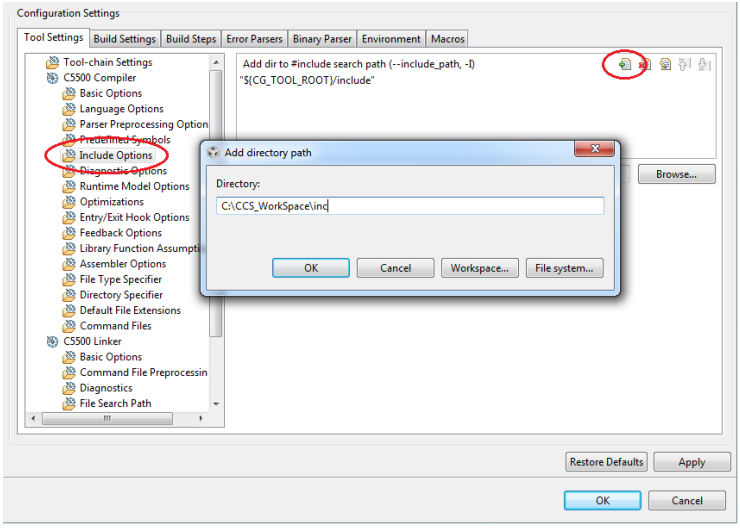


Figura 12. "Include Options"

4.3. SD Boot

El kit de desarrollo permite cargar los programas directamente desde la SD evitando así el uso del depurador del CCS. Esta característica es de gran utilidad para aplicaciones autónomas de bajo consumo, pues elimina la necesidad de un computador para la ejecución de los programas desarrollados.

Para poder realizar el *boot* de un programa desde la SD se deben seguir los siguientes pasos:

- Copiar la carpeta "SD Boot" contenida en el directorio "Otros" del DVD anexo, al computador. Se recomienda usar un directorio de la unidad de almacenamiento principal como "C:\SD Boot"
- Generar el archivo ".out" del programa con el CCS, usando la herramienta "Build Active Project". Este archivo se encuentra en la carpeta del proyecto en el subdirectorio "Debug" o "Release", según la opción elegida
- Copiar el archivo ".out" a la carpeta escogida en el primer paso
- Editar el archivo "demo.cmd" de esta carpeta de modo que en la sexta línea de este se indique el mismo nombre del archivo ".out" generado
- Abrir la ventana de comandos (terminal) de su sistema operativo –cmd en Windows-. Usando los comandos requeridos cambiar al directorio elegido en el primer paso y una vez allí ejecutar el comando: hex55 demo.cmd
- El paso anterior genera el archivo "bootimg_.bin" en la misma carpeta
- Copiar el archivo "bootimg_.bin" al directorio raíz de la SD

- Poner la SD en el kit de desarrollo y encender

4.4. Manejo de cantidades en punto fijo

En sistemas digitales existen dos formas de representar las cantidades: punto fijo y punto flotante. Dicha característica afecta drásticamente el precio, velocidad, consumo energético y facilidad de programación de un procesador. Sin embargo, es la aplicación y condiciones específicas de un sistema las que definen cuál de estos dos formatos es más apropiado.

Las arquitecturas de punto flotante, a pesar de ser más lentas y costosas, tienen un mayor rango dinámico y son mucho más fáciles de programar, ya que la representación numérica es más "transparente" al programador y similar a la forma en que se manejan las cantidades cotidianamente. En oposición, las arquitecturas de punto fijo son mucho más económicas y rápidas, pero requieren un mayor cuidado en el manejo y operación de las cantidades, ya que la representación numérica es responsabilidad del programador.

Formato Qm.n. Debido a esta "libertad" en la representación numérica de los sistemas de punto fijo, pueden implementarse diversas formas de interpretación numérica de los valores binarios pues a diferencia de su contraparte de punto flotante, en el hardware no hay elementos encargados de manipular el punto decimal. Así, en un registro de L bits se pueden usar arbitrariamente m bits para representar la parte entera, n bits para la parte fraccionaria y opcionalmente, un bit para el signo. Siendo

$$L = m + n + 1 .$$

En los sistemas de DSP de 16 bits es bastante común usar los 15 bits menos significativos para representar la parte fraccionaria y el bit de mayor peso para representar el signo. De esta manera pueden representarse valores numéricos entre -1 y $1 - 2^{-15}$ con una resolución de 2^{-15} unidades. Esta representación se conoce como formato Q15.

En la Figura 13 pueden observarse los pesos de cada bit en dicho formato.

Para convertir un número binario Q15 a decimal, basta sumar los pesos de los bits correspondientes. Y para conocer la representación binaria de un número decimal, basta con multiplicar dicho número por 2^{-15} y convertir el valor entero a binario.

Ejemplo 7 ¿A qué número decimal X corresponde el valor 1000110010011001 en formato Q15?

$$X = -20 + 2-4 + 2-5 + 2-8 + 2-11 + 2-12 + 2-15 = -0.9015808105$$

¿Y cuál es la representación binaria b en formato Q15 del número decimal 0.42136?

$$0.42136 \rightarrow 0.42136*2^{15} \rightarrow 13807.124 \rightarrow 13807$$

$$b = 0011010111101111$$

Luego de revisar el ejemplo se puede ver que este tipo de conversión afecta la resolución de los valores representados, pues estos se deben redondear al entero más cercano para poder ser almacenados en los registros de memoria. Esto puede afectar dramáticamente el comportamiento de algunos de los algoritmos implementados.

Finalmente, una de las ventajas más importantes de trabajar con números fraccionarios es que se evita el desbordamiento por multiplicación ya que al multiplicar cualquier par de números que se encuentren en el intervalo mencionado, la magnitud del resultado será menor o igual que 1. Sin embargo, aún existe la posibilidad del desbordamiento por la suma y por ello los fabricantes de hardware proveen las unidades MAC con los llamados *guard bits*, de modo que sea posible almacenar en estos cantidades cuya magnitud sea mayor que 1, dejando al programador la responsabilidad de corregir el desbordamiento.

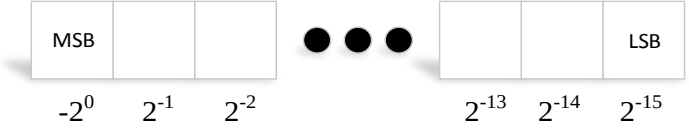


Figura 13. Peso de bits en formato Q15

5. FAMILIARIZACIÓN CON LAS HERRAMIENTAS DE TRABAJO

Con la finalidad de desarrollar satisfactoriamente las prácticas descritas en las secciones siguientes se propone una sesión introductoria que permita al estudiante conocer las funcionalidades básicas del entorno de desarrollo.

En el archivo "Practica1.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

5.1. Objetivos

- Conocer las herramientas básicas del depurador del CCS
- Escribir un programa que permita visualizar una señal senoidal usando la herramienta graficadora
- Implementar las operaciones aritméticas básicas en formato Q15 y corroborar los resultados de estas en el depurador con el uso de *breakpoints*
- Aprender a medir el tiempo de ejecución de una sección del programa

5.2. Conceptos previos

Para poder aplicar los conceptos teóricos vistos anteriormente a un programa escrito en lenguaje C es importante tener diversas consideraciones en el manejo de variables y en la forma de realizar las operaciones entre ellas. Aspectos como: tamaños, tipos de operaciones, rangos y valores máximos, deben tenerse presentes a la hora de escribir el programa pues un código descuidado puede generar resultados incorrectos y comportamientos inesperados y erráticos.

Operaciones aritméticas usando el TMS320C5535. Las operaciones aritméticas más comunes en procesamiento digital de señales son la suma y la multiplicación. Por ello para generar un código eficiente que explote al máximo las capacidades del DSP, estas deben escribirse de una manera especial. En la Tabla 4 se muestran algunas formas óptimas para realizar estas operaciones entre variables de diferentes tamaños. Además en la Tabla 5 puede observarse el tamaño de cada tipo de dato usado en el dispositivo.

Por lo general se prefieren usar variables tipo *short* para realizar bucles, conteos, generación de banderas o para almacenar datos en general y variables tipo *long* para almacenar los resultados de las unidades MAC. Esto resulta en un código más simple y por tanto en mayor eficiencia de ejecución.

Se debe tener especial cuidado en el manejo de la coma decimal ya que la multiplicación de valores fraccionarios afecta la posición de esta y cómo el hardware usado es de punto fijo este no realiza la interpretación de dicho cambio y dicha labor se ve relegada al programador.

Tabla 4. Operaciones básicas

Tipo de operación	Código en C
16bit * 16bit → 32bit	int a, b; long c; c = (long) a*b;
32bit ± 16bit * 16bit → 32bit	int a, b; long c; c = c ± ((long) a*b);
16bit ± 16bit → 16bit	int a, b, c; c = a ± b;
32bit ± 32bit → 32bit	long d, e, f; f = d ± e;

16bit * 16bit → 16bit	int a, b, c;
	long d;
	d = (long) a * b;
	c = (int) (d >> 15);

En el archivo "main.c" presente en el directorio "Codigos/Practica1" se encuentra un código funcional e ilustrativo que al ser programado en el DSP permite interiorizar las características del CCS que se describen a continuación.

El depurador del CCS cuenta con diversas herramientas que permiten verificar el comportamiento de un programa con mayor detalle, brindando la posibilidad de analizar y ejecutar el código línea a línea e incluso instrucción por instrucción.

Controles. Todo depurador debe contar con comandos básicos que permitan controlar la ejecución del programa. En el CCS, dichos controles se encuentran en la parte superior de la ventana "Debug". Algunos de estos son:

Run: Esta opción permite comenzar o reanudar la ejecución del programa hasta el siguiente *breakpoin*t o hasta el final del mismo. La opción "Free Run" realiza la misma tarea pero omitiendo cualquier pausa

Halt: Permite pausar el programa

Terminate All: Esta opción termina el programa completamente y cierra automáticamente la ventana de depuración

Step Over: Ejecuta el código de a una línea cada vez

Step Into: Permite ingresar a una función para ejecutarla línea por línea

Step return: Ejecuta una función libremente hasta el final

Tabla 5. Tamaño de variables en el C5535

Tipo de variable	Tamaño en bits
char	16
int	16
short	16
long	32
long long	40
float	32
double	64

Lectura de variables. Otra característica fundamental de un buen depurador es la posibilidad de brindar una herramienta que permita revisar el comportamiento de las variables a lo largo de la ejecución del programa. En el CCS existen dos ventanas fundamentales que permiten realizar esta tarea: *Local y Watch*. Estas dos ventanas permiten conocer el valor almacenado en alguna variable, la dirección de memoria en la que se encuentra almacenada y el tipo de dato de la misma.

La ventana *Local*, que se activa por defecto, muestra automáticamente todas las variables locales que se han declarado al interior de una función que se encuentre en ejecución. La segunda ventana, *Watch*, debe ser activada a través de la opción "Show View" en la pestaña "Window" y permite monitorear cualquier variable que se halla declarado. Para ello, se debe hacer clic en "<new>" y escribir el nombre de la variable de interés.

Breakpoints. Son elementos externos al código y propios de cada entorno de desarrollo. Ellos representan una pausa en la ejecución de un programa en una línea de código específica. Esto permite visualizar el estado de las variables y registros del DSP en el instante en que se alcanza dicha interrupción. Para crear o eliminar un *breakpoint* basta con hacer doble-clic sobre el número de la línea de código deseada.

Manejo del graficador del CCS. El Code Composer Studio cuenta con una herramienta graficadora que permite visualizar datos almacenados en la memoria del DSP durante la ejecución de un programa. Para utilizar óptimamente este recurso se suelen usar *breakpoints* en los puntos de interés.

Durante la depuración del programa es posible abrir la ventana del graficador en la opción "Graph" del menú "Tools" seleccionando el tipo de gráfico que se desea realizar (para esta práctica se seleccionará el tipo "Single Time"). Al seleccionar el tipo de gráfica se abre una ventana como la mostrada en la Figura 14 en la cual se pueden configurar los parámetros deseados.

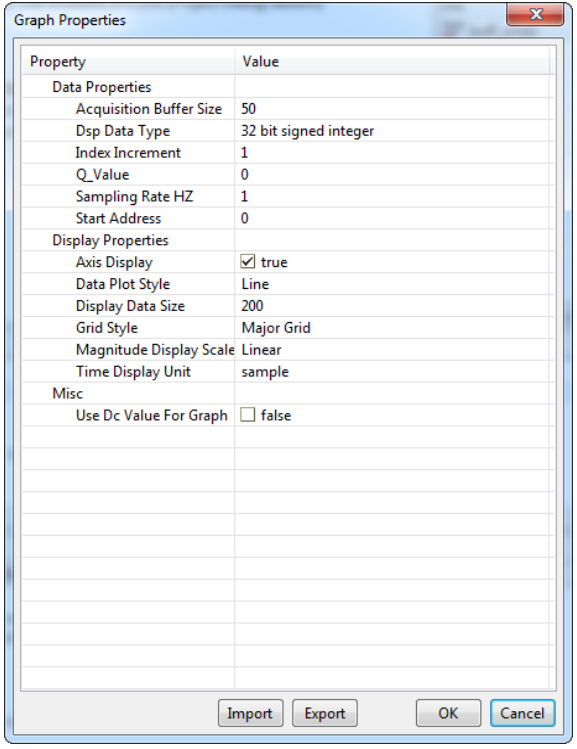


Figura 14. Propiedades del graficador

Para poder graficar es importante conocer la cantidad y tipo de datos que se desean graficar. Además, es necesario que dichos datos se hallen en posiciones consecutivas de memoria. Esto puede ser garantizado usando un arreglo. Con esta información, se llenan los siguientes campos:

- *Acquisition Buffer Size*: Longitud del arreglo de datos
- *Dsp Data Type*: Tipo de datos
- *Start Adress*: Nombre o dirección del arreglo
- *Sampling Rate HZ*: Frecuencia de muestreo de la señal (opcional)

Medida de los ciclos de reloj. Con el CCS es posible realizar un conteo de los ciclos de máquina que toma ejecutar una sección del programa. Para ello es necesario habilitar la herramienta "Clock" del menú "Target". Esto activa un ícono en forma de reloj en la barra inferior de la ventana de depuración del CCS.

Para medir los ciclos de reloj de una parte del código deben establecerse 2 *breakpoints* que delimiten el inicio y el final de la sección a analizar, luego se debe correr el programa hasta alcanzar el primero de los *breakpoints*, hacer doble-clic sobre el reloj para resetearlo y continuar la ejecución hasta el siguiente punto de

control. Así, el número de ciclos aparecerá en el sitio ya indicado junto al ícono del reloj.

5.3. Resultados

Al programar el archivo "main.c" y aplicar los conceptos descritos en el numeral anterior puede graficarse fácilmente la señal contenida en la variable "senoidal". Si se escogen los parámetros correctos, dicha gráfica debería ser como la mostrada en la Figura 15.

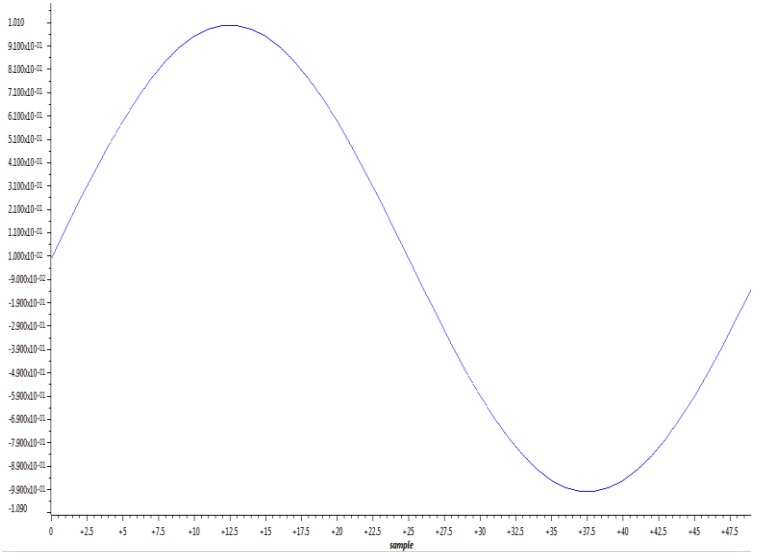


Figura 15. Señal senoidal graficada con el CCS

6. SÍNTESIS DE SEÑALES BÁSICAS

La síntesis es una herramienta de procesamiento que permite la creación de diversas señales directamente desde un hardware o un software. Dicha técnica es de gran importancia en los sistemas digitales modernos para generar señales de prueba con alta precisión, usadas para verificar el funcionamiento de los algoritmos antes de que estos sean sometidos a señales reales. Dichas señales permiten también la modulación de diferentes parámetros como retardos, fases, frecuencias y amplitudes, y en sistemas de mayor complejidad permiten la creación de señales de alta riqueza armónica que han transformado el mundo musical.

En esta práctica se trabajará la síntesis aditiva ya que permite generar fácilmente las formas de onda básicas: senoidal, triangular, cuadrada y diente de sierra. Esto permite una mejor enseñanza de los conceptos relacionados al procesamiento digital de señales. Además, se explicarán los fundamentos de las LUT para implementar la síntesis en tiempo real.

En el archivo "Practica2.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

6.1. Objetivos

- Generar las señales básicas mediante síntesis aditiva usando la librería matemática "math.h"
- Generar las señales básicas mediante el uso de *Lookup Tables* (LUT)
- Reconocer de manera auditiva y gráfica los efectos del *aliasing*
- Comparar los tiempos de ejecución de cada tipo de síntesis

6.2. Conceptos previos

El principio de la síntesis aditiva se basa en la suma de señales senoidales de diferentes amplitudes, frecuencias y fases. De este modo, para poder producir las señales básicas ya mencionadas es necesario que se cumplan ciertas relaciones entre los armónicos y las amplitudes de cada señal. En la Tabla 6 se observan dichas relaciones. En ella f_S corresponde a la frecuencia de muestreo, f a la frecuencia fundamental, y n al n-simo armónico de la señal.

Teóricamente para poder recrear alguna de estas señales exactamente es necesario sumar infinitos armónicos. En la práctica esto no es posible debido a la restricción en la frecuencia máxima que impone la tasa de muestreo y a las limitaciones de memoria y de velocidad, finitas en los dispositivos. Esto tiene como resultado una diferencia entre la forma de onda teórica y la generada por este método. Pero ya que el oído humano sólo puede percibir frecuencias entre 20 Hz y 20 kHz, es posible encontrar un número finito de armónicos que resulte en una aproximación bastante acertada de la onda.

Tabla 6. Series geométricas de señales básicas

Señal	Relación armónica
Senoidal	$f(k) = \sin(2\pi fk / f_s)$
Cuadrada	$f(k) = \sum_{n=1}^{\infty} \frac{1}{2n-1} \sin(2\pi fk(2n-1) / f_s)$
Triangular	$f(k) = \sum_{n=1}^{\infty} \frac{-1^{n+1}}{(2n-1)^2} \sin(2\pi fk(2n-1) / f_s)$
Sierra (positiva)	$f(k) = \sum_{n=1}^{\infty} \frac{1}{n} \sin(2\pi fkn / f_s)$
Sierra (negativa)	$f(k) = \sum_{n=1}^{\infty} \frac{-1^{n+1}}{n} \sin(2\pi fkn / f_s)$

Al implementar estas aproximaciones en el DSP se tiene un inconveniente al sumar cierta cantidad de armónicos, pues la amplitud de la señal puede hacerse mayor que la unidad en algunos puntos y dadas las restricciones del formato Q15 es necesario escalar la amplitud para que esta se mantenga en el intervalo [-1, 1). El factor de escala que se deberá aplicar depende entonces de la señal sintetizada y del número de armónicos empleado.

Otro problema surge cuando se intentan sintetizar las señales mencionadas en tiempo real. Debido a las capacidades del DSP y a la cantidad de operaciones que este debe realizar en relación al número de armónicos para encontrar una muestra de la onda, muchas veces resulta impracticable sintetizar las señales con un número aceptable de armónicos. Es allí donde se hace necesario recurrir a otro método de síntesis.

Lookup Table. Una LUT es un arreglo de valores que se suele calcular al inicio de un programa o antes de este y que es almacenado en memoria para reducir los costos computacionales de un proceso que requiera cálculos complejos, ya que estos se sustituyen por una sencilla lectura de datos.

En síntesis como en otras técnicas de procesamiento digital, esto resulta bastante útil, ya que puede emplearse una LUT para almacenar los valores de un ciclo completo de cualquier forma de onda. De este modo se puede variar la frecuencia de la onda generada cambiando la velocidad con la cual se recorre la tabla y aplicando técnicas matemáticas como la interpolación -en caso de ser requeridas- para mejorar la calidad de la señal resultante.

Interpolación. Es una técnica matemática que permite estimar cualquier punto perteneciente a una función, a partir de un conjunto de muestras discretas conocidas. Mediante el uso de diferentes métodos numéricos es posible aproximar los valores presentes entre dos muestras consecutivas empleando una línea cuyos puntos sean fáciles de calcular.

Estas aproximaciones se hacen comúnmente utilizando polinomios de grado m, requiriendo así $m+1$ puntos para realizar la estimación de los puntos de cada intervalo. Un polinomio de grado mayor garantiza una transición suave entre los puntos. Sin embargo, representa una mayor exigencia en el procesamiento y además puede presentar características oscilatorias indeseables. Por ello, no necesariamente un polinomio de orden mayor representa mejor alguna función y su escogencia dependerá de la aplicación específica de dicha técnica. En procesamiento digital de señales se usan generalmente los polinomios de grado 1 (línea recta) y 3. Prefiriendo la línea recta por su bajo costo computacional y simplicidad aritmética.

Para un par de puntos consecutivos x_0 y x_1, con valores y_0 y y_1, puede encontrarse para el punto x_i, perteneciente al intervalo $[x_0, x_1)$, el valor

$$y_i = m * (x_i - x_0) + y_0,$$

donde

$$m = {(y_1 - y_0)}\Big/{(x_1 - x_0)}\ .$$

Esta ecuación puede simplificarse cuando se trabaja con LUT, pues los valores x_i corresponden a un valor entero que representa una posición de la tabla, por lo que $(x_1 - x_0) = 1$. De este modo, la ecuación anterior puede escribirse como:

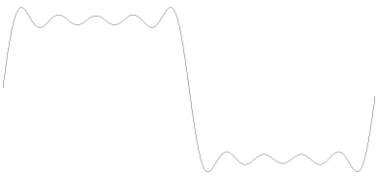
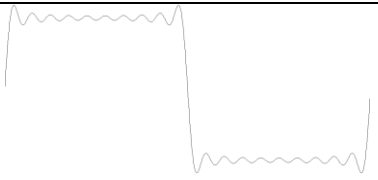
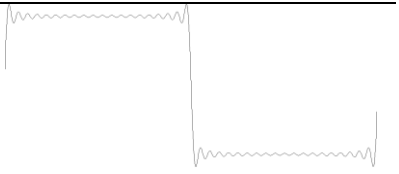
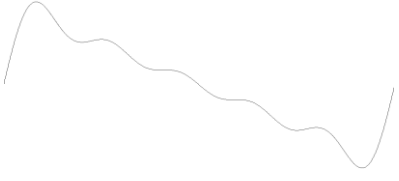
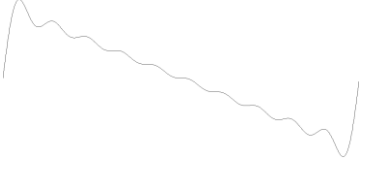
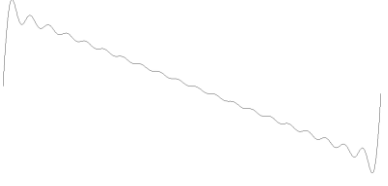
$$y_i = (y_1 - y_0) * (x_i - x_0) + y_0\ .$$

6.3. Resultados

Debido al alto costo computacional de las operaciones en punto flotante, no es posible sumar armónicos a partir del llamado de funciones trigonométricas de la librería “math.h”, ya que para un valor óptimo de frecuencia de muestreo el tiempo de las operaciones excede el periodo entre muestras. Esto impide realizar una buena aproximación de las señales básicas.

En la Tabla 7 puede observarse la forma de onda obtenida al graficar las señales básicas descritas usando diversos números de armónicos.

Tabla 7. Aproximación de las señales básicas

Señal	5 armónicos	10 armónicos	20 armónicos
Cuadrada			
Triangular			
Sierra			

7. TRANSFORMADA DISCRETA DE FOURIER

La transformada de Fourier es una de las herramientas más poderosas creadas en el procesamiento de señales. Este algoritmo matemático permite conocer la representación espectral de cualquier señal y a partir de allí analizarla o modificarla.

El desarrollo de esta práctica se centra en la transformada rápida de Fourier, o FFT, ya que este algoritmo permite calcular la DFT y la IDFT de una señal de una manera óptima en aplicaciones de tiempo real.

En el archivo "Practica3.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

7.1. Objetivos

- Escribir un programa que permita realizar la FFT y la IFFT de una señal
- Entender y comprender los datos generados a partir del algoritmo de la FFT
- Obtener y graficar el espectro de magnitud de la FFT
- Conocer y aplicar el concepto de ventanas a señales en el tiempo
- Calcular la FFT e IFFT aprovechando el acelerador por hardware que ofrece el DSP

7.2. Conceptos previos

Uno de los algoritmos más populares para el cálculo de la FFT es el Cooley-Tukey. En su forma más simple, consiste en partir la FFT de una señal de N puntos en 2 de $\mathrm{N}/2$ puntos, luego, en 4 de $\mathrm{N}/4$ puntos y así sucesivamente hasta obtener FFT's de tan solo 1 dato.

Para poder implementar dicho algoritmo es necesario que la longitud de la señal sea una potencia entera de 2, es decir 4, 8, 16, 128, etc. Además, el conjunto de datos a procesar debe ser previamente reorganizado mediante un algoritmo de permutación llamado *bit reversing*.

En la Figura 16 se puede observar el procedimiento requerido para realizar una FFT de longitud 8. Nótese que a pesar de que los datos no conservan la secuencia original, el resultado posee el orden correcto. El núcleo de este algoritmo, conocido como mariposa, es mostrado en la Figura 17. Esta puede entenderse simplemente como una FFT de 2 puntos y su cálculo requiere además de los 2 valores de entrada, un tercer elemento, conocido como *twiddle factor*. Estos elementos pueden conocerse a partir de la ecuación

$$W_N^k = e^{\frac{-j2\pi k}{N}},$$

y por lo general son calculados previamente y almacenados en LUT.

La FFT es una operación realizada sobre un conjunto de números complejos, que a su vez retorna otro grupo de números complejos que representan la magnitud y la fase de la señal original. Si se toma un punto de esta transformada como $z = a + b * j$, se puede calcular la fase $\emptyset$, de la componente espectral de dicho punto como

$$\emptyset = tan^{-1}(b/a),$$

y la magnitud como

$$|z| = \sqrt{a^2 + b^2}$$

Manejo de cantidades complejas. El cálculo de la FFT de una señal requiere realizar operaciones con números complejos. Sin embargo el compilador de C del DSP usado no cuenta con un soporte para el manejo de estas cantidades. Por esto, es necesario idear una manera de crear, interpretar y manipular este tipo de datos.

Para representar un número complejo es posible generar una entidad que pueda almacenar tanto la parte real como la imaginaria del número empleando el formato Q15 para ambas.

Para ello, se puede definir un nuevo tipo de dato `CInt16` como sigue:

```
typedef struct _CInt16
{
        Int16 re;
        Int16 im;
} CInt16;
```

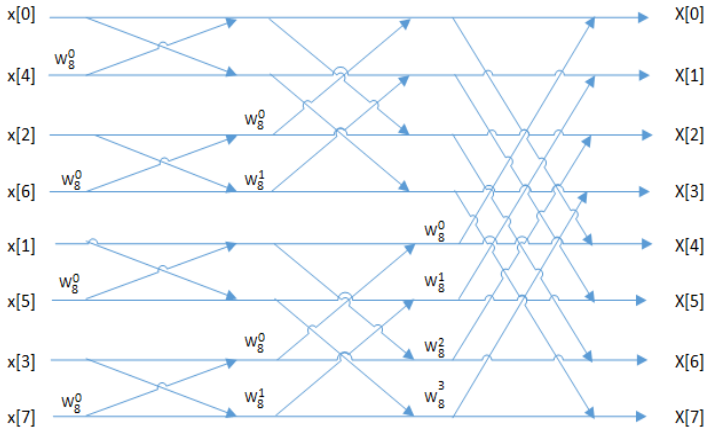


Figura 16. Diagrama de una FFT de longitud 8

Y para definir una variable de este tipo y obtener la parte real e imaginaria puede escribirse:

```
CInt16 z = {3210, -5879};
        //crear    el    numero    complejo
        //z = 0.097961426 - j 0.179412842
Int16 real, imag;
real = z.re;
imag = z.im;
```

```
// recordar los operadores "." y "->" que
// permiten acceder a los datos de una
// estructura
```

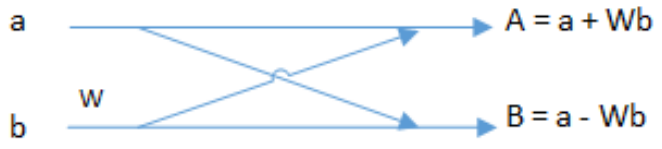


Figura 17. Diagrama de mariposa

Una manera de definir las operaciones básicas (suma, resta, multiplicación y conjugado) es creando una función para cada una de ellas. Sin embargo, esto puede resultar en un código muy ineficiente debido al *overhead* que debe ejecutar el DSP para el llamado y retorno de dichas funciones; especialmente si estas son usadas dentro de ciclos con muchas iteraciones. Debido a esto, se recomienda definir estas operaciones como macros. Así el preprocesador del compilador reemplazará cada aparición del macro por el código asociado, evitando las instrucciones adicionales mencionadas.

- Usando macros:

```
#define _CAdd(a, b) \
      ((CInt16) {.re = a.re + b.re, .im \
      = a.im + b.im})
#define _CSub(a, b) \
      ((CInt16) {.re = a.re - b.re, .im \
      = a.im - b.im})
#define _CMult(a, b) \
      ((CInt16) {\.re = (Int16) \
      (((Int32) a.re * b.re) - \
      ((Int32) a.im * b.im)) >> 15), \
      .im = (Int16)((((Int32) a.im \
      * b.re) + ((Int32) a.re * b.im)) \
      >> 15)})
#define _CConj(a) \
      ((CInt16) {.re = a.re, .im = - \
      a.im})
```

- Usando funciones:

```
CInt16 cadd (CInt16 a, Cint16 b)
{
      CInt16 res;
      res.re = a.re + b.re;
      res.im = a.im + b.im;
      return res;
}

CInt16 csub (CInt16 a, Cint16 b)
{
      CInt16 res;
      res.re = a.re - b.re;
      res.im = a.im - b.im;
      return res;
}

CInt16 cmult (CInt16 a, Cint16 b)
```

```
{
        CInt16 res;
        res.re = (Int16) ( ( ( (long) a.re
        * b.re ) - ( (long) a.im * b.im ) )
        >> 15 );
        res.im = (Int16) ( ( ( (long) a.im
        * b.re )
        + ( (long) a.re * b.im ) ) >> 15 );
        return res;
}

CInt16 cconj (CInt16 a)
{
        CInt16 res;
        res.re = a.re;
        res.im = -a.im;
        return res;
}
```

En los archivos "complex.c" y "complex.h", presentes en la carpeta "Codigos\Practica3" del DVD anexo se encuentra una forma de implementar el manejo de cantidades complejas y algunas operaciones básicas con ellas.

Pensando en aplicaciones de tiempo real, el kit de desarrollo eZdsp™ cuenta con un procesador –HWAFFT- externo al núcleo del DSP, que permite calcular de manera eficiente la FFT y la IFFT de una señal de hasta 1024 puntos. Incluso los 512 c*omplex twiddle factors* requeridos para implementar dichos algoritmos se encuentran almacenados previamente en las LUT del co-procesador.

La función de C `Uint16 hwafft_Npts` (donde `N` es una potencia de 2, hasta 1024) permite calcular la transformada de Fourier de una señal. A continuación se listan en orden los elementos que se requieren para el llamado de dicha función:

- `Int32 *data`: elemento complejo que apunta al arreglo de datos a procesar
- `Int32 *scratch`: elemento complejo que apunta a un arreglo en el que se almacenan los resultados intermedios durante la ejecución del algoritmo
- `Uint16 fft_flag`: Esta bandera indica si se va a ejecutar una FFT -0- o una IFFT -1-
- `Uint16 scale_flag`: Esta bandera indica si se realiza escalamiento -0- durante el cálculo de cada mariposa

Además, esta función retorna un valor de '0' o '1' que indica al programador si el resultado final se encuentra almacenado en el arreglo `data` -0- o en el arreglo `scratch` -1-.

Debido al uso de elementos `Int32` para poder manejar los elementos complejos, el co-procesador HWAFFT requiere que la parte real de cada punto se encuentre en los 16 bits más significativos (bit 31 a bit 16) de la variable y que la parte imaginaria se encuentre en los 16 bits menos significativos (bit 15 a bit 0). De esta forma, basta con usar operaciones de truncamiento, desplazamiento o casting para poder extraer o

almacenar datos de tipo `Int16`, a partir de un elemento `Int32`, así:

- Escritura: Suponga que los elementos `ParteReal` y `ParteImaginaria` fueron previamente declarados e inicializados correctamente como `Int16`.

```
Int32 Cmplx32 = (((Int32) ParteReal)
      << 16) | ((Int32) ParteImaginaria);
```

- Lectura: Suponga que el elemento `Cmplx32` fue previamente declarado e inicializado correctamente como `Int32`.

```
Int16 ParteReal = (Int16) (Cmplx32 >> 16);

Int16 ParteImaginaria = (Int16) (Cmplx32 &
0x0000FFFF);
```

El HWAFFT también cuenta con la función `void hwafft_br`, optimizada para realizar la operación de *bit reversing* en los elementos a procesar. A continuación se listan en orden los elementos que se requieren para el llamado de dicha función:

- `Int32 *data`: elemento complejo que apunta al arreglo de datos a procesar
- `Int32 *data_br`: elemento complejo en el que se almacena el resultado del *bit reversing*
- `Uint16 *data_len`: longitud del vector a procesar

Finalmente, el uso de este co-procesador requiere condiciones especiales de memoria para el almacenamiento de los datos que se pasan a las funciones. La posición de memoria a la que apunta la variable `data_br` de la función `hwafft_br` y la variable `data` de la función `hwafft_Npts`, deben ser almacenadas de tal forma que $log_2 4N$ ceros se encuentren en los bits menos significativos de la dirección de memoria. Para esto es requerido el uso de la directiva del compilador `#pragma DATA_SECTION()` que permite reservar espacios específicos de memoria.

En el archivo "main.c" del directorio "Codigos\Practica3" del DVD anexo, se muestra la forma de usar dicha directiva para separar el espacio de memoria requerido. Adicionalmente, los archivos "hwafft.h" y "C5535.cmd" (presentes en esta misma carpeta) deben ser agregados al proyecto para el uso del acelerador. Es importante revisar su contenido para comprender el funcionamiento del HWAFFT.

Ventanas. Debido a la resolución limitada de la FFT, restringida a la relación entre la frecuencia de muestreo y la longitud de la transformada como f_s/N, resulta necesario en muchas ocasiones acondicionar las señales que se van a analizar para poder obtener de ellas una representación más acertada de la información. Por ello, las ventanas sirven como una herramienta que permite superar varios de los problemas que surgen al tratar de calcular discretamente características que son naturalmente continuas en una señal.

Para conocer la FFT de un conjunto de datos es necesario tomar una cantidad finita de muestras N. Por lo cual es bastante probable que el valor de la señal en la primera y última muestras sean diferentes, y más aún, que no sean valores cercanos a cero. Debido a que el algoritmo de la FFT asume que la señal a procesar es de carácter periódico, esta discontinuidad ocasiona que en el espectro calculado aparezcan componentes de alta frecuencia -asociados a los saltos abruptos- que pueden alejar las amplitudes calculadas de las que realmente contiene la señal. Este fenómeno conocido como *frequency leakage*, es la principal razón para el uso de las ventanas en el procesamiento digital de señales.

Básicamente una ventana es una señal conocida temporal y espectralmente, de longitud finita, cuyos valores suelen acercarse suavemente a cero al inicio y al final de un intervalo determinado, y que son además cero fuera de este. Normalmente suelen tener un espectro de carácter impulsivo, conformado por un lóbulo principal en el centro del espectro e infinitos lóbulos laterales de amplitud reducida (responsables del *frequency leakage*).

Para aplicar una ventana a una señal basta con multiplicar punto a punto los valores de esta con los valores de la señal antes de realizar la FFT de los datos. El resultado de esta técnica, es la presencia replicada del espectro de la ventana en cada componente de frecuencia de la transformada. O en otros términos: es la convolución de ambos espectros. Algunas de las ventanas más usadas (de longitud 'N') y las funciones que las describen se relacionan en la Tabla 8.

Tabla 8. Tipos comunes de ventanas

Tipo de ventana	Función
Triangular	$w(k) = 1 - \left\| \frac{k - \frac{N-1}{2}}{\frac{N-1}{2}} \right\|$
Hanning	$w(k) = 0.5 * (1 - \cos\frac{2\pi k}{N-1})$
Hamming	$w(k) = 0.54 - 0.46 * \cos\frac{2\pi k}{N-1}$
Welch	$w(k) = 1 - \left(\frac{k - \frac{N-1}{2}}{\frac{N-1}{2}} \right)^2$
Lanczos	$w(k) = sinc\left(\frac{2k}{N-1} - 1\right)$

7.3. Resultados

Aprovechando las formas de onda sintetizadas en prácticas anteriores como señales de prueba, se muestran en la Tabla 9 los resultados de aplicar una transformada de Fourier de 256 puntos a una señal senoidal y a una señal diente de sierra de 20 armónicos, ambas con frecuencia fundamental de 440Hz, usando el algoritmo propio y las funciones del HWAFFT.

En la Tabla 10, se muestran comparativamente tres tipos de ventanas, junto a la transformada de Fourier conseguida en cada caso usando el algoritmo propio para una señal diente de sierra de 20 armónicos con frecuencia fundamental de 440Hz. La diferencia encontrada entre estos espectros y el espectro original de la señal, se debe esencialmente a que la longitud escogida para las ventanas es muy pequeña en comparación a la frecuencia fundamental de la señal de este ejemplo, aumentando el efecto del *frequency leakage*. Además, únicamente la ventana Hanning permite diferenciar los picos de los armónicos de la señal, ya que de estas, es la ventana con los lóbulos laterales de menor amplitud. Para las otras ventanas, el espectro pareciera entonces ser continuo.

Tabla 9. Comparación de los espectros de magnitud

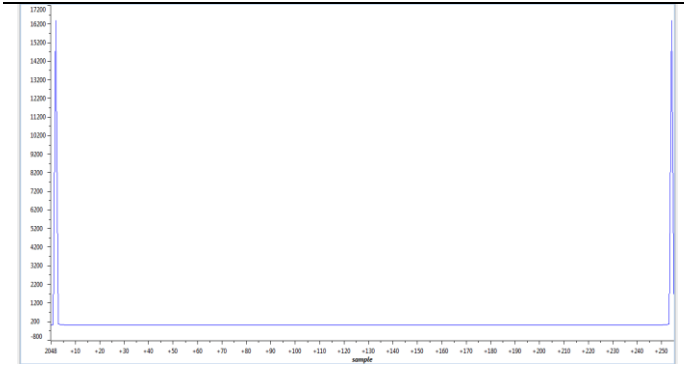
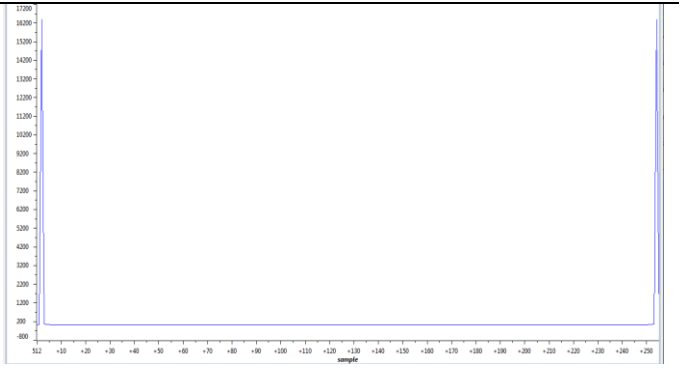
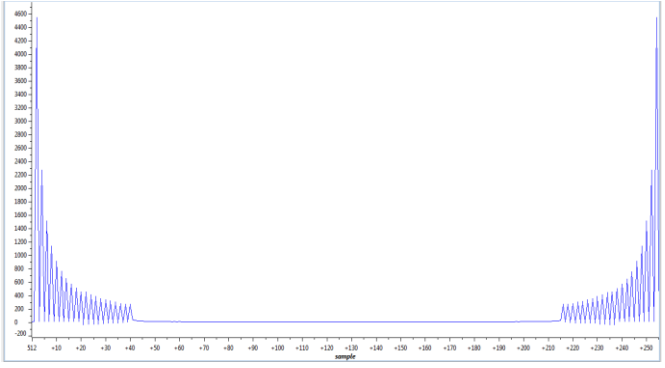
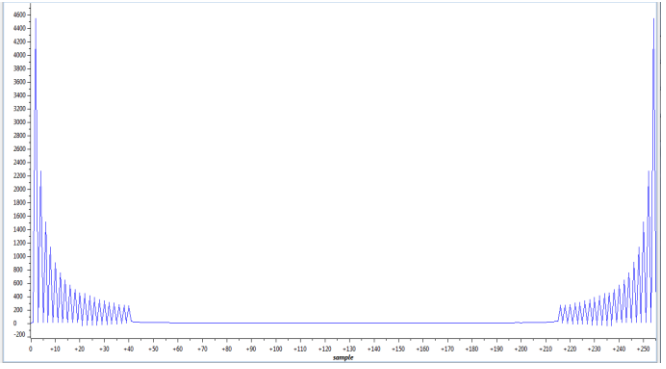
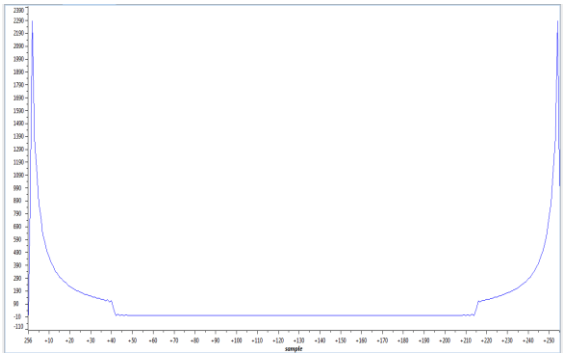
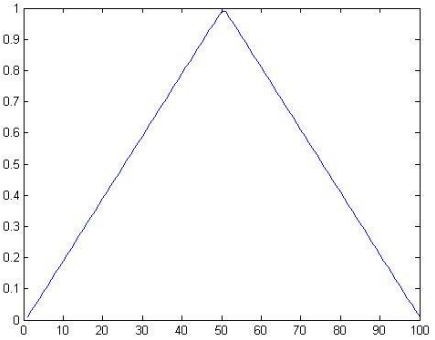
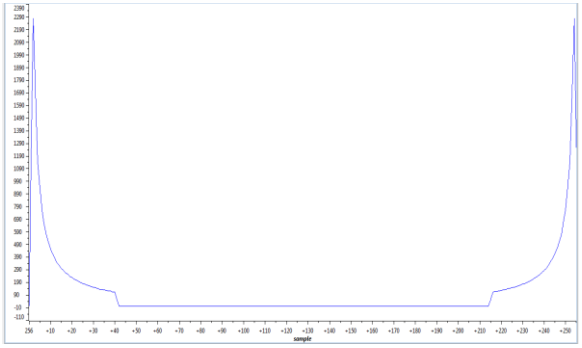
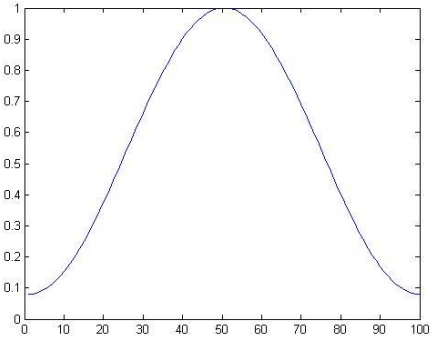
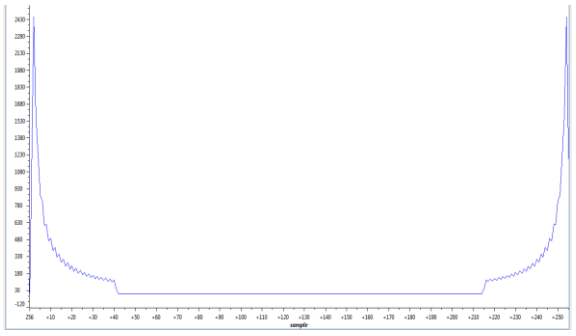
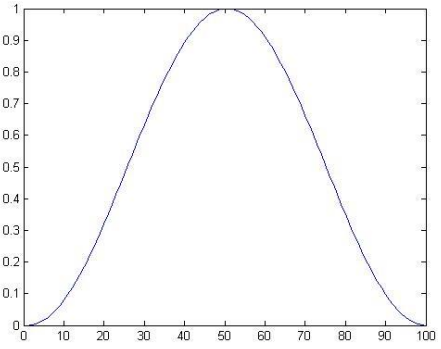
Algoritmo propio	Algoritmo del HWAFFT
	
	

Tabla 10. Comparación en el uso de ventanas en una FFT

Transformada de Fourier	Ventana implementada	
		Triangular
		Hamming
		Hanning

8. FILTROS FIR

Los filtros FIR, son filtros digitales caracterizados por tener una respuesta al impulso de duración finita y ser siempre estables, pues todos sus polos se hallan al interior del círculo unitario (criterio de estabilidad), específicamente en el origen. Estos filtros están definidos por la ecuación en diferencias

$$y[k] = b_0 x[k] + b_1 x[k-1] + \cdots + b_n x[k-n]$$

en la cual los elementos b_i son conocidos como los coeficientes del filtro.

La implementación de este tipo de filtros es bastante sencilla pues consiste únicamente en la acumulación de los productos de los coeficientes del filtro con las muestras retardadas de la señal de entrada.

En el archivo "Practica4.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

8.1. Objetivos

- Manejar e implementar buffers circulares
- Implementar un filtro FIR usando la forma directa
- Implementar un filtro FIR por convolución usando la librería "Dsplib"
- Utilizar la CSL (*Chip Support Library*) para el manejo de datos desde el módulo "MMC/SD"
- Verificar el comportamiento de un filtro FIR con ayuda de la FFT

8.2. Conceptos previos

Es importante tener en cuenta algunos aspectos sobre el rendimiento y los efectos que tienen en el procesamiento las operaciones de punto fijo. El principal elemento a tener en cuenta es que el desbordamiento puede ocurrir fácilmente, ya que el cálculo de la salida del filtro requiere la adición de múltiples términos en formato Q15. Para evitar esto es necesario escalar los coeficientes del filtro y eventualmente los valores de la entrada. Pero esta solución -y sobre todo cuando el orden del filtro es considerablemente grande- va introduciendo en la señal filtrada errores debidos al redondeo y escalamiento de los términos. Por ello es muy común que el software de diseño empleado para obtener los coeficientes se encargue de escalar estos términos de manera que se minimice la posibilidad de un desbordamiento.

Buffers circulares. El funcionamiento de los filtros digitales requiere del almacenamiento de los estados internos del proceso. Es decir, requiere conservar tantas muestras previas como sea el orden del filtro para poder calcular los nuevos valores de salida. Para realizar esta tarea es bastante recomendable hacer uso de un buffer –o arreglo- que permita realizar la lectura y escritura de nuevos elementos de una manera eficiente y que a su vez elimine

la necesidad de realizar muchos movimientos de memoria al actualizar los estados del filtro. Es allí donde surge el concepto del buffer circular.

Para la implementación de un filtro es necesario tener en cuenta en que espacio de memoria se encuentran tanto la muestra más reciente como las muestras precedentes necesarias para el cálculo de la salida. También es fundamental actualizar dichas posiciones de memoria en cada ciclo iterativo del filtro para que cada valor guarde una correspondencia temporal con las demás muestras. Normalmente dicho almacenamiento y "desplazamiento" de datos se haría de la siguiente forma en el caso de un filtro de orden 4:

```
Int16 x[5];
x[0] = NuevaEntrada;
// Procesamiento usando el vector x
…
//Actualización de los datos
x[4] = x[3];
x[3] = x[2];
x[2] = x[1];
x[1] = x[0];
```

Es evidente que al incrementar el orden del filtro el manejo de los datos se vuelve muy ineficiente ya que se necesitan leer, mover y escribir tantos elementos como sea el orden de este, aun cuando solamente se esté almacenando una muestra en cada ciclo.

Para solventar dicha ineficiencia se ha propuesto una forma de acceder y relacionar los datos mediante el uso de punteros "móviles". Así, como se muestra en la Figura 18, se usan dos punteros para indicar la ubicación de la muestra más reciente y la más antigua. Y para almacenar un nuevo dato en el buffer basta reemplazar el valor más antiguo y actualizar los punteros de modo que reflejen correctamente el nuevo estado del buffer. De esta manera se logran leer y escribir los elementos sin necesidad de moverlos de su espacio de memoria (Orfanidis, 1995).

```
#define ORDEN_BUFF  50
...
Int16 datos[ORDEN_BUFF], NuevaMuestra;
short *p_start, *p_end;
short i;
// Se inicializa el buffer
for(i = 0; i < ORDEN_BUFF; i++)
{
       datos[i] = 0;
}
// Se inicializan los punteros
p_start = datos;
p_end = datos + (ORDEN_BUFF – 1);
...
// Ciclo iterativo
*p_end = NuevaMuestra; // Almacena el dato
                       //actual
// Actualizacion de los punteros
p_start = p_end;
p_end = (p_end > datos)?
       p_end – 1 : p_end + ORDEN_BUFF - 1;
```

Figura 18. Movimiento de punteros en un buffer circular

Forma directa. Una de las formas de implementar un filtro FIR es aplicando el modelo descrito en el diagrama de bloques de la Figura 19. A partir de dicho diagrama, se puede calcular el valor de la salida en cualquier instante de tiempo como

$$y[k] = \sum_{n=0}^{N} b_n * x[k-n] \ .$$

Luego de analizar la ecuación anterior, se hace evidente la alta probabilidad que existe de que ocurra un desbordamiento debido a la operación de acumulación que se debe efectuar en cada etapa del filtrado. Para resolver este problema se han ideado diversas soluciones. La más fácil de implementar consiste en escalar el resultado de la sumatoria entre el orden del filtro. De esta forma se puede asegurar que el máximo valor posible sea ±1. Aunque esta solución no resulta bastante práctica, pues sigue existiendo el riesgo de desbordamiento en cálculos intermedios y además requiere efectuar divisiones entre enteros.

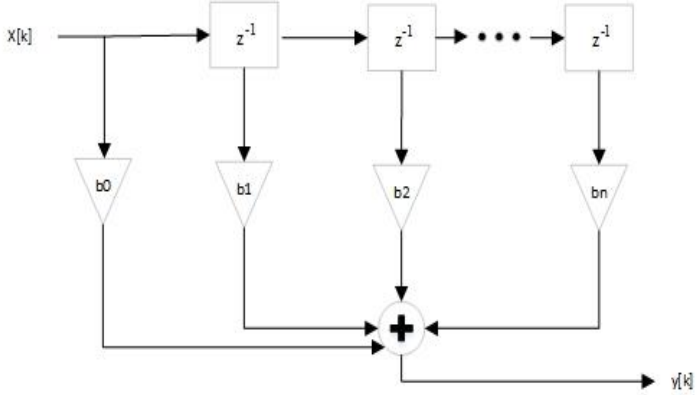


Figura 19. Forma directa de un filtro FIR

Otra solución bastante efectiva consiste en multiplicar la entrada por una constante menor que 1 que evite que la suma de términos se desborde. Dicha constante puede conseguirse como la suma de los valores absolutos de los coeficientes del filtro o también como el cuadrado de estos mismos valores. Sin embargo, esta medida puede aumentar significativamente el ruido de cuantización deteriorando la calidad de la señal filtrada. También es posible escalar los coeficientes del filtro previamente, pero dada la precisión finita del hardware esto ocasiona que el filtro práctico difiera apreciablemente del diseñado. Aun así, este método representa una muy buena alternativa para prevenir los problemas de *overflow*.

Filtrado por convolución. Aparte de las características ya mencionadas, un filtro FIR presenta la particularidad de que su respuesta al impulso, no solo es finita, sino que posee los mismos valores que los coeficientes del filtro. Por esto, se suele aprovechar dicha característica para aplicar el principio de convolución y realizar el filtrado de una señal.

Idealmente, podría realizarse este filtrado convolucionando la señal de entrada de longitud L con la respuesta de un filtro de orden M. Produciendo así una tercera señal de longitud $L + M$. Pero muchas veces esto no es posible, ya que en la práctica suelen tenerse señales de gran longitud o señales de tiempo real y los sistemas DSP cuentan con recursos de memoria y procesamiento limitados. Por ello es necesario procesar la señal de entrada en bloques de menor longitud que puedan ser manejados cómodamente por el hardware.

Existen principalmente dos métodos de procesamiento que permiten realizar este filtrado en bloques: *overlap and add y overlap and save*. Su principio de funcionamiento se puede observar en la Figura 20 y en la Figura 21, respectivamente. De estos dos métodos es importante revisar detalladamente el de *overlap and save*, pues es el algoritmo empleado por la función de convolución de la librería "Dsplib".

Overlap and save. Este método de procesamiento permite realizar la convolución discreta de dos señales, dividiendo una señal de gran longitud en varios bloques de menor tamaño.

La idea del método es calcular segmentos cortos de longitud L de la señal de salida que al ser concatenados generan la señal deseada. Para ello es necesario convolucionar los bloques cortos de la entrada con el filtro, teniendo en cuenta que cada nuevo bloque se forma juntando las L muestras más recientes, precedidas por las M últimas muestras del bloque anterior. Siendo M el orden del filtro que se está implementando y asumiendo como cero las muestras que preceden al primer bloque (Smith, 1999).

Chip Support Library. La CSL, es una librería de Texas Instruments desarrollada con la finalidad de facilitar la configuración y el manejo de periféricos en diferentes familias de DSP. La versión que se puede descargar de manera gratuita desde la página de TI, "Low Power", permite usar en el eZdsp™

elementos como: leds, pulsadores, *display*, módulo SD, puertos de expansión análogos y digitales, entre otros, de manera fácil y rápida, gracias a la documentación incluida. Es importante tener en cuenta la versión del CCS y de la CSL para evitar problemas de compatibilidad al compilar la librería (si se está utilizando la versión 4 del IDE, es necesario usar la versión 3.01, o anteriores de la API).

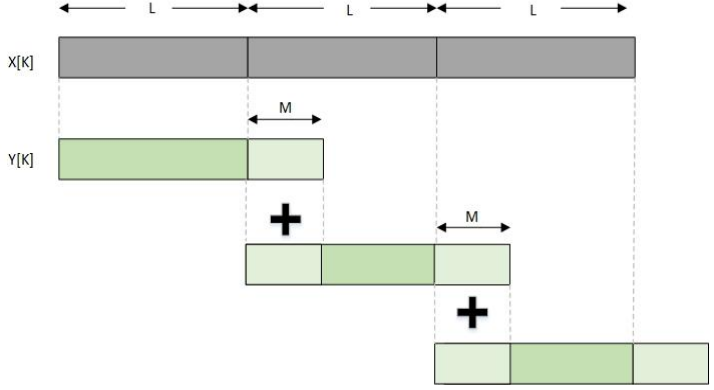


Figura 20. *Overlap and add*

Con el ánimo de acercar al estudiante a las diferentes herramientas de trabajo dispuestas por los fabricantes, se usará en esta práctica el módulo MMC/SD para leer una señal de ruido blanco que posteriormente será manipulada según los objetivos de esta práctica. En los archivos "SD.c" y "SD.h", disponibles en el directorio "Codigos\Practica4" del DVD anexo se puede encontrar una rutina de inicialización del módulo para el manejo de la SD. Para usar dicha unidad, la tarjeta debe encontrarse en formato FAT32 y su escritura debe ser realizada en modo *Little Endian*. Además es necesario realizar el procedimiento de inclusión de archivos y librerías para el uso de la CSL, tal como se explicó en la sección 4 de este documento.

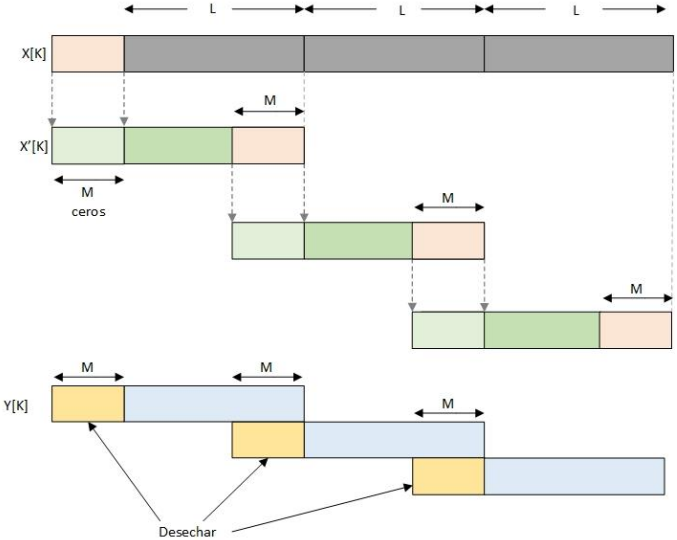


Figura 21. *Overlap and save*

8.3. Resultados

En el archivo “FIR_directa.wav” del directorio “Audios” presente en el DVD anexo se puede apreciar el resultado de filtrar el sonido de una guitarra usando un filtro pasabajos implementado con la forma directa. Similarmente, en el archivo “FIR_convol.wav” se encuentra el resultado de filtrar la misma fuente sonora con un filtro pasaaltos implementado mediante la función de convolución de la Dsplib. En el archivo “guitarra_FIR.wav” se puede escuchar el sonido de la guitarra sin filtrar.

En la Figura 22 se puede observar la señal de ruido leída desde la SD. Seguidamente se observa la FFT aplicada a la señal filtrada con un filtro FIR pasabajos.

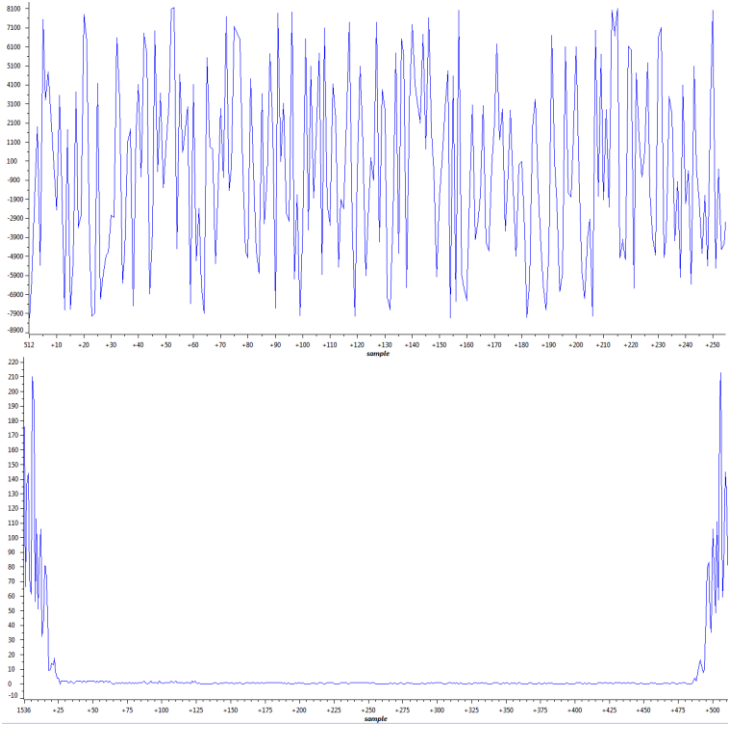


Figura 22. Comparativo del ruido blanco

9. FILTROS IIR

Los filtros IIR, son filtros digitales caracterizados por tener una respuesta al impulso de duración infinita. Estos no son siempre estables ya que su configuración de polos puede estar por fuera del círculo unitario. Los filtros IIR están definidos por la ecuación en diferencias

$$\sum_{m=0}^{M} a_m y[k-m] = \sum_{n=0}^{N} b_n x[k-n],$$

en la cual los elementos a_m y $b_n$son los coeficientes del filtro.

Con este tipo de filtros es posible conseguir una transición más rápida entre las bandas de paso y de rechazo para un orden menor del filtro, en comparación a los filtros tipo FIR.

En el archivo "Practica5.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

9.1. Objetivos

- Implementar filtros IIR en forma directa y en forma canónica
- Implementar filtros bicuadráticos, sencillos y en cascada
- Implementar osciladores recursivos
- Reconocer auditivamente el comportamiento de los filtros empleando una señal sintetizada tipo *chirp*

9.2. Conceptos previos

La principal diferencia entre los filtros tipo FIR e IIR, es la presencia de lazos de realimentación, es decir, lazos que llevan la salida del filtro a los estados internos del mismo. El uso de estos elementos permite el diseño de filtros con mayor diversidad en su respuesta de frecuencia, como los filtros *notch*, *shelving* y *peaking*, entre otros. Sin embargo, su diseño debe realizarse con un mayor cuidado al ser estos más susceptibles a inestabilidades.

La función de transferencia que describe este tipo de filtros es la siguiente:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_n z^{-n}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_m z^{-m}}.$$

Dado que $H(z)$ es una función racional, es posible conseguir diferentes representaciones de la misma empleando herramientas algebraicas como factorización o fracciones parciales. Esto permite obtener diversas estructuras para la implementación del filtro. Cada una con propiedades determinadas.

Forma directa. Al obtener la ecuación en diferencias directamente de la función de transferencia, se obtiene la forma directa como se observa en la Figura 23. La característica más importante de esta estructura es la presencia de un solo sumador, que acumula tanto los términos directos como los recursivos.

Su ecuación en diferencias es:

$$y[k] = -a_1 y[k-1] - a_2 y[k-2] \ldots - a_m y[k-m] + b_0 x[k] + b_1 x[k-1] + \cdots + b_n x[k-n].$$

Dada la complejidad de los filtros IIR, es común introducir en el análisis de estos sistemas un nuevo tipo de variables $v_n[k]$ y $w_n[k]$, llamadas variables de estado. Con ellas es posible definir la dinámica del sistema a partir de los valores presentes a la salida de cada bloque de retardo.

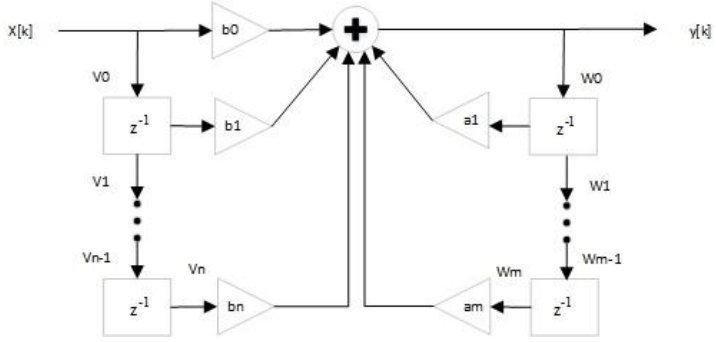


Figura 23. Forma directa del filtro IIR

Debido a la cantidad de retardos que son necesarios para implementar la forma directa (aun cuando solo se usa un acumulador), es preferible usar otra representación que permita economizar recursos en los sistemas digitales.

Forma canónica o forma directa II. Pensando la función de transferencia $H(z)$, como el producto de dos elementos: $N(z)$ y $\frac{1}{D(Z)}$, es posible reorganizar dichos términos para obtener la estructura mostrada en la Figura 24. Gracias a esto, se consigue reducir el número de retardos requeridos de: $n+m$ a $k = MAX(n,m)$, únicamente añadiendo un acumulador.

Forma en cascada. Aplicando factorización a los polinomios del numerador y el denominador de la función de transferencia $H(z)$, es posible separar dichos elementos como el producto de secciones de segundo orden organizados en forma canónica, conocidas como filtros bicuadráticos -Figura 25. Estos tipos de estructuras son ampliamente utilizadas en el procesamiento digital gracias a su eficiencia y a que reducen considerablemente los errores de redondeo presentes en los procesadores de punto fijo cuando son implementados en cascada. También permiten prevenir el *overflow* de una mejor manera, ya que la salida de cada etapa puede ser escalada fácilmente (Aboagye, 1999).

Problemas de redondeo. El diseño de filtros en software matemáticos genera usualmente coeficientes en formato de punto flotante de doble precisión. Esto requiere que el hardware en el cual se implementen cuente con una longitud de palabra de mínimo 64 bits. Pero en la práctica los DSP suelen ser de punto fijo y de 16 o 24 bits. Esta diferencia en la representación de las cantidades ocasiona que los filtros diseñados y los que realmente se implementan en el hardware difieran considerablemente, al punto de llegar a ser inestables o producir respuestas en

frecuencia y fase completamente diferentes a las deseadas (Manolakis & Ingle, 2011).

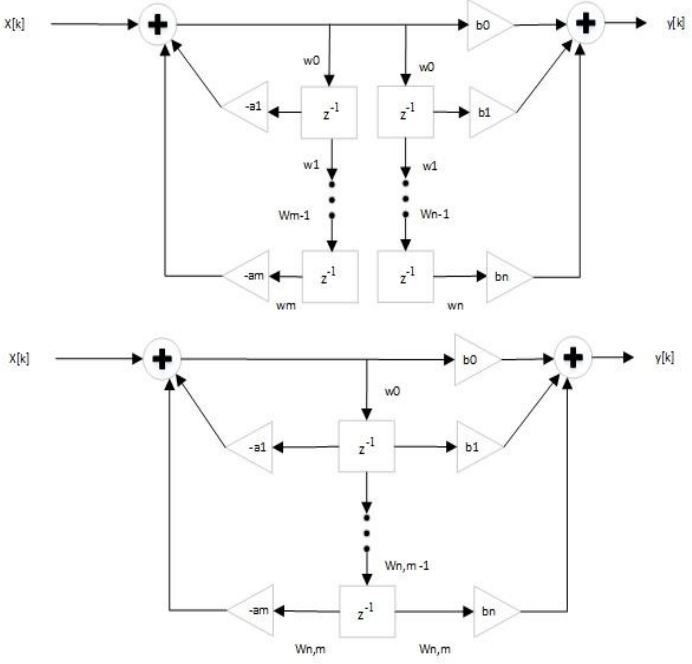


Figura 24. Forma canónica del filtro IIR

Para contrarrestar estos efectos muchos software permiten modelar los filtros de acuerdo a la representación numérica de punto fijo requerida. Esto permite anteponerse a los errores de redondeo para aplicar las correcciones pertinentes. Otra posible solución, es modificar la implementación del filtro en el hardware hacia formas en cascada y canónicas, preferiblemente (Wilson, 1993).

En la Figura 26 se pueden observar los efectos de la cuantización sobre un filtro IIR implementado en forma directa, y la diferencia en la respuesta de frecuencias que esto conlleva.

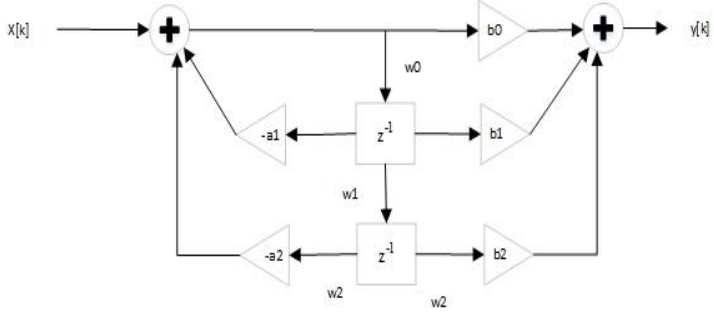


Figura 25. Forma bicuadrática del filtro IIR

Osciladores recursivos. Gracias a la respuesta de duración infinita de los filtros IIR es posible generar a partir de estos señales periódicas generadas recursivamente. Normalmente esta técnica solo es usada para sintetizar señales senoidales ya que para otras más complejas (diente de sierra, cuadrada, triangular, etc.) se requieren filtros de ordenes elevados, cuya implementación resulta mucho menos eficiente que el uso de LUT.

Una señal senoidal puede ser generada como la respuesta al impulso del filtro con función de transferencia

$$H(z) = \frac{R\sin(\omega_0)z^{-1}}{1-2R\cos(\omega_0)z^{-1}+R^2z^{-2}}.$$

De igual manera, una señal cosenoidal puede generarse a partir de la función de transferencia

$$H(z) = \frac{1-R\,cos(\omega_0)z^{-1}}{1-2R\,cos(\omega_0)z^{-1}+R^2z^{-2}}.$$

Normalmente no se suele usar este tipo de señal sinusoidal ya que al comenzar en '1' puede generar un "clic" en el audio escuchado.

En estas ecuaciones el valor R representa la constante de amortiguamiento que define la velocidad con la que se extingue la señal generada. Si se cumple que $R = 1$ la señal oscilará infinitamente (Kuo & Lee, 2001).

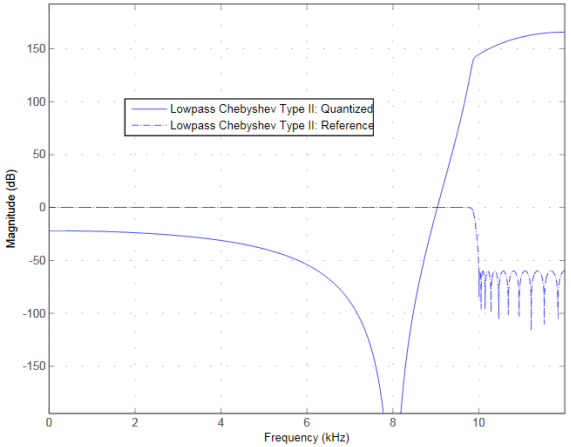


Figura 26. Efectos de la cuantización en un filtro IIR

9.3. *Resultados*

En el archivo "IIR.wav" del directorio "Audios" presente en el DVD anexo se puede apreciar el resultado de filtrar una onda *chirp* usando un filtro pasabajos implementado en forma directa y en cascada. Primero se puede oír la onda sin ningún tipo de procesamiento, seguidamente se escucha la misma señal al ser filtrada primero por la implementación en forma directa y luego por la implementación en cascada. En este audio pueden apreciarse claramente los errores producidos al cuantizar los

coeficientes de los filtros, así como la mejora que muestra para el procesamiento el uso de filtros bicuadráticos. En la Figura 27 se pueden observar las respuestas de ambos filtros. La respuesta superior corresponde a una implementación en forma directa y la respuesta inferior a una en cascada.

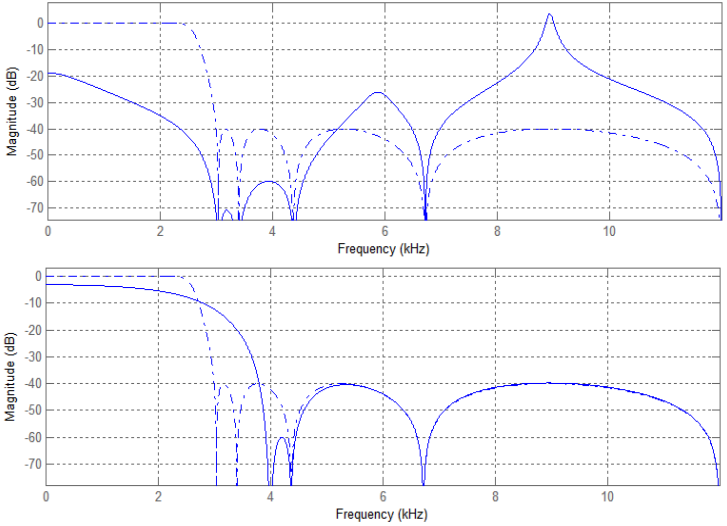


Figura 27. Estructuras y cuantización de un filtro IIR

En este mismo directorio, en el archivo "IIR_seno.wav" puede escucharse una señal senoidal de 440 Hz sintetizada recursivamente, la misma puede observarse en la Figura 28.

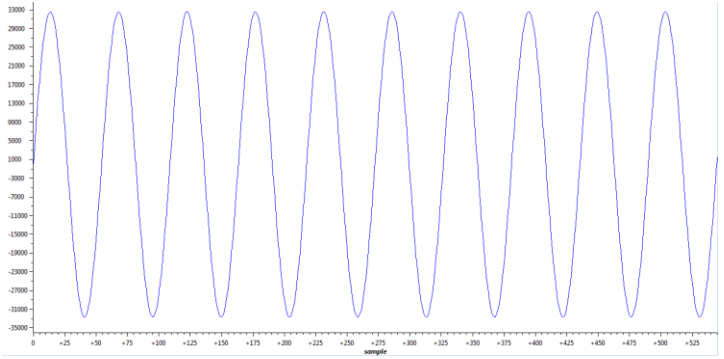


Figura 28. Síntesis por filtro IIR de una señal senoidal

10. FILTROS ADAPTATIVOS Y FILTROS PASA-TODO

Un filtro adaptativo es un sistema variable en el tiempo que modifica sus parámetros de acuerdo a una señal de entrada cualquiera $x[k]$ y a una señal de referencia $d[k]$, en respuesta a un algoritmo de optimización –Figura 29. Normalmente estos parámetros variables son los coeficientes de los filtros FIR o IIR.

Por otro lado, un filtro pasa-todo es un filtro IIR de cualquier orden que tiene una respuesta constante en el espectro de frecuencia pero que modifica la fase de dichas componentes. Estos filtros son usados entre otras cosas, para compensar los desfases generados por otros sistemas o para corregir problemas de inestabilidad que puedan tenerse en ciertos filtros. Los filtros pasa-todo son conocidos también como "ecualizadores de fase".

En el archivo "Practica6.pdf" del directorio "Guias" presente en el DVD anexo se encuentra la guía correspondiente para el desarrollo de esta práctica.

10.1. Objetivos

- Conocer las diferentes estructuras de los filtros adaptativos
- Implementar un filtro adaptativo FIR mediante el algoritmo LMS
- Conocer algunas de las aplicaciones de los filtros pasa-todo
- Implementar un filtro *notch* a partir de un filtro pasa-todo

10.2. Conceptos previos

Existen diversas estructuras que permiten implementar filtros adaptativos. Dicha clasificación está definida básicamente por la relación existente entre la señal de entrada y la referencia. Cada una de ellas responde a una aplicación específica.

Identificación de un sistema. Es muy usual que en el procesamiento de señales se desconozcan los modelos que describen un sistema. Una forma de aproximar el comportamiento de dichos elementos consiste en implementar un filtro adaptativo de tal forma que este se modifique hasta parecerse al sistema desconocido, tal como se muestra en la Figura 30.

Se puede saber que la función de transferencia del filtro se ha acercado a la del sistema real, cuando la señal de error es muy pequeña o nula.

Identificación inversa de un sistema. Similarmente a la implementación anterior, es posible configurar la estructura del filtro de tal forma que este se modifique hasta comportarse inversamente al sistema desconocido, como se observa en la Figura 31. Este método requiere de un bloque adicional de retardo de la señal de entrada que sirve para mantener la causalidad del

sistema. Dicho retardo debe ser similar al introducido por el sistema desconocido.

Similarmente a la estructura anterior, se puede saber que el filtro se comporta de la manera esperada cuando la señal de error es casi nula.

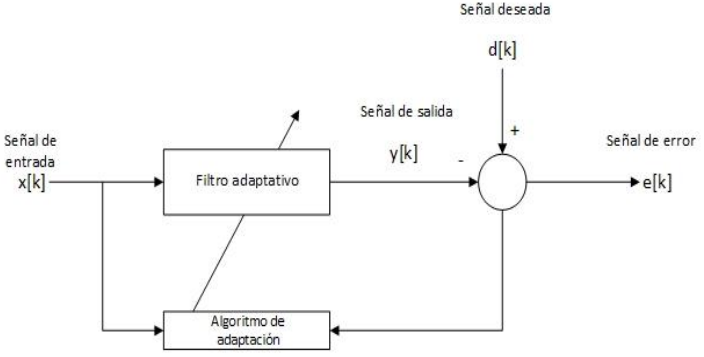


Figura 29. Diagrama general de un filtro adaptativo

Cancelación de ruido. Con el fin de eliminar las señales indeseadas de un sistema es común implementar la estructura de un filtro adaptativo como se muestra en la Figura 32. En esta configuración la entrada del filtro adaptativo es una señal de ruido de características similares al que se desea quitar y la señal deseada es alimentada con aquella que se requiere limpiar. Con este método la señal de error no tiende a cero, sino a ser la señal deseada. Es decir la señal sin ruido.

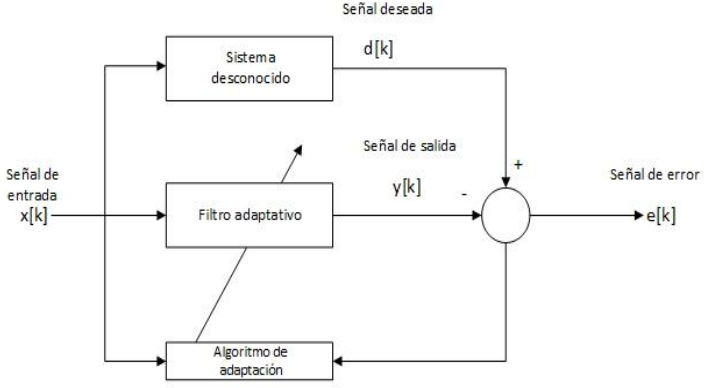


Figura 30. Identificación de un sistema usando filtros adaptativos

Predicción. Teniendo en cuenta que esta configuración solo funciona con señales periódicas de baja frecuencia, un filtro adaptativo puede configurarse como se muestra en la Figura 33 de tal forma que pueda predecir el comportamiento de una señal de entrada.

Considerando que el orden del *delay* sea menor al orden del filtro adaptativo, es posible usar esta configuración para eliminar ruido de una señal de cambio lento, como lo es la voz humana. Así, la

señal $e[k]$ contendría la señal de la cual se ha sido removido el ruido.

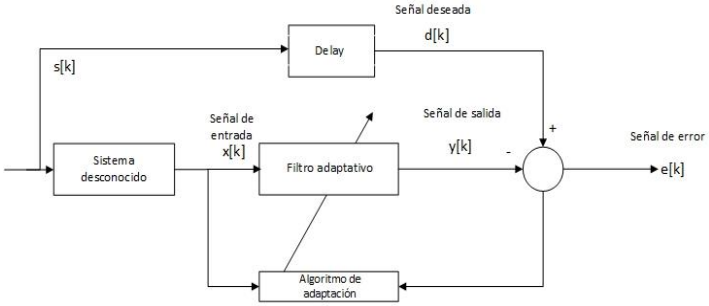


Figura 31. Identificación inversa usando filtros adaptativos

Algoritmos de adaptación. Ya que los filtros IIR pueden llegar a ser inestables, es común implementar los filtros adaptativos con estructuras tipo FIR. Así, además de reducir los cálculos a un solo arreglo de coeficientes, se evitan consideraciones adicionales que resulten en un código demasiado complejo.

Básicamente un algoritmo de adaptación debe responder a la optimización de la función

$$J(k) = E[e^2(k)] \, .$$

Es decir, el valor mínimo de la esperanza estadística del error al cuadrado.

De esta ecuación se derivan dos algoritmos principales para el cálculo de la función mencionada.

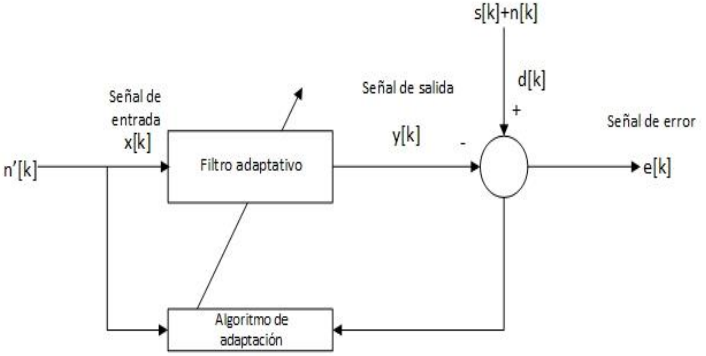


Figura 32. Cancelación de ruido usando filtros adaptativos

El algoritmo *Recursive Least Square* (RLS) permite una mayor tasa de convergencia de los coeficientes al involucrar cálculos matemáticos de gran complejidad (Haykin, 2013). Debido a esto, no suele ser implementado en procesamiento en tiempo real, ya que requeriría un hardware de gran capacidad para ser ejecutado.

Por otro lado, el algoritmo *Least Mean Square* (LMS) permite una implementación bastante simple con buenos resultados, involucrando operaciones mucho más sencillas y fácilmente computables por el kit de desarrollo eZdsp™. Con este método es posible calcular los nuevos valores del filtro como

$$b_n[k+1] = b_n[k] + \mu * e[k] * x[k],$$

donde el error instantáneo es

$$e[k] = d[k] - \sum_{i=0}^{N} b_i[k] * x[k-i],$$

y el parámetro 'μ' (*step size*) se relaciona directamente con la velocidad de convergencia del filtro. Para valores bajos de este, el filtro tarda mucho en responder y no resulta útil, y para valores altos el filtro oscila. Normalmente se suelen dar valores a μ entre 0.002 y 0.005.

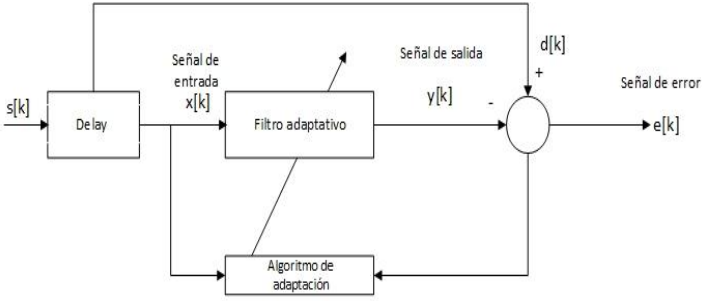


Figura 33. Predicción usando filtros adaptativos

Filtros pasa-todo. Estos filtros se caracterizan por tener sus polos y ceros en pares conjugados recíprocos. Su función de transferencia compleja es entonces

$$A(z) = e^{j\theta} \prod_{k=1}^{M} \frac{\gamma_k^* - z^{-1}}{1 - \gamma_k z^{-1}}.$$

Generalizando dicha función para ser de carácter real, se tiene que:

$$A(z) = \frac{z^{-M} D(z^{-1})}{D(z)}.$$

Los grupos de polinomios obtenidos a partir de esta ecuación se conocen como "imagen especular" (*mirror-image*) y para un filtro de segundo orden, tienen la forma:

$$A(z) = \frac{a_2 + a_1 z^{-1} + z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}.$$

Esta particularidad resulta en dos cualidades muy importantes para la implementación en tiempo real y de punto fijo de dichos filtros. La primera permite disminuir el número de multiplicaciones requeridas para calcular la salida del filtro de $2M + 1$ -para un filtro en forma directa- a tan solo $M + 1$. La segunda es la conservación de la respuesta en frecuencias del filtro aun cuando se han cuantizado los coeficientes.

Filtros complementarios. "Dos filtros son llamados complementarios, si las bandas de paso de uno corresponden a las bandas de rechazo de otro" (Regalia, Mitra, & Vaidyanathan, 1988). Es decir, dos filtros $B(z)$ y $C(z)$ que cumplen el siguiente par de ecuaciones en términos de filtros pasa-todo $A_x(z)$ se consideran con dicha cualidad:

$$B(z) + C(z) = A_1(Z)$$

y

$$B(z) - C(z) = A_2(Z).$$

Este par de filtros son muy útiles en técnicas de *multirate processing* como solución a los filtros *anti-aliasing* y *anti-imaging*. También son muy comunes en otras técnicas de procesamiento en bandas como filtros *crossover*.

Filtros notch. La diferencia de fases producida por un filtro pasa-todo de orden 2, en el intervalo $[0, f_s/2]$ es de -2π. Esto quiere decir que existe un valor de frecuencia F_c para el cual el desfase es exactamente π. Así, si se implementa un sistema con función de transferencia

$$G(z) = \frac{1}{2}[1 + A(z)],$$

la evaluación en el punto correspondiente a F_c tendría atenuación infinita. En aplicaciones prácticas el espectro de un sistema $G(z)$ es como se muestra en la Figura 34.

Parametrizando los coeficientes de la función de transferencia de un filtro *notch* de orden dos como

$$G(z) = \frac{1}{2} * \frac{(1+k_2)+2k_1(1+k_2)z^{-1}+(1+k_2)z^{-2}}{1+k_1(1+k_2)z^{-1}+k_2z^{-2}},$$

se puede modificar la frecuencia F_c mediante el parámetro

$$k_1 = -\cos\left(\frac{2\pi F_c}{f_s}\right),$$

y el ancho de banda de -3 dB, BW, alrededor de F_c como

$$k_2 = \frac{1-\tan(\pi\, BW/f_s)}{1+\tan(\pi\, BW/f_s)}.$$

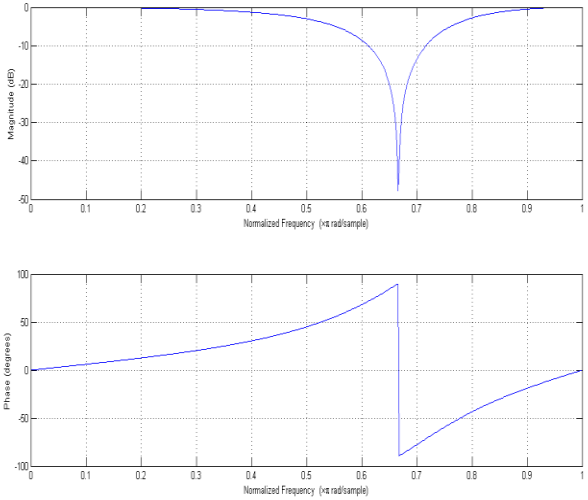


Figura 34. Espectro de un filtro *notch*

10.3. Resultados

En el archivo “guitarra_adaptativo.wav” del directorio “Audios” presente en el DVD anexo se puede apreciar una señal de guitarra capturada en un espacio ruidoso. La primera mitad del audio no ha sido procesada, mientras que la segunda ha sido modificada usando un filtro adaptativo. En este mismo directorio, en el archivo “notch.wav” puede escucharse una señal tipo *chirp* que ha sido filtrada usando un filtro *notch* implementado con un filtro pasa-todo. El archivo “chirp.wav” contiene la onda original.

11. MULTIRATE PROCESSING

Para la producción y el procesamiento musical profesional es necesario que la frecuencia de muestreo de las señales sea de al menos 96 kHz. Esto minimiza los efectos del *aliasing* que puedan presentarse en el proceso de grabación debido a la riqueza armónica de los sonidos capturados. Además permite que el producto final, re-muestreado a 44.100 Hz exceda un poco la respuesta del oído para obtener una calidad óptima. Pero este no es el único tipo de audio o de señal existente y dicha frecuencia de muestreo puede resultar excesiva para representar otro tipo de ondas –o componentes de otras más complejas- de menor ancho de banda. Para economizar recursos digitales del hardware se ha creado una técnica llamada *multirate processing* (o procesamiento en multi-frecuencia), que permite separar las señales en bandas de frecuencia para ser tratadas de acuerdo a sus requerimientos mínimos necesarios.

11.1. Objetivos

- Comprender e implementar las técnicas de: *up-sampling* y *down-sampling*
- Utilizar la CSL para el manejo de los pulsadores de la tarjeta
- Conocer la teoría de *re-sampling* y *delays* fraccionales
- Implementar filtros *anti-aliasing* y *anti-imaging*
- Conocer las ventajas del procesamiento en multi-frecuencia

11.2. Conceptos previos

En general los datos que ingresan al DSP han sido previamente acondicionados por el dispositivo de captura (códec de audio), de tal forma que las muestras puedan ser manipuladas directamente por el programador sin que este deba preocuparse por efectos como el *aliasing* o la cuantización. Sin embargo, si la aplicación específica requiere modificar dicha tasa de muestreo, la señal debe acondicionarse nuevamente desde el software antes de poder ser tratada.

Down-sampling. Este proceso permite reducir la frecuencia de muestreo de una señal de entrada f_s en un factor entero D. Según lo descrito por el teorema de Nyquist, la nueva frecuencia máxima representable será

$$f_N' = \frac{f_s}{2\,D},$$

y dado que el contenido de frecuencias de la señal original supera este nuevo límite, es necesario filtrar esta antes de realizar el proceso de *down-sampling* para evitar los efectos del *aliasing*. De esta manera, el filtro *anti-alias* requerido ha de tener una frecuencia de corte f_N' o inferior a esta, dependiendo de la respuesta del filtro.

Para implementar el *down-sampling* basta tomar una de cada D muestras de la señal filtrada. La Figura 35 muestra el diagrama de bloques correspondiente a este proceso.

Uno de los mayores usos de esta técnica es la implementación de filtros FIR o IIR, ya que la respuesta de los mismos mejora al disminuir la frecuencia de muestreo. Así, la respuesta de un filtro de menor orden puede superar la de uno de mayor orden implementado a una frecuencia de muestreo mayor –Figura 36.

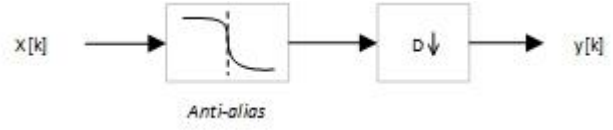


Figura 35. Diagrama de bloques del *down-sampling*

Up-sampling. Análogo al procedimiento anterior, esta técnica permite aumentar la frecuencia de muestreo de una señal de entrada en un factor entero L. Para implementar dicha técnica se requieren insertar $L-1$ ceros entre muestras de la señal original.

El cambio abrupto entre los datos originales y los ceros resulta en la aparición de componentes de alta frecuencia que deben ser removidos para conservar el ancho de banda. Los filtros empleados para esto se conocen como filtros *anti-imaging* y su efecto se traduce en la interpolación de los $L-1$valores presentes entre las muestras originales. Dichos filtros deben ser pasabajos con frecuencia de corte

$$f_N' = \frac{f_S L}{2}.$$

La Figura 37 muestra el diagrama de bloques básico para la implementación del *up-sampling*.

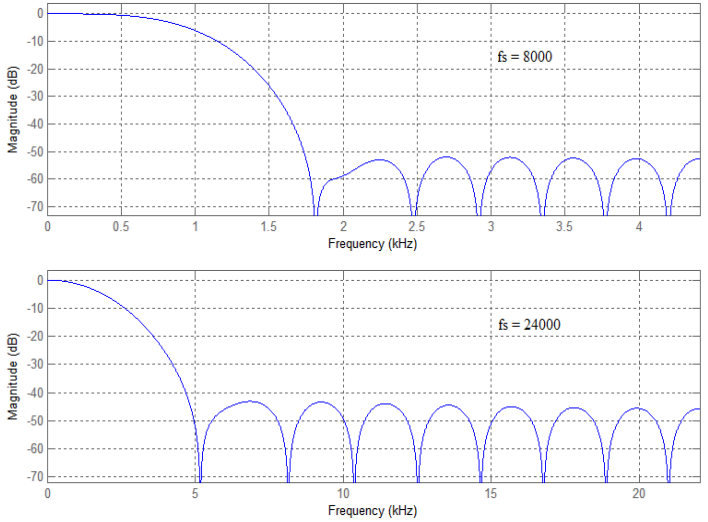


Figura 36. Comparación de la respuesta de un filtro FIR

Procesamiento por bandas. Las técnicas descritas anteriormente permiten descomponer y analizar una señal determinada en bandas de frecuencia. Para realizar esto es necesario implementar

el diagrama de bloques mostrado en la Figura 38. Esta técnica es ampliamente utilizada en algoritmos de compresión digital y en ecualizadores gráficos.

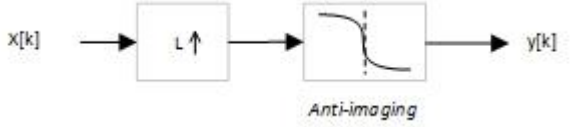


Figura 37. Diagrama de bloques del *up-sampling*

Delay fraccional. En algunas aplicaciones es necesario retardar una señal en una cantidad que no es múltiplo entero de la frecuencia de muestreo, como en *phasers*, *flangers* o en la generación de efectos binaurales. Para implementar un *delay* racional $R = N/D$, es necesario implementar un bloque de retardo de longitud N seguido de un bloque de *down-sampling* con factor D. Es importante que la señal que se quiere retardar R muestras, tenga las condiciones adecuadas en su espectro para que no se produzca *aliasing* en el proceso de *down-sampling.*

11.3. Resultados

En la Tabla 11 se puede apreciar la compresión y expansión del espectro de una señal cuadrada de 10 armónicos con frecuencia fundamental de 440 Hz. Se puede apreciar en dicha tabla que la frecuencia fundamental aparece en la 9ª, 36ª y 2ª muestra respectivamente, para una FFT de 512 puntos en cada caso. Debido a la simetría de la transformada de Fourier, solo se ha graficado medio espectro. Además se pueden apreciar componentes de alta frecuencia en los espectros, debido a la ausencia de ventanas previas al procesamiento de los datos.

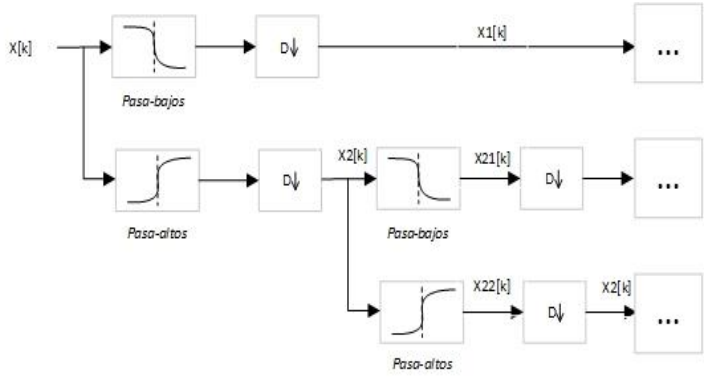
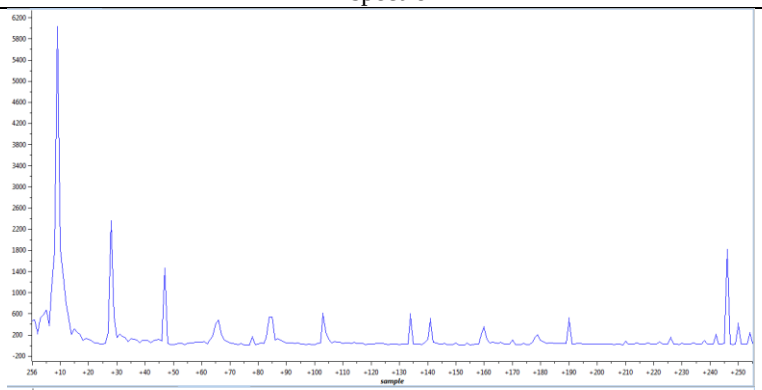
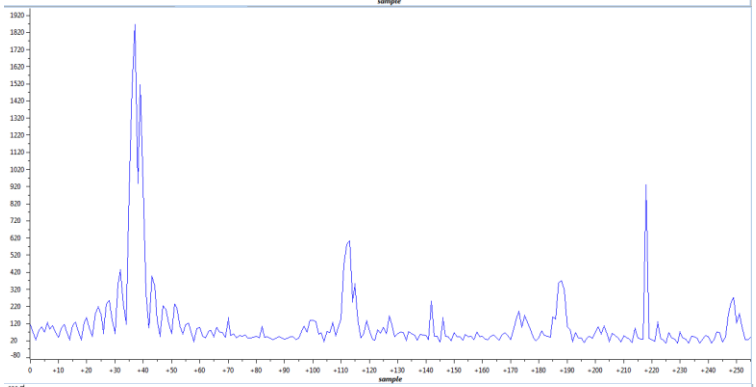
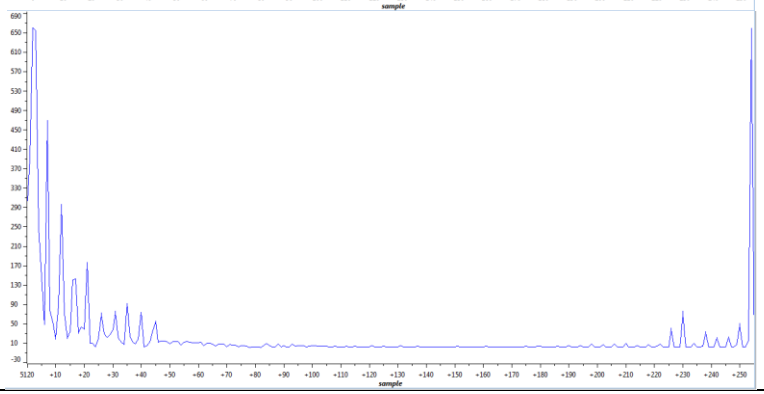


Figura 38. Diagrama de procesamiento en bandas

En los archivos "multirate_filter_24000.wav" y "multirate_filter_8000.wav" del directorio "Audios" presente en el DVD anexo se puede apreciar el resultado de filtrar la señal de una guitarra con un par de filtros FIR (con $f_c = 1\ kHz$ y de orden 10), con frecuencias de muestreo $f_{S1} = 24000$ y $f_{s2} = 8000$, respectivamente. Para poder comparar los resultados y apreciar claramente la diferencia de la respuesta en frecuencia, en el archivo "guitarra_multirate.wav" se encuentra el sonido original.

Tabla 11. Espectros de una señal aplicando técnicas de *multirate processing*

Proceso	Espectro
Bypass	
Down-sampling	
Up-sampling	

12. SUGERENCIAS ADICIONALES PARA EL DESARROLLO DEL CURSO

La elaboración de un curso no debe limitarse a la selección de los contenidos teóricos o prácticos del mismo. Diferentes aspectos como: la evaluación, las herramientas usadas para la enseñanza, la intensidad horaria y el contexto en el que se desarrolla deben tenerse en cuenta para que la transmisión del conocimiento sea óptima. Por esta razón se exponen a continuación otros aspectos que permitan la creación de un curso integral en procesamiento digital de señales.

12.1. Temas adicionales

Las secciones anteriores describen los temas fundamentales que deberían ser abordados en un curso básico en procesamiento digital de señales. Sin embargo, la duración de un semestre académico (aproximadamente 16 semanas) y la intensidad horaria semanal correspondiente a un curso de pregrado (4 o 5 horas), impiden el desarrollo de temas adicionales de gran valor académico y práctico en esta rama de la ingeniería.

Se deja a consideración del docente la profundización en los temas ya descritos o la inclusión de alguno de los temas que se proponen a continuación, en caso de que el desarrollo del curso lo amerite:

- Programación en lenguaje *assembly*: Familiarizar al estudiante con el set de instrucciones del lenguaje del dispositivo y comparar (donde sea pertinente) la eficiencia de este en la ejecución de un código
- *Boot* desde la memoria *Flash*: Revisar el procedimiento que permita cargar el programa desde la memoria *Flash* para aplicaciones autónomas
- Modulaciones en amplitud y frecuencia: Revisar los conceptos y técnicas de las modulaciones que permitan la implementación de estas desde el hardware
- Envolventes temporales: Implementar una moduladora ADSR desde el hardware para síntesis de señales u otras aplicaciones musicales
- Ecualizadores gráficos y paramétricos: Construir con ayuda de componentes externos diferentes ecualizadores que puedan ser usados en tiempo real y programados en el DSP
- Codificación y cuantización: Revisar las técnicas que permitan cambiar los niveles de cuantización de una señal y conocer los efectos y aplicaciones de esta técnica
- Algoritmos de compresión y expansión: Conocer los fundamentos matemáticos y aplicaciones de los algoritmos e implementarlos en el DSP
- STFT, transformada discreta del coseno, *wavelet transform* y otras transformadas para análisis espectral:

Mostrar las diferentes técnicas de análisis así como las ventajas y aplicaciones específicas de cada una

- Tramado: Revisar los efectos sobre el ruido de cuantización, de la adición de *Dither* a una señal

12.2. Trabajo final

Para evaluar el aprendizaje general del curso, se propone la realización de un trabajo final de tema libre de mayor complejidad que cada una de las prácticas realizadas. En este, se espera que los estudiantes empleen varios de los conceptos enseñados y que usen para ello los diversos módulos y librerías disponibles en el kit de desarrollo. Como:

- Leds, *display*, pulsadores, módulo SD
- Entradas y salidas de audio estéreo
- Dsplib
- CSL y los módulos: SPI, I2S, I2C, etc., para interactuar con elementos externos al kit, analógicos y digitales
- *Hardware accelerator*
- Puerto de expansión

Además, se sugiere que los temas tentativos de este trabajo sean presentados con anterioridad al docente a través de un anteproyecto, para que este evalúe su viabilidad y competencias. Es importante que dentro de su desarrollo, el trabajo final contenga la investigación de temas y elementos diferentes a los vistos durante el curso. Algunas ideas de trabajo, con un alcance similar al esperado por parte de los estudiantes, son:

- Compresor de audio MP3 (u otro formato)
- Detector de tiempo en piezas musicales
- Banco de efectos para guitarra (Saebi & Moses)
- *Time stretching*
- Afinador de guitarra automático
- Sintetizador musical

12.3. Evaluación

Ya que el desarrollo del curso se centra en los aspectos prácticos del procesamiento digital de señales, se sugiere una calificación que de mayor peso a esta parte del curso. Sin embargo, también es importante que sean evaluadas de alguna manera las bases teóricas tanto de sistemas y señales como del lenguaje C. En la Tabla 12 se muestran las actividades evaluativas que podrían llevarse a cabo durante el curso, así como los pesos sugeridos para cada una. Se propone que todas las prácticas tengan el mismo valor porcentual.

12.4. Cronograma de actividades

En la Tabla 13 se especifica el tiempo y orden sugerido para la enseñanza de las actividades del curso durante las 16 semanas del semestre académico. Además, se sugiere que la intensidad del curso sea de 5 horas semanales, separadas en un bloque de 2 horas y en otro de 3 horas. La intención de esta cantidad de horas, es que los conceptos previos y las prácticas se puedan completar en clase y el trabajo independiente sea mínimo.

Tabla 12. Porcentajes evaluativos

Tema	Peso
Quiz Sistemas y Señales	5 %
Quiz lenguaje C	5 %
Prácticas	70 %
Trabajo final	20 %

Tabla 13. Cronograma del curso

Actividad	Semana 1	Semana 2	Semana 3	Semana 4	Semana 5	Semana 6	Semana 7	Semana 8	Semana 9	Semana 10	Semana 11	Semana 12	Semana 13	Semana 14	Semana 15	Semana 16
Bases S y S																
Bases C																
Familiarización																
Síntesis																
FFT																
FIR																
IIR																
Adaptativos																
Multirate																
Final																

13. POTENCIAL

Las diversas instituciones académicas tienen la misión de estar constantemente en la búsqueda de tecnologías y temáticas pertinentes al contexto social, económico y político de los programas académicos para formar profesionales íntegros que puedan responder a las exigencias del medio. Por esto es importante para la Universidad Pontificia Bolivariana la presencia de una materia en procesamiento digital de señales contenida en el plan de estudios de Ingeniería Electrónica (y afines), tal y como sucede en otras grandes universidades del mundo, que contribuya al desarrollo de áreas como: aeronáutica, ingeniería civil, telecomunicaciones, informática, biomedicina, meteorología y por supuesto, audio.

Se espera que el desarrollo de este trabajo de grado siembre en la facultad y en los nuevos ingenieros un gusto por el audio que vaya mucho más allá de la creación y el estudio de un curso. Y que a partir de allí puedan gestarse semilleros y grupos de investigación de los que surjan proyectos e invenciones de gran impacto para nuestro medio.

14. CONCLUSIONES

Una de las herramientas que más facilitó el desarrollo del trabajo, fue la escogencia asertiva del módulo de desarrollo. No solo por sus capacidades, sino por la cantidad y excelente calidad de la documentación disponible, las librerías y los foros activos alrededor del TMS320C5535 eZdsp™.

Otro factor clave para la realización del trabajo, fue el uso del lenguaje C en lugar del lenguaje *assembly* para la programación de las prácticas. C permite una portabilidad y abstracción de códigos mucho más intuitiva y fácil. Además, facilita la interpretación de códigos ajenos, permitiendo así una mayor divulgación de información y contenidos, disponible abiertamente en la web. Esto permite un mejor y mayor avance al enfrentar problemas en cada aplicación.

El hardware usado responde a las necesidades académicas del curso, sin embargo, es deseable que se pueda contar con una herramienta de mayores prestaciones que permita abordar tópicos de mayor complejidad. Con esto dicho y con los temas adicionales que se propusieron, sería muy posible la construcción a futuro de un curso avanzado en procesamiento digital de señales. Estos posibles equipos podrían elegirse de la misma familia del hardware usado (como el TMS320C6455 o el TMS320C6747, de Texas Instruments), con la finalidad de extender fácilmente el curso desarrollado con los temas adicionales, además de servir óptimamente para el uso en grupos o semilleros de investigación.

El objetivo de la docencia y de este trabajo de grado va más allá de la enseñanza de los temas propuestos. Al desarrollar el proyecto, se hace notorio que aun cuando el área principal de estudio (el audio) es apasionante, algunos de los temas pueden resultar difíciles de explicar y de aprender. Por esto, aparte de pensar en el desarrollo de los contenidos de cada práctica, también fue importante la búsqueda de un método agradable y motivador de transmitir cada temática a los posibles estudiantes del curso.

Luego de finalizar la investigación y escritura del trabajo de grado se tiene la certeza de que se han cumplido satisfactoriamente los objetivos propuestos inicialmente. Sin embargo, al analizar minuciosamente los diversos componentes del proyecto pueden encontrarse puntos que no se tuvieron en cuenta desde el inicio o que pueden ser desarrollados con mayor profundidad. Esto no denota una incompletud del trabajo, por el contrario, fue la completud del mismo la que permitió el reconocimiento de temáticas que por su extensión y riqueza requerirían un curso de mayor extensión y alcances para ser abordadas adecuadamente.

REFERENCIAS

Aboagye, A. K. (1999). Overflow avoidance techniques in cascaded IIR filter implementations on the TMS320 DSP's. *SPRA509*. Dallas, Texas, Estados Unidos: Texas Instruments.

Embree, P. M. (1995). C algorithms for real-time DSP. Upper Saddle River, New Jersey, Estados Unidos: Prentice Hall.

Haykin, S. O. (2 de Junio de 2013). Adaptive Filter Theory. *5*. Upper Saddle River, New Jersey, Estados Unidos: Prentice Hall.

Kuo, S. M., & Lee, B. H. (2001). *Real-Time Digital Signal Processing. Implementations, aplications and experiments with the TMS320C55x.* Chichester, Inglaterra: John Wiley & Sons, LTD.

Lathi, B. P. (1998). *Signal Processing & Linear Systems.* Carmichael, California, Estados Unidos: Berkeley Cambridge Press.

Manolakis, D. G., & Ingle, V. K. (2011). *Applied Digital Signal Processing.* Cambridge, Reino Unido: Cambridge University Press.

Orfanidis, S. J. (Agosto de 1995). Introduction to Signal Processing. Upper Saddle River, New Jersey, Estados Unidos: Prentice Hall.

Proakis, J. G., & Manolakis, D. G. (1996). *Digital Signal Processing Principles, Algortihms, and Applications* (Tercera ed.). Upper Saddle River, New Jersey, Estados Unidos: Prentice-Hall, Inc.

Regalia, P. A., Mitra, S. K., & Vaidyanathan, P. P. (Enero de 1988). The Digital All-Pass Filter: A Versatile Signal Processing Building Block. *Proceedings of the IEEE, 76*(1), 19-37.

Saebi, P., & Moses, P. (s.f.). *Wireless Embedded Sensing System Lab*. Recuperado el 12 de Febrero de 2015, de http://wiesel.ece.utah.edu/redmine/projects/dfx/wiki

Smith, S. W. (1999). The Scientist and Engineer's Guide to Digital Signal Processing. *2*. San Diego, California, Estados Unidos: California Technical Publishing.

Texas Instruments Incorporated. (Agosto de 2001). TMS320C55x DSP Programmer's Guide. Dallas, Texas, Estados Unidos.

Texas Instruments Incorporated. (Agosto de 2011). TMS320C5535, 'C5534, 'C5533, 'C5532 Fixed-Point Digital Signal Processors. *SPRS737C*. Dallas, Texas, Estados Unidos.

Texas Instruments Incorporated. (Diciembre de 2011). TMS320C55x Optimizing C/C++ Compiler v 4.4. User's Guide. *SPRU281G*. Dallas, Texas, Estados Unidos.

Wilson, R. (Septiembre de 1993). Filter Topologies. *AES, 41*(9), 667-678.

AUTORES

Luis Eduardo *MARTÍNEZ FERRÍN*. Egresado próximo a graduarse del programa de Ingeniería Electrónica. Diplomado en docencia universitaria. Socio de la empresa de amplificadores, pedales y guitarras Lovell Musiclab, donde trabaja como Ingeniero Electrónico. Músico, guitarrista y melómano empírico.

Alejandro ROSSO PÉREZ. Egresado próximo a graduarse del programa Ingeniería Electrónica. Diplomado en docencia universitaria. Estudiante de música en la academia Studio Ensamble. Flautista, guitarrista y aprendiz de *Lutheria*.