UNIVERSIDAD DE EL SALVADOR

FACULTAD DE INGENIERÍA Y ARQUITECTURA

ESCUELA DE INGENIERÍA ELÉCTRICA

SÍNTESIS DE SONIDOS UTILIZANDO LA DSP BLACKFIN BF533

PRESENTADO POR:

RAÚL ANÍBAL ALBERTO MARTÍNEZ

NÉSTOR ALEXANDER MURILLO GÓMEZ

PARA OPTAR AL TÍTULO DE:

INGENIERO ELECTRICISTA

CIUDAD UNIVERSITARIA, SEPTIEMBRE 2016

UNIVERSIDAD DE EL SALVADOR

RECTOR INTERINO :

LIC. JOSÉ LUIS ARGUETA ANTILLÓN

SECRETARIA GENERAL :

DRA. ANA LETICIA ZAVALETA DE AMAYA

FACULTAD DE INGENIERÍA Y ARQUITECTURA

DECANO :

ING. FRANCISCO ANTONIO ALARCÓN SANDOVAL

SECRETARIO :

ING. JULIO ALBERTO PORTILLO

ESCUELA DE INGENIERÍA ELÉCTRICA

DIRECTOR :

ING. ARMANDO MARTÍNEZ CALDERÓN

UNIVERSIDAD DE EL SALVADOR

FACULTAD DE INGENIERÍA Y ARQUITECTURA

ESCUELA DE INGENIERÍA ELÉCTRICA

Trabajo de Graduación previo a la opción al Grado de:

INGENIERO ELECTRICISTA

Título :

SÍNTESIS DE SONIDOS UTILIZANDO LA DSP BLACKFIN BF533

Presentado por :

RAÚL ANÍBAL ALBERTO MARTÍNEZ

NÉSTOR ALEXANDER MURILLO GÓMEZ

Trabajo de Graduación Aprobado por :

Docente Asesor :

DR. CARLOS EUGENIO MARTÍNEZ CRUZ

San Salvador, Septiembre 2016

Trabajo de Graduación Aprobado por:

Docente Asesor :

DR. CARLOS EUGENIO MARTÍNEZ CRUZ

ACTA DE CONSTANCIA DE NOTA Y DEFENSA FINAL

En esta fecha, martes 13 de septiembre de 2016, en Laboratorio de Comunicaciones de la Escuela de Ingeniería Eléctrica, a las 9:00 a.m. horas, en presencia de las siguientes autoridades de la Escuela de Ingeniería Eléctrica de la Universidad de El Salvador:

Firma:

1. Ing. Armando Martínez Calderón
   Director

Firma:

2. MSc. José Wilber Calderón Urrutia
   Secretario

Y, con el Honorable Jurado de Evaluación integrado por las personas siguientes:

Firma:

1- Dr. Carlos Eugenio Martínez Cruz

2- MSc. Salvador de Jesús German

3- Ing. Walter Leopoldo Zelaya Chicas

Se efectuó la defensa final reglamentaria del Trabajo de Graduación:

SÍNTESIS DE SONIDOS UTILIZANDO LA DSP BLACKFIN BF533

A cargo de los Bachilleres:

- ALBERTO MARTÍNEZ RAÚL ANÍBAL
- MURILLO GÓMEZ NÉSTOR ALEXANDER

Habiendo obtenido en el presente Trabajo una nota promedio de la defensa final: 9.0

(NUEVE PUNTO CERO)

AGRADECIMIENTOS.

¡A Dios sea la Gloria y Honra! Con esta primera frase quiero agradecer profundamente a mi Dios, por permitirme terminar mi carrera universitaria, aunque fue largo el camino nunca me abandonó, proveyendo todo lo necesario a través de mis padres y amigos. Y en agradecimiento dispongo mi carrera para sus planes y sus proyectos porque me declaro fiel servidor suyo.

A mis padres, Víctor Alberto y Ester de Alberto quienes siempre confiaron y creyeron en mí, estando siempre atentos de las necesidades que la carrera me pedía y en muchas ocasiones con mucho esfuerzo y amor me proveyeron de todo lo necesario, dándome siempre lo mejor. A mis hermanas Carina Alberto y Jennifer Alberto por cuidarme, animarme y brindarme de su ayuda en todo tiempo. Y a mi demás familia que de una u otra forma estuvieron pendientes de toda esta travesía, muchas gracias.

Agradezco a mis amigos por estar siempre pendientes y darme de su apoyo, animándome a continuar y llegar a este día. Gracias amigos por sus oraciones. También agradezco a mis compañeros y en especial a mi compañero de trabajo de graduación Néstor Murillo por toda su colaboración y apoyo para alcanzar esta meta, les deseo a cada uno de ustedes todo lo mejor. Así también al grupo de profesores de la escuela de Ingeniería Eléctrica por compartir de su conocimiento y a mi asesor al Dr. Carlos Martínez por ayudarnos en esta última etapa de la carrera, a todos ellos muchas gracias.

Raúl Aníbal Alberto Martínez.

AGRADECIMIENTOS.

Primero quiero agradecer a mi Dios por permitirme llegar hasta ésta etapa de mi vida, que me ha cubierto en su manto sagrado desde que comencé ésta carrera, por darme la motivación, salud y paciencia necesaria para poder culminar este trabajo.

A mis padres, quienes formaron en mí una persona con valores, deseos de superación y cuyos consejos, además de su apoyo económico han sido parte esencial en el cumplimiento de este objetivo.

A mis hermanos que son un ejemplo a seguir y siempre me brindan su apoyo en cada aspecto de mi vida, y me incentivaron a terminar esta carrera a pesar de adversidades que en algún momento se presentaron.

A mi compañero de tesis y gran amigo Raúl Alberto, a quién he tenido el privilegio de conocer y compartir a lo largo de estos años como estudiante y haber formado un equipo de trabajo para la realización de ésta tesis.

A mi asesor Dr. Carlos Martínez, quién gracias a su ayuda y asesoría fueron de vital importancia en el éxito de ésta investigación, y también por su amistad y conocimientos brindados en sus cursos como catedrático de la universidad de El Salvador.

A mis compañeros de la universidad, a quienes tuve el honor de conocer en cada una de las materias de ésta carrera y por estar presentes hasta el final.

Néstor Alexander Murillo Gómez

ÍNDICE GENERAL

Capítulo 1: Introducción 13

1.1 Planteamiento del problema 14
1.2 Antecedentes 14
1.3 Justificación 15
1.4 Objetivos 15
1.4.1 Objetivo general 15
1.4.2 Objetivos específicos 16
1.5 Alcances 16
1.6 Organización 17

Capítulo 2: Descripción de hardware del KIT ADSZ-BF533 EZlite 18

2.1 Descripción de hardware 18
2.1.1 Unidad aritmética 18
2.1.2 Unidad aritmética de direcciones 19
2.1.3 Unidad de control 19
2.1.4 Archivo de registros (datos y direcciones) 20
2.1.5 Memoria 21
2.1.6 Pulsadores y Ledes 21
2.1.7 Temporizadores 22
2.1.8 Banderas Programables 23
2.1.9 Interrupciones 23
2.1.10 Acceso directo a memoria 24
2.1.11 Interfaz para periféricos serie e Interfaz para periféricos paralela 24
2.1.12 Controlador de puerto serial (SPORT). 24
2.1.13 Controlador de puerto UART 25
2.1.14 Interfaz de audio 25
2.1.15 Interfaz de video 26
2.2 Tareas especiales generadas por hardware 28
2.2.1 Modos de operación en el procesador ADSP-BF533 28
2.2.2 Tipos de reseteo 29
2.2.3 Métodos de arranque en el procesador ADSP-BF533 30
2.2.4 Administración dinámica de potencia del ADSP-BF533 30

Capítulo 3: Lenguaje de programación Chuck y entorno de desarrollo VisualDSP++ 32

3.1 Lenguaje de programación Chuck 32
3.1.1 Entorno de desarrollo de Chuck 32

3.1.2 Operadores y operaciones 34
3.1.3Tipos, valores y variables 35
3.1.4 Objetos 36
3.1.5 Unidades analizadoras (UAnae) 37
3.1.6 Unidades generadoras (UGens) 38
3.1.7 Librerías (API) 39
3.2 Entorno de desarrollo VisualDSP++ 40
3.2.1 Herramientas de desarrollo 40
3.2.2 Entorno de desarrollo 40

Capítulo 4: Resultados de la implementación de la síntesis modal en el software Chuck y en el KIT ADSZ-BF533 EZlite 47

4.1 Síntesis modal en software Chuck 47
4.1.1 Descripción de la síntesis modal en software Chuck 47
4.1.2 Resultado de la síntesis modal en Chuck 56
4.2 Síntesis modal en el KIT ADSZ-BF533 EZlite 60
4.2.1 Descripción de la síntesis modal en el KIT ADSZ-BF533 EZlite 61
4.2.2 Resultado de la síntesis modal en el KIT ADSZ-BF533 EZlite 73
4.3 Comparación de resultados 75

Capítulo 5: Conclusiones y líneas futuras 78

5.1 Conclusiones 78
5.2 Líneas Futuras 80

ANEXOS 81

Anexo A. Síntesis Modal 82
Anexo B. Resonador Biquad 87
Anexo C. Modelo “PIPELINE” 89
Anexo D. Arquitectura RISC y Arquitectura de memoria en procesadores Blackfin . 92
D.1 Arquitectura RISC 92
D.2 Arquitectura de memoria en procesadores Blackfin 93
Anexo E. Representación Fraccionaria 94

GLOSARIO 95

REFERENCIAS BIBLIOGRÁFICAS 99

LISTA DE FIGURAS

Figura 1. Diagrama del núcleo del procesador 20
Figura 2. Ledes y pulsadores de propósito general 22
Figura 3. Canales de entrada y salida de audio 26
Figura 4. Canales de entrada y salida de video 26
Figura 5. Ubicación de algunos de los elementos de la tarjeta 27
Figura 6. Entorno de desarrollo del lenguaje Chuck 33
Figura 7. Ejemplo del operador => en Chuck 34
Figura 8. Ejemplo de asignación de un valor a una variable en Chuck 34
Figura 9. Operaciones aritméticas en Chuck 35
Figura 10. Valores literales en Chuck 36
Figura 11. Definición de una nueva clase en Chuck 37
Figura 12. Inicialización de una unidad analizadora en Chuck 37
Figura 13. Ejemplo de la activación de la FFT cada 1024 muestras 38
Figura 14. Acceso a la configuración del parámetro frecuencia de la unidad generadora 38
Figura 15. Compilar y ensamblar [7] 40
Figura 16. Diagrama de enlazador [7] 41
Figura 17. Usando el cargador para crear un archivo de salida [7] 42
Figura 18. Ventana de proyectos de VisualDSP++ 43
Figura 19. Ventana de edición de proyectos en VisualDSP++ 44
Figura 20. Ventana de salida 45
Figura 21. Ventana de depuración mostrando el código ensamblador del proyecto 46
Figura 22. Inicio del script FFTFindModes.ck 49
Figura 23. Aplicación de la FFT y espectro de potencia 50
Figura 24. Sumatoria de los diferentes espectros de potencia [10] 51
Figura 25. Énfasis en las altas frecuencias 51
Figura 26. Búsqueda de modos en FFTFindModes.ck 52
Figura 27. Código en Chuck para imprimir los modos en consola 53
Figura 28. Inicio de la clase ModalSynth 53
Figura 29. Excitación para ModalResnth.ck 54
Figura 30. Resonadores Biquad en Chuck [10] 54
Figura 31. Configuración de parámetros de frecuencia, ganancia y Q 55

Figura 32. Función que calcula Q en ModalResynth.ck 55
Figura 33. Forma de onda del sonido de un perol metálico 56
Figura 34. 20 modos principales del audio: perol_metálico.wav 57
Figura 35. Forma de onda de “residue.wav” 58
Figura 36. Parte del espectro de potencia de perol_metálico.wav 59
Figura 37. Forma de onda del sonido sintetizado realizado en Chuck 59
Figura 38. Funciones inicializadoras en el main.c 61
Figura 39. Ilustración de la conexión entre la computadora y la DSP 62
Figura 40. Parte del código que atiende la interrupción al presionar SW4 63
Figura 41. Función Process_Data() 64
Figura 42. Función fft_btc() 66
Figura 43. Parte del código que atiende la interrupción al presionar SW5 67
Figura 44. Calculo de coeficientes del resonador Biquad 70
Figura 45. Resonadores Biquad en la DSP [10] 70
Figura 46. Parte del código en modos.c donde se calculan los coeficientes 71
Figura 47. Resto del código de los resonadores Biquad 71
Figura 48. Parte final del código que implementa los resonadores 72
Figura 49. Mensajes de inicio de la ejecución del proyecto 73
Figura 50. Modos encontrados por la DSP 74
Figura 51. Forma de onda del sonido sintetizado por la DSP 74
Figura 52. Mensaje para indicar que se ha presionado SW7 74
Figura 53. Comparación de los primeros 500 elementos de los espectros de Potencia para una FFT de 4096 puntos 75
Figura 54. 4 Modos principales en Chuck configurado a una FFT de 4096 muestras 76
Figura A.1 Sistema masa/resorte con amortiguamiento [12] 83
Figura A.2 Gráfica del resultado de la ecuación A.3 [12] 85
Figura A.3. Forma de onda y Espectro de Potencia en donde se pueden localizar los modos principales [12] 86
Figura A.4 Residuo y Espectro de Potencia [12] 86
Figura B.1 Diagrama de bloques del Biquad 88
Figura C.1 Ilustración del “PIPELINE” 90
Figura C.2 Diagrama del “PIPELINE” ejecutado en el Blackfin BF533 [3] 91
Figura D.1 Diagrama en bloques de arquitectura de memoria del procesador Blackfin 93

LISTA DE TABLAS

Tabla 1. Registros accedidos en modo usuario 28
Tabla 2. Métodos de arranque del procesador 30
Tabla 3. Asociación de archivos en carpetas del proyecto 44
Tabla 4. Coeficientes para el resonador Biquad 69
Tabla E.1 Representación fraccionaria fract16 94
Tabla E.2 Representaciones Fraccionarias 94

CAPITULO 1: INTRODUCCION.

En el presente trabajo de graduación, se implementan algoritmos básicos de procesamiento de señales en un KIT ADSZ-BF533 EZLITE, para sintetizar sonido partiendo de sus modos resonantes. Este KIT es una tarjeta de desarrollo y que está conformada por un procesador de la familia Blackfin y un conjunto de periféricos de entre los cuales se pueden mencionar entradas y salidas para audio y video analógicas.

Además este KIT cuenta con su respectivo entorno de desarrollo llamado VisualDSP++, en el cual se escriben, editan, depuran, compila y carga los diferentes códigos que conforman los proyectos que se ejecutan sobre la tarjeta. Estos códigos muy bien pueden estar escritos en lenguaje C/C++ y/o ensamblador. El lenguaje de programación a utilizar queda al criterio del programador para optimizar la aplicación, en este trabajo la aplicación se realiza en lenguaje C en su totalidad.

Además se estudia esta misma técnica de sintetizado que ha sido implementada en un lenguaje de programación llamado Chuck, el cual es un lenguaje orientado al procesamiento de audio en tiempo real. Este lenguaje le permite al usuario realizar procesamiento de audio por medio de sus diferentes objetos y clases ya definidas por defecto en sus librerías. Este estudio es una referencia con que son comparados los resultados obtenidos al sintetizar un mismo sonido.

Todo lo anterior es realizado para aperturar nuevas áreas de investigación, teniendo como principal elemento el KIT ADSZ-BF533 EZLITE de *Analog Devices* con que cuenta la escuela de ingeniería eléctrica y así aprovechar este tipo de recursos.

1.1 Planteamiento del problema.

Se busca implementar en una tarjeta de desarrollo del tipo Blackfin BF-533 algoritmos de sintetizado de sonidos. Las aplicaciones son de una algorítmica básica, basado en modelos sinusoidales.

Los algoritmos son simulados previamente en herramientas tipo software tales como: Matlab y Chuck, en donde se verifican los resultados obtenidos en el proceso de sintetizado. Luego las aplicaciones son escritas en lenguaje C en el entorno de desarrollo VisualDSP++, que es el software encargado de la comunicación entre el kit de desarrollo y el ordenador (PC).

1.2 Antecedentes.

Los trabajos de graduación en el área de procesamiento de señales han sido muy pocos comparados con otras áreas de Ingeniería Eléctrica. Uno de los factores en la baja realización de estos trabajos es el hardware necesario para realizar el procesado de señales. El último trabajo de graduación fue implementado en un procesador digital de señales (DSP, en sus siglas en inglés) DSP56L811, fabricado por Motorola Inc. En ese DSP se implementó un filtro adaptativo para la cancelación de ruido basado en el algoritmo LMS (*Least Mean Square*) [1].

El DSP DSP56L811 de Motorola está diseñado solamente para el análisis de señales en tiempo real, es decir, su potencial radica en que es un procesador aritmético pero con la desventaja que no es posible realizar aplicaciones que involucren señales de audio y video.

Entre las características más importantes de este DSP se encuentran: Eficiente dispositivo DSP de 16 bit, dos acumuladores de 36 bit (incluyendo los bits de extensión), un compilador de C eficiente y soporta variables locales, subrutinas e interrupciones, de longitud limitada y un reset externo para aplicar reset por hardware.

Este modelo fue comercializado alrededor del año 1996 y ya es considerado como obsoleto. Hoy en día existen DSP´s cuyos periféricos permiten la manipulación de señales que están relacionados con audio y video, como lo permite el KIT ADSZ – BF533 EZLITE de *Analog Devices*.

1.3 Justificación.

El procesamiento digital de señales es una rica y amplia área que hasta la fecha ha carecido de suficiente atención. Mediante el presente trabajo de graduación se busca abrir una importante línea de investigación que permita desarrollar aplicaciones necesarias en las más diversas áreas de la ingeniería eléctrica. Se selecciona el tema de sintetizar sonidos debido a que es un tema relevante, de interés para un público amplio y de una dificultad adecuada para un trabajo de graduación. El éxito de este trabajo pretende abrir paso a nuevos desarrollos, más sofisticados y orientados a la solución de problemas.

1.4 Objetivos.

1.4.1 Objetivo general.

- Sintetizar sonidos mediante un procesador digital de señales.

1.4.2 Objetivos específicos.

- Desarrollar algoritmos básicos de procesamiento de señales.
- Desarrollar algoritmos básicos de modelado de sonidos.
- Desarrollar algoritmos básicos de modelado de voz.
- Implementar algoritmos en un procesador de señales.
- Aplicar técnicas de modelado de fenómenos físicos.

1.5 Alcances.

Este trabajo de graduación pretende construir las bases necesarias en cuanto a conocimiento y manejo a nivel de hardware y de software del KIT ADSZ –BF533 EZLITE de *Analog Devices*, para que en un futuro se puedan realizar aplicaciones más avanzadas y sofisticadas que resuelvan problemas que impliquen el procesamiento digital de señales.

De igual manera introducir herramientas de procesado señales tal como lo es Chuck, el cual permite procesamiento de audio en tiempo real, sin la necesidad de adquirir un hardware específico, lo que permite desarrollar aplicaciones en un menor tiempo y costo dado a que no se tendría que trabajar con hardware.

También, se busca que el sonido resultante por medio del método de síntesis de sonido sea lo más parecido al sonido de origen, para ello es necesario que la aplicación basada en algoritmos logre extraer la mayor cantidad de características del sonido a sintetizar, como por ejemplo: su frecuencia, intensidad, duración, etc.

Y por último demostrar que la principal fortaleza del KIT ADSZ –BF533 EZLITE radica en el procesamiento en tiempo real, ofreciendo eficiencia y rapidez al momento de realizar diferentes tareas de procesamiento de audio

1.6 Organización.

El trabajo de graduación se divide en cinco partes, cada una de ellas claramente diferenciada por medio de capítulos. La primera parte consiste en la introducción y en la presentación de las generalidades de este trabajo.

El segundo capítulo presenta la descripción de los componentes de hardware que tienen el KIT ADSZ –BF533 EZLITE tales como: el procesador Blackfin y las partes que lo componen, banderas programables, memoria, interrupciones, entradas y salidas analógicas de audio y video, controladores y puertos.

La siguiente parte, en el capítulo tres se presenta el software que se ha utilizado a lo largo de este trabajo, los cuales son Chuck y el VisualDSP++. Chuck como un lenguaje de programación de procesamiento de audio y VisualDSP++ como el entorno de desarrollo empleado para enlazar la DSP y la computadora. Se menciona brevemente sus principales características que cada uno posee.

La cuarta parte o el capítulo cuatro se presentan los resultados obtenidos de las diferentes pruebas realizadas en Chuck y la DSP. De las cuales se exponen los datos arrojados; luego se procede a aplicar algoritmos básicos de procesamiento sobre la señal de entrada, y la comparación entre el sonido sintetizado resultante en Chuck y en la DSP.Y por último, el capítulo cinco contiene las conclusiones finales de este trabajo, así como las líneas futuras de investigación e implementación de nuevas aplicaciones siempre relacionadas con este tema. Todo esto es expuesto a continuación.

CAPITULO 2: Descripción de hardware del KIT ADSZ-BF533 EZLITE.

El KIT ADSZ-BF533 EZLITE, es una tarjeta de procesamiento digital, en donde se pueden desarrollar aplicaciones multimedia (audio y video). Este KIT está compuesto por un procesador (ADSP-BF533) de alto desempeño y varios periféricos, los que permiten obtener la señal digital (audio o video) a partir de una señal analógica. Tanto el procesador y los periféricos son controlados por medio de un conjunto de instrucciones y de configuraciones preestablecidas para diferentes modos de funcionamiento.

2.1 Descripción de hardware.

El procesador ADSP-BF533 es miembro de la familia de productos Blackfin, incorporados por *Analog Devices/Intel Micro Signal Architecture*[2]. Este procesador cuenta con un solo núcleo y se puede dividir en 4 partes: unidad aritmética, unidad aritmética de direcciones, unidad de control y archivos de registros (datos y direcciones).

2.1.1 Unidad aritmética.

La unidad aritmética, está compuesta por ALU's (unidad lógica aritmética), MAC's (multiplicador/acumulador), unidad de desplazamiento (*barrelshifter*), registros acumuladores y registros de datos [3].

Las ALU's (2 de 40 bits y 4 de 8 bits para video) son las unidades que están diseñadas para realizar operaciones aritméticas y lógicas para 16, 32 y 40 bits de punto fijo. Sus instrucciones incluyen operaciones tales como la suma, resta, AND, OR, NOT, funciones de valor absoluto, redondeo, etc.

Las MAC (16x16 bits) permiten realizar operaciones de multiplicación y multiplicación/acumulación de punto fijo. Sus instrucciones operan datos de 16 bits y produce resultados de 32 bits. Cada MAC tiene un acumulador de 40 bits.

2.1.2 Unidad aritmética de direcciones.

La unidad aritmética de direcciones (AAU) está formada por dos generadores de direcciones de datos (DAG0 y DAG1), archivo de punteros de registros y set de registros DAG (*Data Address Generators*). Esta unidad se encarga de generar las direcciones tanto de memoria como de puertos E/S. Las direcciones generadas son de 32 bits y ésta unidad puede generar o realizar búsquedas de 2 direcciones al mismo tiempo [3].

Los registros DAG están compuestos por 4 tipos de registros: registros índice I[0:3] (4 registros índices que van desde I0 hasta I3), registros modificador M[0:3] (4 registros modificadores partiendo desde M0 hasta M3), registros base B[0:3] (4 registros bases que van desde B0 hasta B3) y registros longitud L[0:3] (4 registros de longitudes partiendo desde L0 hasta L3). Cada uno de estos registros tiene una longitud de 32 bits.

Entre los archivos de punteros, ésta unidad contiene dos tipos diferentes. Uno es el puntero del stack o pila (SP) y el otro es el puntero de trama (FP).

2.1.3 Unidad de control.

Los componentes de la unidad de control son un secuenciador de programa, que es el encargado de controlar el flujo del código de un programa, es decir, provee constantemente la dirección de la siguiente instrucción a ser ejecutada por otras partes del procesador. Además maneja estructuras tales como: lazos, subrutinas, saltos, interrupciones e inactividad (Idle).

2.1.4 Archivos de registro (datos y direcciones).

Los Archivos de registros (datos y direcciones), son registros que se utilizan de manera general para manejar direcciones (registros P [0:5]) (6 registros de direcciones que van desde P0 hasta P5) y para datos (registros R [0:7]) (8 registros de datos partiendo desde R0 hasta R7). El ancho de cada uno de estos registros es de 32 bits, aunque se pueden utilizar como registros independientes de 16 bits (parte alta y baja).

En la Figura 1 se muestra un diagrama de bloque en donde se ilustra el núcleo del procesador.

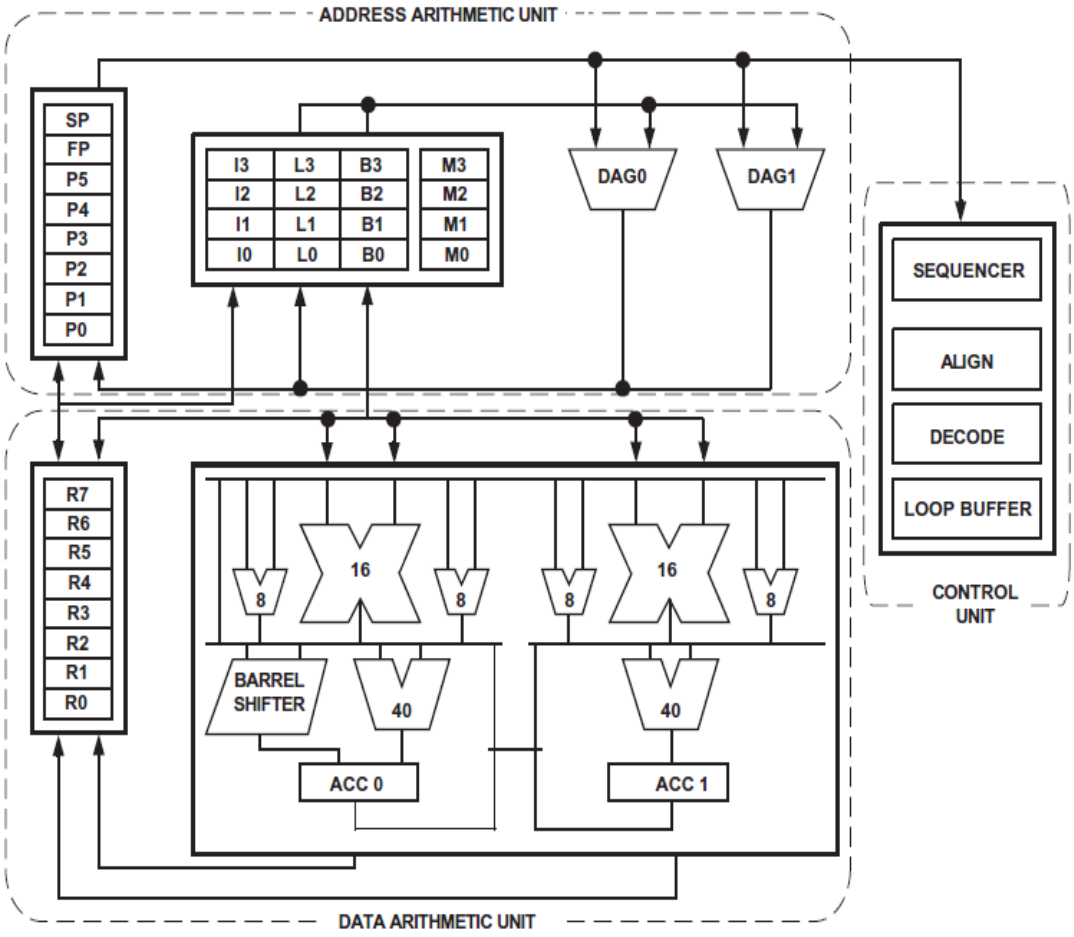


Figura 1. Diagrama del núcleo del procesador.

2.1.5 Memoria.

En el ADSP-BF533, la memoria es del tipo jerárquico, en donde la memoria de Nivel 1 (L1) es la única que está dentro del chip (On-Chip) y la de Nivel 2 (L2) se encuentra afuera del chip (Off-Chip).

La memoria de Nivel 1 está conformada por tres partes: memoria para las instrucciones, memoria para datos y "*scratchpad*" [3].

La memoria de instrucciones L1, se utiliza para almacenar instrucciones. El núcleo puede leer instrucciones mediante un bus de 64 bits de ancho. Y su capacidad para almacenar instrucciones es de 80 Kbyte de memoria SRAM (*Static Random Access Memory*).

La memoria de Datos L1, es donde se guardan datos y son dos bancos SRAM de 32 Kbyte (Banco de Datos A y Banco de Datos B). La memoria "*Scratchpad*" se utiliza para almacenamiento datos y su tamaño es de 4 KB [3].

La memoria de Nivel 2 o memoria externa, se le accede mediante la Unidad de Interfaz de Bus Externo (EBIU). Esta memoria es tipo SDRAM y la conexión entre ella y la EBIU es de 16 bits.

2.1.6 Pulsadores y Ledes.

La tarjeta cuenta con 4 pulsadores y 6 Ledes para propósitos generales. Los pulsadores (SW4-SW7) se pueden configurar mediante las banderas programables (PF8 – PF11) y los Ledes (LED 04 – LED 09) mediante la configuración de la memoria flash. Cabe mencionar que la memoria flash cuenta con dos bancos de memoria (flash A y flash B) y para cada uno de ellos hay dos puertos.

Específicamente el grupo de Ledes se controlan por el puerto B del banco Flash A. En la Figura 2 se muestra los pulsadores y Ledes de propósito general.

Figura 2. Ledes y pulsadores de propósito general.

2.1.7 Temporizadores.

La tarjeta posee 3 temporizadores para propósito general, además de un temporizador del núcleo y un temporizador tipo "*watchdog*". Los temporizadores de propósito general pueden ser configurados para trabajar en modo modulación por ancho de pulso (PWM), conteo y captura de ancho de pulso (WDTH) y en modo de evento externo (EXT).

El temporizador del núcleo es usado para generar interrupciones periódicas para una variedad de funciones de tiempo del sistema.

El temporizador *watchdog* se usa para implementar funciones de guardián. Un temporizador *watchdog* por software puede mejorar la disponibilidad del sistema al generar un evento al núcleo sí es que el temporizador expira antes de que sea actualizado por software.

El procesador cuenta con un reloj interno y que puede llegar a trabajar hasta a 133 MHz con un oscilador de 27 MHz. Sí se trabaja con este reloj del procesador (SCLK) el periodo máximo para el conteo del temporizador es de 32.2 segundos [3].

2.1.8 Banderas programables.

La tarjeta utiliza el sistema de banderas programables para manipular algunas de las funciones de sus periféricos. En total cuenta con 16 banderas (PF). Entre algunas funciones que forman parte de este sistema se pueden mencionar: Los pulsadores para propósito general (SW4-SW7), configuración para el codificador y decodificador de video (ADV7171 y ADV7183), y pines para el SPI y PPI [3].

Para configurar a cada una de estas banderas programables, la tarjeta cuenta con registros de 16 bits, entre los cuales están: registro de datos (FIO_FLAG_D), registro de estado (FIO_FLAG_S), registro de limpieza (FIO_FLAG_C) y registro de dirección (FIO_FLAG_DIR).

2.1.9 Interrupciones.

Cuando se declara a un periférico como una entrada, este periférico podría hacer una solicitud de interrupción. Siendo una interrupción un evento que cambia el curso de la ejecución actual del procesador y pasa a ejecutar un código especifico.

Para controlar las diferentes interrupciones que se podrían dar, el procesador utiliza un controlador de interrupciones del sistema (SIC) y un controlador de eventos del núcleo (CEC). Estos dos controladores establecen la prioridad y controlan las interrupciones del sistema. Cuando existe una petición de interrupción, el registro SIC_ISR guarda esa petición hasta que sea atendida, sí se ha habilitado al periférico para realizar peticiones (SIC_IMASK). Los controladores determinan la prioridad de esta con referencia a otras que pudieran estar sin atender [3].

2.1.10 Acceso directo a memoria.

El procesador ADSP-BF533 usa el Acceso Directo a Memoria (DMA) para transferir datos dentro de los espacios de memoria, o entre un espacio de memoria y un periférico. El procesador puede especificar las operaciones de transferencia de datos y retornar al procesamiento mientras el controlador integrado DMA lleva a cabo las transferencias de los datos independientemente de la actividad del procesador [3].

2.1.11 Interfaz para periféricos serie e Interfaz para periféricos paralela.

SPI es una interfaz serie síncrona full-duplex, que está compuesto por cuatro pines (dos pines de datos, seleccionador y de reloj), y funciona como un registro de desplazamiento en serie en donde transmite y recibe bits de datos (un bit a la vez).

Hay varios dispositivos que se pueden conectar a esta interfaz entre los cuales están otros microcontroladores, convertidores A/D y D/A, pantallas LCD y FPGA [3].

La interfaz PPI es half-duplex con capacidad para 16 bits de datos. Para un mayor rendimiento se deberá de trabajar con datos de 8 bits, porqué estaría manipulando dos muestras al mismo tiempo [3].

2.1.12 Controlador de puerto serial (SPORT).

En la tarjeta se encuentran dos puertos serie síncronos idénticos (SPORT0 y SPORT1). Estos puertos proporcionan interfaces tanto para entrada como de salida a una amplia variedad de dispositivos. Cada uno de ellos cuenta con dos grupos de pines, los cuales se programan por separado.

Cada uno de los puertos (SPORT0 y SPORT1) tiene la capacidad de transmitir serie de datos de entre 3 a 32 bits de longitud, también permiten el control de la interfaz de audio (I^2S y TDM) [3].

2.1.13 Controlador de puerto UART.

El transmisor/receptor universal asíncrono (UART) convierte los datos entre los formatos de serie y paralelo. Cada una de las palabras de datos que serán transmitidas o recibidas por el UART debe de tener un bit de inicio y al menos un bit de parada. Este periférico es full-duplex, pero soporta también el protocolo IrDAhalf-duplex [3].

2.1.14 Interfaz de audio.

Esta interfaz está compuesta por un códec de audio AD1836, el cual ofrece tres canales de audio estéreo de salida y dos canales de entrada en audio estéreo. La frecuencia de muestreo pueden ser de 48 y 96 kHz con resoluciones de 16, 24 y 32 bits.

Este códec posee cuatro canales ADC (*Analog-to-Digital Converter*), configurados como dos pares estéreo independientes y seis canales DAC (*Digital-to-Analog Converter*) como tres pares estéreo independientes. Cada canal de salida tiene su propio atenuador programable, ajustable en 1024 pasos lineales [5].

El procesador puede transferir datos al códec de audio mediante 2 modos. El primero de ellos es por interfaz serie de dos cables (TWI) y en este modo el códec puede operar a una frecuencia de 96 kHz, pero limita sus salidas a solamente dos.

El segundo modo es el modo multiplexado por división de tiempo (TDM) y en este modo a diferencia del anterior, su frecuencia es de 48 kHz, pero con la ventaja de utilizar los tres canales de salida [4]. En la Figura 3 se identifican cada una de las entradas y salidas de audio.

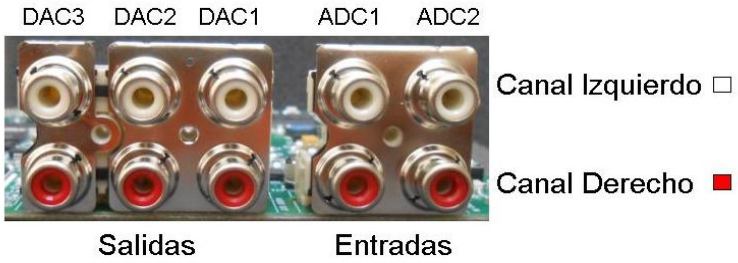


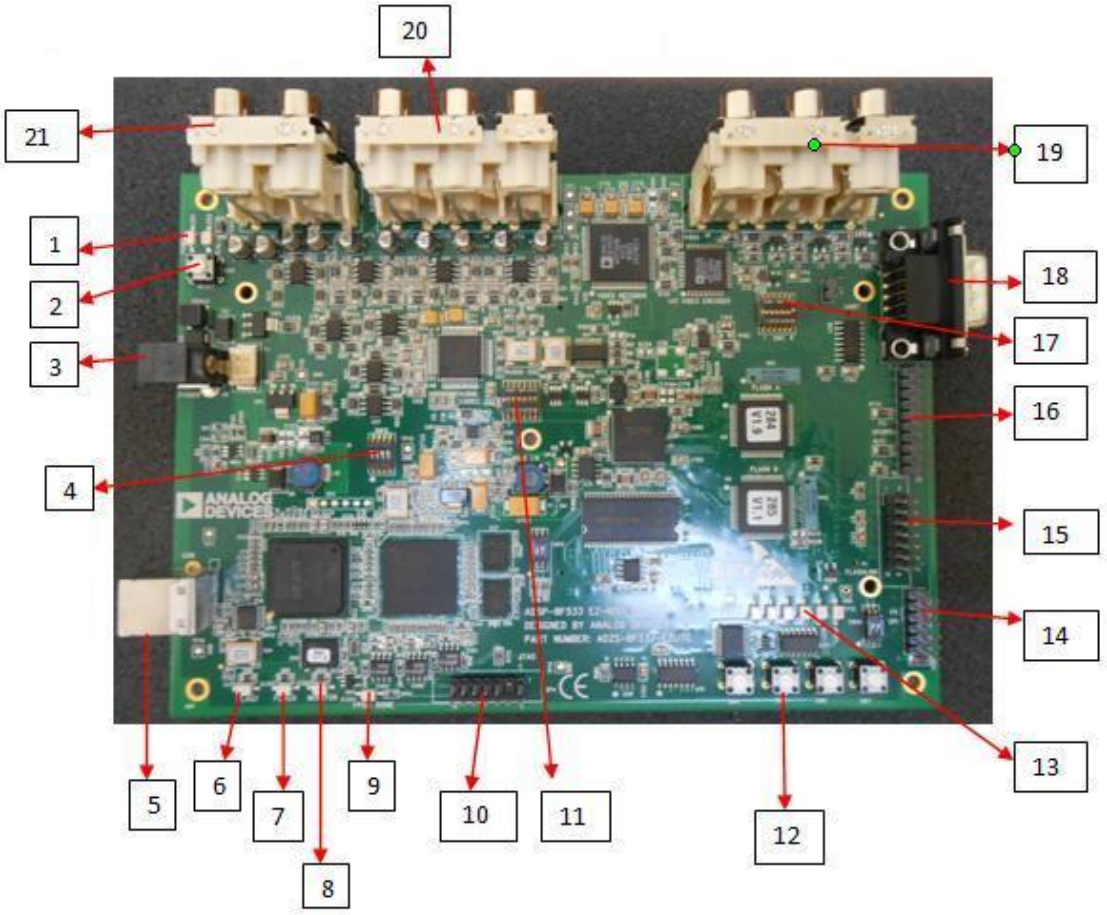
Figura 3. Canales de entrada y salida de audio.

2.1.15 Interfaz de video.

La tarjeta cuenta con un codificador de video (ADV7171) que proporciona tres canales de salidas analógicas y un decodificador de video (ADV7183) qué de igual manera proporcionan tres canales de entrada de video analógicas.

El codificador de video requiere un reloj de referencia de 27 MHz para el funcionamiento estándar (CCIR601). Cuando funciona en modo de píxeles cuadrados requiere una entrada de reloj de 24.54 MHz para NTSC y para la operación de PAL, se requiere una entrada de reloj de 29.5 MHz. En la configuración PAL el decodificador soporta 768 píxeles activos por línea y en NTSC soporta 640 píxeles activos por línea [4]. El decodificador de igual manera necesita de un reloj fijo de 27 MHz. Y automáticamente detecta si es NTSC/PAL la señal de entrada. La Figura 4 muestra las tres entradas y tres salidas que conforman la interfaz de video.

F*igura 4.* Canales de entrada y salida de video.



1. Ledes indicadores de encendido (Power) y reset.
2. Pulsador para reset.
3. Conector para fuente de potencia.
4. DIP Switch 12 para configurar SPORT0.
5. Connector USB.
6. ZLED1 indicador para FLAG0.
7. ZLED2 indicador para FLAG1.
8. ZLED3 monitoreo de la comunicación vía USB.
9. ZLED4 indicador para FPGA.
10. Conector JTAG.
11. DIP Switch 9 para configurar pulsadores y audio.
12. Pulsadores (SW4-SW7) de Propósito General.
13. Ledes (LED04-LED09) de Propósito General.
14. Conector SPI.
15. Conector Flash LINK.
16. Conector SPORT1.
17. DIP Switch 3 para configuración de video.
18. Puerto UART.
19. Entrada y Salida de Video.
20. Salidas de Audio.
21. Entrada de Audio.

Figura 5. Ubicación de algunos de los elementos de la tarjeta.

2.2 Tareas especiales generadas por hardware.

2.2.1 Modos de operación en el procesador ADSP-BF533.

Como se ha mencionado antes este KIT posee un procesador de la familia Blackfin de *Analog Devices*, este tipo de procesador posee tres modos de operación: Modo Usuario, Modo Supervisor y Modo Emulación.

En el modo usuario es el que normalmente se utiliza para aplicaciones y programas. El procesador se encuentra en este modo cuando no está en reposo o de reinicio, y cuando no está atendiendo a una interrupción, excepción o evento de emulación. Además este modo se utiliza para procesar código de nivel de aplicación que no requiere acceso explicito a los registros del sistema. Cualquier intento de acceder a los registros del sistema restringidos provoca un evento de excepción. La Tabla 1 muestra los registros que pueden ser accedidos en modo usuario.

Registros del Procesador	**Nombre de Registro**
Registros de Datos	R[0:7], A[0:1]
Registro de Punteros	P[0:5], SP, FP, I[0:3], M[0:3], L[0:3]. B[0:3]
Registros de Secuencia y Estado	RETS, LC[0:1], LT[1:0], LB[0:1], ASTAT, CYCLES, CYCLES2

Tabla 1. Registros accedidos en modo usuario.

El modo supervisor tiene acceso completo y sin restricciones a todos los recursos del sistema del procesador, incluyendo todos los recursos de emulación. Y por último el procesador puede entrar en modo emulación sí se produce un evento de emulación externa o sí la instrucción EMUEXCPT se emite.

El procesador permanece en modo emulación hasta que la rutina de servicio ejecuta una instrucción RTE, el procesador cambia a modo usuario.

Además de los eventos que causan que el procesador entre a un modo de operación específico, hay dos estados en los cuales no hay ningún tipo de procesamiento; entre ellos están: el estado de reseteo y el estado de inactividad o IDLE [3].

2.2.2 Tipos de reseteo.

El procesador del ADSP-BF533 tiene cinco formas de resetear el núcleo [3]:

1. *Reseteo mediante hardware.* Este reseteo tiene el propósito de reiniciar el núcleo del procesador y los demás periféricos, así como el controlador dinámico de manejo de poder (DPMC).
2. *Reseteo del software del sistema.* Este tipo de reseteo no afecta al núcleo, tampoco se inicia la secuencia de arranque; sino más bien solo resetea los periféricos y la mayoría de los DPMC.
3. *Reseteo por temporizador "watchdog*". Programando el temporizador causa un reseteo tanto para el núcleo y los DPMC, se puede leer el registro SWRST, para conocer si fue realmente el temporizador "watchdog", el que ha causado el reseteo en el núcleo.
4. *Reseteo por doble falla del núcleo.* Este reseteo causa que el núcleo y los periféricos se reinicien debido a que el núcleo del procesador ha entrado en un estado de doble error, en este caso también el registro de reseteo por software (SWRST) puede ser leído para determinar si el reseteo ha sido debido a una doble falla en el núcleo.

5. *Reseteo por software del núcleo.* En este caso solo se resetea el núcleo del procesador, los periféricos se mantienen en el estado en el que se encuentran en ese momento.

2.2.3 Métodos de arranque en el procesador ADSP-BF533.

Esta tarjeta contiene en su procesador, un kernel el cuál configura un periférico específico para que el núcleo del procesador arranque.

Existen 4 métodos para el arranque, los cuales se detallan a continuación en la Tabla 2, con las posibles combinaciones que pueden tomar estos bits del registro BMODE [0:1]:

Combinación de bits	**Método de arranque**
00	Se ejecuta desde la memoria externa de 16 bits
01	Es efectuado desde la memoria flash externa de 8 o 16 bits
10	Trabaja como un dispositivo esclavo SPI, y un microprocesador es usado como anfitrión para arrancar el procesador ADSP-BF533
11	Usa una ROM de arranque para configurar el código de arranque y carga de la EPROM serie SPI (8, 16, o rango de direcciones de 24 bits)

Tabla 2. Métodos de arranque del procesador [3].

2.2.4 Administración dinámica de potencia del ADSP-BF533.

La funcionalidad en cuanto a la administración de potencia que el procesador Blackfin de esta tarjeta puede ofrecer es: control de voltaje y controlador de administración dinámico de potencia.

El control de voltaje, está basado en el consumo de potencia de la tarjeta de acuerdo a lo que requiera alguna aplicación en particular.

El controlador de administración dinámico de potencia, también es conocido como DPMC, posee cuatro modos de operación, entre ellos están: modo Full On, modo Active, modo Sleep y modo Deep Sleep [3].

En el modo *Full On*, es el que la tarjeta por medio de su procesador normalmente utiliza para la ejecución de las aplicaciones, se obtiene el máximo desempeño, esto quiere decir todo funciona a la máxima velocidad.

En el modo *Active*, se hace el siguiente ajuste: CCLK = SCLK = CLKIN, esto quiere decir que la frecuencia del reloj de entrada es igual tanto para el reloj del núcleo como para el reloj del sistema.

El modo *Sleep*, es aquel en el cual el acceso directo a memoria (DMA) es permitido pero solo a la memoria externa, debido a CCLK es decir, el reloj del núcleo es deshabilitado. Para que este modo regrese al estado Active o Full On es necesario un evento de *WAKEUP*.

El modo *Deep Sleep*, el procesador entra en un estado donde el núcleo del procesador y todos los demás periféricos son deshabilitados y la única forma de regresar de este modo es únicamente haciendo un reseteo de hardware.

A parte de los cuatro modos de operación anteriormente mencionados, hay un estado de hibernación en el que se apaga la alimentación interna pero mantiene los dispositivos de E/S asíncronos encendidos. Cuando un periférico no es usado, puede deshabilitarse para disminuir el consumo de potencia, esto se logra deshabilitando el reloj de un periférico en específico.

CAPITULO 3: Lenguaje de programación Chuck y entorno de desarrollo VisualDSP++.

En este capítulo, se da a conocer el lenguaje de programación Chuck, en el cual se ha implementado el algoritmo de la síntesis modal, que sirva de referencia para comparar los resultados que se han obtenido del desarrollo de este mismo algoritmo en el KIT de desarrollo.

Luego de esto, se describen los elementos que componen al entorno de desarrollo VisualDSP++ del KIT ADSZ-BF533 EZLITE.

3.1 Lenguaje de programación Chuck.

Chuck es un lenguaje de programación de propósito general, destinado para síntesis de audio en tiempo real y programación de gráficos multimedia [6]. Fue desarrollado por Ge Wang y Perry Cook en el año 2007, actualmente existen diferentes cursos en algunas universidades de Estados Unidos, donde este lenguaje y sus aplicaciones están siendo enseñadas. Las plataformas que soportan a Chuck son: MacOs X, Linux y Windows. Puede ser descargado y utilizado gratuitamente.

3.1.1 Entorno de desarrollo de Chuck.

La miniAudicle es un entorno de desarrollo integrado y liviano para el lenguaje de programación Chuck. Este entorno puede ser utilizado junto a la consola y la maquina virtual o como un entorno independiente sí se desea manejar por líneas de comando la ejecución.

La Máquina Virtual, es la encargada de permitir la modificación del código del programa mientras se está ejecutando, sin detener o reiniciar y experimentar durante el tiempo de ejecución. A este tipo de programación se le conoce por "*On the Fly*" [6].

Chuck utiliza una consola para interactuar con el usuario, sobre ésta consola son presentadas las tareas ejecutadas como por ejemplo la depuración de los códigos, y la impresión de texto por medio de los operadores "<<<" y ">>>". En la Figura 6 se muestra cada una de las ventanas que conforman el entorno de desarrollo para Chuck.

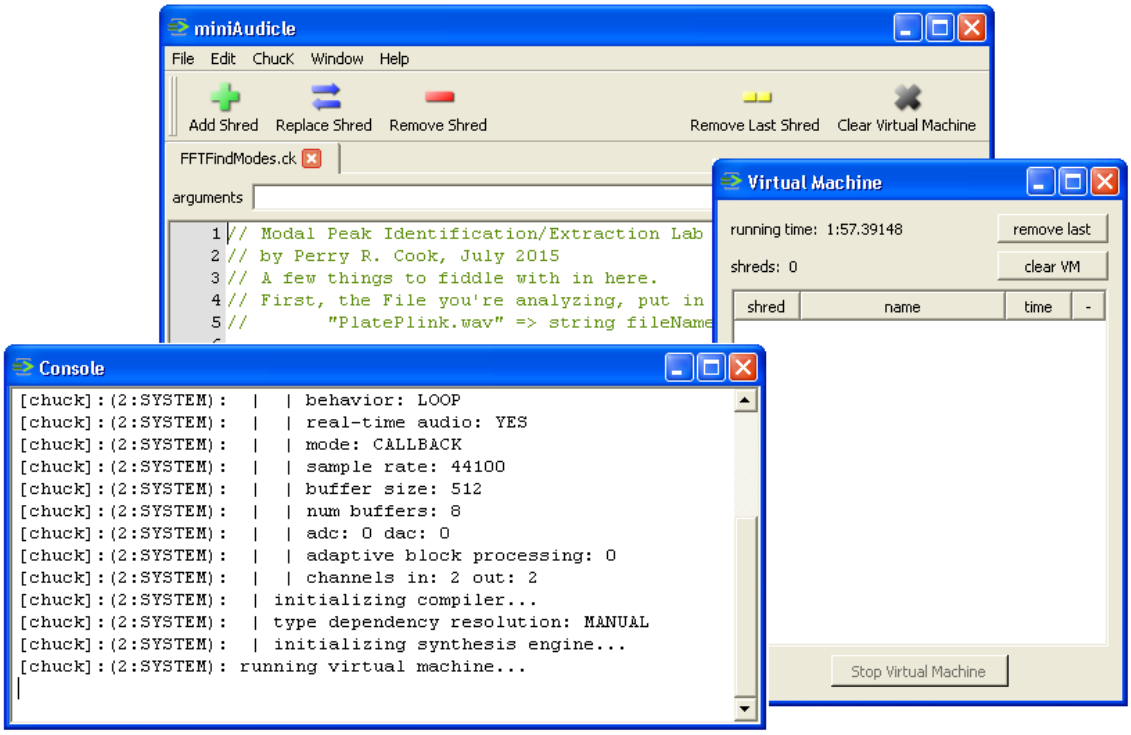


Figura 6. Entorno de desarrollo del lenguaje Chuck.

3.1.2 Operadores y operaciones.

El lenguaje Chuck, tiene sus propios operadores, bloques de construcciones como las unidades analizadoras (*Unit Analyzers*), unidades analizadoras de objetos (*UAna Objects*), unidades generadoras (*Unit Generators*) y librerías estándares (*API*). Chuck tiende a sobrecargar masivamente el operador "=>", dependiendo del tipo de dato que éste envuelve, permite varias acciones.

Un ejemplo de cómo se utiliza este operador se muestra en la Figura 7:

```
// una unidad generadora, el flujo de señal es evidente
// en este caso, => conecta dos unidades generadoras
SinOsc b => Gain g => Biquad f => dac;
```

Figura 7. Ejemplo del operador => en Chuck.

En Chuck, no hay ningún operador de asignación estándar (=), como lo hay en otros lenguajes de programación. Sí se desea asignar un valor a una variable, se debe de utilizar el operador Chuck [6], tal como lo muestra la Figura 8:

```
// asigna 4 a la variable foo
4 => int foo;
```

Figura 8. Ejemplo de asignación de un valor a una variable en Chuck.

Solo hay una limitante para este operador y es que solo puede ser utilizado para los tipos primitivos (float, int, time, dur y void), para los tipos objetos y clases se debe de utilizar el operador "@=>", este operador también funciona de la misma manera con los tipos primitivos de Chuck. Para realizar operaciones aritméticas se utilizan los operadores ya conocidos seguidos del operador Chuck. Para las operaciones de resta y división se debe de tener el cuidado porque no son operaciones conmutativas.

Por ejemplo en la Figura 9 se muestra la manera correcta de realizar estas operaciones:

```
//agregar 4 a la variable foo y asignar el resultado en foo
4 +=> foo;
//restar 10 a foo y asignar el resultado en foo
//recuerde esto es (foo-10) y no (10-foo)
10 -=> foo;
//duplicar el valor de foo y asignar el resultado en foo
2 *=> foo;
//dividir 4 entre foo y guardar el asignar en foo
//recuerde esto es (foo/4) y no (4/foo)
4 /=> foo;
```

Figura 9. Operaciones aritméticas en Chuck.

Al igual que en otros lenguajes de programación Chuck permite utilizar operadores lógicos entre dos operandos, dando como resultado 0 o 1. Y operaciones de bits tales como el desplazamiento [6].

3.1.3 Tipos, valores y variables.

Chuck es un lenguaje fuertemente tipado, lo que significa que los tipos de datos se resuelven en tiempo de compilación. Este tipo de sistema ayuda a imponer precisión y claridad en el código, y, naturalmente, se presta a la organización de programas complejos [6].

Los tipos primitivos o intrínsecos son los que son datos simples (que no tienen atributos de datos adicionales). Los tipos primitivos de Chuck son: int (entero con signo), float (punto flotante de doble precisión), time (tiempo), dur (duración) y void (sin tipo).

Los valores literales son explícitamente expresados en el código y son asignados a un tipo por el compilador.

La Figura 10 muestra algunos ejemplos de valores literales:

```
//int
      42
//int (hexa)
      0xaf30
//float
      1.323
//dur
5.5::second
```

Figura 10. Valores literales en Chuck.

Las variables son ubicaciones en la memoria que contienen datos. Las variables tienen que ser declaradas en Chuck antes de ser utilizadas. Es posible asignar un valor a una varibale como ya se mencionó con anterioridad utilizando el operador Chuck (=>).

3.1.4 Objetos.

En Chuck se hace el manejo de objetos bajo las convenciones de Java y C++ [6]. Esto significa: definición de clases personalizadas como un nuevo tipo de objeto e inicializar este objeto, Chuck soporta herencia polimórfica, todas las variables son referencia de objetos (como en Java), pero en instancia se asemeja a C++, hay una librería de clases por defecto y todos los objetos heredan de la clase (como en Java).

Dentro de Chuck ya existe un conjunto de clases definidas previamente, entre las cuáles pueden encontrarse: *Object* (clase base para todos los objetos de Chuck), *Event* (mecanismo base de sincronización), *Shred* (base abstracta para un proceso no primitivo en Chuck) y *UGen* (clase base para unidad de generación).

Las clases encapsulan un conjunto de comportamientos y datos. Para definir un nuevo tipo de objeto, la palabra reservada "*class*" se utiliza seguida del nombre de esta clase.

Por ejemplo la Figura 11 muestra la definición de una clase:

```
//definición de clase
class X
{
      //insertar código
}
```

Figura 11. Definición de una nueva clase en Chuck.

3.1.5 Unidades Analizadoras (UAnae).

Las unidades analizadores son bloques de construcción de análisis, y realizan la función de analizar las señales de audio y/o metadatos de entrada, y producen metadatos de salida como resultados del análisis.

Unidades analizadoras pueden ser enlazadas entre ellas y juntamente con unidades generadoras para lograr así un red de análisis /síntesis. Una de las principales características de este tipo de unidades, es que es necesario utilizar una función "*upchuck()*" para activarlas. Estas unidades son objetos y necesitan ser inicializadas antes de ser utilizadas. La forma de cómo se declaran este tipo de objetos se observa en la Figura 12:

```
// Transformada rápida de Fourier, asignada a una variable f
FFT f;
```

Figura 12. Inicialización de una unidad analizadora en Chuck.

En cualquier programa en Chuck, es necesario avanzar en el tiempo con el fin de sacar muestras de audio a través de las unidades generadoras y crear sonido. Además, es necesario para desencadenar los cálculos de análisis de forma explícita a fin de que se realice cualquier análisis. Para desencadenar explícitamente el cómputo en un punto en el tiempo, la función *upchuck()* es requerida.

Para ejemplificar lo anteriormente dicho se presenta el siguiente ejemplo en la Figura 13, en donde se calcula la FFT cada 1024 muestras [6]:

```
adc => FFT fft => dac;
//ajusta el tamaño de la FFT para ser de 2048 muestras
2048 => fft.size;

while (true){
//luego que pasen 1024 muestras
      1024::samp => now;
//desencadenara el cálculo de la FFT sobre las últimas 2048
      fft.upchuck();
}
```

Figura 13. Ejemplo de la activación de la FFT cada 1024 muestras.

Entre algunos ejemplos de este tipo de bloques ya definidos predeterminadamente en Chuck se tienen: IFFT, DCT, RMS, RollOf, IDCT, Windowing, Flux, etc.

3.1.6 Unidades Generadoras (UGens).

Este tipo de objeto son similares a las UAnae, solo que estas generan señales. Una unidad generadora puede poseer cero o más parámetros de control, para ajustar estos parámetros es necesario utilizar el operador Chuck, por ejemplo en la Figura 14 se muestran estos ajustes:

```
//conecta un oscilador seno hacia el dac
SinOsc osc => dac;
//ajusta la frecuencia de osc a 60.0 Hz
60.0 => osc.freq;
```

Figura 14. Acceso a la configuración del parámetro frecuencia de la unidad generadora.

Entre las unidades generadoras predefinidas en Chuck se encuentran: dac (convertidor digital/analógico), adc (convertidor analógico/digital), Noise (generador de ruido blanco), Impulse, ZeroX (detector de cruces por cero), Biquad (filtro de dos polos y dos ceros), Filter (filtro generico), etc.

3.1.7 Librerías (API).

Chuck provee un conjunto de librerías por defecto, estas librerías contienen funciones muy parecidas a la que se pueden encontrar en C. Por ejemplo la librería "*Std*" incluye funciones como un generador de números aleatorios, valor absoluto, convertidores de unidades, etc.

Otras librerías incluidas por defecto en Chuck se tiene: Math en donde se encuentran funciones que realizan operaciones como truncamiento, exponencial, logaritmo, raíz cuadrada, identidades trigonométricas y constantes matemáticas [6].

3.2 Entorno de desarrollo VisualDSP++.

El KIT ADSP-BF533 EZ-KIT Lite, trae consigo un entorno de desarrollo propio, éste software es el VisualDSP++. Este entorno permite realizar variedad de tareas sobre la tarjeta, posee una gran capacidad de edición de código fuente, opciones en la construcción de proyectos, flexibilidad sobre el espacio de trabajo, es decir permite construir y depurar múltiples proyectos en una sola sesión. Además permite depurar códigos escritos en C, C++ o en ensamblador.

3.2.1 Herramientas de desarrollo.

VisualDSP++ incluye las siguientes herramientas de desarrollo:

- ✓ Compilador.
- ✓ Librerías de rutinas de matemáticas, DSP y C.
- ✓ Ensamblador.
- ✓ Enlazador.
- ✓ Archivador.
- ✓ Separador.
- ✓ Cargador.
- ✓ Simulador.

Cada una de ellas realizan una tarea en especifico, el proceso comienza con los archivos escritos en C, C++, o ensamblador. El compilador organiza cada secuencia de instrucciones o datos dentro de las secciones, que se convierten en el principal componente sobre el cual actúa el enlazador.

El primer paso para producir un archivo ejecutable es compilar o ensamblar archivos fuentes en C, C++ o ensamblador dentro de un archivo objeto. El software de desarrollo VisualDSP++ asigna una extensión .doj para este tipo de archivo. En la Figura 15 se muestra un diagrama para ilustrar este proceso.

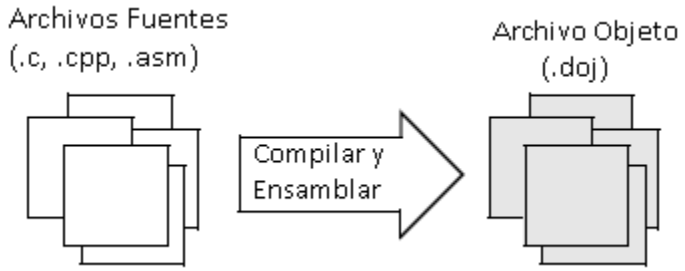


Figura 15. Compilar y ensamblar [7].

El archivo objeto producido por el compilador y por el ensamblador consiste en sí mismo por secciones de entrada. Cada sección de entrada contiene un tipo particular de código fuente compilado/ensamblado. Por ejemplo, una sección de entrada puede consistir en datos o código de programa, como las variables de diferentes anchuras.

Algunas secciones de entrada pueden contener información que permita la depuración a nivel de fuente. El enlazador mapea cada sección de entrada a un segmento de memoria, un intervalo continuo de dirección de memoria en el sistema de la tarjeta.

Cada sección de entrada en el LDF requiere un único nombre, tal como se especifica en el código fuente. Dependiendo sí la fuente es C, C++, o ensamblador, diferentes convenciones son usadas para nombrar a las secciones de entrada.

Luego de tener (compilado y) ensamblado los archivos fuentes en un archivo objeto, se usa el enlazador para combinar los archivos objeto dentro de archivos ejecutables. Por defecto, el desarrollo de software tiene la extensión .dxe.

La Figura 16 muestra un diagrama del enlazador en el VisualDSP++.

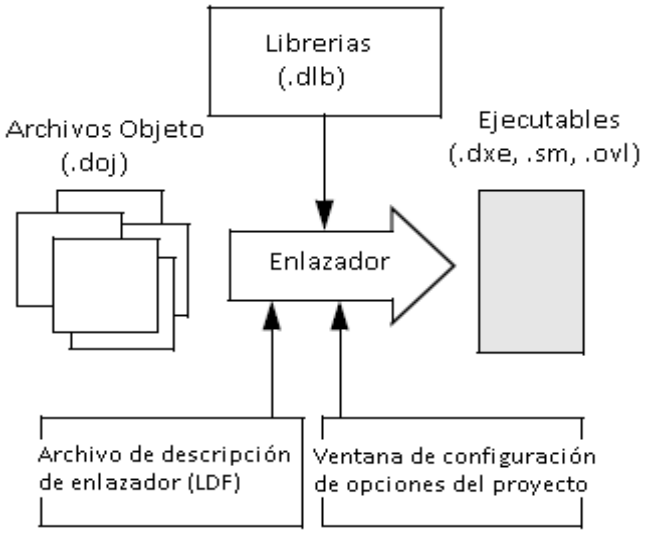


Figura 16. Diagrama de enlazador [7].

Después de depurar el archivo .dxe, se le procesa a través de un cargador o un separador para crear archivos de salida utilizados por el procesador. En general los procesadores Blackfin utilizan el cargador para obtener una imagen de arranque que se puede cargar (archivo .ldr), que reside en la memoria externa al procesador. Para hacer un archivo que pueda cargarse, el cargador procesa los datos de un archivo de inicio del kernel (.dxe) y uno o más archivos ejecutables (.dxe). La Figura 17 ilustra lo anterior:

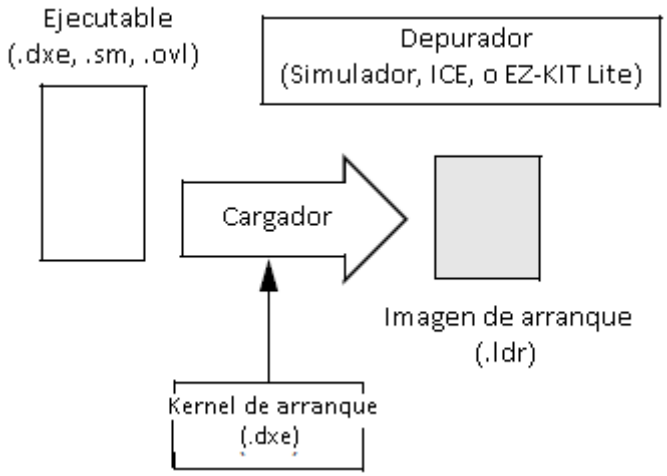


Figura 17. Usando el cargador para crear un archivo de salida [7].

Ahora es necesario mencionar la utilidad del archivador, el cual combina archivos objetos e indexa (o cualquier tipo de archivo) para producir un archivo de librería de búsqueda. Realiza las siguientes operaciones, según las indicaciones de opciones en la línea de comandos del archivador: crea un archivo de librería a partir de archivos objetos, añade uno o más archivos objetos en un archivo de librería existente, elimina un archivo de una librería, extrae un archivo de una librería e imprime el contenido de los archivos objeto de un archivo de librería a la salida estándar.

Y el simulador es un modelo en software del procesador y una de sus principales ventajas es que no necesita un hardware externo para ejecutar los proyectos.

El modelo de software solo simula el procesador, por lo que hace difícil simular con precisión un sistema complejo que envuelve más que un procesador [8].

3.2.2 Entorno de desarrollo.

VisualDSP++ es una intuitiva, y fácil de usar interfaz de programación para procesadores *Analog Devices*. La interfaz está compuesta por cuatro tipos de ventanas: Ventana de proyectos, Ventana de edición, ventana de salida y ventana de depuración [8].

La ventana de proyectos es el área donde el cual se muestran en una forma jerárquica los archivos que conforman un proyecto (.c, .cpp, .asm, .h, .ldf, etc.), VisualDSP++ permite abrir varios proyectos a la vez, pero solo uno de ellos puede estar activo, evitando confusiones al momento de compilar y cargar un proyecto al procesador.

La Figura 18 ilustra la ventana de proyectos de VisualDSP++.

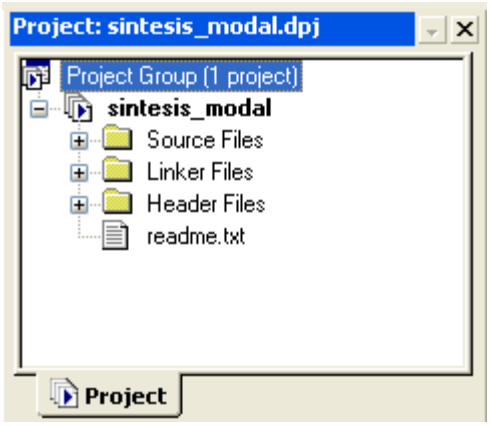


Figura 18. Ventana de proyectos de VisualDSP++.

Esta ventana cuenta con un menú (clic derecho sobre la ventana) en donde le permite al programador crear nuevos proyectos, abrir proyectos, cerrar todos los proyectos, entrar a las opciones o permitir que la ventana sea flotante. VisualDSP++ permite agregar archivos a un proyecto por medio del menú despegable de la ventana o con solo arrastrar el archivo a la carpeta del proyecto respetando la asociación de archivos. La Tabla 3 muestra ésta asociación.

Carpeta	Asociación por defecto
Archivos fuente	.c, .cpp, .cxx, .asm, .dsp, .s
Archivos de cabecera	.h, .hpp, .hxx
Archivos de enlazador	.ldf, .dlb, .doj
Archivos del kernel	.vdk

Tabla 3. Asociación de archivos en carpetas del proyecto.

La ventana de edición, permite editar los archivos que conforman al proyecto. Se pueden abrir cuántas ventanas de edición se desea, solo es necesario dar un doble clic sobre el archivo en la ventana de proyecto. Esta ventana utiliza diferente color de fuente para diferenciar los comentarios, palabras reservadas y string, así como funciones como cortar, copiar, deshacer y pegar. Esta ventana es mostrada en la Figura 19.

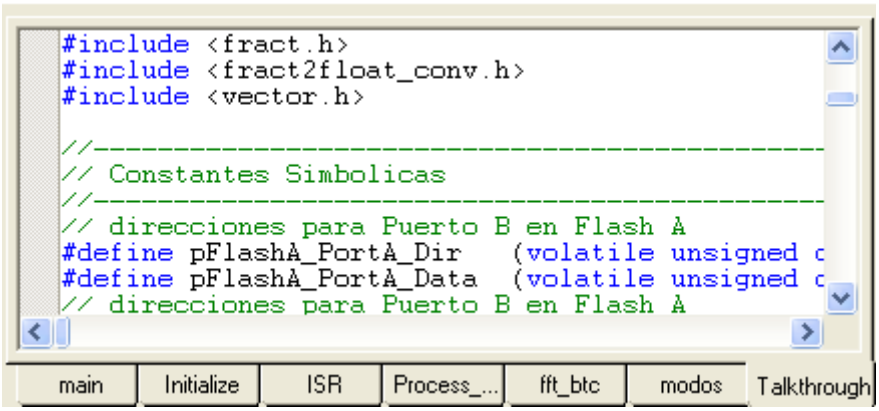


Figura 19. Ventana de edición de proyectos en VisualDSP++.

La ventana de salida, es por donde el VisualDSP++ muestra mensajes de información sobre la compilación y carga de proyectos. Sobre esta ventana existen dos pestañas: la pestaña "*Build*" y "*Console*". La primera pestaña muestra los mensajes de error que se generan al momento de depurar el proyecto y la pestaña de "*Console*" permite ver la salida estándar de los programas C/C++.

Sí se encuentra un error en algunos de los archivos fuentes al momento de depurar el proyecto, se genera una notificación sobre este error, la sintaxis utilizada en cada uno de estos mensajes es comenzando a identificar la herramienta que género la notificación, por ejemplo: ar (archiver), cc (compiler), ea (assembler), el (expert linker), id (idde), li (linker), pp (preprocessor), si (simulator) y xml (custom board support) [8].

Luego de identificar la herramienta se presenta un código de identificación, seguido de la descripción del error. VisualDSP++ ofrece más información sobre cada uno de los errores que él puede generar y para visualizar esta ayuda se debe sombrear el código del error que aparece en la ventana de salida y presionar F1. La Figura 20 muestra como VisualDSP++ presenta los mensajes de error en la ventana de salida.

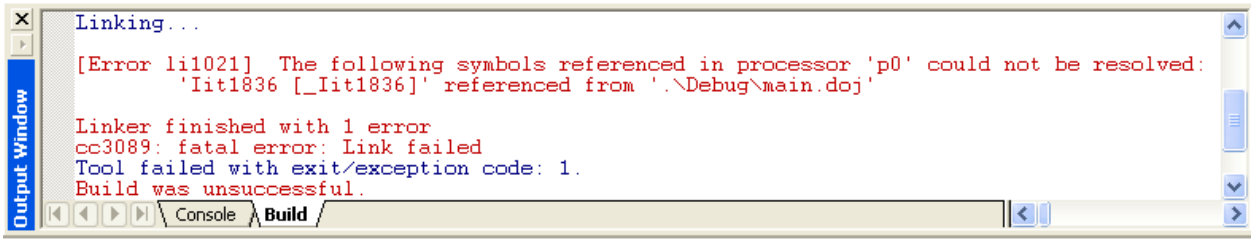


Figura 20. Ventana de salida.

Y por último la ventana de depuración, en si la ventana de depuración no es una sola ventana, sino que son ventanas en donde se muestran la operación del programa en el procesador y los resultados. Entre algunos de los resultados que se pueden presentar se tienen: el código en lenguaje ensamblador (Figura 21), historial de la actividad del procesador durante la ejecución, valores actuales de los registros del sistema, contenido de los canales de telemetría en tiempo real y porcentaje de tiempo utilizado en procesos idle.

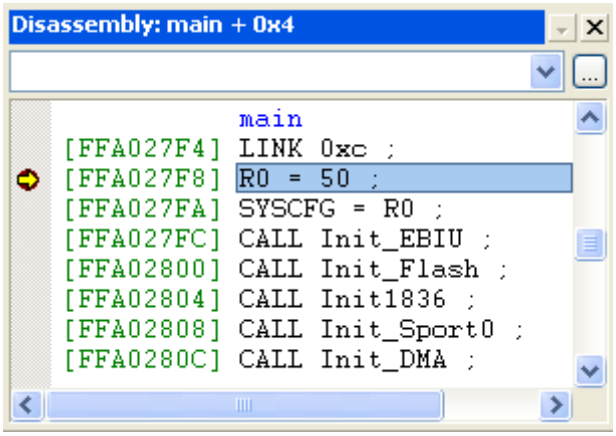


Figura 21. Ventana de depuración mostrando el código ensamblador del proyecto.

VisualDSP++ ofrece un amplio entorno de desarrollo, así como de herramientas que permiten depurar, optimizar y visualizar el comportamiento del proyecto en el procesador, todo esto hace que sea un complemento perfecto a la variedad de periféricos y hardware con que está equipado el KIT.

CAPITULO 4: Resultados de la implementación de la síntesis modal en el software Chuck y en el KIT ADSZ-BF533 EZlite.

En este capítulo se exponen los resultados obtenidos al implementar la síntesis modal en el lenguaje de programación Chuck y en el KIT ADSZ-BF533 EZlite. Iniciando con una breve descripción de cada una de estas aplicaciones para tener en consideración al momento de comparar dichos resultados.

4.1 Síntesis modal en software Chuck.

Brevemente la síntesis modal consiste en sintetizar un sonido a partir de la obtención de los modos resonantes del sonido original, cada modo resonante es representado por una frecuencia y una magnitud. La identificación de los modos resonantes de un sonido se realiza sobre el espectro de potencia de la señal.

4.1.1 Descripción de la síntesis modal en software Chuck.

El desarrollo de esta técnica de sintetizado fue realizado por Perry Cook en el año 2015, y consiste en 2 script: FFTFindModes.ck y ModalResynth.ck. El primero de ellos es el encargado de la adquisición de muestras de audio, procesar esas muestras para obtener el espectro de potencia y a partir de ahí buscar los modos resonantes. Y el segundo script tiene como labor sintetizar el sonido a partir de los modos resonantes utilizando resonadores Biquad.

Uno de los primeros pasos es la adquisición de muestras, el script FFTFindModes.ck requiere un archivo de audio (.wav) previamente grabado y guardado junto en la misma carpeta donde se encuentra el código fuente.

Actualmente existen muchos programas gratis en donde se puede fácilmente realizar esta grabación, y para tal propósito se ha utilizado el software Audacity [9], el cual además de ser gratis, ofrece muchas herramientas de análisis que no se abordarán en este trabajo.

La frecuencia de muestreo con que trabaja Chuck es variable, esta frecuencia puede ser modificada ingresando al menú de preferencias de la ventana miniAudicle. Y para este propósito se establece a una frecuencia de muestreo de 48 kHz.

La pista de audio es asignada a un *string* con el nombre de "*fileName"*, en este script se coloca una sentencia *if*, esto por sí se introduce argumentos por medio de la consola. Se define el número de modos que se buscarán. Este número puede variar sin ningún problema, solo con la variante que a mayor número de modos el sonido será más similar al original.

Además se declaran algunos arrays en donde se guardarán la localización de los picos y la magnitud de estos y una variable más llamada "*LOBE*", la cual se menciona su utilidad más adelante.

Luego de esto, se declara un buffer de sonido "*in*", el cual está asignado a una unidad analizadora de objetos (*UAna Objects*), esta unidad realiza la transformada rápida de Fourier y tiene como nombre "*fft*". Se obtiene la dirección de la ubicación de la pista de audio, es leída y guardada en el buffer de audio. Hay una sentencia para controlar que la pista de audio existe, sí esta sentencia se cumple automáticamente se termina el programa.

En la Figura 22 se muestra una parte de este script, en donde se realizan las instrucciones necesarias para ejecutar las tareas anteriormente mencionadas y la declaración de algunas variables.

```
"perol metalico.wav" => string fileName;
if (me.args()) me.arg(0) => fileName;

20 => int NUM_PEAKS;

int peakloc[NUM_PEAKS];
float peaks[NUM_PEAKS];
16 => int LOBE;

SndBuf in => FFT fft => blackhole;
me.dir()+fileName => in.read;
if (in.samples() < 1)  me.exit();
```

Figura 22. Inicio del script FFTFindModes.ck

Ahora para obtener el espectro de potencia de la señal capturada en la etapa anterior, es necesario utilizar la transformada rápida de Fourier sobre la muestras de audio. A partir de la magnitud de la FFT se consigue el espectro de potencia.

Entonces una vez que se han guardado las muestras, se necesita ahora configurar algunas variables que servirán al momento de realizar la transformada rápida de Fourier. El tamaño o la cantidad de muestras para la FFT, será de 2^{14} muestras (16,384), a un mayor tamaño, mejor será la aproximación en frecuencia. Se utiliza la ventana Blackman-Harris y su tamaño a un cuarto del tamaño de la FFT.

En Chuck, al término del script FFTFindModes.ck se genera un archivo .wav, en el cual se guarda la señal resultante, luego de habérsele quitado los modos principales a la señal original, para este proceso, es necesario utilizar una *UAna Objects* que realice la IFFT.

En la Figura 23, se observa el operador "@=>", el cual asigna a "*blob*" para que sea él que guarde el resultado de la FFT, "*blob*" es una *Uana Objects*, la cual es un objeto que guarda datos o resultados asociados con una unidad analizadora como lo es la FFT.

```
for (int i; i < NUM_FRAMES; i++)  {
    (SIZE/8) :: samp => now;
    fft.upchuck() @=> blob;
    blob.fvals() @=> float mag_spec[];
    for (0 => int i; i < SIZE/2; i++)    {
        mag_spec[i]*mag_spec[i] +=> histo[i];
    }
    numBufs++;
```

Figura 23. Aplicación de la FFT y espectro de potencia.

En Chuck este tipo de objetos (*UAna Objects, Unit Analyzers y Unit Generators*) poseen diferentes funciones, dependiendo del tipo de objeto que sea. Un objeto como "*blob*" se puede tener acceso a los datos que él guarda por medio de una función (el tipo de dato que guarda depende de la unidad analizadora con que esté vinculado). La instrucción "blob.fvals()" accede a los datos tipo flotante que "*blob*" almacena, estos datos se guardan en un array también del mismo tipo (flotante) llamado "*mag_spec[]*", estos datos ya son la magnitud de la FFT de la pista de audio.

Se utiliza un lazo *for* para calcular el espectro de potencia de la señal (pista de audio), y se guardan en "*histo[]*", por ser la FFT una señal periódica cada 2π, "*histo[]*" solo es la mitad del tamaño total de la FFT, es decir 8,192 elementos.

Dado, a que el rango dinámico de frecuencias de la señal con la que se está trabajando es pequeño, se puede realizar una sumatoria de cada uno de los diferentes resultados del espectro de potencia, teniendo poca repercusión en el resultado final.

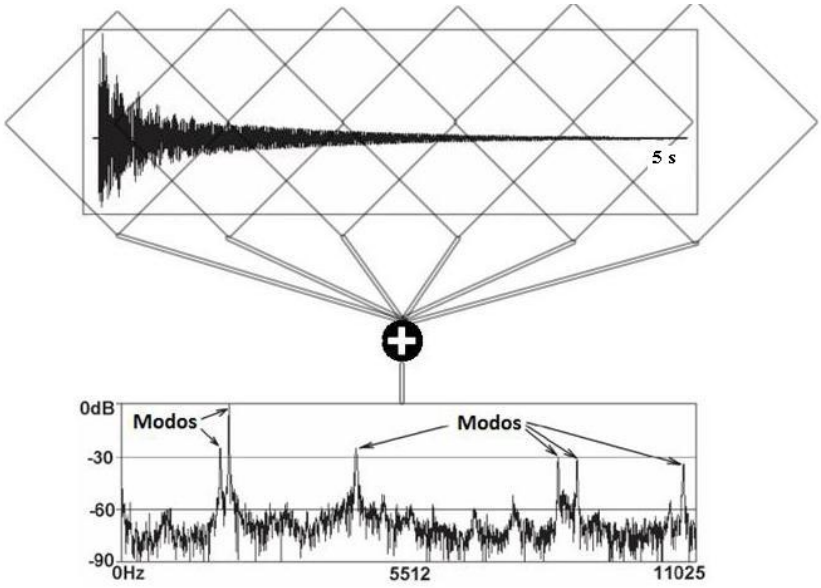


Figura 24. Sumatoria de los diferentes espectros de potencia [10].

En la Figura 24, cada uno de los rombos son la representación del proceso de la obtención del espectro de potencia comenzando por la FFT, como se puede apreciar hay un cierto traslape entre cada uno de los rombos, esto en parte es una técnica que utiliza Chuck para modificar el tiempo de duración de la pista sin modificar el tono [10]. Luego de obtener el espectro de potencia de cada uno de los marcos o *frames*, Chuck suma ese resultado para obtener el espectro de potencia total.

```
for (0 => int i; i < SIZE/2; i++)  {
i *=> histo[i];
}
```

Figura 25. Énfasis en las altas frecuencias.

Por último, se hace un énfasis en las frecuencias altas, esto es porque el límite inferior del rango audible de nuestro oído es de 20 Hz, entonces frecuencias menores a esta no son de interés. La Figura 25 muestra dicho énfasis.El script FFTFindModes.ck, tiene como objetivo final encontrar los modos de una pista de audio, como se menciona al inicio se puede configurar el número de modos a ser encontrados. La forma en cómo este script presenta al usuario los modos es por la ubicación en frecuencia y la magnitud normalizada. Estos dos datos representan a un modo, y son los requeridos por el siguiente script para sintetizar el sonido original.

En la figura 26, se utiliza un lazo *while* para encontrar los n valores máximos (n es igual al número de modos) del array "*histo[]*", al encontrar al primer máximo, la variable "*norm*" toma el valor de la magnitud, y sigue así sucesivamente encontrando los máximos valores de este array. En los arrays "*peakloc[]*" y "*peaks[]*" es donde se guardan la posición (frecuencia) y la magnitud normalizada (con respeto al primer máximo) de cada uno de los n modos.

.

```
0 => int pk;
while (pk < NUM_PEAKS)  {
    0.0 => peak;
    -1 => peakloc[pk];
    for (0 => int i; i < SIZE/2; i++)    {
        if (histo[i] > peak) {
            histo[i] => peak;
            i => peakloc[pk];   }    }
    peak / peakloc[pk] => peak;
    if (pk == 0) peak => norm;  peak/norm => peaks[pk];
<<< "Peak", pk, peaks[pk], (second/samp)*peakloc[pk]/SIZE >>>;
    peaks[pk]*peaks[pk]/(peakloc[pk]*peakloc[pk]) +=> power;
    zeroOut(histo,peakloc[pk]);
    pk++;   }
```

Figura 26. Búsqueda de modos en FFTFindModes.ck.

Se utiliza un lazo *for* para mandar a imprimir a consola, la frecuencia y la magnitud normalizada de cada uno de los modos (Figura 27).

```
for (0 => pk; pk < NUM_PEAKS; pk++)  {
    histo[peakloc[pk]]/max => peaks[pk];
<<< "[",(second/samp)*peakloc[pk]/SIZE,",",peaks[pk],"],"
>>>;
}
```

Figura 27. Código en Chuck para imprimir los modos en consola.

Para concluir, solo se menciona que el script utiliza 3 diferentes funciones para crear el archivo "*residue.wav*", las funciones son "*zeroOut()*", "*sortPeaks()*" y "*squelchComplex()*", además de la IFFT y la transformada z. El propósito de la creación de este archivo (*residue.wav*) se menciona más adelante. El script ModalResynth.ck, es el encargado de la creación de los resonadores Biquad, de hacerlos resonar (excitarlos) y el resultado mandarlo al convertidor digital analógico (*DAC*) de Chuck.

Primeramente ModalResynth.ck declara una clase bajo el nombre "*ModalSynth*" y dentro de esta clase define los resonadores y su respectiva configuración, además de eso define el tipo de excitación que se utilizará para cada uno de los resonadores y algunas funciones propias de la clase. En la Figura 28 se muestra el inicio de este script y de la declaración de la clase.

```
class ModalSynth extends Chubgraph {
    20 => int NUM_MODES;
    ResonZ modes[NUM_MODES];
[[ 199.182129 , 1.000000 ],
[ 1076.660156 , 0.782176 ],
…
[ 1886.846924 , 0.080443 ]]
@=> float freqsNamps[][];
```

Figura 28. Inicio de la clase ModalSynth.

Chuck por defecto ya tiene definido una unidad generadora, que realiza la función del resonador Biquad y es "*ResonZ*". Los parámetros de control para esta unidad son: frecuencia central y la calidad (Q). La clase "*ModalSynth"* define un array de "*ResonZ*" llamado "*modes[]*", de igual número que la cantidad de modos encontrados por FFTFindModes.ck. Los datos (frecuencia y magnitud normalizada) resultantes de FFTFindModes.ck son ingresados previamente al código copiándolos manualmente desde la consola de Chuck y son guardados en "*freqsNamps[][]*".

```
SndBuf excite;
me.dir() + "Residue.wav" => excite.read;
excite.samples() => excite.pos;
8.0 => excite.gain;
```

Figura 29. Excitación para ModalResynth.ck.

Normalmente se utiliza un impulso o ruido para excitar al resonador Biquad, Chuck utiliza el archivo "*residue.wav*" para obtener un sonido sintetizado con la misma duración y comportamiento como el original. La Figura 29 ilustra cómo se excita el resonador Biquad en Chuck.

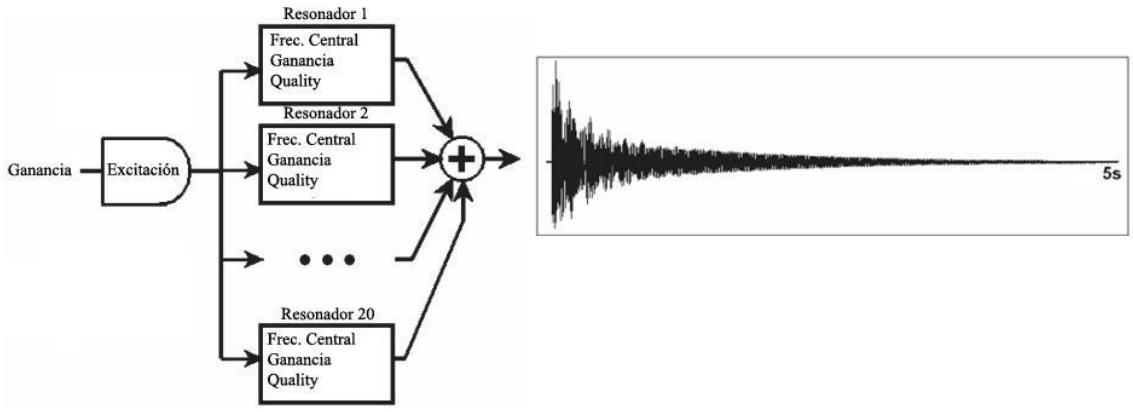


Figura 30. Resonadores Biquad en Chuck [10].

La Figura 30 muestra el diagrama de cómo están colocados en paralelo los resonadores Biquad empleados en Chuck.

```
for (int i; i < NUM_MODES; i++)  {
        excite => modes[i] => dac;
        freqsNamps[i][0] => modes[i].freq;
        setQfromT60(1.0-i/NUM_MODES,freqsNamps[i][0]) => modes[i].Q;
        freqsNamps[i][1] => modes[i].gain;
}
```

Figura 31. Configuración de parámetros de frecuencia, ganancia y Q.

En la declaración de la clase, se utiliza un lazo *for* para excitar uno por uno a los resonadores que están en el array "*modes[]*" y configurar cada uno de sus parámetros como la frecuencia central, Q y la ganancia (Figura 31).

Para el parámetro de la frecuencia, se utiliza el resultado del script anterior y que se encuentra en la matriz "*freqsNamps[][]*", al igual para el parámetro de la ganancia. El parámetro de Q, se utiliza la función "*setQfromT60*", definida a continuación en la Figura 32:

```
fun float setQfromT60 (float tsixty, float centerFreq)  {
        Math.pow(10.0,-3.0/(tsixty*second/samp)) => float rad;
        Math.log(rad) / -pi / (samp/second) => float BW;
        centerFreq / BW => float Q;
return Q;
}
```

Figura 32.Función que calcula Q en ModalResynth.ck.

Además de esta función, se declaran otras funciones más, la función "*whackIt()*" la cual hace un rebobinado de la excitación (residue.wav), esta función debe de cambiar según el tipo de excitación se vaya a utilizar.

La función "*whackItRandom(arg)*", esta función cambia aleatoriamente la ganancia de los modos. La función "*whackIt(arg,arg)*" cambia el orden de los modos y sus ganancias. Todo lo anterior se encuentra dentro de la declaración de la clase *ModalSynth* y una vez terminado esto, solo es necesario declarar una clase de este tipo.

Al ejecutar este script, la instrucción que hace escuchar el sonido sintetizado por las bocinas es "*ModalSynth plate;*", y la segunda instrucción "*plate.whackIt();*" rebobina la excitación (residue.wav), las siguientes instrucciones, son para manipular mediante las funciones declaradas en la clase "*ModalSynth*" los parámetros de los resonadores y obtener diferentes sonidos a partir del sonido sintetizado.

4.1.2 Resultado de la síntesis modal en Chuck.

En esta última parte, son presentados los resultados de los 2 scripts que están escritos en Chuck y que realizan la síntesis modal. Como es necesario introducir una archivo .wav al script FFTFindModes.ck, se ha elegido grabar el audio de cuando es golpeado un perol metálico y la forma de onda de este sonido se observa en la Figura 33.

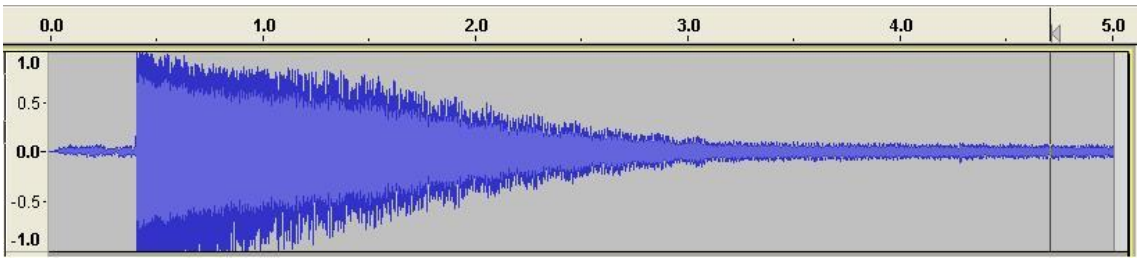


Figura 33. Forma de onda del sonido de un perol metálico.

Esta forma de onda es muy similar a la que es presentada en el anexo A (Figura A.1), esto permite saber que es un buen sonido para ser sintetizado a partir de sus modos.

Al ejecutar el script FFTFindModes.ck, aparece en la consola de Chuck la información de la frecuencia y la ganancia de cada uno de los modos que con anterioridad se han solicitado, (en este ejemplo son los primeros 20). Los modos principales se muestran en la Figura 34.

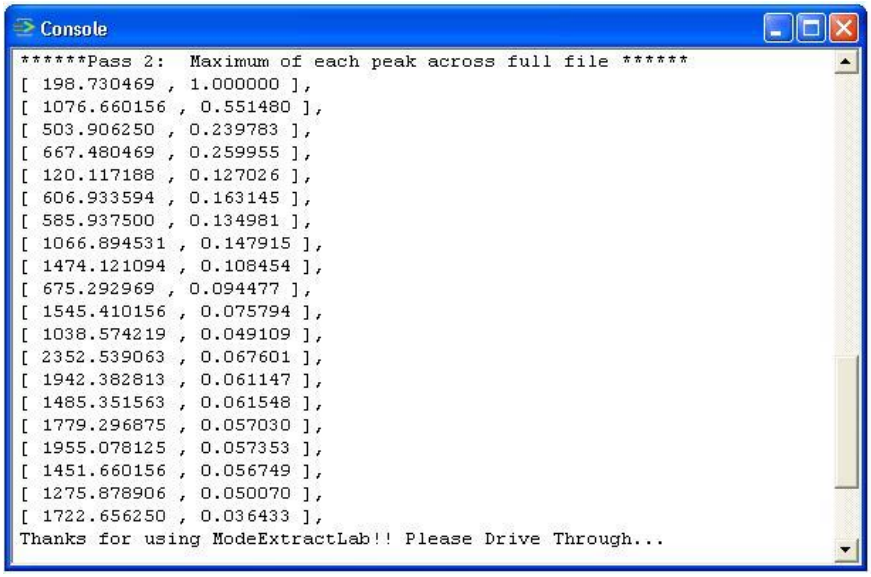


Figura 34. 20 modos principales del audio: perol_metalico.wav

Estos datos deben cortarse y pegarse exactamente como están, en el código del script ModalResynth.ck, para ambos script el número de modos debe de ser el mismo. Además de la información de los modos, FFTFindModes.ck realiza la creación del archivo “residue.wav”, el cual es el sonido original menos los modos encontrados (Figura 35).

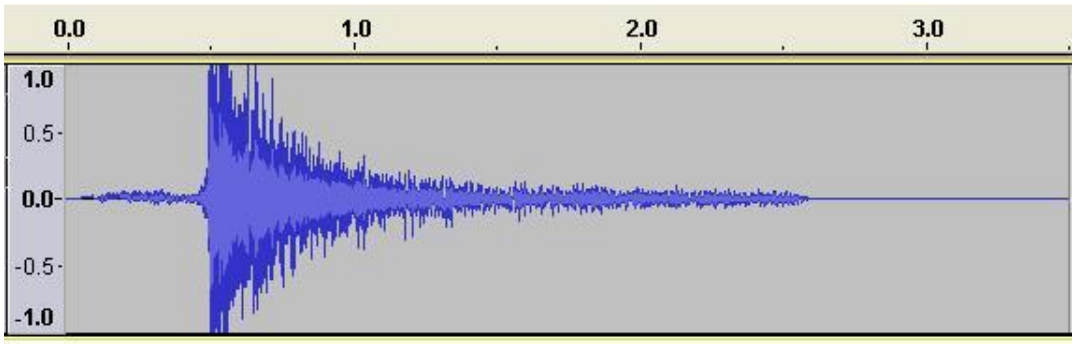


Figura 35. Forma de onda de "residue.wav"

Sí se hace una comparación entre el sonido original (perol_metalico.wav) y el residuo (residue.wav), esta última presenta una forma de onda que tiende a decaer mucho más rápido, recortando su tiempo de duración y la energía almacenada en éste, todo esto es a causa de que sus 20 modos principales han sido sustraídos.

La forma de onda de "*residue.wav*" aún tiene modos dentro de sí, es por esa razón que aun presenta cierta similitud con la ilustración del anexo A. En la figura 36 se ha utilizado MatLab para graficar el espectro de potencia a partir de los datos que Chuck calcula y guarda en "*histo[]*", puede visualizarse que existen varios modos, unos más grandes que otros. Los más grandes son los que aportan una mayor energía y caracterizan al sonido (Figura 36).

En el script ModalResynth.ck, se debe de ajustar la ganancia de la excitación, esto sí es el caso cuando se utiliza al residuo como la excitación, de caso contario no es necesario y se puede omitir.

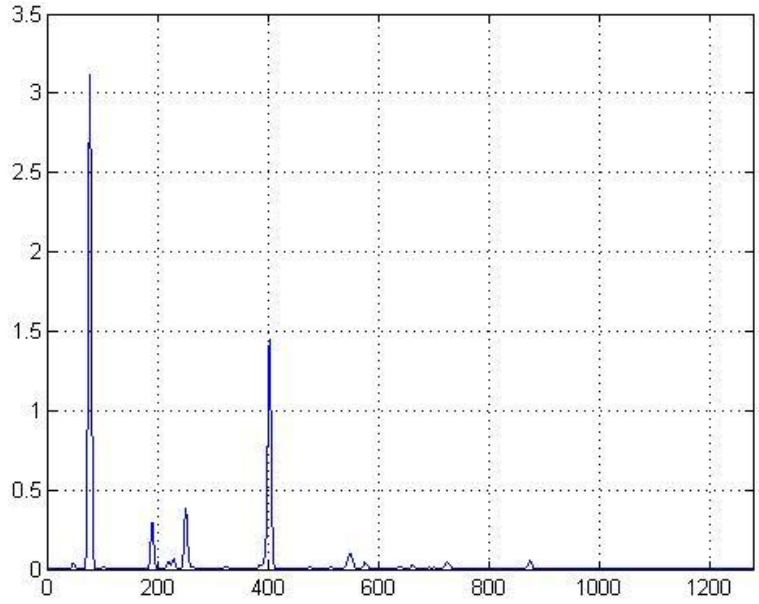


Figura 36. Parte del espectro de potencia de perol_metalico.wav

Con la ayuda de Audacity, se muestra la forma de onda del sonido ya sintetizado en la Figura 37, y se logra apreciar que su duración es similar a la del residuo, pero que su forma es diferente, dado a que ésta señal proviene de la sumatoria de la salida de cada uno de los diferentes resonadores Biquad, como se muestra en la Figura 30.

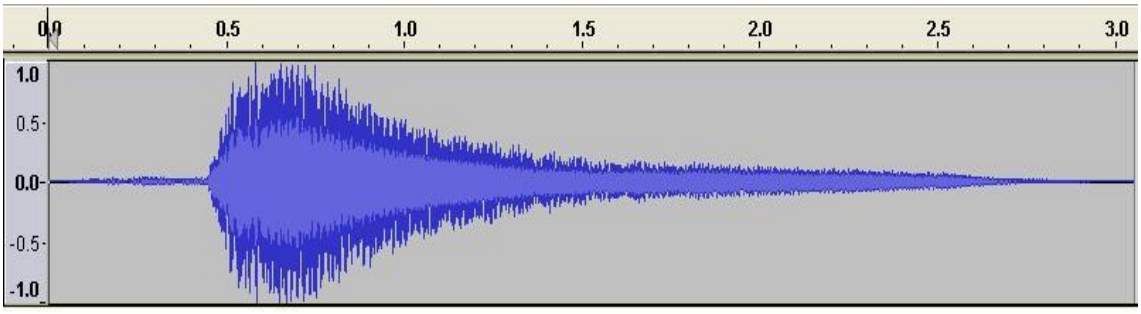


Figura 37. Forma de onda del sonido sintetizado realizado en Chuck.

4.2 Síntesis modal en el KIT ADSZ-BF533 EZlite.

Junto al VisualDSP++, *Analog Devices* provee un conjunto de proyectos de ejemplo, en los cuales la configuración de los diferentes periféricos ya está realizada dependiendo de la aplicación. En aplicaciones de audio, existen 4 proyectos de ejemplo (2 en lenguaje ensamblador y 2 en lenguaje C), cuya diferencia radica en la forma en que se configura el códec AD1836 (I^2S y TDM). Este conjunto de ejemplos, son la base para nuevos proyectos para quienes inician en la programación de este tipo de KIT.

El proyecto de síntesis modal desarrollado en VisualDSP++, consta de 6 archivos escritos en lenguaje C y son: main.c, fft_btc.c, Initialize.c, ISR.c, modos.c y Process_data.c. Agregado a esto un archivo de cabecera: Talkthrough.h en donde se realizan algunas declaraciones de constantes y librerías.

Las características generales de este proyecto se basan en el proyecto de ejemplo "*Talkthrough*", en el cual se inicializa el SPORT0 para establecer un enlace entre el ADSP-BF533 (procesador) y el códec AD1836. El códec AD1836 se configura en modo TDM y el SPORT0 para recibir/transmitir muestras de audio a partir del códec. Luego de ser procesadas estas muestras por el ADSP-BF533, son colocadas en un buffer de transmisión y regresan al códec para ser transmitidas por el convertidor digital analógico. En el archivo "*main.c*", se declaran algunas variables tales como: canales de audio de entrada y salida (iChannel0LeftIn, iChannel0LeftOut, etc.), una tabla de configuraciones para el códec (sCodec1836TxRegs), buffers de recepción y de transmisión de datos (iTxBuffer1 y iRxBuffer1), así como las variables para realizar cada una de las tareas de procesamiento de la señal (buffer_in, buffer_out, histo, B_0, A_0, etc.).

Luego de esto, el programa principal llama a diferentes funciones para realizar las configuraciones restantes. Entre estas configuraciones se puede mencionar la inicialización de la EBIU, memoria flash, códec, SPORT0, DMA, Interrupciones y banderas programables como puede observarse en la Figura 38.

```
sysreg_write(reg_SYSCFG, 0x32);
      Init_EBIU();
      Init_Flash();
      Init1836();
      Init_Sport0();
      Init_DMA();
      Init_Sport_Interrupts();
      Enable_DMA_Sport0();
      Init_Flags();
```

Figura 38. Funciones inicializadoras en el main.c

4.2.1 Descripción de la síntesis modal en el KIT ADSZ-BF533 EZlite.

En este proyecto, se ha declarado 2 tipos de interrupciones, una que la genera SPORT0 cuando recibe una trama de datos de entrada y la otra generada por los pulsadores. La trama de datos de entrada, está compuesta por 6 datos tipo entero (*int*) de 32 bits cada uno y se almacenan en el buffer "*iRxBuffer1*". Solo se utilizan dos de estos datos, que corresponde a las entradas ADC1 izquierda y ADC1 derecha, los datos provenientes del ADC2 no son tomados en cuenta.

La forma de cómo se muestrea el sonido, es conectando un speaker entre la salida de audio de una computadora y las entrada RCA de la tarjeta (Figura 39), reproducir el archivo .wav desde la computadora y que la tarjeta muestree a 48 kHz este sonido. Este procedimiento se realiza de esta forma para partir de una misma señal y así comparar ambos resultados (Chuck vs DSP).

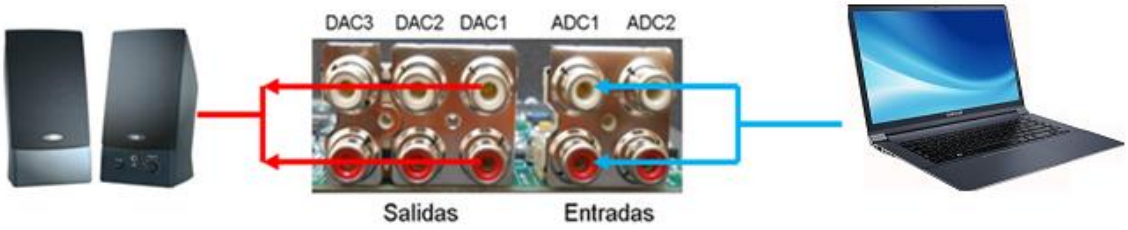


Figura 39. Ilustración de la conexión entre la computadora y la DSP.

La frecuencia de muestreo se puede configurar a 48 kHz y a 96 kHz, la forma de configurar esta frecuencia es por medio de una tabla de registros: "*sCodec1836TxRegs*" y el significado de cada uno de los registros y de las diferentes configuraciones del códec AD1836, pueden ser consultados en su respectiva hoja técnica [5].

Como se mencionó antes, los pulsadores también generan una interrupción, y es que por medio de los pulsadores se modifica una variable de control y su configuración es la siguiente:

1. Al presionar SW4, la variable "*task*"toma le valor de 1 y se habilita la toma de muestras por parte de la DSP a la señal de audio, realiza la FFT y calcula el espectro de potencia en tiempo real.
2. Al presionar SW5, la variable "*task*" toma el valor de 2 y se buscan los modos en el array "*histo[]*".
3. Al presionar SW6, la variable "*task*" toma el valor de 3 y se genera la respuesta al impulso de los resonadores Biquad.
4. Al presionar SW7, la variable "*task*" toma el valor de 4 y reinicia algunas variables para obtener la respuesta al impulso nuevamente.

Entonces para empezar a muestrear la señal de entrada, es necesario presionar SW4 y en la consola del VisualDSP++ aparece una pequeña notificación, la cual indica el tiempo de inicio para que sea reproducido el archivo de audio en la computadora. La desactivación de la toma de muestras es ejecutada automáticamente, esto para evitar no interferir con alguna operación que se esté realizando al momento de detenerlo.

```
if (task == 1){
      iChannel0LeftIn = iRxBuffer1[INTERNAL_ADC_L0];
      iChannel0RightIn = iRxBuffer1[INTERNAL_ADC_R0];
      Process_Data();
}
```

Figura 40. Parte del código que atiende la interrupción al presionar SW4.

En la figura 40, se muestra parte del proceso del muestreo de la señal, "*iChannel0LeftIn*" e "*iChannel0RightIn*" son variables en donde se guarda una muestra para cada canal de audio, dentro de estas variables se encuentra datos de 24 bits que representan la magnitud de la señal muestreada.

A diferencia de Chuck, la tarjeta DSP posee una memoria limitada, es por esa razón que se decide a no guardar ningún archivo de audio dentro de su memoria y a cambio explotar una de sus principales características, la cual es el procesamiento de señales en tiempo real.

A partir de esta característica, esta etapa se hace en simultáneo junto a la anterior (adquisición de muestras). Entre las tareas de procesamiento, que es necesario realizar, está la FFT y en las librerías de procesamiento de señales con que cuenta la DSP se puede encontrar muchas funciones para este tipo de procesamiento.

En la librería "*filter.h*", se encuentran algunas funciones útiles como: filtro FIR, filtro IIR, FFT, IFFT, convolución, compresión y expansión. Como se necesita realizar la FFT a la señal de entrada, es necesario utilizar la función "*rfft_fr16*", la cual realiza la transformada rápida de Fourier de una señal real, cabe mencionar que todas las funciones de las librerías DSP están optimizadas y que trabajan con el formato fract16. Es decir que se debe convertir los datos de 32 bits a formato fract16.

En la figura 41, se llama a una función llamada "*Process_Data ( )*", esta función se define en el archivo del mismo nombre y es la encargada de ir haciendo la conversión entre datos de 32 bits a datos en formato fract16, además de ir guardando estos datos en un array ("*buffer_in[]*").

```
void Process_Data(void)
{
      if (sample < NUMPOINTS){
            buffer_in[sample]=(iChannel0LeftIn >> 8)/AT;
            sample = sample + 1;
      }
```

Figura 41. Función Process_Data().

Los datos que se guardan en las variables "*iChannel0LeftIn*" e "*iChannel0RightIn",* son datos tipo enteros de 32 bits, pero en realidad los datos provenientes del códec solo son de 24 bits, es decir hay 8 bits que podemos eliminar sin ningún inconveniente. Para hacer esto se utiliza el operador ">>", el cual realiza una desplazamiento de bits.

Después de realizar el desplazamiento, solo necesitamos una conversión de escala entre los datos de 24 bits y fract16.

Como en el muestreo solo es una representación numérica de la magnitud, no habrá ningún problema al realizar este cambio de escala. El formato fract16 tal como se muestra en la Tabla E.2, es un formato en el cual se utilizan 16 bits para representar a un número, y cuyo rango de valor en decimal es -1.0 a 0.9999.

Conociendo esto se calcula un factor de escala y la manera de calcular es la siguiente:

$$AT = \frac{Lim.\ sup.\ de\ rep.\ 24\ bits\,(hexa\,)}{Lim.\ sup.\ de\ rep.\ fract\ 16(hexa\,)} = \frac{0xFFFFFF}{0x7FF8} \qquad \text{Ecuación 1.}$$

$$AT = 0x200 = 512$$

"*AT*" es una constante con un valor de 512, y es la que se utiliza para modificar la escala de los datos de entrada.

En la Figura 41, solo se realiza esta conversión a los datos provenientes de "*iChannel0LeftIn*", además de guardar solo las muestras de este canal, esto se realiza porque en ambos canales es la misma información, entonces para ser más eficiente y no procesar un mismo dato dos veces, se decide solo trabajar con las muestras provenientes de uno de los dos canales (muy bien pudo haber sido "*iChannel0RightIn"*).

El tamaño del array "*buffer_in[]*", está definido por la constante "*NUMPOINTS*", y es la misma constante que define el tamaño (2^{12} muestras) de la FFT, realizada por la función "*rfft_fr16*". Al completar el número de muestras requeridas para hacer la FFT, se llama a la función "*fft_btc*", esta función está declarada en el archivo del mismo nombre (fft_btc.c). Dentro de esta función, se declaran algunos de los argumentos que necesita la función "*rfft_fr16*" de la librería "*filter.h*" del DSP.

```
void fft_btc(void){
      int i,j;
      int wst = 1;
      int n = NUMPOINTS;
      int block_exponent;
      int scale_method = 1;
      twidfftrad2_fr16(w, NUMPOINTS);
      rfft_fr16(buffer_in, buffer_out, w, wst, n, &block_exponent,
scale_method);
      for (i=0; i<NUMPOINTS; i++) {
            mag[i] = cabs_fr16(buffer_out[i]);
      }
      for (i=0; i<NUMPOINTS/2;i++){
                  histo[i] = histo[i] + (mag[i]*mag[i]);
      }  }
```

Figura 42. Función fft_btc().

En la Figura 42 se muestra el código de la función "*rfft_fr16*", la cual transforma la señal de entrada del dominio del tiempo al dominio de la frecuencia mediante el uso de la FFT radix-2. El tamaño del array de entrada ("*buffer_in[]*") y el de salida ("*buffer_out[]*") debe ser igual al número de puntos ("*NUMPOINTS*") de la FFT. Entre los argumentos para esta función se necesita una tabla de rotación o *twiddle*("*w[]*"), los cuales son coeficientes compuestos por +coseno y –seno y que pueden ser inicializados por medio de la función "*twidfftrad2_fr16*". La variable "*wst*", es para controlar sí la tabla de rotación o *twiddle* coincide en tamaño con la FFT. El argumento "*scale_method*" controla la forma de como la función aplicará la escala cuando se calcula la transformada de Fourier. Las opciones disponibles son de escala estática (dividiendo la entrada entre 2), escala dinámica (dividiendo la entrada entre 2, sí el mayor valor de entrada absoluta es mayor o igual que 0.5) o sin escala [7].

Luego de calcular la FFT de un conjunto de muestras, se procede a encontrar la magnitud de la FFT, para esta tarea es necesario auxiliarse de la función "*cabs_fr16*", para finalmente obtener el espectro de potencia (cuadrado de la magnitud de la FFT) y guardarlo en "*histo[]*".

Todo este proceso se realiza en tiempo real (captura de muestras, FFT, cálculo de la magnitud y obtención del espectro de potencia), y por esa razón el tamaño de la FFT es ajustada al máximo, tal que permita realizar todas estas tareas sin que interfieran una con otra. Esta característica proviene de la arquitectura del procesador y de la técnica que utiliza para procesar cada una de estas instrucciones conocida como "PIPELINE" [3]. De la misma manera que se realiza en Chuck, el espectro de potencia es sumado en la variable "*histo[]*".

En el archivo modos.c, se encuentra declarada la función "*modos ()*", esta función es llamada para atender la interrupción generada cuando se presiona el SW5 (Figura 43), pero antes se realiza el énfasis en las altas frecuencias como se realizó en Chuck.

```
if (registro == 0x0200){
task = 2;
for (i=0;i<NUMPOINTS/2;i++){
      histo[i]=histo[i]*i;  }
modos();
   printf("%i MODOS [Frecuencia] [Magnitud]\n",NUM_PEAKS);
for (i=0;i<NUM_PEAKS;i++){
      printf("\t[%f,\t%f],\n",loc_pic[i],pic[i]);
}  task = 0;
}
```

Figura 43. Parte del código que atiende la interrupción al presionar SW5.

Dentro de esta función, la dinámica es similar a la ejecutada en Chuck, que por medio de lazos *for* y *while* se busca en el array "*histo[]*" los n valores más altos, al encontrar cada uno de ellos, se guarda su ubicación (frecuencia) y su magnitud normalizada en las variables "*loc_pic[]*" y "*pic[]*" respectivamente. El número de modos a encontrar, puede ser modificado al ingresar al archivo "*Talkthrough.h*" y cambiar el valor de la constante "*NUM_PEAKS*". Al retornar de esta función, se imprime en consola, los valores de la frecuencia y de la magnitud de cada uno de los modos encontrados.

Hay 3 formas de implementar los resonadores Biquad en la DSP, 2 de esas formas se implementan y solo una se logra acoplar a las necesidades. Una de las formas es utilizando la función *"iir_fr16*" que se encuentra en la librería de la DSP "*filter.h*". Esta función implementa un filtro Biquad (*direct form II*) de respuesta al impulso infinito y además también existe otra función en esta misma librería, y es la función "*iirdf1_fr16*". Esta función es similar a la anterior, con una única diferencia, el algoritmo cambia a *direct form I* [3].

El único inconveniente, es que estas dos funciones necesitan que entre sus argumentos se defina la longitud del array de entrada y salida. Entonces como la frecuencia de muestreo es de 48 kHz, para tener un segundo de sonido sintetizado se necesitaría un array de salida y de entrada de 48,000 elementos. Tal cantidad de elementos en un array provoca al momento de ejecutar el programa una excepción desconocida.

Así que la solución a este problema es implementar el filtro de respuesta al impulso infinito a partir de su ecuación, esto permite trabajar sin restricción en cuanto a que los datos deben de estar en formato fract16 y que no es necesario guardar todo los elementos de la respuesta, sino solo se necesita al elemento actual de la salida, colocarlo en el buffer de transmisión de datos "*iTxBuffer1*" y que sea convertido por el DAC de la tarjeta.

La ecuación que se implementa es la siguiente:

$$y(i) = B_0 * x(i) + B_1 * x(i-1) + B_2 * x(i-2) + A_1 * y(i-1) + A_2 * y(i-2) \qquad \text{Ecu. 2.}$$

Donde:

Coeficiente	**Valor**	**Coeficiente**	**Valor**
A_0	1	B_0	1
A_1	$-2R\cos(2\pi F_c T)$	B_1	0
A_2	R^2	B_2	-R

Tabla 4. Coeficientes para el resonador Biquad.

El código que realiza el cálculo de cada uno de los coeficientes del resonador (Figura 44), se encuentra al final del archivo "*modos.c*", esto se realiza así para ya tener listo los coeficientes antes que se presione SW6. Además de esto, la cantidad máxima de resonadores Biquad que se pueden implementar en la DSP es de 4, un número mayor a este, perjudica al sonido por el tiempo en que se tarda en realizar todo este grupo de operaciones.

```
for (i=0; i < NUM_PEAKS; i++){
                    Fc = loc_pic[i];
                    Bw =Fc/Q;
                    R = expf(-3.14159*Bw*Ts);
                    A_2[i]=R*R;
                    A_1[i]=-2*R*cos(2*PI*Fc*Ts);
                    A_0[i]=1.0;
                    B_2[i]=-R;
                    B_1[i]=0.0;
                    B_0[i]=1.0; }
```

Figura 44. Calculo de coeficientes del resonador Biquad.

Al presionar el SW6, la variable "*task*" toma el valor de 3 y se permite atender la interrupción generada por este pulsador. Como excitación se utiliza un impulso con una amplitud de 2097151.0, esta amplitud ayudara a tener a la salida un sonido lo suficientemente fuerte.

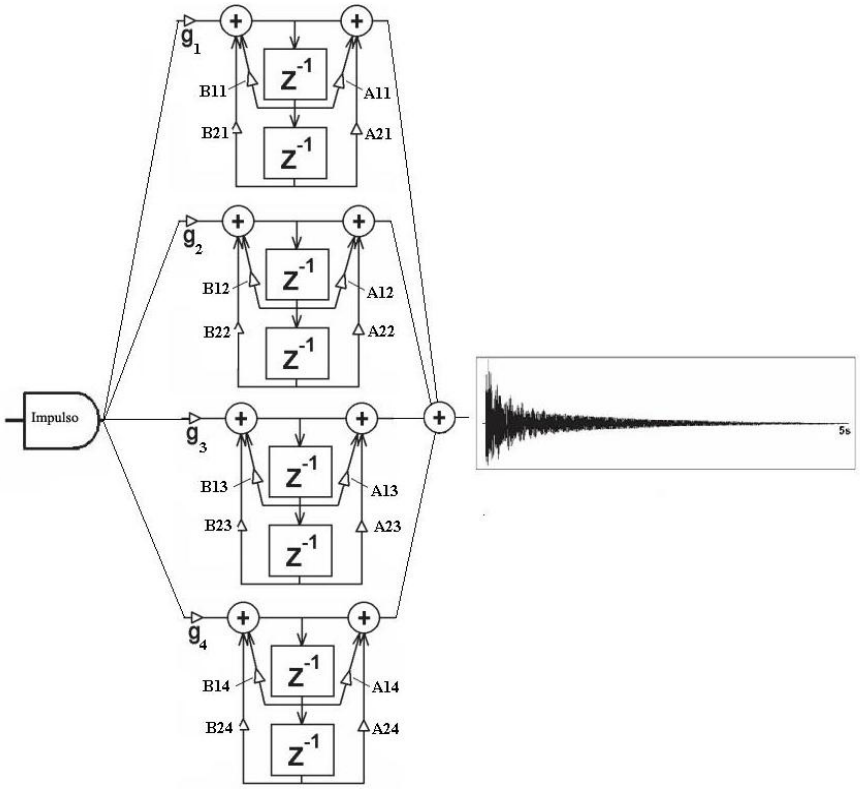


Figura 45. Resonadores Biquad en la DSP [10].

En la Figura 46 se muestra parte del código que realiza la ecuación del resonador Biquad para la primera muestra, que es un impulso de amplitud bastante grande, y como no se tienen muestras anteriores, la ecuación se ve reducida.

```
for (frecuencia =0; frecuencia < NUM_PEAKS; frecuencia++){
      Fc = loc_pic[frecuencia];
      gain = pic[frecuencia];
      if (m == 0){
            Xm = 2097151.0;
            Ym = B_0[frecuencia]*Xm;
            Xm_1[frecuencia] = Xm;
            Ym_1[frecuencia] = Ym;
      }
```

Figura 46. Parte del código en modos.c donde se calculan los coeficientes.

```
if ((m>0) && (m<duracion) ){
      Xm = 0.0;
      Ym = (B_0[frecuencia]*Xm) + (B_1[frecuencia]*Xm_1[frecuencia]) +
(B_2[frecuencia]*Xm_2[frecuencia]) - (A_1[frecuencia]*Ym_1[frecuencia]) -
(A_2[frecuencia]*Ym_2[frecuencia]);
      Xm_2[frecuencia] = Xm_1[frecuencia];
      Xm_1[frecuencia] = Xm;
      Ym_2[frecuencia] = Ym_1[frecuencia];
      Ym_1[frecuencia] = Ym;
      }
SALIDA[frecuencia] = Ym*gain;
}
```

Figura 47. Resto del código de los resonadores Biquad.

Para el resto de muestras, hay una ejecución de la ecuación del Biquad completa (Figura 47), para controlar que tan grande es la duración del sonido, se utiliza la variable "*duración*", en la cual se guarda un número múltiplo de 48,000.

Por la recursividad de la ecuación, se declara un array para cada una de los términos y coeficientes B_0, B_1, B_2, A_1, A_2, Xm_1, Xm_2, Ym_1 y Ym_2, que guarda estos términos para cada una de las frecuencias centrales. Esto se realiza de esta forma para evitar confundir términos que han sido calculados con diferentes coeficientes y obtener una señal resultante errónea.

Al termino de las sentencias *if* realizadas, existe un último array, en él se guardan los términos Ym. Los términos Ym son los datos que han sido calculados por medio de la ecuación del Biquad con los coeficientes de cada una de las frecuencias de los 4 modos predominantes. En esta parte los términos Ym son multiplicados por la ganancia de su respectivo modo resonante, encontradas por la función "*modos ()*". Al final cada uno de estos términos son sumados y guardados en la variable "*bocina*", la cual a su vez es guardada en el buffer de transmisión y es reproducida por las bocinas luego de ser convertida por el convertidor digital analógico (DAC) del KIT ADSP-BF533 EZ-KIT Lite como se muestra en la Figura 48.

```
for (frecuencia =0; frecuencia < NUM_PEAKS; frecuencia++){
      bocina = bocina + SALIDA[frecuencia];
      }
      iTxBuffer1[INTERNAL_DAC_L0] = (signed int)(bocina*0x1FF);
            m = m + 1;
            bocina = 0.0;
            if (m > duracion){
                  task = 0;
            }
      }
}
```

Figura 48. Parte final del código que implementa los resonadores.

4.2.2 Resultado de la síntesis modal en el KIT ADSZ-BF533 EZlite.

En esta última parte, luego de hacer las conexiones necesarias como se muestra en la Figura 39, se procede a seguir los siguientes pasos: en el VisualDSP++, abrir el proyecto de la síntesis modal, compilar el proyecto presionando F7 (VisualDSP++ carga automáticamente un ejecutable en la tarjeta) y ejecutarlo presionando F5, esto desplegará en la ventana de salida del VisualDSP++ un mensaje de bienvenida y las tareas que se ejecutarán al presionar cada uno de los pulsadores de la tarjeta. La Figura 49 muestra la ventana de salida del VisualDSP++ con los mensajes escritos por el proyecto.

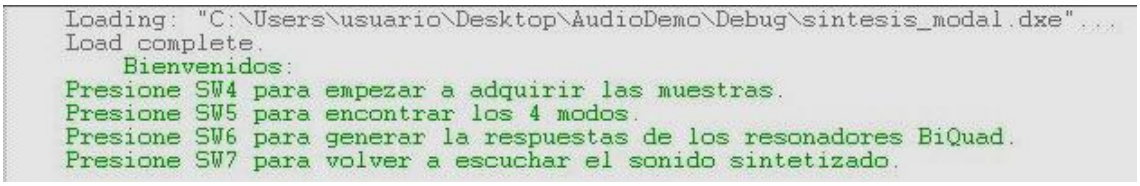


Figura 49. Mensajes de inicio de la ejecución del proyecto.

Se presiona SW4 y se notifica por medio de la ventana de salida el inicio del tiempo para capturar las muestras.

Se reproduce la pista de audio desde la computadora y al término del tiempo de captura de muestras (alrededor de 5 segundos) automáticamente se muestra la notificación en la consola. Recordar que la forma de onda que está siendo muestreada por la DSP es la misma mostrada en la Figura 33.

Presionar el SW5 para que se encuentren los 4 modos predominantes del sonido, al calcular los modos son impresos en la ventana de salida tal como se logra observar en la Figura 50.

```
El tiempo para la toma de muestra comienza  AHORA!
Fin del tiempo de toma de muestras

4 MODOS [Frecuencia] [Magnitud]
    [199.218750,    1.000000],
    [1078.125000,   0.465189],
    [667.968750,    0.116155],
    [503.906250,    0.091807],
```

Figura 50. Modos encontrados por la DSP.

Al presionar SW6 se escucha el sonido sintetizado. Y la forma de onda del resultado de la síntesis (Figura 51).

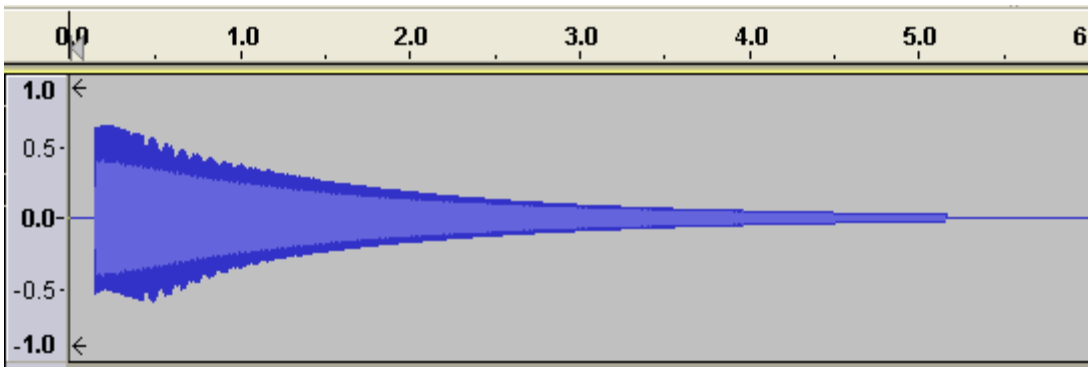


Figura 51. Forma de onda del sonido sintetizado por la DSP.

Y por último, presionar SW7 para volver a generar el sonido, cuantas veces el usuario desee. Cada vez que se presione este pulsador, un mensaje se estará mostrando en la consola, cabe señalar que se debe de esperar cierto tiempo (5 segundos aprox.) para volver a presionarlo sino el sonido producido es distorsionado. La figura 52 muestra cuando SW7 es presionado.

```
4 MODOS [Frecuencia] [Magnitud]
    [199.218750,    1.000000],
    [1078.125000,   0.411827],
    [667.968750,    0.109897],
    [1066.406250,   0.099861],
Sintetizando sonido...
Sintetizando sonido...
```

Figura 52. Mensaje para indicar que se ha presionado SW7.

Cabe mencionar que una vez que el programa está siendo ejecutado, y sí se desea sintetizar algún otro sonido, es necesario detener la ejecución del proyecto (Shift+F5) y volver a compilar, cargar, ejecutar y presionar SW4 para que vuelva a tomar muestras y repetir todo el proceso antes mencionado.

4.3 Comparación de resultados.

Al término de las secciones anteriores, se procede a la comparación de los datos que se han calculado mediante el lenguaje de programación Chuck y la DSP. Para comenzar se comparan los modos resonantes.

La DSP tienen la capacidad de encontrar tantos modos se desean encontrar, solo es necesario configurar el numero de ellos en el archivo "*Talkthrough.h*". Estos modos se determinan al obtener el espectro de potencia de la señal de entrada.

Al obtener los espectros de potencia, una generada por Chuck y la otra por la DSP de una misma señal de entrada, se obtiene la siguiente gráfica (Figura 53).

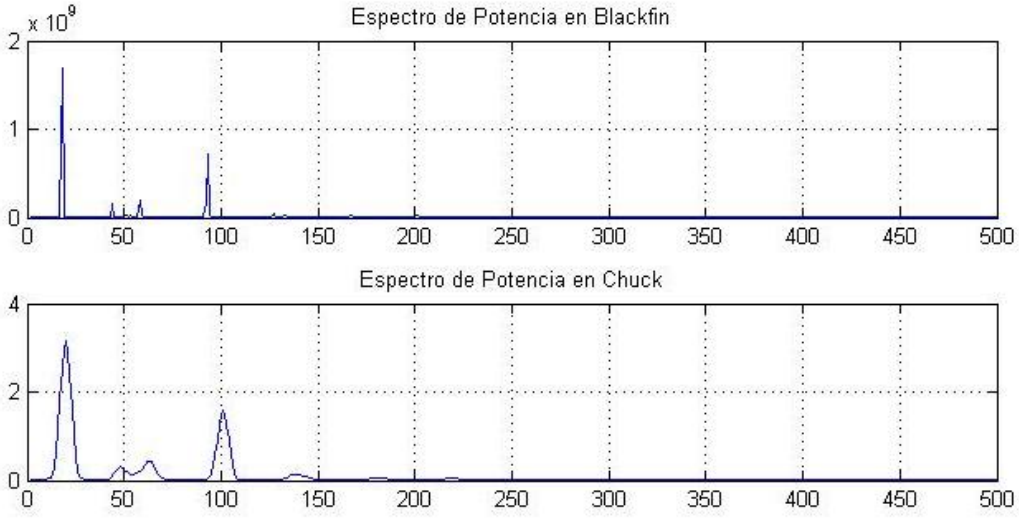


Figura 53. Comparación de los primeros 500 elementos de los espectros de Potencia para una FFT de 4096 puntos.

De la Figura anterior, se logra apreciar que el espectro de potencia tanto en Chuck como en la DSP tiende a tener la misma forma, solamente que el espectro de Chuck los picos son más anchos, esto se debe a que en Chuck se cuenta con la totalidad de las muestras, además de que al momento de realizar la FFT, hay un leve traslape es decir hay muestras que vuelven a ser tomadas en cuenta en este procesamiento. Mientras que en la DSP se toma de la rapidez con que se ejecutan las instrucciones para ir realizando este procesamiento cada 4096 muestras, pudiendo perder algunas, pero que no repercute mucho en el resultado, es decir en la obtención de los modos resonantes. Además, la diferencia entre amplitudes de ambos espectros de potencia radica en cómo son representados los datos provenientes de sus respectivos convertidores analógico digital, en Chuck utiliza números flotantes de doble precisión, mientras que la DSP números enteros de 32 bits.

Para comparar sí los valores de frecuencia y magnitud de cada uno de los modos encontrados por la DSP son aceptables, se ha configurado el script FFTFindModes.ck de Chuck a un tamaño de la FFT a 4096 puntos para encontrar los 4 modos principales. Al terminar todo el proceso Chuck muestra sus resultados como puede verse en la Figura 54.

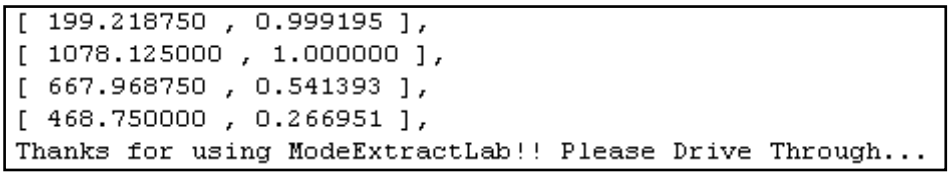


Figura 54. 4 Modos principales en Chuck configurado a una FFT de 4096 muestras.

De la Figura anterior, se puede observar que son casi idénticos los valores de las frecuencias de los modos que fueron encontrados por la DSP (Figura 52), no así las magnitudes. Pero al compararlas con las encontradas por Chuck cuando el tamaño de la FFT era de 16384 puntos (Figura 34), son muy similares.

Esto permite asegurar que los modos resonantes encontrados por la DSP se encuentran en un margen aceptable.

Ahora la forma de onda del sonido sintetizado en la DSP, no es tan idéntica a la original (Figura 33) y tampoco a la sintetizada en Chuck (Figura 37), esto a razón de que la excitación que se utilizó en la DSP es un impulso, mientras que en Chuck fue utilizando el residuo, es decir que solo se guardan los valores de las frecuencias y magnitud de los 4 modos principales del sonido original, provocando que el sonido resultante sea similar, no en su forma (envolvente) pero si en sus frecuencias obtenida de sus modos resonantes.

Capítulo 5: Conclusiones y líneas futuras.

5.1 Conclusiones.

- La síntesis de sonidos en el KIT ADSZ-BF533 EZLITE, se logra gracias a la implementación de algoritmos básicos, estos mismos son punto de partida para realizar otro tipo de procesamiento de audio. Al término de este trabajo se ha observado que el KIT muestra un buen desempeño al realizar este tipo de procesamiento. Por lo que se concluye que podría ser una buena opción para desarrollar otro tipo de aplicaciones de procesamiento de audio.
- Las librerías existentes en la DSP, permiten realizar muchas tareas de procesamiento de una forma eficiente y rápida, para alcanzar estas características el procesador hace uso del formato fract16, tal formato le es beneficioso al procesador, pero que restringen el rango de valor representado que la señal puede adquirir (-1.0 a 0.999), por lo que fue necesario comprender bien este tema y realizar un cambio de escala a la señal de entrada y salida para lograr utilizar las funciones predefinidas en la DSP.
- Al comparar los resultados del lenguaje de programación Chuck y la DSP, se logra apreciar que Chuck provee una mayor información sobre la señal de entrada, no presenta limitantes de memoria y no hace uso del fract16, por lo que no hay ninguna condición a los datos de entrada, contiene una amplia variedad de clases y objetos ya preestablecidos que ayudan y facilitan al procesamiento de señales de audio en tiempo real. Todo lo anterior también dependerá de los recursos con que cuenta la computadora en donde está instalado.

- La DSP cuenta con funciones en donde se puede implementar el resonador Biquad, pero que no lograron satisfacer las necesidades, por lo que se recurrió a implementar el resonador por medio de su ecuación, esto proporcionó el resultado que se buscaba, no así la calidad, dado a que al implementar el resonador desde su ecuación provocó que la eficiencia de la DSP se redujera y esto limitó el número de resonadores que se tendrían al mismo tiempo. Con lo anterior no quiere decir que las funciones que se encuentra en la DSP no funcionan adecuadamente, sino que el uso de estas dependerá de la aplicación y del objetivo a conseguir, cabe destacar que esta DSP no está fabricada para sintetizar sonidos sino para filtrado.
- De ambos sonidos sintetizados ninguno fue exactamente igual al original, esto puede verse a simple vista si se observa la forma de onda. El sonido que se produjo en Chuck fue el que se aproximó más al original, pero fué porque se utilizó como fuente de excitación el residuo de la señal original (“residue.wav”), además por contar con un mayor número de modos resonantes. El sonido sintetizado por la DSP en cambio, sólo contó con 4 modos resonantes, aunque la DSP muy bien puede encontrar los modos resonantes que le soliciten, no así cuando tienen que implementar los resonadores Biquad, teniendo como máximo a 4 resonadores y en el caso de la excitación, se utilizó un impulso.

5.2 Líneas Futuras.

- Seguir trabajando en el código para hacerlo más eficiente y así lograr obtener un mejor sonido sintetizado. Se podría utilizar las herramientas de desarrollo con las que cuenta el VisualDSP++, para visualizar sí existen cuellos de botella en alguna parte del código y optimizarlo.
- Investigar sobre otros tipos de sintetizado que pueden ser implementados en el KIT, que simulen el comportamiento de cuerdas (1D), tambores (2D) y la cavidad bucal (3D), entre otros. Este tipo de sintetizado se han realizado en Chuck y entre algunas de las unidades que vienen por defecto, existen instrumentos sintetizados a partir de estos conceptos de dimensiones.
- Investigar sobre el procesamiento de imágenes, hay muchas áreas bastante conocidas por utilizar este tipo de procesamiento, tales como la seguridad biométrica, así como áreas no tan conocidas y que pueden abrir nuevos campos de investigación como lo es la astronomía. Sí bien es cierto en la DSP no es posible realizar un paso constante de muestras de video así como se ha realizado en audio, permite la captura de imágenes y la implementación de algoritmos de procesamiento en éstas.

ANEXOS

Anexo A. Síntesis Modal.

La síntesis obtiene sonidos a partir de medios no acústicos; variaciones de voltaje en el caso de la síntesis analógica o en las señales digitales por medio de programas de cómputo. El término síntesis se refiere a producir sonidos “similares” al de un instrumento musical real, e incluso a crear sonidos nuevos al mezclar los sonidos obtenidos por medio de alguna metodología en particular. Se menciona a continuación algunos métodos de síntesis de sonido.

La Síntesis aditiva es el método usado para crear timbres, estos timbres están formados por cantidades de variables de armónicos o parciales que cambian a lo largo del tiempo en referencia a un tono o frecuencia fundamental. Se le llama parcial, a las ondas que complementan a la onda de frecuencia fundamental para crear un timbre [11].

La Síntesis substractiva es el método en donde una señal es generada por un oscilador para luego ser filtrada, dicha señal puede tener diferentes formas por tanto, su contenido armónico es variable. Las señales base de la síntesis substractiva, deben ser ricas en armónicos, estas señales pueden poseer cualquier forma de onda aunque las formas básicas como la triangular, diente de sierra, cuadrada son las usualmente usadas, menos la senoidal por su espectro pobre.

La técnica de síntesis granular se basa en sintetizar sonidos a partir de pequeños elementos de señal en el dominio del tiempo llamados sonidos atómicos o granos. Los granos pueden tener una duración comprendida entre un milisegundo y más de cien milisegundos.

La síntesis mediante tabla de ondas es el tipo de síntesis conocido como “PCM” o “WAVETABLE (TABLA DE ONDA)”, los sonidos muestreados se almacenan directamente tras un cuantificador PCM sin ningún tipo de codificación, por lo que su tamaño en memoria es elevado. Los sistemas que implementan síntesis por tabla de ondas tiene como parámetro más crítico el espacio en memoria y por tanto, todos los esfuerzos de desarrollo en esta síntesis han tenido como objeto primordial dicho espacio.

La síntesis de amplitud modulada (AM) consiste en modificar la amplitud de la señal portadora en función de la señal moduladora, en este caso la portadora es una señal periódica de alta frecuencia (tono) y la señal moduladora es una señal también periódica con un rango de frecuencia entre los 20 Hz y 20KHz que es el rango de frecuencia audible para el oído humano.

La Síntesis por modulación de frecuencia (FM) es el tipo de síntesis donde se varía la frecuencia de la señal portadora respecto a otra señal que es considerada como la señal moduladora, con lo que se genera finalmente una onda modulada en frecuencia (FM). Ahora para comenzar a explicar en qué consiste la síntesis modal, se considera un sistema mecánico tal como se presenta en la Figura A.1.

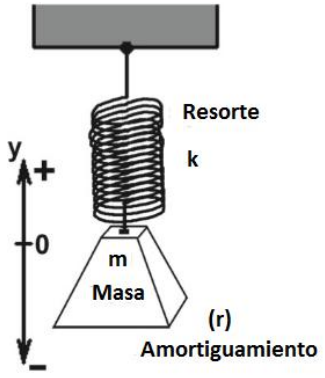


Figura A.1. Sistema masa/resorte con amortiguamiento [12].

En este sistema la masa se designa por “m”. Un resorte ideal se caracteriza por la fuerza requerida para desplazarlo desde la posición de reposo: Esta fuerza por unidad de distancia se designa por “k”. El desplazamiento de la masa se designa por “Y”, lo que es positivo para ascendente es la colocación y negativo para el desplazamiento hacia abajo. Esta elección de signo/dirección es arbitraria, y podría ser seleccionado para ser el contrario. Cualquier sistema real tendrá pérdidas, tales como el calor perdido en la compresión y estiramiento del resorte, y un poco de resistencia al viento de la masa a medida que avanza. Las pérdidas totales son designadas por “r”. Es posible escribir las ecuaciones de Newton simples para el sistema (segunda ley de Newton), dando [12]:

$$-ky - mg - rv = ma \qquad \text{Ecuación A.1}$$

De la ecuación anterior, se sustituye la velocidad y la aceleración como la primera y segunda derivada de la posición con respecto al tiempo respectivamente. Obteniendo lo siguiente:

$$-ky - r\frac{dy}{dt} = m\frac{d^2y}{dt^2} \qquad \text{Ecuación A.2.}$$

La ecuación A.2, se podría discretizar mediante aproximaciones [12]:

$$y(n) = y(n-1)(2m + Tr)/(m + Tr + T^2k) - y(n-2)m/(m + Tr + T^2k)$$

Ecuación A.3.

Esta ecuación se podría implementar en un código de programación por ejemplo en C y al graficar se tendría un resultado como el siguiente, tal como se muestra en la Figura A.2.

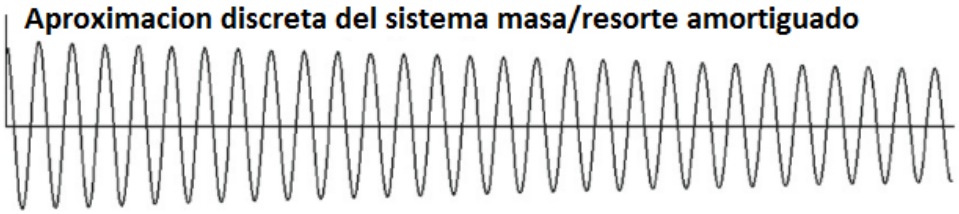


Figura A.2. Gráfica del resultado de la ecuación A.3 [12].

El sistema masa/resorte amortiguado muestra un único “modo” (una disminución exponencial de coseno en una sola frecuencia). La combinación de osciladores resonantes (modos) constituye la base de la síntesis modal, síntesis aditiva sinusoidal, y algunas formas de modelado [12].

La síntesis modal está basada en el hecho de que cualquier objeto que posee la capacidad de producir sonido, puede ser modelado y ser representado como un conjunto de subsistemas vibrantes, que quedan definidos por su caracterización modal. Estos subsistemas están acoplados entre si y pueden responder a excitaciones externas. No todos los sonidos se pueden sintetizar utilizando este método. El sonido debe de presentar cierto comportamiento senoidal decreciente, tal como se muestra en la Figura A.2. Una de las formas de cómo detectar dichos modos de un sonido, es por medio del espectro de potencia (el cuadrado de la magnitud de la FFT), estos modos al ser retirado de la señal, queda lo que se conoce como residuo.

Como un ejemplo práctico sobre síntesis modal, se muestra la Figura A.3, en donde se muestra la forma de onda del sonido y sus modos en el espectro de potencia. De ahí se le son removidos los modos principales y queda lo que se ha denominado como el residuo, tal y como se aprecia en la Figura A.4.

Cabe mencionar que la energía, la amplitud y la duración de este residuo serán menores a comparación del sonido original [12].

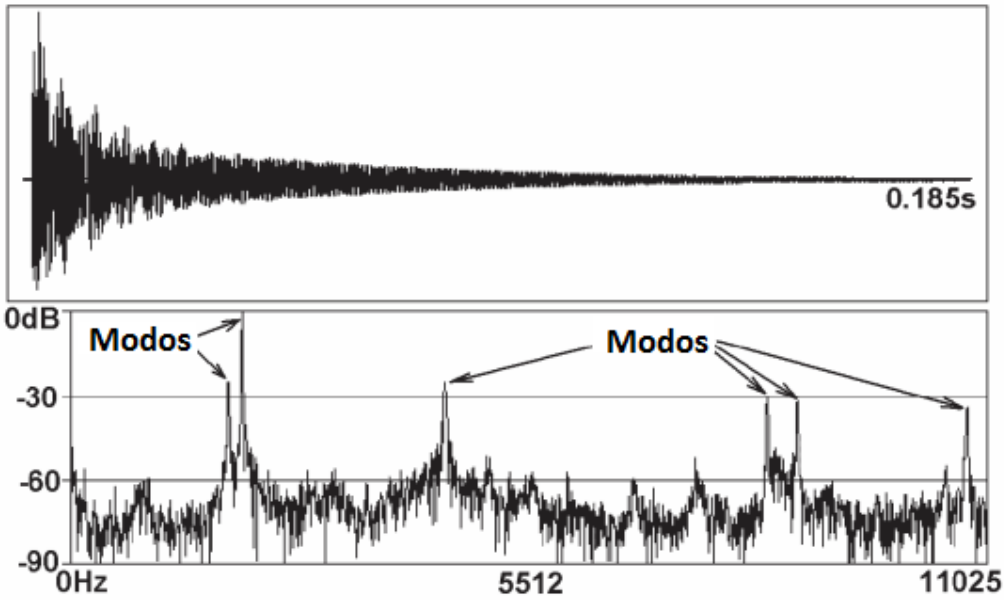


Figura A.3. Forma de onda y Espectro de Potencia en donde se pueden localizar los modos principales [12].

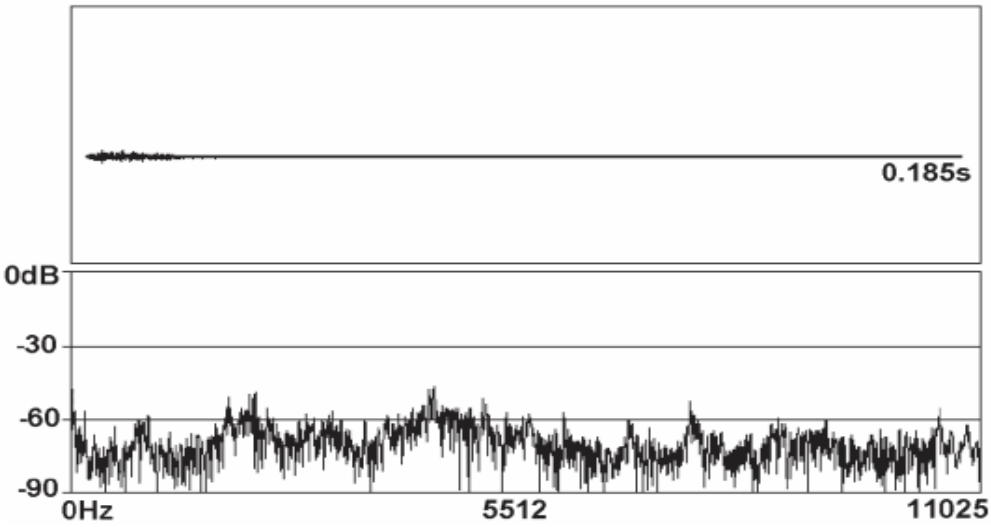


Figura A.4. Residuo y Espectro de Potencia [12].

Anexo B. Resonador Biquad.

Se entiende por filtro digital al hardware o software que es usado para eliminar ciertos rangos de frecuencias en una señal o para la supresión de ruido en la misma.

El filtro digital emplea un procesador digital que efectúa operaciones matemáticas en valores muestreados de la señal. El procesador puede ser de propósito general, tal como cualquier ordenador personal, un procesador digital de señales (DSP) o una FPGA programable.

La señal de entrada analógica debe ser muestreada y digitalizada usando un convertidor analógico-digital. El resultado de este proceso son números binarios que representan los valores sucesivos muestreados.

Estos valores son transferidos al procesador, el cual realiza operaciones matemáticas en estos valores; dichas operaciones pueden ser filtros de promediado de la muestra actual con la anterior muestra, así como también multiplicaciones por constantes de los valores de entrada o de instantes anteriores almacenados en la memoria del procesador.

Una forma especial de filtro es el Biquad [13], llamado así porque el numerador y el denominador de la función de transferencia son a la vez de segundo orden, o polinomios de segundo grado en Z^{-1}. El Biquad en el tiempo y en el dominio de la transformada Z se parece a:

$$y(n) = g(x(n) + a_1 x(x-1) + a_2 x(n-2) - b_1 y(n-1) - b_2 y(n-2)) \quad \text{Ecu. B.1}$$

$$\frac{Y}{X} = \frac{g(1+a_1 Z^{-1}+a_2 Z^{-2})}{1+b_1 Z^{-1}+b_2 Z^{-2}} \quad \text{Ecu. B.2}$$

Dependiendo de los valores de los coeficientes a y b, los polos y ceros se pueden colocar en posiciones bastantes arbitrarias en todo el plano z, pero no complemente arbitraria sí los coeficientes a y b son números reales (no complejo). Para el uso práctico en el procesamiento de sonido, hay una formula del Biquad que se ocupa más directamente con los parámetros del resonador [13]:

$$\frac{Y}{X} = \frac{g(1-2r_z \cos(2\pi Freq_z T)Z^{-1}+r_z{}^2 Z^{-2})}{1-2r_p \cos(2\pi Freq_p T)Z^{-1}+r_p{}^2 Z^{-2}} \qquad \text{Ecu. B.3}$$

$$y(n) = g(x(n) - 2r_z \cos(2\pi Freq_z T)\, x(n-1) + r_z{}^2 x(n-2) + 2r_p \cos(2\pi Freq_p T)\, y(n-1) - r_p{}^2 y(n-2) \qquad \text{Ecu. B.4}$$

Esto describe los coeficientes del filtro en función de un parámetro de amortiguamiento exponencial (r_z para los ceros y r_p para los polos) y una frecuencia central de la resonancia (anti resonancia para los ceros), que es la $Freq_z$ para los ceros y $Freq_p$ para los polos. Ahora podemos controlar aspectos del filtro de manera más directa a partir de estos parámetros, sabiendo que una vez que nos decidimos por r_z y $Freq_z$, podemos convertir a a_1, a_2, b_1 y b_2 directamente.

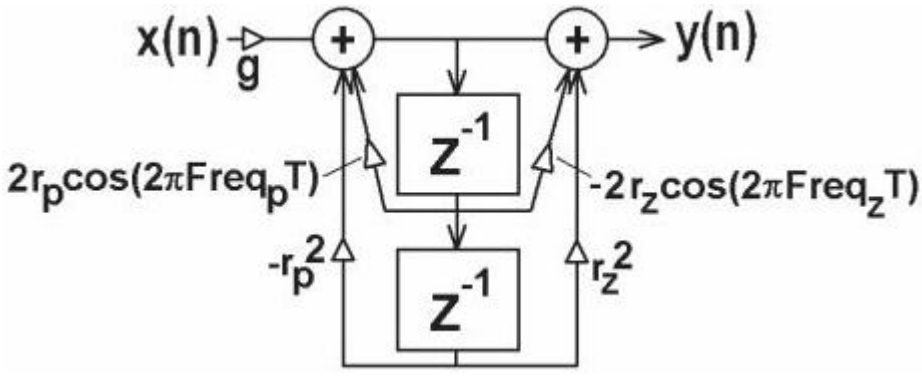


Figura B.1. Diagrama de bloques del Biquad.

Anexo C. Modelo "PIPELINE".

Para ejecutar las instrucciones, el ADSP-BF533 usa un modelo de encauzamiento entrelazado (*interlocked pipeline*). La estructura pipeline consiste en descomponer la ejecución de cada instrucción en varias etapas para poder empezar a procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez [3].

Cada una de las etapas de la instrucción usa en exclusiva un hardware determinado del procesador, de tal forma que la ejecución de cada una de las etapas en principio no interfiere en la ejecución del resto.

Por ejemplo, para un "PIPELINE" genérico con cuatro estados o fases: Búsqueda (Fetch), Decodificación (Decode), Ejecución (Execute) y Escritura (Write-Back), como se pueden ver en la Figura C.1, en el ciclo 0, hay cuatro instrucciones representadas por los cuadros de color amarillo, azul, rojo, verde y están esperando a ser ejecutados. En el ciclo 1, la instrucción amarilla está siendo sacada de la memoria (fetched), en el ciclo 2 la instrucción amarilla está siendo decodificada y a la vez la instrucción azul está siendo sacada de memoria y así sucesivamente hasta completar las cuatro instrucciones.

Se puede observar que el ciclo 4, todas las instrucciones están en una fase diferente del "PIPELINE", es decir, el procesador está trabajando con 4 instrucciones a la vez, pero en etapas diferentes, aumentando así la eficiencia en cuanto a la ejecución de instrucciones, porque se va a tener más instrucciones finalizadas en menos ciclos de reloj, a diferencia de un procesador no segmentado que tendría que esperar que una instrucción finalice, para poder iniciar una nueva instrucción.

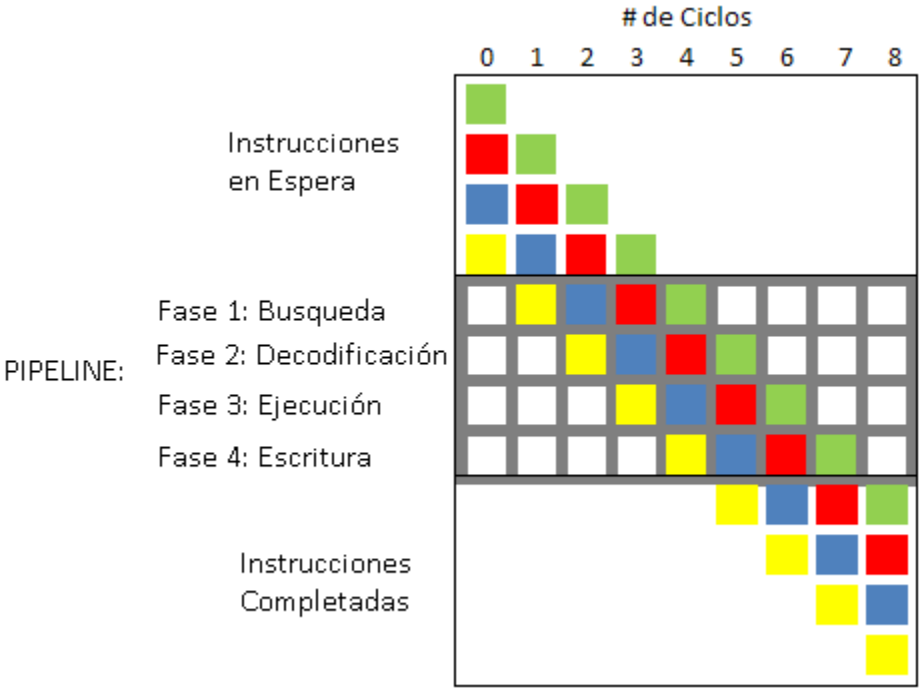


Figura C.1. Ilustración del "PIPELINE"

El ADSP-BF533 tiene un "pipeline" de 10 etapas, las cuales se explican a continuación:

1. *Búsqueda de la Instrucción 1 (IF1).*En esta etapa se emite la dirección de la instrucción al bus IAB, y a partir de esa dirección empieza a comparar la etiqueta.
2. *Búsqueda de la Instrucción (IF2).*Esta etapa espera los datos de la instrucción.
3. *Búsqueda de la Instrucción (IF3).*Esta etapa se encarga de leer la instrucción desde el bus IDB y la alinea.
4. *Decodificación de la instrucción (DEC).*Esta etapa decodifica las instrucciones.
5. *Cálculo de la Dirección (AC).*Calcula la dirección del dato y del objetivo.

6. *Búsqueda de Datos 1 (DF1).*Esta etapa se encarga de emitir la dirección del dato al bus DA0 y DA1, para empezar a comparar la etiqueta del caché de datos.
7. *Búsqueda de Datos 2 (DF2).*Lee los archivos del registro.
8. *Ejecución 1 (EX1).*Esta etapa lee los datos desde el bus LD0 y LD1, y posteriormente multiplica y ejecuta las instrucciones de video.
9. *Ejecución 2 (EX2).*Ejecuta y Completa las instrucciones tales como: desplazamiento, sumas, etc.
10. *Escritura (WB).*El resultado en los archivos del registro, bus SD y luego actualiza los punteros.

En la siguiente Figura se muestra el diagrama del "PIPELINE" que se ejecuta en el KIT.

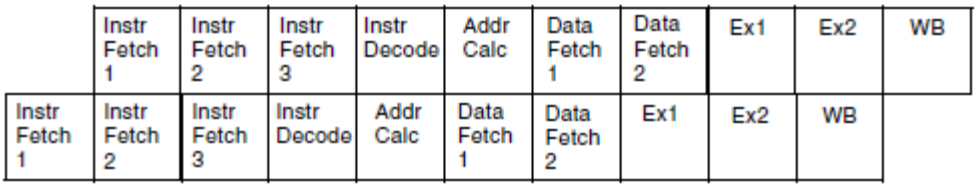


Figura C.2. Diagrama del "PIPELINE" ejecutado en el Blackfin BF533 [3].

En el ADSP-BF533, el secuenciador de programa es el encargado de determinar la dirección siguiente de la instrucción, examinando la actual dirección en ejecución y también examinando el estado actual del procesador.

Anexo D. Arquitectura RISC y Arquitectura de memoria en procesadores Blackfin.

D.1 Arquitectura RISC.

RISC (*Reduced Instruction Set Computer, de sus singles en inglés*), es un tipo de arquitectura que posee un conjunto reducido de instrucciones de computadora. Los procesadores que poseen esta arquitectura son llamados también: procesadores RISC, los cuáles se distinguen por tener un set de instrucciones con características determinadas; una de las características importantes de estos procesadores es el hecho de que sólo las instrucciones de carga y almacenamiento tienen acceso a la memoria de datos, así como también poseen instrucciones de un tamaño predeterminado y en formatos pequeños. Además, los procesadores RISC disponen de muchos registros de propósito general [14].

En los procesadores RISC, las instrucciones son más sencillas de implementar y los bloques lógicos que traducen a las instrucciones utilizan menos espacio logrando así aumentar la frecuencia de funcionamiento del sistema. Gracias a esta arquitectura, la unidad de control que es la encargada que bloques funcionales como la unidad aritmética lógica sea más sencilla comparada con otras arquitecturas [15].

En cuanto a desventajas en estos procesadores se tiene la baja potencia que ofrecen en cuanto a las operaciones que se aceleran mucho debido a instrucciones complejas matemáticas asociadas a simulaciones, tratamiento de la señal, video e imágenes; en estos casos es mejor contar con otro tipo de arquitecturas que posean instrucciones más complejas que puedan tratar con cientos de datos y escribirlos en memoria de una sola vez [15].

Los procesadores Blackfin de Analog Devices están desarrollados a partir de un set de instrucciones de 16 ó 32 bits RISC. Todos los miembros de la familia Blackfin comparten el mismo núcleo, con la única diferencia de la capacidad de memoria, la frecuencia de operación, así como también el consumo de potencia y el número de periféricos que estos poseen.

D.2 Arquitectura de memoria en procesadores Blackfin.

La memoria del procesador Blackfin se basa en una arquitectura Harvard modificada, dónde los espacios de direcciones de memoria se organizan en una estructura jerárquica para proporcionar una buena relación costo/rendimiento [16].

Al ser de arquitectura Harvard modificada, conlleva a que los buses tanto de datos como de instrucciones estén separados entre sí, lo que significa que el procesador pueda acceder a dos o más memorias de buses simultáneamente, además de permitir el rápido intercambio de datos entre el núcleo y la memoria L1 (datos e instrucciones).

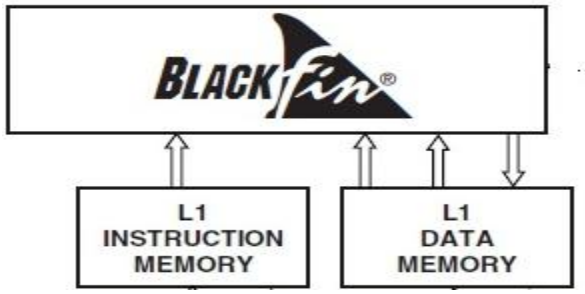


Figura D.1. Diagrama en bloques de arquitectura de memoria del procesador Blackfin.

En la *Figura D.1* se puede observar que la memoria L1 del procesador Blackfin se divide en una memoria de instrucciones y una de memoria de datos, las cuales poseen buses dedicados para comunicarse con el procesador lo que genera un flujo de datos constantes, y así brindarle la mayor rapidez y rendimiento al usuario.

Anexo E. Representación Fraccionaria.

El procesador ADSP-BF533 posee una unidad de punto fijo, es llamado a menudo máquina de punto fijo por ser de 16 bits. Las DSP de punto fijo es el arte de representación de los números reales con números enteros. Supongase que se tiene enteros de 16 bits. Suele decirse esto representa el numero 322767 a -32768. Los códigos binarios se asignan a los números de esta manera [17]:

Binario	Numero
0000 0000 0000 0001	1
0000 0000 0000 0010	2
0111 1111 1111 1111	32767
1000 0000 0000 0000	-32768

Tabla E.1. Representación fraccionaria fract16.

La ventaja del procesador, al ser de punto fijo, es el bajo consumo de energía, una mayor rapidez de procesado y por supuesto el costo para adquirirlos es mucho menor que un procesador de punto flotante.

Vía software es posible hacer que un procesador de punto fijo emule las operaciones de los procesadores de punto flotante, pues se sabe que estos procesadores ofrecen mayor precisión en los cálculos.

Formato	# de bits enteros	# de bits fraccionarios	Valor Máximo Positiva (0x7FFF) en Decimal	Valor Max. Neg. (0x8000) en Decimal	Valor de 1 LSB (0x0001) en decimal
1.15	1	15	0.999969482421875	-1.0	0.000030517578125
2.14	2	14	1.999938964843750	-2.0	0.000061035156250
3.13	3	13	3.999877929687500	-4.0	0.000122070312500

Tabla E.2. Representaciones Fraccionarias.

GLOSARIO.

AAU (*Address Arithmetic Unit*), unidad aritmética de direcciones, encargada de generar las direcciones tanto de memoria como de los puertos de entrada y salida (E/S).

ADC (*Analog to Digital Converter*), convertidor análogico a digital, es un sistema que convierte una señal análogica en una señal digital.

Algoritmo LMS (*Least Mean Square algorithm*), se usa en filtros adaptativos para encontrar los coeficientes del filtro que permiten obtener el valor esperado mínimo del cuadrado de la señal de error, definida como la diferencia entre la señal deseada y la señal producida a la salida del filtro.

ALU (*Arithmetic Logical Unit*), unidad aritmética lógica, es un circuito digital que calcula operaciones aritméticas como la suma, resta; operaciones lógicas tales como OR, AND, NOT, así como funciones de valor absoluto y redondeo.

API (*Application Programming Interface*), interfaz de programación de aplicaciones, es el conjunto de subrutinas, funciones y procedimientos (o métodos en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

BUFFER Ubicación de memoria, reservada para el almacenamiento momentáneo de información digital, mientras espera a ser procesada.

DAC (*Digital to Analog Converter*), convertidor digital a analógico, es un sistema que convierte una señal digital en una señal analógica.

DAG (*Data Address Generator*), generador de datos de direcciones, es el mecanismo que genera las direcciones de memoria temporal para los datos que se transfieren entre la memoria y los registros en un procesador digital de señales.

DMA (*Direct Memory Access*), conocido como acceso directo a memoria, ésta característica permite al procesador mover datos entre la memoria y un periférico o entre memoria y memoria sin necesidad de intervención del procesador.

DPM (*Dynamic Power Management*), manejo dinámico de energía. Característica del procesador Blackfin para manejar dinámicamente el voltaje y la frecuencia de operación, para mantener un adecuado consumo de energía para cada aplicación.

DSP (*Digital Signal Processor*), es un procesador digital de señales, el cuál posee un conjunto de instrucciones, un hardware y un software optimizado para aplicaciones que requieran operaciones aritméticas a muy alta velocidad.

EBIU (*External Bus Interface Unit*), unidad de interfaz de bus externo, conecta la memoria externa al procesador ADSP-BF533.

EPROM (*Erasable Programmable Read Only Memory*), memoria de sólo lectura programable borrable, éstas memorias se programan mediante un dispositivo electrónico que proporcionan voltajes superiores a los normalmente utilizados en circuitos integrados. Una vez programada esta memoria se puede borrar solamente ante la exposición de una luz ultravioleta fuerte en el cristal de la parte alta del encapsulado.

FETCH Fase o estado dentro de la operación de ejecución de una instrucción relacionada a la búsqueda de la instrucción.

FFT (*Fast Fourier Transform*), transformada rápida de Fourier, algoritmo eficiente usado en el tratamiento digital de señales que permite calcular la transformada discreta de Fourier (DFT) y su inversa.

FPGA (*Field Programmable Gate Array*), arreglo de compuertas programables, es un dispositivo programable que contiene bloques de lógica cuya interconexión y funcionalidad puede ser configurada ”in situ” mediante un lenguaje de descripción especializado.

FULL-DUPLEX Modo de transmisión de datos que permite el flujo en ambas direcciones simultáneamente por el mismo canal. Existen dos frecuencias una para transmitir y otra para recibir.

HALF-DUPLEX Modo de transmisión de datos en dónde los datos pueden transmitirse en ambas direcciones en un portador de la señal, pero no al mismo tiempo.

Inactividad o IDLE Es una instrucción que provoca que el procesador deje de operar y mantenga su estado actual hasta que una interrupción lo reactive.

MAC Unidad de multiplicación – acumulación, permite realizar operaciones de multiplicación y multiplicación/acumulación de punto fijo.

NTSC (*National Television System Committee*), Comité Nacional de Sistema de Televisión, es el sistema de televisión analógico que se ha empleado en América del Norte, América Central, la mayor parte de América del Sur, Japón, entre otros.

PAL (*Phase Alternating Line*), línea de fase alternada, es el sistema de codificación utilizado en la transmisión de señales de televisión analógica en color en la mayor parte del mundo. Utilizado en la mayor parte de países europeos, africanos y asiáticos.

PIPELINE PIPELINE o segmentación es una secuencia de operación del bus durante la ejecución de una instrucción en donde se descompone el proceso de ejecución de una instrucción en varias etapas para poder empezar a procesar una instrucción diferente en cada una de ellas y trabajar con varias a la vez.

PPI (*Parallel Peripherical Interface*), interfaz de periféricos paralelo, es un periférico que se encuentra en el procesador Blackfin y se utiliza para conectar pantallas LCD, sensores CMOS y cualquier otro dispositivo paralelo de alta velocidad.

PWM (*Pulse Width Modulation*), modulación por ancho de pulso, es una técnica en la que se modifica el ciclo de trabajo de una señal periódica, ya sea para transmitir información a través de un canal de comunicaciones o para controlar la cantidad de energía que se envía a una carga.

ROM (*Read Only Memory*), memoria de sólo lectura, es un medio de almacenamiento que permite solo lectura de información y no su escritura, independientemente de la presencia o no de una fuente de energía.

SCRATCHPAD Memoria interna de nivel 1 de alta velocidad que se utiliza para el almacenamiento temporal de los cálculos, datos y otros trabajos en curso.

SIMD (*Single Instruction Multiple Data*), Una sola instrucción múltiples datos, es una técnica empleada para conseguir paralelismo a nivel de datos.

SPI (*Serial Peripheral Interface*), interfaz de periféricos serie, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos.

SPORT Puerto Serial, ofrece una interfaz tanto para entrada como de salida para una amplia variedad de dispositivos periféricos seriales.

SRAM (*Static Random Access Memory*), memoria estática de acceso aleatorio, es un tipo de memoria RAM basada en semiconductores, capaz de mantener los datos mientras siga energizada, todo esto sin necesidad de un circuito de refresco.

TDM (*Time Division Multiplexing*), Multiplexación por División de Tiempo, usado especialmente en los sistemas de transmisión digitales, en dónde el ancho de banda total del medio de transmisión es asignado a cada canal durante una fracción del tiempo total.

UART (*Universal Asynchronous Receiver-Transmitter*), Transmisor-Receptor Asincrono Universal, es el dispositivo que controla los puertos y dispositivos serie, se encuentra en la placa base o en la tarjeta adaptadora del dispositivo.

Watchdog timer Temporizador “perro guardián”, es un mecanismo de seguridad que provoca un reset del sistema en caso de que éste se haya bloqueado.

REFERENCIAS BIBLIOGRÁFICAS.

[1]. W. L. Zelaya Chicas, "Diseño de un filtro digital adaptativo como cancelador de ruido basado en el algoritmo LMS," Trabajo de graduación Ing. Eléctrica, Esc. de Ing. Eléctrica, Univ. de El Salvador, San Salvador, Enero 2004.

[2]. *Blackfin Embedded Processor ADSP-BF531/ADSP-BF532/ADSP-BF533*. Rev. E, Analog Devices, Norwood, Mass., 2007.

[3]. *ADSP-BF533 BlackfinProcessor Hardware Reference*. Rev. 3.2, Analog Devices, Norwood, Mass., 2006.

[4]. *ADSP-BF533 EZ-KIT Lite Evaluation System Manual*. Rev. 3.0, Analog Devices, Norwood, Mass., 2006.

[5]. *Preliminary Technical Data AD1836*, Analog Devices, 2001.

[6]. *The Chuck Manual*. Ge Wang, Perry Cook, 2007.

[7]. *VisualDSP++ 5.0 C/C++Linker and Utilities Manual*. Rev. 2.0, Analog Devices,Norwood, Mass., 2007.

[8]. *VisualDSP++ 5.0 User's Guide*. Rev. 3.0, Analog Devices, norwood, Mass., 2007.

[9]. Audacity. Disponible en: www.audacityteam.org, [En línea]. [Última consulta 18/09/2016].

[10]. Cook, Perry R., *Real Sound Synthesis for Interactive Applications*, 1st ed. A K Peters/CRC Press, 2015. VitalBook file. Ch. 2.

[11]. Javier. (2009, Jul. 21). *Tutorial rápido de sonido e informática musical.* Disponible en: http://audiosintesis.blogspot.com [En línea]. [Última consulta 18/09/2016].

[12]. Cook, Perry R., *Real Sound Synthesis for Interactive Applications*, 1st ed. A K Peters/CRC Press, 2015. VitalBook file. Ch. 4.

[13]. Cook, Perry R., *Real Sound Synthesis for Interactive Applications*, 1st ed. A K Peters/CRC Press, 2015. VitalBook file. Ch. 3.

[14]. Wikipedia, La enciclopedia libre, "*Reduced instruction set computing".* Disponible en: https://es.wikipedia.org/wiki/Reduced_instruction_set_computing [En línea] [Ultima consulta 18/09/2016].

[15]. A. L. Sánchez Iglesias**, "***¿Qué es un procesador RISC?****".** Disponible en: http://computadoras.about.com/od/Tecnologias/a/Procesador-Risc.htm [En línea] [Ultima consulta 18/09/2016].

[16]. D. Zuluaga Arias, C. Zuluaga Arias, E. A. Quintero Salazar, "ARQUITECTURA DE LA DSP BLACKFIN ADSP- BF533 DE ANALOG DEVICES Y SU VIABILIDAD EN EL PROCESAMIENTO DIGITAL DE IMAGENES*"*. Disponible en: *https://dialnet.unirioja.es/descarga/articulo/4723694.pdf* [En línea] [Ultima consulta 18/09/2016].

[17]. Rowetel, "Fixed Point Scaling – Low Pass Filter Example". Disponible en: http://www.rowetel.com/?p=1245 [En línea] [Ultima consulta 18/09/2016].