

# CENTRO DE ENSEÑANZA TÉCNICA INDUSTRIAL



IMPLEMENTACIÓN DE SISTEMAS DE PROCESAMIENTO DE  
SEÑALES E IMÁGENES MEDIANTE LA TARJETA DE  
DESARROLLO ATLYS DE XILINX Y SIMULINK.

“PROTOTIPO”

SUSTENTANTE:  
DIEGO ARMANDO HERNÁNDEZ RAMÍREZ

CARRERA:  
TECNÓLOGO EN ELECTRÓNICA Y COMUNICACIONES.

ASESOR:  
MTRO. JOSÉ MARÍA VALENCIA VELASCO

GUADALAJARA, JAL. A 11 DE JULIO DE 2015.



## **Hoja de Aprobación.**

**Uso de Simulink y la Tarjeta de desarrollo Atlys de Xilinx para la implementación de Sistemas de Procesamiento de Señales e imágenes.**

Diego Armando Hernández Ramírez.

**Asesor técnico:** Mtro. José María Valencia Velasco.

**Asesor metodológico:** Mtro. Cuauhtémoc Rafael Aguilera Galicia.

## **Resumen**

Placeholder, TBD;

## Agradecimientos

# Índice

<b>I Introducción.</b>	<b>16</b>
1. Planteamiento del problema.	17
2. Propósito.	18
2.1. Objetivo general. . . . .	18
2.2. Objetivos específicos. . . . .	18
<b>II Antecedentes.</b>	<b>19</b>
3. Procesamiento Digital de Señales.	20
3.1. Elementos de un sistema DSP. . . . .	20
3.1.1. Teorema de muestreo. . . . .	21
3.1.2. Cuantización. . . . .	22
3.2. Sistemas Discretos Lineales e Invariantes en el Tiempo.. . . . .	23
3.2.1. Convolución. . . . .	24
3.2.2. Filtros FIR. . . . .	26
3.2.3. Filtros IIR. . . . .	28
4. Plataformas FPGA.	30
4.1. Descripción general. . . . .	30
4.2. Estructura general de las FPGA. . . . .	31
4.3. Arquitectura de la familia de FPGAs Xilinx Spartan-6. . . . .	33
4.4. Kit de desarrollo Atlys de Digilent. . . . .	37

<b>5. Plataformas de Software para el diseño de sistemas DSP.</b>	<b>38</b>
5.1. Introducción a MATLAB y Simulink. . . . .	38
5.1.1. MATLAB. . . . .	38
5.1.2. Simulink. . . . .	40
5.2. Xilinx System Generator for DSP. . . . .	41
5.2.1. Descripción general. . . . .	41
5.2.2. Co-Diseño de Hardware/Software en el ambiente de Xilinx System Generator for DSP. . . . .	43
5.3. Introducción a Xilinx ISE y PlanAhead. . . . .	44
5.3.1. Reseña de Xilinx ISE. . . . .	44
5.3.2. PlanAhead. . . . .	47
<b>III Marco metodológico.</b>	<b>49</b>
<b>6. Filtrado básico de señales - Introducción práctica a Sysgen.</b>	<b>50</b>
6.1. Diseño de un filtro FIR pasabajas para la eliminación de ruido en una señal senoidal. . . . .	50
6.2. Especificaciones del modelo del filtro FIR en Matlab. . . . .	50
6.3. Simulación del sistema en Sysgen. . . . .	53
6.4. Implementación del sistema en el kit Atlys. . . . .	60
<b>7. Procesamiento de Audio - Algoritmo de eco.</b>	<b>63</b>
7.1. Filtros FIR peine o <i>comb filter</i> . . . . .	63
7.2. Implementación en FPGA del códec de audio AC97. . . . .	63
7.3. Desarrollo del modelo en Sysgen. . . . .	64
7.4. Implementación en el kit Atlys. . . . .	70
<b>8. Procesamiento de imágenes - Detección de bordes.</b>	<b>75</b>
8.1. Características técnicas y arquitectura del modelo a implementar. . . . .	75
8.2. Algoritmo del filtro Sobel. . . . .	75
8.3. Descomposición del espacio de colores RGB. . . . .	76
8.4. Modelo de conversión RGB a escala de grises. . . . .	78
8.5. Operador Sobel usando filtros FIR. . . . .	82
8.6. Implementación y resultados. . . . .	87

<b>IV Conclusiones.</b>	<b>90</b>
<b>Referencias</b>	<b>101</b>
<b>Bibliografía</b>	<b>101</b>

# Índice de figuras

3.1. Diagrama a bloques de un Sistema de procesamiento Digital de Señales. . . . .	21
3.2. Banda de espectro limitado típica [13]. . . . .	21
3.3. Ejemplo de cuantización en una señal senoidal. Los puntos marcados con una «x» representan la señal original, mientras que las coordenadas marcadas con «↔» muestran la señal cuantizada . . . . .	22
3.4. Función delta graficada. . . . .	24
3.5. Función de respuesta al impulso graficada. . . . .	25
3.6. Convolución de dos vectores en Matlab. . . . .	26
3.7. Respuesta al impulso de un filtro FIR con $b_0 = 0.9$ . . . . .	27
3.8. Estructura en Forma Directa de un Filtro FIR. Imagen creada usando el paquete TikZ de LATEX	28
3.9. Estructura en Forma Directa de un Filtro IIR. Imagen creada usando el paquete TikZ de LATEX	30
4.1. Arquitectura general de las FPGA [14]. . . . .	32
4.2. LUT de seis entradas [2]. . . . .	33
4.3. Diagrama a bloques de una FPGA Spartan-6 [2]. . . . .	34
4.4. Bloque Lógico Configurable. . . . .	35
4.5. Estructura detallada de un SLICE [2]. . . . .	36
4.6. Block RAM de doble puerto [2]. . . . .	37
4.7. Kit de desarrollo Atlys Spartan-6 de Digilent, obtenida del sitio web del fabricante[6]. . . .	38
5.1. Interfaz Gráfica del Usuario de MATLAB. . . . .	39
5.2. Entorno de modelado gráfico Simulink de MATLAB. En este ejemplo se puede observar, como se lleva a cabo la ejecución de un algoritmo de detección de elementos en una imagen, utilizando solamente 8 bloques del <i>Image Processing Toolbox</i> de Simulink. . . . .	40
5.3. Flujo de diseño con System Generator de Xilinx, imagen obtenida del documento “ <i>Wireless communications: from systems to silicon</i> ”, del autor Raghu Rao. Wireless Systems Group, Xilinx Inc, 2008. . . . .	42
5.4. Ejemplo de blocksets incluidos en el toolbox de Sysgen, en su versión ISE 14.7. . . . .	42
5.5. Diagrama a bloques del flujo de implementación mediante <i>Hardware-in-the-loop</i> . . . . .	43
5.6. Ejemplo de implementación de un diseño completo utilizando el toolbox de Sysgen en Simulink. Imagen tomada de la guía de usuario “ <i>System Generator for DSP User Guide, UG640</i> ”. Se puede observar como el procesador Microblaze, puede transmitir y recibir impulsos del modelo en Sysgen. . . . .	44

5.7. Interfáz principal del <i>ISE Project Navigator</i> . . . . .	45
5.8. Diagrama a flujo del proceso de compilación en FPGA, elaborado usando el paquete <i>TikZ</i> de <i>LATEX</i> . . . . .	46
5.9. Interfáz gráfica de PlanAhead, imagen obtenida del paquete Xilinx ISE. . . . .	48
6.1. Gráfica derivada de la función diseñada en Matlab para este ejercicio. . . . .	51
6.2. Respuesta en magnitud del filtro FIR diseñado, donde se puede observar la atenuación de frecuencias mayores a $f_c = 7\text{KHz}$ . . . . .	53
6.3. Resultado de la implementación del filtro FIR, donde se puede observar la señal resultante, de un color más oscuro, en oposición a la señal contaminada. La señal resultante está totalmente reconstruida, lo que valida la implementación. . . . .	53
6.4. Mensaje de instalación del toolbox de Sysgen en Matlab R2013a. . . . .	54
6.5. Instanciación del System Generator token. . . . .	54
6.6. Bloques de entrada/salida que acotan la implementación en el FPGA del algoritmo a diseñar. . . . .	55
6.7. Generadores de función senoidal utilizados para generar la señal a filtrar. . . . .	56
6.8. Parámetros de configuración para la componente base de entrada al sistema de filtrado. . . . .	57
6.9. Parámetros de configuración para la componente de ruido de entrada al sistema de filtrado. . . . .	57
6.10. Configuración del bloque FIR Compiler. . . . .	58
6.11. Interconexión de los bloques de Simulink/Sysgen para la elaboración del filtro FIR. . . . .	59
6.12. Parámetro de tiempo de simulación para este ejercicio. . . . .	59
6.13. Resultado de la simulación del sistema utilizando los bloques de Sysgen. . . . .	59
6.14. Generación de la representación del kit Atlys en Simulink, el cual contiene los recursos de hardware necesario para ejecutar el algoritmo del filtro FIR. . . . .	60
6.15. Conexión del bloque de Co-simulación. . . . .	61
6.16. Comparación entre el resultado de la simulación en contraste con la ejecución en FPGA como co-procesador. . . . .	61
6.17. Diagrama esquemático de la implementación del modelo del filtro FIR en FPGA. . . . .	62
6.18. Utilización de recursos de la FPGA para la implementación del filtro FIR. . . . .	62
7.1. Diagrama a bloques del filtro peine o <i>comb filter</i> . . . . .	63
7.2. Diagrama a bloques de la implementación del controlador para el códec AC97 en Verilog. . . . .	64
7.3. Configuración del bloque <i>From multimedia File</i> para servir como fuente de sonido. . . . .	65
7.4. Configuración del tipo de dato usado por los puertos de entrada al filtro peine. . . . .	66

7.5.	Filtro comb (FIR) con retroalimentación realizado en Sysgen. . . . .	67
7.6.	Parámetros de la ganancia del filtro comb. . . . .	67
7.7.	Ganancia de retroalimentación para estabilizar el sistema. . . . .	68
7.8.	Conversión del filtro comb en subsistemas y replicación para formato estéreo. . . . .	68
7.9.	Interconección del sistema de eco finalizado. . . . .	69
7.10.	Gráfica resultante del eco aplicado al archivo de audio. La señal de la parte superior representa el audio sin procesar, mientras que la señal inferior muestra el audio procesado, el cual se compone de un desfase y una amplitud diferente a la original. . . . .	69
7.11.	Vista jerárjica de las fuentes utilizadas para el proyecto. . . . .	72
7.12.	Archivos de Verilog que contienen el hardware realizado en Sysgen para implementar el algoritmo de procesamiento de audio. . . . .	73
7.13.	Esquemático completo de la implementación del modelo de procesamiento de audio. . . .	73
7.14.	Utilización y frecuencia máxima de trabajo de la implementación final. . . . .	74
7.15.	Conexiones a la tarjeta Atlys para ejecutar el algoritmo desarrollado. . . . .	74
8.1.	Imagen utilizada para la implementación del filtro Sobel. . . . .	75
8.2.	Parámetros de configuración del bloque ' <i>Image From File</i> '. . . . .	77
8.3.	Configuración de los bloques ' <i>Transpose</i> ', ' <i>Reshape</i> ', ' <i>Frame Conversion</i> ' y ' <i>Unbuffer</i> ', respectivamente. . . . .	77
8.4.	Conexión de los bloques necesarios para convertir la matriz 2D en un vector de 1D necesario para cada una de las componentes del espacio de color. . . . .	78
8.5.	Conexión entre la fuente de imagen hacia el bloque de conversión de la matriz 2D a vectores por espacio de color. . . . .	78
8.6.	Configuración para las entradas RGB al sistema en FPGA. . . . .	79
8.7.	Entradas registradas usando bloques de Sysgen, con sus configuraciones originales. . . .	80
8.8.	Configuración de los multiplicadores por el valor constante requerido para R, G y B respectivamente. . . . .	80
8.9.	Modelo completo que realiza la conversión RGB a escala de grises de la imagen obtenida por el bloque ' <i>Image From File</i> '. . . . .	81
8.10.	Parámetros necesarios para el bloque ' <i>Convert 1-D to 2-D</i> '. . . . .	81
8.11.	Reconstrucción de la imagen procesada por los bloques FPGA. . . . .	82
8.12.	Resultado esperado de la conversión RGB a escala de grises. . . . .	82
8.13.	Coeficientes del filtro FIR para el eje X de la imagen. . . . .	83

8.14. Configuración del tipo de dato de los coeficientes para el filtro FIR. . . . .	84
8.15. Coeficientes del filtro FIR para el eje Y de la imagen. . . . .	84
8.16. Conexión de los filtros FIR y sus terminales. . . . .	85
8.17. Adición de los valores calculados por ambos filtros FIR. . . . .	85
8.18. Modelo para obtener el valor absoluto de los datos arrojados por los filtros FIR. . . . .	86
8.19. Modelo completo del filtro Sobel. . . . .	86
8.20. . . . .	87
8.21. Resultados de la simulación del modelo del filtro Sobel. . . . .	87
8.22. Bloque de Co-Simulación del filtro Sobel, que se ejecuta en FPGA. . . . .	88
8.23. Conexión del modelo completo para FPGA, el cual ejecuta el resultado simulado y el obtenido por el hardware reconfigurable. . . . .	88
8.24. Resultado de la ejecución en FPGA del modelo del filtro Sobel. . . . .	88
8.25. Gráfica de utilización de recursos en la FPGA al implementar el modelo del operador Sobel.	89
.26. Información digital convertida en un tren de impulsos .[18] . . . . .	92
.27. Representación gráfica de una señal producida por el efecto de <i>retención de orden cero</i> en el proceso de transformación A/D [19]. . . . .	93
.28. Estructura de un DAC resistivo de ponderación binaria. Imagen adaptada de: [17] . . . . .	93
.29. Estructura de un DAC capacitivo de ponderación binaria. . . . .	94
.30. DAC R-2R red en escalera. . . . .	94
.31. Modelo en NGSpice de un DAC ideal de 4 bits. . . . .	94
.32. Selección del tipo de proyecto en PlanAhead. . . . .	96
.33. Modelo de ejemplo en Sysgen. . . . .	97

## **Índice de cuadros**



## **Parte I**

### **Introducción.**

## 1. Planteamiento del problema.

En la actualidad la electrónica está presente en prácticamente en todos los aspectos de nuestra vida a través de una gran infinidad de dispositivos y sistemas: teléfonos inteligentes, monitores de ritmo cardiaco, cámaras fotográficas, televisores, automóviles, refrigeradores, computadoras, etc.

Todos estos dispositivos realizan de manera interna la manipulación e interpretación de señales eléctricas, que en otras palabras es lo que se conoce como procesamiento de señales. El procesamiento de una señal puede aplicarse, por ejemplo, en el reconocimiento de voz para determinar quién es la persona que habla; para determinar, mediante una imagen, piezas defectuosas en una línea de producción o para la protección de información (encriptación).

El procesamiento de señales involucra la realización de operaciones matemáticas sobre las señales, las cuales son llevadas por sistemas cuya única función es precisamente el llevar a cabo esas operaciones, los procesadores digitales de señales (DSP, por sus siglas en inglés) y los arreglos programables (FPGA, por sus siglas en inglés) son los encargados de ello.

En muchas aplicaciones de procesamiento de señales se requiere una velocidad de procesamiento elevada (por ejemplo procesamiento de video) por lo que, debido al paralelismo de su operación, los FPGA son aptos para ser utilizados en ellas[9].

Con el fin de explotar las ventajas que los FPGA poseen y poderlos aplicar de una manera eficaz en el procesamiento de señales es necesario contar con sólidos conocimientos principalmente en metodologías de diseño digital e implementación matemática de algoritmos; estos conocimientos deben adquirirse desde la academia, puesto que es el tiempo ideal en que el futuro ingeniero o arquitecto de sistemas puede ir desarrollando, a través de la experimentación, las habilidades necesarias para crear prototipos en el que se involucre el procesamiento de señales.

Teniendo como objetivo principal el recortar la curva de aprendizaje, las empresas líderes en FPGA como Xilinx, Altera y Synopsys proporcionan plataformas de trabajo que puede interactuar con Matlab (software especializado que permite la implementación y prueba de algoritmos). De esta forma, el alumno puede poner en práctica de forma ágil y sin complicaciones los conocimientos adquiridos en las áreas de procesamiento digital de señales.

Muchas veces la información que el fabricante proporciona sobre sus plataformas de trabajo es escasa y

poco concreta, lo que puede impactar negativamente en el interés del alumno, provocando que los conocimientos y conceptos no queden del todo entendidos.

## 2. Propósito.

### 2.1. Objetivo general.

- Describir el proceso de implementación de un sistema de procesamiento de señales e imágenes mediante hardware reconfigurable (FPGA) y la tarjeta de desarrollo Atlys.

### 2.2. Objetivos específicos.

- Facilitar el diseño e implementación de un sistema de procesamiento de audio en tiempo real, basado en el desarrollo de un algoritmo de eco, así como un sistema de detección de bordes en una imagen basado en el algoritmo Sobel, ambos utilizando bloques de Xilinx System Generator para Simulink.
- Mostrar la conversión de los algoritmos matemáticos básicos que intervienen en el procesamiento de señales, a hardware en FPGA, haciendo uso de la abstracción que proporciona Simulink
- Describir las técnicas de implementación más eficientes para obtener el mayor rendimiento sobre la familia FPGA Spartan 6 utilizada en la tarjeta Atlys
- Diseñar las propiedades intelectuales (IPs) más comunes en el tratamiento de señales tales como bloques de filtros FIR, IIR y convoluciones, utilizando los entornos de programación de MATLAB® y Xilinx®.
- Explicar los diferentes métodos de ejecución del hardware diseñado en Simulink®, sobre la tarjeta Atlys.
- Justificar el uso de MATLAB/Simulink® y Xilinx/ISE® para el diseño e implementación de algoritmos complejos en contraste con el uso tradicional de HDL puro.

## **Parte II**

### **Antecedentes.**

### 3. Procesamiento Digital de Señales.

#### 3.1. Elementos de un sistema DSP.

El procesamiento digital de señales es una de las tecnologías más vanguardistas que ha marcado varios segmentos tecnológicos como las comunicaciones digitales, ciencias médicas, diseño de radares, reproducidores de música de alta fidelidad, por nombrar sólo algunas. Esta área se distingue de todas las demás dentro de las Ciencias de la Computación, por el tipo de datos único que utiliza: **las señales [18]**.

El propósito principal de un sistema de procesamiento digital de señales es manipular matemáticamente algún tipo de información tomada del mundo real. Dicha información es naturalmente analógica, es decir, la representación de sus valores y funciones son continuos, por lo que necesita ser previamente digitalizada antes de ser procesada. Algunos ejemplos de señales comúnmente utilizados son voz, audio, video, temperatura y presión.

Generalmente las operaciones que se realizan sobre dichas señales son adiciones, sustracciones, multiplicaciones y divisiones, mismas que se deben ejecutar muy rápidamente con el fin de generar una salida mucho más precisa[3]. Esto se hace para cumplir una amplia variedad de objetivos, tales como: mejoras en la visualización de imágenes, reconocimiento y generación de voz, compresión de datos para almacenamiento y transmisión, etc.

El procesamiento digital de señales analógicas puede ser descrito en tres etapas:

- La señal analógica es *digitalizada*, es decir, se *muestrea* y cada muestra es a su vez, *cuantizada* a un número finito de bits. Este proceso es llamado **conversión analógico-digital**.
- Los muestreros digitalizados son procesados por un *Procesador Digital de Señales*.
- Las muestras resultantes de la etapa de procesamiento, se convierten de nuevo a un formato *analógico* mediante alguna técnica de reconstrucción analógica (**conversión digital-analógico**).

Un sistema DSP típico se muestra en la Figura 3.1.

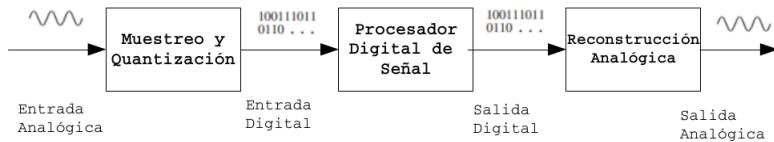


Figura 3.1: Diagrama a bloques de un Sistema de procesamiento Digital de Señales.

Las secciones a continuación describen con más detalle, los teoremas y técnicas fundamentales que intervienen en los sistemas DSP.

### 3.1.1. Teorema de muestreo.

El *teorema de muestreo* establece que para una representación precisa de una señal  $x(t)$  por sus muestras de tiempo  $x(nT)$ , dos condiciones deben cumplirse:

- La señal  $x(t)$  debe ser de banda limitada, es decir, su espectro de frecuencia debe ser limitada para contener las frecuencias hasta cierta frecuencia máxima, digamos  $f_{\max}$ , y sin frecuencias más allá de eso. Un espectro típico de banda limitada se muestra en la Figura 3.2.
- La frecuencia de muestreo  $f_s$  debe ser elegida para ser al menos el doble de la frecuencia máxima  $f_s$ , es decir,

$$f_s \leq 2f_{\max} \quad (3.1)$$

- En términos del intervalo del tiempo de muestreo:

$$T \leq \frac{1}{2f_{\max}} \quad (3.2)$$

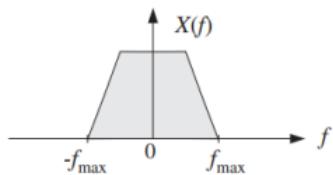


Figura 3.2: Banda de espectro limitado típica [13].

La velocidad de muestreo mínima permitida por el teorema de muestreo, que es (4.1), es llamada *Tasa de muestreo de Nyquist*. Para valores arbitrarios de  $f_s$ , la magnitud de  $\frac{f_s}{2}$  es llamada *Frecuencia de Nyquist*. Esta frecuencia también define las frecuencias de corte de los filtros pasa bajas que se requieren en las operaciones en DSP-.

### 3.1.2. Cuantización.

El muestreo y la cuantización son requisitos indispensables para cualquier operación de procesamiento digital en señales analógicas. Una señal digital es una secuencia de números<sup>1</sup> en donde cada muestra es representada por un número finito de dígitos<sup>2</sup>. La cuantización es el proceso de convertir una señal discreta en el tiempo de amplitud continua, a una señal digital, representando cada valor muestreado como un número finito de dígitos. La Figura 3.3 muestra la representación gráfica de una señal cuantizada.

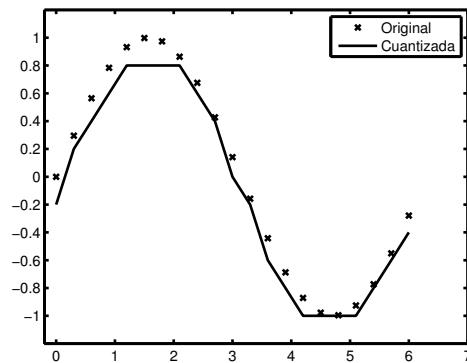


Figura 3.3: Ejemplo de cuantización en una señal senoidal. Los puntos marcados con una «x» representan la señal original, mientras que las coordenadas marcadas con «-» muestran la señal cuantizada .

Este proceso induce un error llamado *error de cuantización*, el cual se debe al cambio en la representación de la señal original, de un valor continuo a un set de valores discretos. En términos matemáticos, la operación de cuantización de las muestras  $\mathbf{x}(\mathbf{n})$  se denota como  $\mathbf{Q}[\mathbf{x}(\mathbf{n})]$ . Tomando  $\mathbf{x}_q(\mathbf{n})$  como la secuencia de muestras cuantizadas, el sistema completo queda como:

$$\mathbf{x}_q(\mathbf{n}) = \mathbf{Q}[\mathbf{x}(\mathbf{n})] \quad (3.3)$$

<sup>1</sup>Esta secuencia es conocida como **Muestra**.

<sup>2</sup>En el diseño de sistemas DSP, se dice que la señal es de **precisión finita** cuando la representación está dada por un número finito de datos.

El error de cuantización es representado por la secuencia  $e_q(n)$ , como la diferencia entre los valores cuantizados y el valor de la muestra actual:

$$e_q x(n) = x_q(n) - x(n) \quad (3.4)$$

### 3.2. Sistemas Discretos Lineales e Invariantes en el Tiempo..

La relación entrada/salida de los sistemas *Lineales e Invariantes en el Tiempo* (LTI, por sus siglas en inglés) está definida por la *convolución* en tiempo discreto de la respuesta del impulso finito aplicado a la entrada del sistema.

Los sistemas LTI pueden ser clasificados en dos tipos, dependiendo de si su respuesta al impulso tiene duración finita o infinita, estos son: *Respuesta al Impulso Finito* (FIR, por sus siglas en inglés) o *Respuesta al Impulso Infinito* (IIR, por sus siglas en inglés).

En términos matemáticos, se dice que un sistema es lineal cuando, por ejemplo, dos señales de entrada  $x_1(t)$  y  $x_2(t)$  tienen salidas  $y_1(t)$  y  $y_2(t)$  respectivamente. Entonces, la salida del sistema al impulso  $\alpha_1 x_1(t) + \alpha_2 x_2(t)$  es  $\alpha_1 y_1(t) + \alpha_2 y_2(t)$ . Es decir, cumplen con las reglas de *homogeneidad* y *superposición*.

También, un sistema es invariante en el tiempo cuando  $y(t)$  es la salida correspondiente a  $x(t)$ , entonces para cada  $\tau$ ,  $y(t - \tau)$  es la salida que corresponde a  $x(t - \tau)$ . Es decir, si agregamos un retraso a la entrada o salida, el resultado debe ser exactamente el mismo.

Además, los *sistemas LTI* deben ser causales, lo que significa que la salida del sistema no puede anticipar la entrada del mismo, tal que, para todo impulso de entrada  $\delta(t)$ , la salida  $h(t) = 0$  mientras  $t < 0$ .

Otra característica que debe cumplir, es la de ser un sistema sin memoria. Se dice que un sistema tiene memoria cuando la señal de salida depende de las entradas pasadas y/o futuras. Por ejemplo, la ecuación del cálculo del voltaje de un resistor representa un sistema sin memoria dado que  $v(t) = R_i(t)$ , mientras que el voltaje en un capacitor representa un sistema con memoria por su ecuación  $v(t) = v(t_0) + \int_{t_0}^t i(\tau) d\tau$ .

En términos de hardware, todos los *sistemas LTI* pueden ser implementados a base de sumadores, multiplicadores y unidades de retraso. Estos sistemas son fácilmente realizables dado que se basan a partir de un número finito de elementos, como los antes mencionados.

Las siguientes subsecciones resumen conceptos importantes que son utilizados en cualquier aplicación y diseño de sistemas DSP.

### 3.2.1. Convolución.

La convolución es la forma matemática de combinar dos señales para formar una tercera. Es la técnica más importante en el Procesamiento Digital de Señales. Usando la estrategia de la descomposición del impulso, los sistemas pueden ser descritos por una señal llamada *respuesta al impulso*. La convolución es importante porque relaciona las tres señales de interés: la señal de entrada, la señal de salida y la señal de respuesta al impulso.

Un punto fundamental a comprender de los sistemas DSP, es que estos trabajan descompiniendo la señal de entrada en simples componentes aditivos, cada uno de estos componentes se pasa a través de un sistema lineal, y los componentes de salida resultantes son sintetizados, o en otras palabras, sumados. Esta descomposición se puede hacer de dos formas distintas: *Descomposición por impulsos* y *Descomposición por el método de Fourier*. Cuando la Descomposición por impulsos es utilizada, el procedimiento se puede describir matemáticamente utilizando la **convolución**.

Esta operación se basa en dos términos importantes en los sistemas DSP. El primero es la **función delta**, simbolizada por  $\delta[n]$ . La función delta es un impulso normalizado, que es, la muestra número cero con un valor asignado de una unidad, mientras que las otras muestras tienen asignadas un valor de cero. Por esta razón, la función delta es llamada **unidad de impulso**.

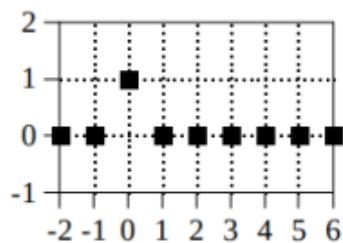


Figura 3.4: Función delta graficada.

Por otra parte, la **respuesta al impulso** es la señal que existe en el sistema cuando una función delta es aplicada a la entrada. Si dos sistemas son diferentes en cualquier manera, ambos tendrán diferentes respuestas al impulso. Comunmente, las respuestas a la entrada y salida son llamadas  $x[n]$  y  $y[n]$ , la respuesta al impulso es usualmente llamada  $h[n]$ .

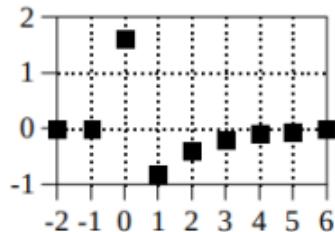


Figura 3.5: Función de respuesta al impulso graficada.

En otras palabras, una señal de entrada  $\mathbf{x}[\mathbf{n}]$ , entra en un sistema linear con una respuesta al impulso  $\mathbf{h}[\mathbf{n}]$ , resultando en una señal de salida  $\mathbf{y}[\mathbf{n}]$ . En forma matemática, la ecuación de convolución de una sola muestra queda como se muestra a continuación:

$$\mathbf{y}[\mathbf{n}] = \mathbf{x}[\mathbf{n}] * \mathbf{y}[\mathbf{n}] \quad (3.5)$$

Tomando la ecuación 4.5 como guía, siendo  $\mathbf{x}[\mathbf{n}]$  una señal de  $N$  puntos, desplazándose desde 0 a  $N-1$ , y  $\mathbf{h}[\mathbf{n}]$  una señal de  $M$  puntos desplazándose de 0 a  $M-1$ [20], la convolución de ambas señales es una señal  $N+M-1$  ejecutándose desde 0 a  $N+M-2$ , esto es:

$$\mathbf{y}[\mathbf{i}] = \sum_{\mathbf{j}=0}^{M-1} \mathbf{h}[\mathbf{j}] \mathbf{x}[\mathbf{i}-\mathbf{j}] \quad (3.6)$$

La ecuación anterior representa la definición formal de la convolución, la cual también es conocida como la *suma de convolución* o *convolución discreta*.

Un ejemplo de la convolución tomando como vectores de entrada  $\vec{v}_{actual} = [v_1, v_2, \dots, v_n]$  y  $\vec{v}_{anterior} = [v_n, v_{n-1}, \dots, v_1]$  se muestra en la **Figura 3.6**.

En el **Apéndice C** se muestra el script interactivo que se puede ejecutar en el ambiente de Matlab, el cual fue tomado de [5]. Este script muestra el resultado de la convolución discreta de dos impulsos  $\mathbf{y}[\mathbf{n}] = \delta[\mathbf{n}] * \delta[\mathbf{n} - 1]$ , tomando como vector a  $\mathbf{n} = [1, 0, 1]$ .

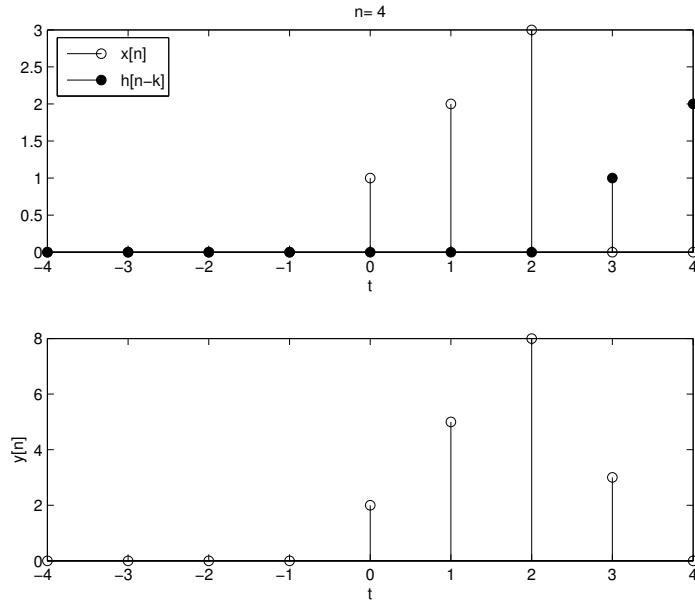


Figura 3.6: Convolución de dos vectores en Matlab.

### 3.2.2. Filtros FIR.

Un filtro digital de *Respuesta Finita al Impulso* (FIR) es un *sistema LTI* si es definido por un conjunto de coeficientes constantes. La salida de un filtro FIR de orden (o longitud)  $L$ , a la respuesta de impulso unitario aplicado a la entrada  $\mathbf{x}[n]$ , está dada por una versión finita de la ecuación de convolución, como se muestra a continuación:

$$\mathbf{y}[n] = \mathbf{f}[n] * \mathbf{x}[n] = \sum_{k=0}^{L-1} \mathbf{f}[k] \mathbf{x}[n-k] \quad (3.7)$$

Para un tren de impulsos en el dominio del tiempo a la entrada  $\mathbf{x}[n]$ , la ecuación queda como:

$$\mathbf{y}[n] = \mathbf{b}_0 \mathbf{x}(n) + \mathbf{b}_1 \mathbf{x}(n-1) + \dots + \mathbf{b}_{M-1} \mathbf{x}(n-M+1) \quad (3.8)$$

$$= \sum_{k=0}^{M-1} \mathbf{b}_k \mathbf{x}(n-k) \quad (3.9)$$

En otras palabras, la respuesta al impulso consiste sólo de respuesta en los coeficientes, procedida y antecedida por ceros (el filtro producirá una respuesta que irá decayendo a cero y se mantendrá en ese estado, de ahí el nombre característico de este filtro). Matemáticamente se puede expresar esta respuesta al impulso con la siguiente ecuación:

$$\mathbf{h}(\mathbf{n}) = \begin{cases} b_n, & 0 \leq n \leq M - 1 \\ 0, & \text{others} \end{cases} \quad (3.10)$$

Gráficamente, esta ecuación se puede representar como se muestra en la **Figura 3.7**.

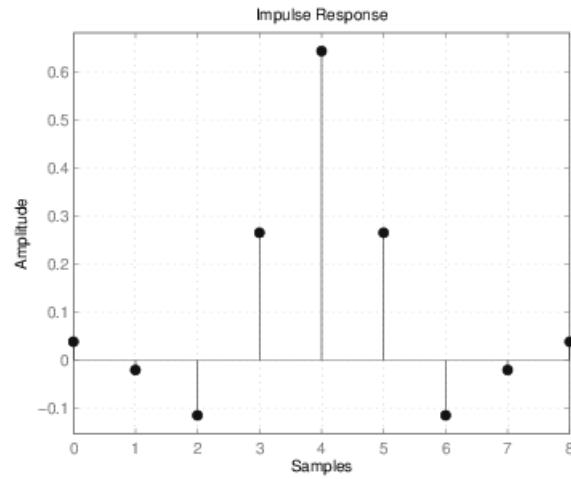


Figura 3.7: Respuesta al impulso de un filtro FIR con  $\mathbf{b}_0 = 0.9$ .

Este tipo de filtros son más populares en cuanto a implementación, dado que cuentan con características muy útiles, entre las cuales destacan:

- **Fase Lineal:** Esta propiedad implica que la fase es una función lineal de la frecuencia. Esto asegura que las señales de todas las frecuencias se retrasan en la misma cantidad de tiempo, eliminando la posibilidad de distorsión de fase.
- **Estabilidad:** Para una entrada finita, la salida siempre es finita, además son no recursivos, es decir, no hay una conexión de retroalimentación envuelta en la estructura del filtro.

Existen muchos métodos de implementación de estos filtros, la estructura más básica es conocida como *forma directa*, la cual consta de utilizar la ecuación en diferencia, no recursiva, mostrada en (4.7)[12], lo que es

equivalente a la sumatoria convolucional. La estructura se muestra en la **Figura 3.8**, el cual representa un filtro con un número de coeficientes  $b_{0..3} + 1$ .

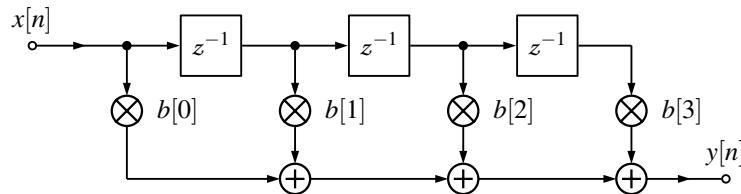


Figura 3.8: Estructura en Forma Directa de un Filtro FIR. Imagen creada usando el paquete TikZ de L<sup>A</sup>T<sub>E</sub>X

En términos generales, existen cuatro formas básicas de implementación de este tipo de filtros, las cuales son:

- Método por ventanas (Rectangular, Barlett, Hanning, Hamming, Blackman y Kaiser).
- Muestreo en frecuencia.
- Aproximación de Chebyshev y algoritmo de intercambio de Remez (conocido como método de Rizado Constante).
- Mínimos Cuadrados.

La implementación más popular utilizada en FPGA es la del Método por Ventanas debido a que muchos de los paquetes de software incluidos en las herramientas de diseño de filtros, utilizan este método como el principal, lo cual recorta el tiempo de desarrollo del mismo.

### 3.2.3. Filtros IIR.

El sistema de Respuesta Infinita al Impulso (*IIR* por sus siglas en inglés) se caracteriza por utilizar las muestras de la señal de salida en instantes anteriores en adición a las muestras presentes más las muestras pasadas de la misma función de salida, es decir, este filtro cuenta con lazos de *retroalimentación* y *anticipación*, por lo que es conocido como un **sistema discreto recursivo**, a diferencia del filtro FIR que se caracteriza por ser **no recursivo**, esto significa que la función de salida del filtro puede continuar indefinidamente aún cuando la entrada tienda a cero y permanezca en cero.

El filtro IIR es un sistema de clase lineal, invariante en el tiempo que se representa en función de una ecuación en diferencia:

$$y[n] = \sum_{k=1}^N a_k y[n-k] + \sum_{k=0}^M b_k x[n-k] \quad (3.11)$$

La primera suma representa la parte “auto-regresiva” o la parte IIR, y la segunda suma representa el “promedio móvil” o la parte FIR[4].

Estos filtros *IIR* tienen una relación con los demás filtros en tiempo continuo. Ambos tienen respuesta al impulso de longitud infinita y son descritos por funciones de transferencia racionales en el dominio de la frecuencia. El método en que estos filtros se diseñan, a partir de estructuras discretas, es usando métodos de aproximación numérica, como los métodos de *Chebyshev*, *Butterworth* o *Bessel*, además de que es posible utilizar un gran número de tablas pre diseñadas, disponibles en los libros afines al tema en cuestión[16]. Dado que este tipo de filtros no serán utilizados en el marco metodológico de este documento, sólo se abarcarán las características fundamentales de ellos.

El formato de implementación del filtro IIR utilizando el **método de Chebyshev** minimiza la diferencia absoluta entre la respuesta en frecuencia ideal y la actual sobre toda la banda de paso mediante la incorporación de rizo uniforme en la banda de paso. La transición desde la banda de paso hacia la banda de corte es más rápida que en el formato *Butterworth*.

El filtro de **Butterworth** representa la mejor aproximación de un filtro ideal pasa bajas en frecuencias analógicas. La respuesta en la banda de paso y la banda de corte es máximamente plana[8].

El filtro de **Bessel** no se puede implementar digitalmente utilizando muchas las herramientas comerciales de modelado matemático, por esta razón, no es muy común en el campo del diseño DSP. Además, estos filtros requieren un orden más alto, por lo que la utilización de elementos discretos para su implementación se dispara, volviendo al filtro de *Bessel* muy complicado en términos de implementación.

La estructura de un filtro IIR se puede representar mediante la ecuación (4.11), en donde se puede observar que es similar a tener un filtro FIR que comprende los coeficientes de **b** y otro filtro FIR inverso con los coeficientes de **a**, como se muestra en la **Figura 3.9**.

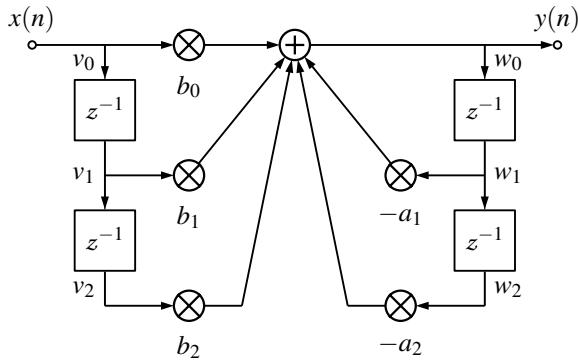


Figura 3.9: Estructura en Forma Directa de un Filtro IIR. Imagen creada usando el paquete TikZ de L<sup>A</sup>T<sub>E</sub>X

## 4. Plataformas FPGA.

### 4.1. Descripción general.

La Matriz de Compuertas Programables en Campo (FPGA) es un circuito integrado reconfigurable que puede ser utilizado para diseñar circuitos digitales. La configuración de la FPGA es normalmente especificada usando lenguajes de descripción de hardware como SystemVerilog o VHDL y después es traducida, mediante herramientas de síntesis, a un formato binario en el cual se encuentra la información de ruteo e interconexiones necesarias para que el dispositivo ejecute las funciones lógicas para el cual fue diseñado. Esta propiedad de ser reconfigurable y poder ejecutar múltiples tareas tan complejas o sencillas, de forma paralela, ofrece una significante ventaja en muchas aplicaciones como por ejemplo en el diseño de circuitos integrados donde a diferencia del prototipado con ASIC, en donde los diseñadores no tienen la flexibilidad de hacer modificaciones al prototipo después de que el chip ha sido manufacturado, en el FPGA es posible y muy común el modificar algunas partes del circuito después de que el proyecto ha sido concluido.

La arquitectura de una FPGA se basa en Bloques de Matrices Configurables (CLBs, por sus siglas en Inglés), las cuales proporcionan la lógica programable y una jerarquía de interconexiones reconfigurables para interconectar los CLBs entre sí. Además de estos componentes básicos, las FPGA actuales contienen bloques de memoria internos, controladores para interfaces externas de alta velocidad como memorias DDR y bloques físicos de PCI Express, así como bloques optimizados para operaciones de DSP, microprocesadores físicos y algunas funciones especiales más, que dependen del enfoque o familia de la FPGA, todo esto en el mismo silicio.

La tendencia reciente en la tecnología FPGA es trabajar con arquitecturas de hardware en alto nivel de abstracción, agregarle bloques DSP, procesadores embebidos y transductores de alta velocidad para formar

un Sistema Programable en Chip completo (SoPC). Además, las FPGA toman ventaja del paralelismo natural del hardware, ya que exceden el poder computacional de los Procesadores Digitales de Señales, rompiendo el paradigma de la ejecución secuencial y lograr un mayor rendimiento.

Una de las aplicaciones principales de las FPGA es poder ejecutar y modificar arquitecturas digitales múltiples veces, hasta que se ha cumplido el objetivo del prototipo que se estableció al principio, sin ser necesario recurrir a los costosos procesos de fabricación de Circuitos Integrados personalizados. Gracias a esto, se pueden implementar diseños de manera incremental e incluso hacer cambios iterativamente en cuestión de horas en lugar de semanas. También, debido a la creciente oferta de herramientas de diseño en alto nivel, se ha decrementado la curva de aprendizaje y con frecuencia, estas herramientas incluyen valiosas Propiedades Intelectuales (IP) para control y procesamiento de señales avanzadas.

Existe una numerosa cantidad de fabricantes pero sólo dos tipos de FPGAs: Reprogramables (basadas en SRAM o Flash) y Programables una sola vez (OTP). Las FPGAs basadas en SRAM necesitan una memoria de configuración y no retienen los datos cuando son desconectadas de la fuente de alimentación. Las que son basadas en Flash, no necesitan una memoria externa para almacenar la configuración y la pueden mantener aún cuando el dispositivo no está energizado. Anteriormente, las FPGAs basadas en Flash tenían la característica de ser OTP, pero hoy en día existen dispositivos basados en esta tecnología que pueden ser reprogramados tales como las MAX 10 de Altera.

En la próxima sección se cubrirá a detalle, la arquitectura de la familia Spartan6 de Xilinx, ya que es esta la que se encuentra en el kit de desarrollo Atlys de Digilent.

## 4.2. Estructura general de las FPGA.

Como se mencionó anteriormente, las FPGA modernas ofrecen una serie de componentes que son de gran utilidad al momento de diseñar sistemas digitales. Estos básicamente son:

- Bloques Lógicos Configurables (CLB) para poder implementar funciones lógicas así como registros.
- Memoria en Chip (On-chip memory) que provee almacenamiento de datos dentro del FPGA, generalmente es reducido dado que el área de construcción de memoria en el silicio, tiende a ocupar gran parte de este.

- Propiedades Intelectuales físicas, tales como controladores Ethernet MAC, Transductores, Multiplicadores optimizados, bloques DSP, Procesadores, Controladores de memoria externa DDR, PCIe endpoint físico, etc.
- Recursos de manejo de reloj que generen las frecuencias necesarias para controlar dispositivos como los antes mencionados y que además, puedan ser distribuidos dentro de la FPGA. Esto es muy importante al momento de diseñar sistemas con un alto índice de transferencia de datos.
- Bloques de entrada y salida que comuniquen a la FPGA al mundo exterior.
- Recursos de ruteo para proveer la interconectividad de los Bloques Lógicos Configurables internos y las Propiedades Intelectuales.

La **Figura 4.1** muestra la arquitectura típica de una FPGA con los bloques de construcción básicos. Es importante mencionar que algunos elementos como las Block RAM, bloques DSP, controladores de memoria, etcétera, que se muestran en la imagen, son construidos sobre el mismo silicio, sin quitarle espacio a los elementos lógicos. También cabe mencionar que, las Tablas de Búsqueda, mejor conocidas como Look Up Tables (LUT) que hay dentro de los bloques lógicos son usadas para crear funciones de lógica combinacional, pero también pueden ser configuradas como memorias RAM o registros de desplazamiento. Esta es una forma muy eficiente de inferir dichos registros sin tener que usar los elementos de almacenamiento.

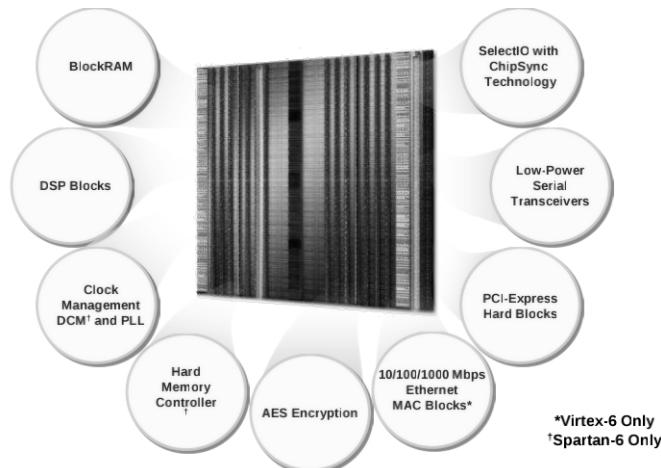


Figura 4.1: Arquitectura general de las FPGA [14].

### 4.3. Arquitectura de la familia de FPGAs Xilinx Spartan-6.

La familia Spartan-6 está fuertemente enfocada a proveer capacidades de integración de sistemas con el menor costo posible para aplicaciones de alto volumen, es decir, es una línea de dispositivos que tienen bloques de comunicación de alta velocidad como PCI Express, controladores de memoria externa DDR3 y Ethernet, entre otros. Además de una vasta densidad de elementos lógicos y registros disponibles que van desde 3,840 hasta 147,443 celdas lógicas, dependiendo del dispositivo seleccionado por el diseñador. Consumen la mitad de la potencia comparado con la familia anterior de FPGAs Spartan 3, gracias a que están construidas con una avanzada tecnología de 45nm. Esta línea de FPGAs son muy populares ya que son el balance óptimo entre costo, potencia y rendimiento[23].

La innovación más notable en estas FPGA es la re estructuración de la arquitectura interna para implementar LUTs de 6 entradas y doble registro de salida en cada LUT, esto significa que una sola LUT puede implementar funciones lógicas de  $2^6 = 64$ bits, como por ejemplo, una RAM de 64 bits o un registro de desplazamiento de 32 bits. Anteriormente, la arquitectura se basaba en LUTs de 4 entradas, como se muestra a continuación.

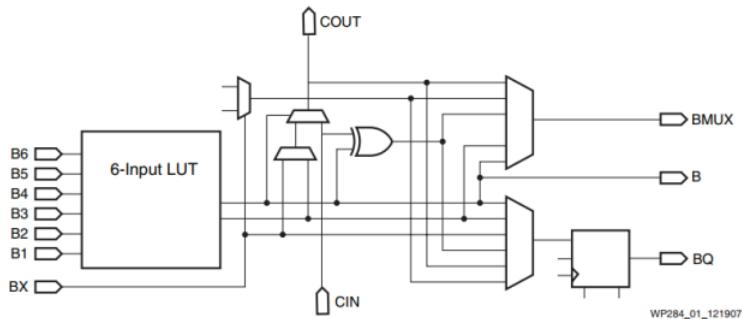


Figura 4.2: LUT de seis entradas [2].

Además, la familia Spartan-6 incluye bloques de memoria RAM (BRAM) de 18Kb, una optimización de dispositivos DSP48A1 los cuales sirven para ejecutar cálculos complejos de manera paralela, controladores físicos de memoria SDRAM, bloques de manejo de reloj internos mejorados para poder generar las frecuencias necesarias para controladores de alta velocidad, así como opciones de configuración y seguridad de IP más avanzados.

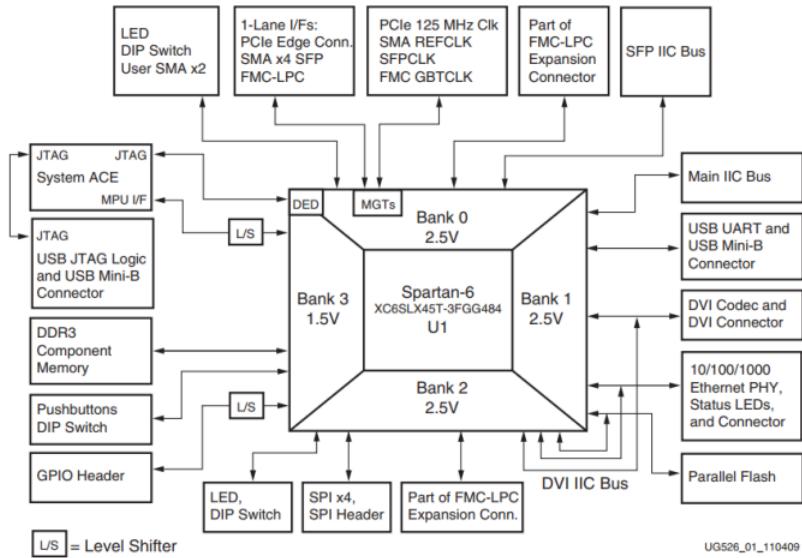


Figura 4.3: Diagrama a bloques de una FPGA Spartan-6 [2].

Debido a la construcción en 45nm, se han podido incorporar una mayor cantidad de CLBs en esta familia de FPGAs. Los CLBs son los recursos lógicos principales necesarios para implementar circuitos secuenciales y combinatorios. Cada elemento CLB es conectado a una matriz de switches programables para acceder a otra matriz de ruteo como se muestra en la Figura 4. Cada elemento CLB contiene un par de SLICEs. Estos dos SLICEs no tienen una conexión directa entre si. Cada SLICE tiene un bloque de acarreo en cadena (carry chain).

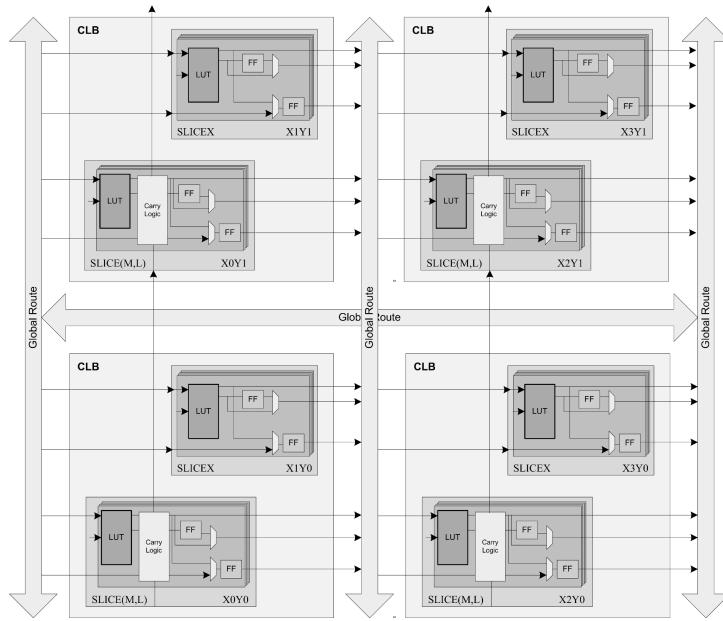


Figura 4.4: Bloque Lógico Configurable.

Cada SLICE contiene cuatro LUTs, cuatro elementos de almacenamiento (flip flop), un amplio número de multiplexores y un bloque de acarreo lógico. Esos elementos son usados por todos los SLICES para proveer las funciones lógicas, aritméticas y algunos tipos de memoria ROM. Adicionalmente, algunos SLICES pueden implementar dos funciones adicionales: almacenar datos al adoptar la función de RAM distribuida y desplazar datos adoptando la función de registro de desplazamiento de 32 bits. Son llamados SLICEM (por Memoria), los comunes son llamados SLICEL (Por Lógico).

La Figura describe con detalle la arquitectura de cada SLICE en un CLB. Los multiplexores antes mencionados sirven para proveer la conectividad entre los recursos lógicos que rodean a los CLBs, mientras que la red de elementos de acarreo en cadena dentro de los CLBs, hacen la función de ruteo para generar sumadores lógicos más eficientes.

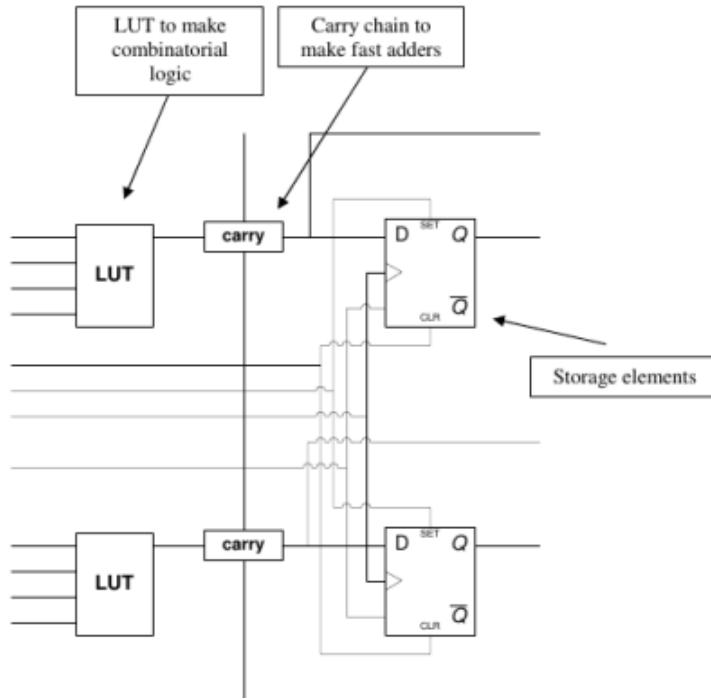


Figura 4.5: Estructura detallada de un SLICE [2].

Los dispositivos Spartan-6 cuentan con un gran número de memorias BRAM de 18Kb, las cuales están construidas por dos memorias controladas independientemente de 9Kb cada una. Estas memorias son colocadas en columnas, el número total de ellas depende del tamaño del dispositivo Spartan-6. Las memorias BRAM se pueden utilizar en cascada para habilitar implementaciones de mayor tamaño de Kilo bits, con un pequeño impacto en el timing. La Figura 6 muestra una BRAM en cascada con dos distintos puertos de lectura y escritura.

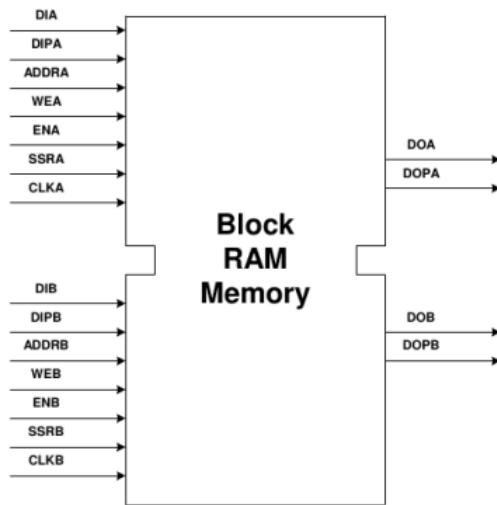


Figura 4.6: Block RAM de doble puerto [2].

#### 4.4. Kit de desarrollo Atlys de Digilent.

El sistema de desarrollo Atlys fabricado por Digilent bajo el programa Xilinx University Program es la plataforma seleccionada para cumplir con los objetivos de esta investigación. El kit Atlys es una plataforma avanzada para el desarrollo de sistemas digitales. Está basada en un dispositivo FPGA Xilinx Spartan-6 LX45, la familia LX está optimizada para implementaciones lógicas de alto rendimiento[6].

Los periféricos que se incluyen son de gama alta, tales como controladores Gigabit Ethernet, entrada y salida de video HDMI, memoria RAM DDR2 de 128 Megabytes de almacenamiento en un bus de 16 bits, además de puertos de USB host y entrada y salida de audio a través de un códec AC97. Este kit es compatible con toda la suite de diseño de Xilinx ISE, por lo que se pueden implementar una gran variedad de diseños para diferentes áreas de investigación con un costo relativamente bajo.

Las principales características técnicas de esta plataforma se muestran a continuación:

- FPGA Spartan-6 LX45-3 con 6,822 slices, cada una contiene 4 LUTS de 6 entradas y 8 flip-flops, con una frecuencia máxima de ejecución a 500MHz.<sup>3</sup>
- 58 DSP slices (elementos lógicos optimizados para operaciones de DSP).
- Oscilador CMOS a 100 MHz como reloj base para PLL/DCM.

<sup>3</sup>Para una mayor comprensión de estas características técnicas, referirse a la sección “4.3 Arquitectura de la familia de FPGAs Xilinx Spartan-6” de este documento.

- Códec de audio AC97 con líneas de entrada/salida de audio y ariculares.
- Dos entradas y dos salidas de video HDMI.

Entre algunas otras. En la **Figura 4.7** se puede observar una imagen de esta plataforma.

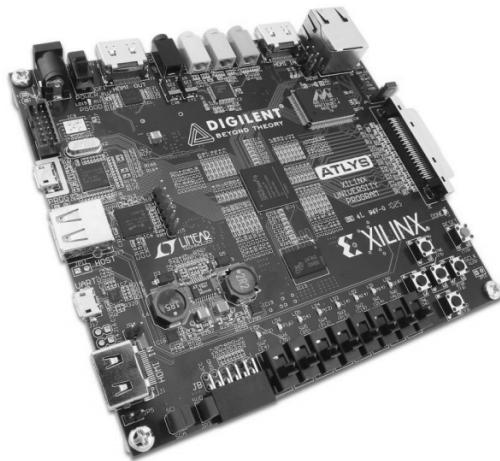


Figura 4.7: Kit de desarrollo Atlys Spartan-6 de Digilent, obtenida del sitio web del fabricante[6].

## 5. Plataformas de Software para el diseño de sistemas DSP.

### 5.1. Introducción a MATLAB y Simulink.

#### 5.1.1. MATLAB.

El nombre de esta herramienta proviene de recortar y unir las palabras **MAT**rix **LAB**oratory (Laboratorio de Matrices), lo cual nos da una referencia de la metodología que utiliza. MATLAB es un sistema interactivo cuyo elemento básico son matrices las cuales, no requieren ser dimensionadas. Está enfocado a resolver problemas técnicos computacionales a través de un ambiente completo donde los problemas y soluciones son expresados en una notación matemática familiar[24]. Además, la integración de herramientas para la adquisición de datos externos tales como sonidos e imágenes, así como el análisis a través de aplicaciones de visualización como gráficas, hacen de este lenguaje uno de los mas atractivos en el área de las ciencias aplicadas.

Lo que hace de MATLAB una herramienta muy popular en el ámbito académico e industrial, es la rapidez con la que se pueden implementar y resolver algoritmos de mediana a gran complejidad. Esto se debe en gran medida a la facilidad que brinda el intérprete interactivo para crear scripts conocidos como Archivos-M (M-files, por sus siglas en Inglés) que ayudan a describir sistemas con eficacia, así como el gran número de funciones prediseñadas incluidas en los complementos conocidos como «cajas de herramienta» o Toolboxes, los cuales contienen colecciones de funciones para un gran rango de disciplinas tales como procesamiento de señales, sistemas de control, redes neuronales, lógica difusa, por mencionar algunas. Todo esto ayuda a que el usuario implemente sus modelos matemáticos en cuestión de horas y no días, comparado con lenguajes compilados no interactivos como C o Fortran.

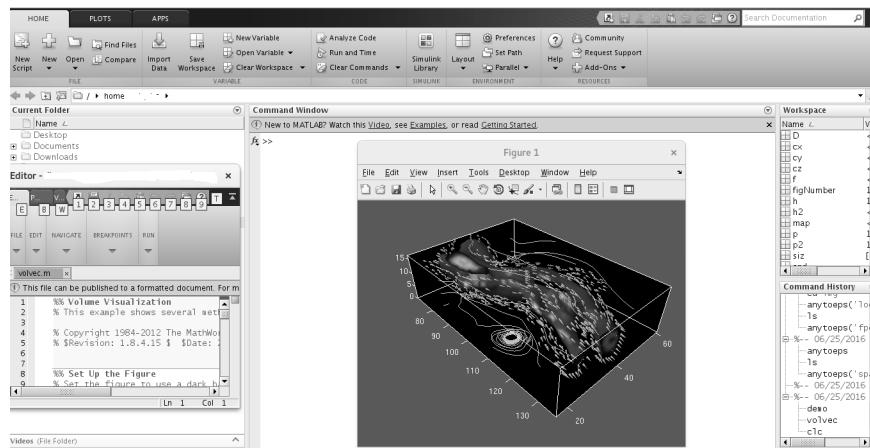


Figura 5.1: Interfaz Gráfica del Usuario de MATLAB.

MATLAB es considerado un lenguaje de programación de alto nivel, debido a la gran abstracción de datos con los que trabaja. Algunas propiedades importantes de esta herramienta son:

- El tipo de datos por defecto es una matriz de doble precisión, lo que significa que se pueden representar números desde 0 a  $1.7977e + 308$ .
- Es un lenguaje orientado a objetos, lo cual resulta en una mejora en el manejo de la complejidad de aplicaciones y estructuras de datos.
- Los algoritmos diseñados en este paquete de software, pueden ser convertidos a código en lenguaje C, HDL y/o PLC, para ser ejecutados en dispositivos embebidos.

- Matlab en conjunto con simulink, tienen soporte para la herramienta de diseño DSP de Xilinx, lo cual convierte a este ambiente en uno de los más atractivos en este ámbito.

### 5.1.2. Simulink.

MATLAB tiene un toolbox adicional llamado Simulink que proporciona el modelado, la simulación y el análisis de sistemas dinámicos, dentro de un entorno gráfico. Este software permite el diseño modular y jerárquico, ayudando al usuario en el manejo y creación de sistemas complejos que son conceptualmente simplificados.

Debido a este amplio uso, la capacidad de diseñar y verificar las implementaciones de hardware desde el mismo ambiente de Simulink, se traduce en una mayor ventaja para el prototipado rápido de nuevas teorías y diseños. Añadiendo las múltiples capas de software y librerías que habilitan la comunicación del ambiente gráfico de Simulink con el mundo exterior, proceso conocido como simulación de *hardware-in-the-loop* (co simulación con módulos de adquisición de datos en hardware, que estimulan entradas y/o envían resultados al modelo de Simulink), proporciona un enorme beneficio adicional en el diseño y verificación del algoritmo sin arriesgar pérdidas de hardware o errores de implementación[7].

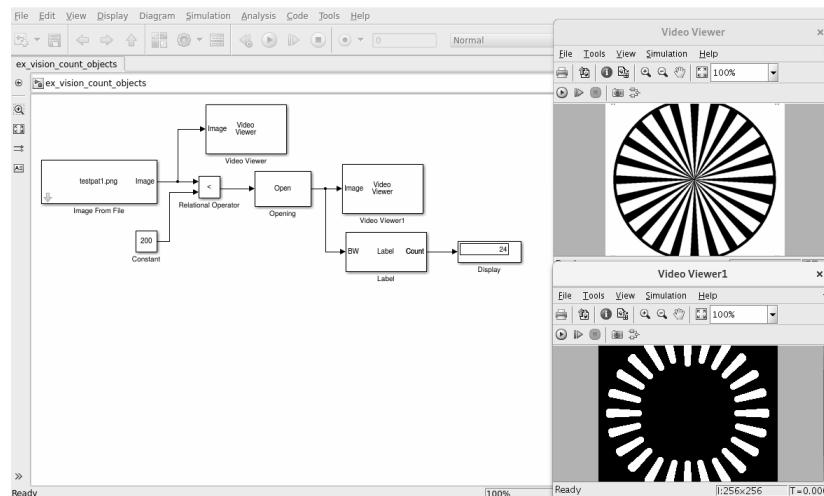


Figura 5.2: Entorno de modelado gráfico Simulink de MATLAB. En este ejemplo se puede observar, como se lleva a cabo la ejecución de un algoritmo de detección de elementos en una imagen, utilizando solamente 8 bloques del *Image Processing Toolbox* de Simulink.

Sin embargo, el comportamiento de la simulación matemática de Simulink en comparación con la implementación en el hardware, no es exacto. Simulink permite que cálculos de punto flotante complejos sean

completados en un solo paso “virtual”, mediante el proceso de hacer más lenta la simulación para permitir cálculos precisos,

En comparación, la implementación en hardware con FPGA requiere un paso de tiempo fijo pre-definido que funciona en tiempo real, mientras que las tasas de muestreo de información se pueden ajustar en diferentes puntos dentro del hardware para permitir la ejecución asíncrona. Cada uno de estos procesos se ejecutan de manera coherente con un paso de tiempo consistente y una longitud de palabra fija. La conversión entre estas dos formas de ejecución, es decir, la simulación del algoritmo en Simulink y la ejecución del mismo en elementos de hardware, debe hacerse de una manera sistemática en donde se tenga en cuenta estas diferencias de derivación[11].

## 5.2. Xilinx System Generator for DSP.

### 5.2.1. Descripción general.

El toolbox incluido en la suite de Xilinx ISE, llamado *System Generator for DSP* o en su contracción, *Sysgen*, está pensado para resolver el problema de la interfáz entre el mundo de la simulación matemática de Simulink, y el mundo tangible del hardware en FPGA.

El desarrollo de algoritmos DSP implica transformar las muestras de entrada provenientes de la etapa de conversión A/D, utilizando numerosas operaciones matemáticas complejas sobre dichas señales como la convolución o la aplicación de alguna transformación como la **Transformada Rápida de Fourier (FFT)** y además, es necesario tener el equipo de laboratorio requerido para para poder visualizar los espectros de las señales a través de todo el diseño[22]. Todo este proceso se vuelve una labor muy intensa cuando se utilizan técnicas tradicionales de RTL en un FPGA.

*System Generator* es una herramienta de modelado a nivel sistema que facilita el diseño de hardware en FPGA<sup>4</sup>. Esta herramienta extiende la funcionalidad de Simulink de muchas maneras, con el objetivo de proporcionar un entorno de modelado bastante adecuado para el desarrollo de hardware. Este toolbox proporciona abstracciones de alto nivel que se compilan automáticamente en un FPGA, facilitando la implementación de algoritmos complejos en donde intervengan, por ejemplo, hasta varios bloques de hardware

<sup>4</sup>Se recomienda leer la documentación de Xilinx: “*System Generator for DSP, User Guide (UG640)*” para tener una mayor referencia a todos los conceptos técnicos que intervienen en esta metodología de diseño y que no son tocados por este documento de investigación.

complejos en diferentes dominios de reloj. También proporciona acceso a recursos o macros primitivas subyacentes del FPGA a través de abstracciones de bajo nivel, lo que permite la construcción de diseños altamente eficientes[25].

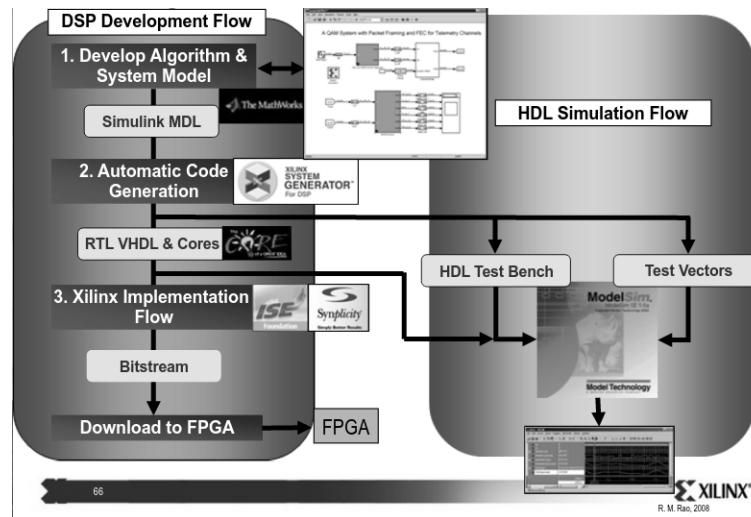


Figura 5.3: Flujo de diseño con System Generator de Xilinx, imagen obtenida del documento “*Wireless communications: from systems to silicon*”, del autor Raghu Rao. Wireless Systems Group, Xilinx Inc, 2008.

System Generator añade librerías o *blocksets* a la interfaz de Simulink, estos se pueden conectar a los demás toolboxes que vienen ya incluidos en la suite de Matlab para generar modelos funcionales de sistemas dinámicos. Estos bloques proveen abstracciones de funciones matemáticas, lógicas, utilización de memorias y funciones DSP que pueden ser fácilmente utilizados en el diseño de sistemas complejos y sofisticados.

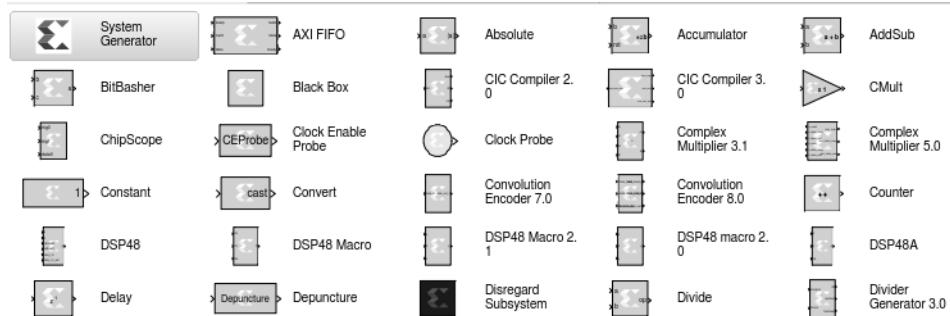


Figura 5.4: Ejemplo de blocksets incluidos en el toolbox de Sysgen, en su versión ISE 14.7.

Una de las características más importantes de este toolbox es que, permite implementar algoritmos ha-

ciendo uso de varios ingredientes. Por ejemplo, se pueden integrar bloques de código en *Verilog* o *VHDL*, así como funciones pre diseñadas de matlab (scripts .m) dentro de las conexiones de Simulink, sin afectar la ejecución de simulación y más importante, garantizando la síntesis de todos estos elementos en el FPGA. Además, este *blockset* trabaja con **precisión de bits** (*bit-accurate*) y **precisión de ciclo** (*cycle-accurate*). La **precisión de bits** produce valores en Simulink que coinciden con los valores producidos en el hardware; la **precisión de ciclo** produce los valores correspondientes en el tiempo correspondiente.

### 5.2.2. Co-Diseño de Hardware/Software en el ambiente de Xilinx System Generator for DSP.

Es posible que el flujo de diseño con esta herramienta de alto nivel no sea tan claro a primera instancia, más para aquellos que tienen experiencia trabajando con los paquetes de software de Matlab y Xilinx por separado. Existen varias formas en las que Sysgen puede ser sintetizado en el kit Atlys, algunos flujos representan ciertas ventajas sobre los demás, o simplemente son mejores formas de lograr el objetivo deseado de una manera más rápida. Los flujos de implementación soportados por este toolbox son:

- **Exploración de Algoritmos mediante simulación *Hardware-in-the-Loop*:** Este flujo está pensado para llevar a cabo una simulación en tiempo real de un prototipado o incluso de un análisis realista del desempeño de un sistema bajo estudio. El bloque de Simulink es sintetizado con una capa extra de hardware que le brinda la capacidad al FPGA de enviar y recibir estímulos directamente al ambiente de simulación, mostrando así una respuesta tangible y cercana a la realidad que el hardware podría brindar. En este modo, la información se puede enviar a través del puerto JTAG o ethernet, dependiendo de la velocidad de transmisión requerida.

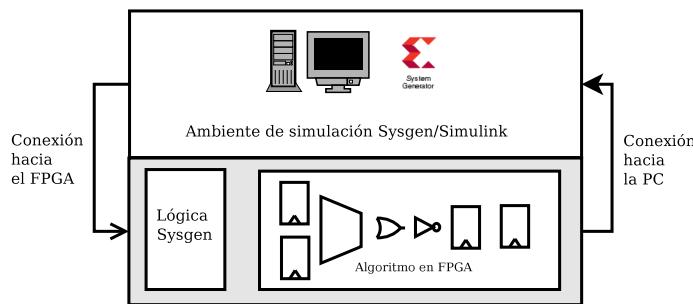


Figura 5.5: Diagrama a bloques del flujo de implementación mediante *Hardware-in-the-loop*.

- **Diseño de una IP como parte de un prototipo más grande:** Se puede empaquetar el sistema previamente modelado y simulado en Sysgen/Simulink para después ser instanciado como parte de un

diseño más grande. *Sysgen* tiene bastante limitaciones a la hora de trabajar con interfaces que necesitan estrictos requerimientos de *timing*. Por ejemplo, para trabajar con interfaces ADC/DAC que generan información a grandes frecuencias de reloj, es mejor utilizar este flujo y combinar el modelo de *Sysgen* con las demás partes necesarias del prototipo, en un solo proyecto. *Sysgen* puede crear un empaquetado en Verilog o VHDL para sintetizar y/o simular un modelo de forma independiente.

- **Implementación de un diseño completo:** La suite de Xilinx contiene herramientas para distintos enfoques, por ejemplo, síntesis lógica con lenguajes de descripción de hardware, sistemas embebidos basados en la familia de procesadores *MicroBlaze/PicoBlaze*, catálogos de IPs para una gran variedad de sistemas ingenieriles y diseño de sistemas en lenguajes de alto nivel. *Sysgen* puede trabajar con todos y cada uno de esos enfoques, para lograr un diseño mucho más estable en un menor tiempo.

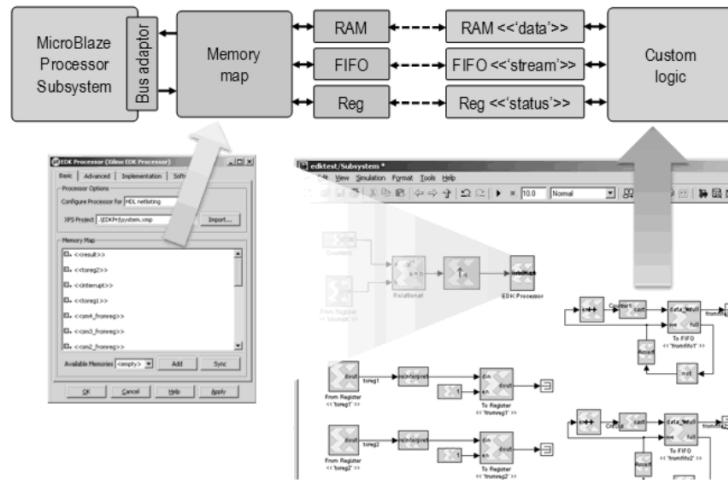


Figura 5.6: Ejemplo de implementación de un diseño completo utilizando el toolbox de *Sysgen* en *Simulink*. Imagen tomada de la guía de usuario “*System Generator for DSP User Guide, UG640*”. Se puede observar como el procesador *Microblaze*, puede transmitir y recibir impulsos del modelo en *Sysgen*.

### 5.3. Introducción a Xilinx ISE y PlanAhead.

#### 5.3.1. Reseña de Xilinx ISE.

El sistema de Xilinx ISE es un entorno de diseño integrado que consiste en un conjunto de programas para crear (o capturar), simular e implementar diseños digitales en un dispositivo FPGA o CPLD de Xilinx únicamente. Todas las herramientas contenidas en este entorno utilizan una interfáz gráfica de usuario que permiten que todos los programas se ejecuten desde la barra de herramientas[15].

Este entorno gráfico mostrado en la Figura 5.7 es conocido como *Project Navigator*; desde esta interfáz se controlan todos los aspectos del diseño, desde la creación de un proyecto y la selección de la familia de FPGA que se va a utilizar, hasta la edición de archivos de descripción de hardware, diseño a base de captura de esquemático y verificación del diseño, sin salir de la interfáz.

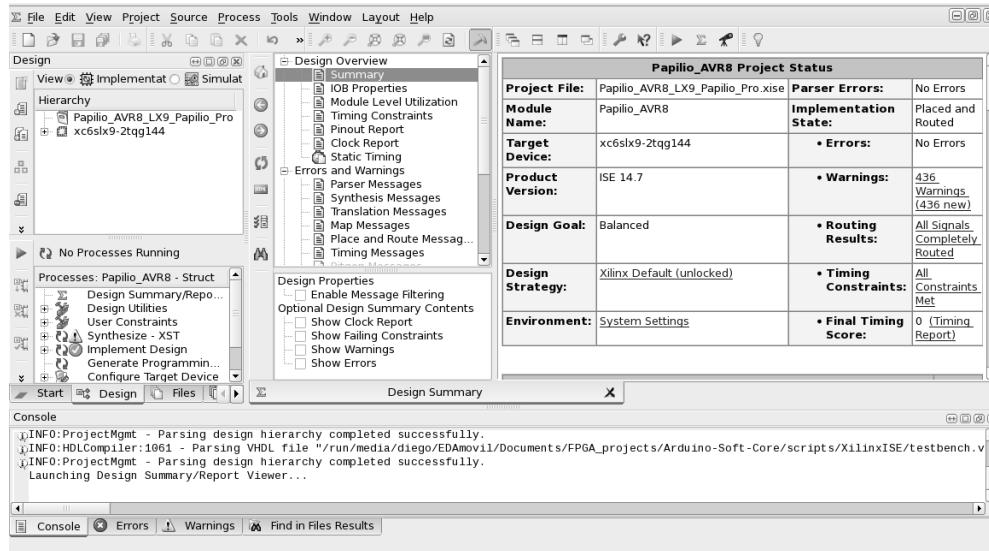


Figura 5.7: Interfáz principal del *ISE Project Navigator*.

Xilinx ISE guía al diseñador en el flujo de construcción en FPGA utilizando el flujo conocido como “*push button*”, esto significa que todo el proceso está definido, automatizado y sólo basta presionar un botón para que la herramienta haga el trabajo de llevar la especificación del diseño de RTL al *bitstream*, mediante la ejecución de algunos procesos en segundo plano, los cuales se enlistan en la **Figura 5.8** y se describen a continuación.

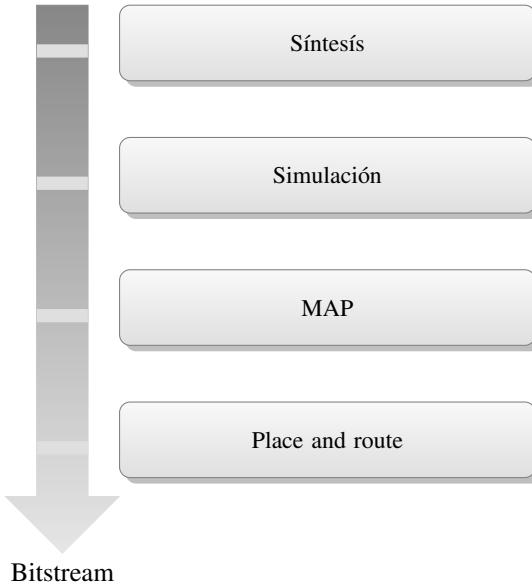


Figura 5.8: Diagrama a flujo del proceso de compilación en FPGA, elaborado usando el paquete *TikZ* de *L<sup>A</sup>T<sub>E</sub>X*.

- **Síntesis:** Este es el proceso de convertir un diseño descrito en Lenguaje de Descripción de Hardware a un netlist con un formato propietario conocido como *Archivo Netlist con Información de restricciones físicas (NGC)*, por sus siglas en Inglés), ejecutando la herramienta llamada *Tecnología de Síntesis de Xilinx (XST)*, por sus siglas en Inglés).
- **Simulación:** Xilinx ISE cuenta con un simulador integrado llamado **ISE Simulator** o **ISim**, el cual soporta Verilog 2001 y VHDL 93, además del uso de ambos lenguajes para diseño y simulación. También ofrece capacidades de visualización y rastreo/análisis de formas de onda tanto analógicas<sup>5</sup> como digitales y depuración de fuentes HDL, entre muchas otras características<sup>6</sup>.
- **Map:** En este paso se translada la salida generada en el proceso de síntesis, a primitivos de la FPGA especificada en el proyecto. Al finalizar, el ejecutable igual que el proceso (**Map**) genera un netlist más, pero en un formato conocido como *Descripción Nativa del Circuito (NCD)* por sus siglas en Inglés). Es importante recalcar que en este proceso es donde se llevan a cabo las optimizaciones de *timing closure*<sup>7</sup> y área, las cuales están relacionadas con el desempeño final del diseño[21].

<sup>5</sup>La visualización de formas de onda analógicas requiere de una licencia adicional.

<sup>6</sup>Para obtener una lista más completa de las características de **ISim**, puede visitar el sitio web <https://www.xilinx.com/products/design-tools/isim.html>

<sup>7</sup>*Timing closure* es el proceso mediante el cual se modifica un diseño FPGA para cumplir con los requisitos de tiempo. Por ejemplo, la frecuencia máxima a la que el prototipo puede trabajar, o la generación de frecuencias correctas para protocolos de baja velocidad como RS232 o SPI.

- **Place and Route (PAR):** Este es el proceso de interconectar los elementos primitivos, a partir de la salida dada por el proceso de Map, para así poder finalizar la implementación. Este es el paso más importante y el que consume un mayor tiempo de ejecución, debido a que la herramienta de Xilinx encargada de hacer la *colocación* de los primitivos en los CLBs ejecutará diferentes algoritmos para obtener el resultado más óptimo, de otra forma el *enrutamiento* será casi imposible de lograr. En este paso, se obtiene el porcentaje exacto de recursos utilizados en la FPGA y la información para generar el *bitstream*.

*Project Navigator* se encarga de ejecutar el flujo de implementación descrito, así como generar reportes para el análisis de resultados y manejar la configuración del kit Atlys, para descargar el *bitstream* generado.

### 5.3.2. PlanAhead.

Hablando en términos más profesionales, el flujo de implementación en FPGA descrito en la subsección anterior se divide en dos procesos globales: *Front-End*, que básicamente abarca la síntesis y simulación, y *Back-end* que abarca Map y PAR.

Generalmente, *PlanAhead*<sup>8</sup> es utilizado por usuarios con más experiencia en el campo de diseño con herramientas de Xilinx, puesto que este ambiente ofrece mejoras para la optimización en el *Back-end* que en el *Project Navigator* no se exponen con la misma facilidad.

Sin embargo, *PlanAhead* provee plugins bastante útiles para el *Front-End*, por ejemplo, usa un potente motor para el análisis de los archivos HDL en el diseño, también puede trabajar con netlist previamente sintetizados y resultados de implementación, además de poder crear diferentes versiones de implementación, con lo que se pueden explorar multiples optimizaciones para un mismo proyecto.

En este documento se utilizará *PlanAhead* sobre *Project Navigator* por las ventajas ya mencionadas, además de que, Xilinx descontinuó la suite ISE para evolucionar a Vivado desde el año 2013. Vivado es una herramienta más poderosa pero no compatible con familias inferiores a la serie 7.

*PlanAhead* es muy similar a la interfáz de Vivado, por lo que los diseños podrán migrarse fácilmente sin necesidad de tener que familiarizarse de nuevo con otra herramienta.

---

<sup>8</sup>Se recomienda leer la guía UG673: “Quick Front-to-Back Overview Tutorial” para mayor detalle en el uso de PlanAhead.

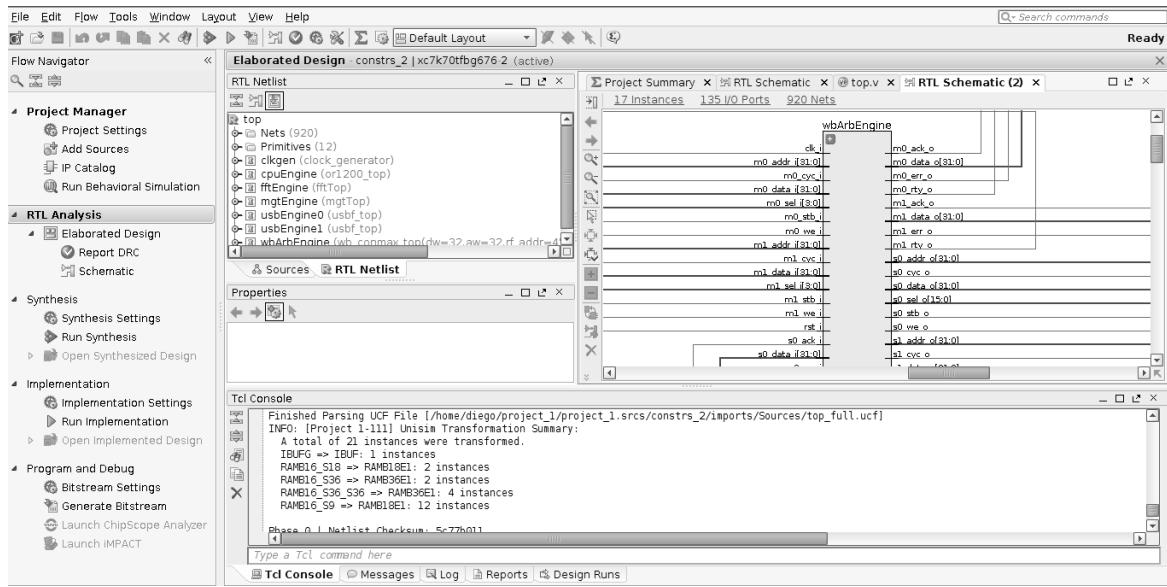


Figura 5.9: Interfáz gráfica de PlanAhead, imagen obtenida del paquete Xilinx ISE.

## **Parte III**

### **Marco metodológico.**

## 6. Filtrado básico de señales - Introducción práctica a Sysgen.

### 6.1. Diseño de un filtro FIR pasabajas para la eliminación de ruido en una señal senoidal.

Una de las aplicaciones más comunes en el mundo del procesamiento digital de señales, es la eliminación de componentes no deseadas que, por algún fenómeno físico, distorsionan la información de la señal en la cual estamos interesados.

A manera de introducción, se utilizará el ambiente de Matlab/Simulink con el toolbox de Sysgen para diseñar un sistema que elimine una componente senoidal de alta frecuencia mezclada a una componente senoidal base, mediante el uso de un filtro FIR pasabajas ejecutándose en el kit Atlys.

### 6.2. Especificaciones del modelo del filtro FIR en Matlab.

Existe una señal de entrada a nuestro sistema  $F_{DSP}$  cuya entrada está representada por la suma de dos señales senoidales de  $f_{senBase} = 1\text{KHz}$  y  $5\text{KHz} \leq f_{senRuido} \leq 15\text{KHz}$  con una amplitud de 1 y 0.5 respectivamente, esta suma de señales representará el fenómeno no deseado.

El sistema  $F_{DSP}$  debe ser capáz de eliminar señales por encima de  $f_{corte} = 7\text{KHz}$ , por lo que el tipo de filtro a implementar debe ser *paso bajo*. Además, la frecuencia de muestreo del sistema debe ser de  $F_s = 48\text{KHz}$ .

1. En primer lugar, es necesario desarrollar la función de las señales  $f_{senBase}$  y  $f_{senRuido}$  para utilizarlas para ejercitarse el filtro FIR, utilizando el script de Matlab que se muestra en el Algoritmo 1, y el resultado de la función se muestra en la Figura 6.1.

**Algoritmo 1** Generación de la señal de excitación al sistema  $F_{DSP}$ .

```

1 % Señal de prueba para ejercitarse el filtro fir.
2 % Parámetros: fsenBase=1KHz, fsenRuido=1KHz:15KHz;
3 % Abase=1, Aruido=0.5;
4 % Fs=48KHz, nmuestras=300;
5 close all; clear all; clc
6
7 Fs      = 48000;
8 nmuestras = 300;
9 Abase   = 1;
10 Aruido  = 0.5;
11 fsenBase = 1000;
12 fsenRuido = 15000;
13
14 F = [fsenBase fsenRuido];
15 A = [Abase Aruido];
16
17 % Generación del vector de tiempo para el muestreo
18 t = (0:nmuestras-1)/Fs;
19
20 % Generación de la señal contaminada
21 Senmix = A * sin(2*pi*F'*t);
22
23 % Conversión del dominio del tiempo a dominio de frecuencia
24 % de la señal Senmix, para efectos de ilustración.
25 Freq    = fft(Senmix);
26 S2      = abs(Freq/nmuestras);
27 S1      = S2(1:NMUESTRAS/2+1);
28 S1(2:end-1) = 2*S1(2:end-1);
29 f       = Fs*(0:(NMUESTRAS/2))/NMUESTRAS;
30
31 % Gráfico de la componente generada en los dominios de la
32 % frecuencia y tiempo
33 figure
34 grid on
35 subplot(2,2,1)
36 plot(Senmix)
37 title('Suma de la señal creada por las dos componentes senoidales')
38 xlabel('Muestras')
39 ylabel('Amplitud')
40 subplot(2,2,2)
41 plot(f,S1)
42 xlabel('f (Hz)')
43 ylabel('|S1(f)|')
44 title('Espectro en amplitud de Senmix')

```

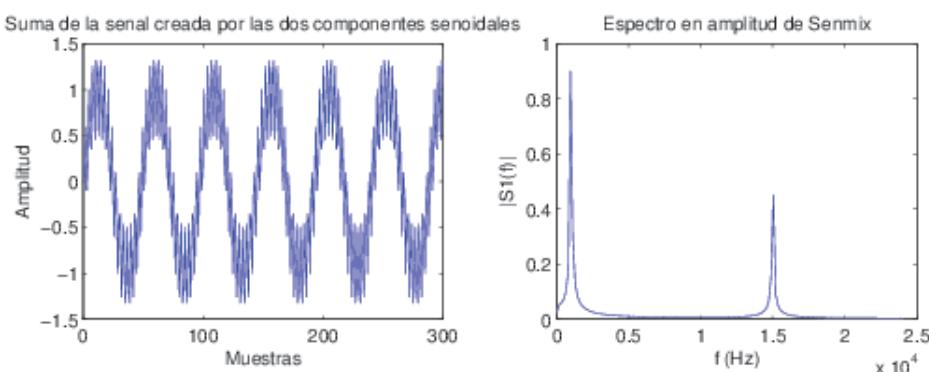


Figura 6.1: Gráfica derivada de la función diseñada en Matlab para este ejercicio.

2. Para el diseño del filtro FIR, se usará el método por ventana de Hamming por la facilidad de implementación discutida en la **sección 3.2.2** de este documento. La ecuación del filtro por ventana de

Hamming queda como sigue:

$$W = 0.54 - 0.46 * \cos(2\pi t/T) \quad (6.1)$$

$$y = x(t) * W(t) \quad (6.2)$$

3. Para efecto de simplificar la implementación y demostrar la versatilidad de los lenguajes de alto nivel, se utilizará la función **fir1(n, Wn)** incluida en el *Signal Processing Toolbox* de Matlab, donde **n** es el orden del filtro y **Wn** es un elemento entre 0 y 1 correspondiente a la frecuencia de Nyquist. Para calcular el valor de **Wn**, necesitamos normalizar la frecuencia de corte para cumplir con el teorema de muestreo discutido en la sección **3.1.1** mediante el uso de las siguientes fórmulas:

$$F_{Nyquist} = \frac{F_s}{2} = \frac{48 * 10^3}{2} = 24 * 10^3 \quad (6.3)$$

$$Wn = F_{corteNormalizada} = \frac{f_s}{F_{Nyquist}} = \frac{7 * 10^3}{24 * 10^3} = 0.2917 \quad (6.4)$$

4. Por lo tanto, el diseño final del filtro queda como se muestra en el Algoritmo 2 y la gráfica de la respuesta en magnitud se muestra en la Figura 6.2

---

#### Algoritmo 2 Implementación del filtro FIR.

---

```

1 close all; clear all; clc
2
3 % Parámetros para el cálculo de implementación
4 Fs = 48000; % Fs
5 Fc = 7000; % Fc
6
7 % Normalización de la frecuencia de corte
8 FNyquist = Fs/2;
9 Wn = Fc/FNyquist;
10
11 % Elaboración del filtro FIR paso bajos usando los parámetros
12 % ya calculados para la normalización de la frecuencia. El orden de este
13 % filtro será de 32.
14 orden = 31; % orden-1, como especifica la función fir1
15 coeficientes = fir1(orden, Wn);
16
17 % Análisis del filtro
18 fvtool(coeficientes, 'Fs', Fs);

```

---

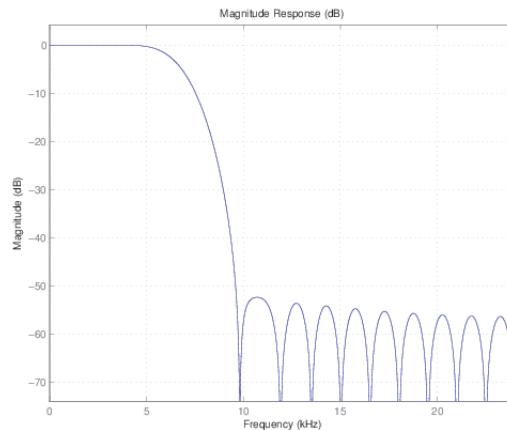


Figura 6.2: Respuesta en magnitud del filtro FIR diseñado, donde se puede observar la atenuación de frecuencias mayores a  $f_c = 7\text{KHz}$ .

5. Por último, para confirmar que el diseño se ha implementado de una manera correcta, aplicamos la señal generada en el **paso 1** al filtro que acabamos de desarrollar, la respuesta estricta debe ser una señal senoidal sin alguna componente por encima de la  $f_c = 7\text{KHz}$ , como se muestra en la Figura 6.3.

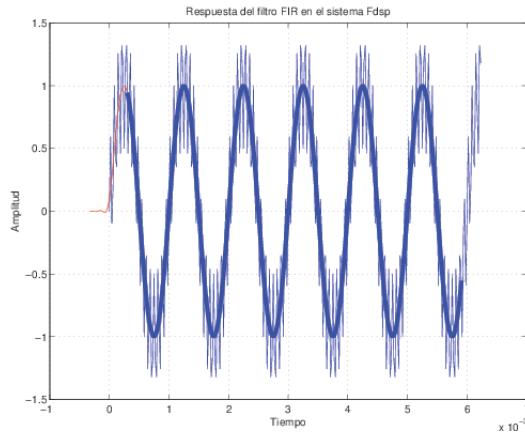


Figura 6.3: Resultado de la implementación del filtro FIR, donde se puede observar la señal resultante, de un color más oscuro, en oposición a la señal contaminada. La señal resultante está totalmente reconstruida, lo que valida la implementación.

### 6.3. Simulación del sistema en Sysgen.

Las instrucciones para transformar el algoritmo que hemos diseñado ya, a un modelo que se pueda ejecutar en la tarjeta de desarrollo Atlys, se muestran a continuación:

1. Para ejecutar Sysgen, no se abre Matlab directamente puesto que el toolbox de Xilinx no aparecerá en las librerías de Simulink. La forma de hacerlo es ir a la barra de **Inicio > Todos los programas >**

**Xilinx Design Tools > ISE > System Generator.** Esto invocará a Sysgen con el toolbox habilitado con la leyenda mostrada en la Figura 6.4.

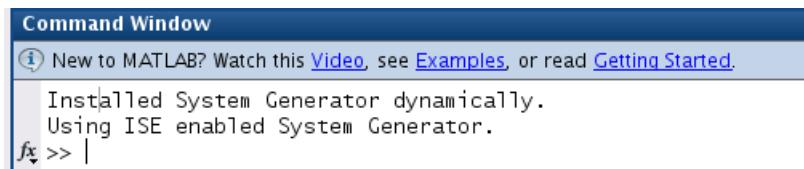


Figura 6.4: Mensaje de instalación del toolbox de Sysgen en Matlab R2013a.

2. Dar clic en el ícono  posicionado en la barra superior de la interfáz de Matlab, esto abrirá una ventana con los toolboxes de Simulink disponibles en la instalación. Los toolboxes de Xilinx se encuentran listados en la pestaña de **Bibliotecas** con los nombres de '**Xilinx Blockset**', '**Xilinx Reference Blockset**' y '**Xilinx XtremeDSP Kit**'.
3. Crear un nuevo modelo seleccionando el ícono '*New model*' o presionando **CTRL+N**.
4. Dentro de esta nueva ventana, el primer paso que siempre se debe realizar es agregar el '*Sysgen token*' desde **Xilinx Blockset > Basic Elements**, como se muestra en la Figura 6.5.

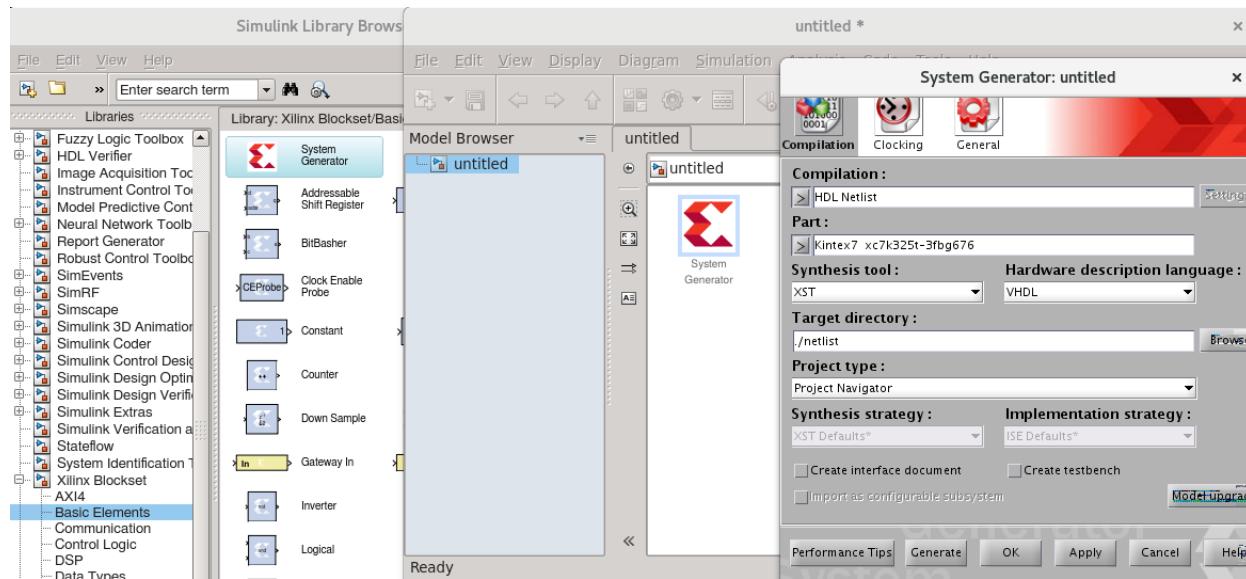


Figura 6.5: Instanciación del System Generator token.

Aquí es necesario resaltar algunas reglas importantes para hacer uso del toolbox de Xilinx. La primera es que, en todo diseño de Simulink donde se involucren bloques de Xilinx, debe existir el '*Sysgen token*', este es un bloque especial que no tiene entrada o salida alguna, pero es la interface que convierte

los vectores de simulación de Matlab a datos que la FPGA pueda comprender. Además, este bloque contiene la información del proyecto que estamos modelando, es decir, aquí se puede definir como el diseño va a ser transformado acorde a los flujos discutidos en la **sección 5.5.2**.

5. Al dar doble clic sobre el 'Sysgen token' se abrirá una ventana donde podremos definir los parámetros de implementación, los cuales definiremos como los que se muestran a continuación:

- **Compilation:** Hardware Co-Simulation > Atlys > JTAG.
- **Synthesis tool:** XST.
- **Hardware Description Language:** Verilog.
- **Project type:** PlanAhead.
- **Synthesis strategy:** PlanAhead Defaults.
- **Implementation Strategy:** ISE Defaults.
- **Clocking > Simulink system period:** 2.0833e-05 (1/48e3), este parámetro es el más importante puesto que si es agregado con un valor incorrecto, el sistema no responderá como se espera. Este periodo define la frecuencia de muestreo a la que el sistema trabaja.

6. Ahora, se necesita agregar uno de los bloques más importantes los cuales definen el módulo que será sintetizado por la FPGA: Los bloques de entrada/salida, conocidos como '*Gateway in*' y '*Gateway Out*'.

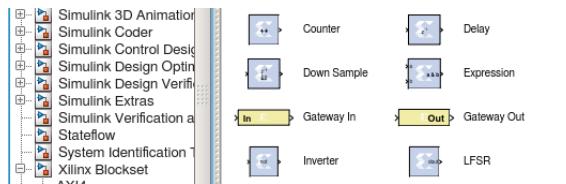


Figura 6.6: Bloques de entrada/salida que acotan la implementación en el FPGA del algoritmo a diseñar.

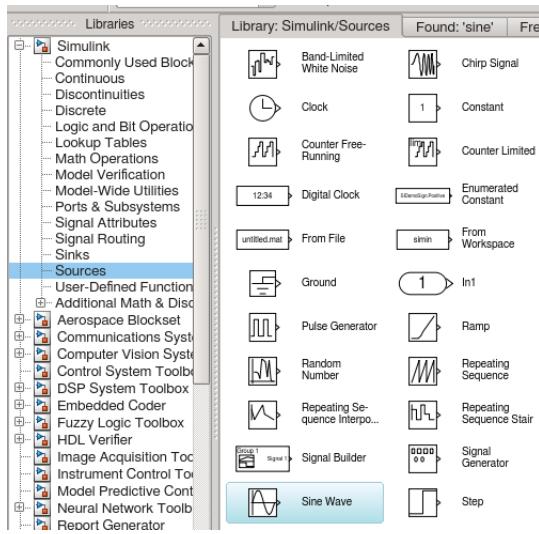
En otras palabras, los únicos bloques que pueden comunicarse directamente con otros toolboxes de Simulink son los *Gateway In/Out*. Este par de bloques transforman los estímulos de Simulink en datos que el FPGA puede procesar, también son capaces de capturar la información del tipo de dato a utilizar (punto fijo o punto flotante), la frecuencia de muestreo e incluso alguna locación física dentro de los puertos de entrada y salida generales de la FPGA, para funcionar como disparadores externos.

Este bloque se configura de la siguiente manera:

- **Output type:** Fixed point.

- **Arithmetic type:** Signed (2's comp).
- **Fixed-point Precision > Number of bits:** 16, **Binary point:** 14.
- **Sample period:** 2.0833e-05.

7. Como se utilizó en el algoritmo de Matlab, será necesario generar la señal que deseamos filtrar, a partir de utilizar dos generadores de señal senoidal del toolbox de Simulink, ubicados en **Simulink > Sources > Sine Wave**, como se muestra en la Figura 6.7.



**Figura 6.7: Generadores de función senoidal utilizados para generar la señal a filtrar.**

Se deben utilizar dos generadores de señal senoidal, con los parámetros mostrados a continuación:

a) Señal base:

- **Amplitude:** 1.
- **Frequency:**  $f_{senBase} * 2\pi = 1e3 * 2 * \pi$ .
- **Sample time:**  $1/Fs = 1/48e3$ .

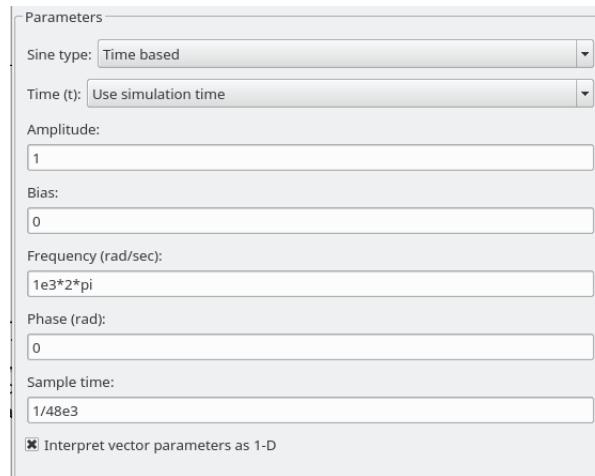


Figura 6.8: Parámetros de configuración para la componente base de entrada al sistema de filtrado.

b) Señal a filtrar:

- **Amplitude:** 0.5.
- **Frequency:**  $f_{senRuido} * 2\pi = 15e3 * 2 * \pi$ .
- **Sample time:**  $1/F_s = 1/48e3$ .

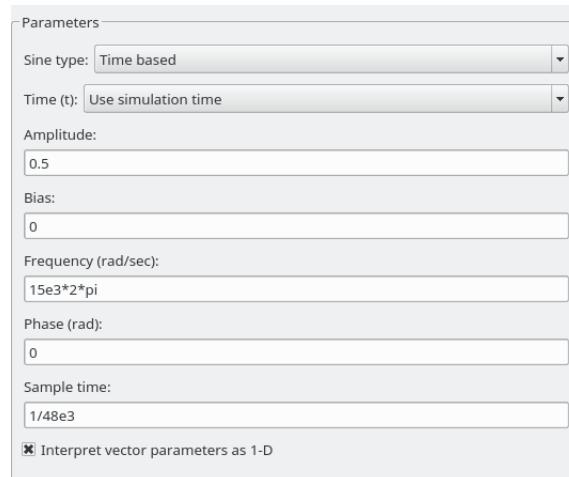


Figura 6.9: Parámetros de configuración para la componente de ruido de entrada al sistema de filtrado.

8. También se necesitará un sumador de señales para mezclar ambas funciones senoidales, el cual se encuentra en **Simulink > Math Operations Sum:** .
9. El bloque que se encargará de filtrar la señal se encuentra en **Xilinx Blockset > DSP > FIR Compiler 5.0**, se debe configurar con los siguientes parámetros:

- **Coefficient Vector** =  $fir1(orden, Wn) = [0.0016, 0.0013, -0.0005, -0.0037, -0.0055, -0.0018, 0.0084, 0.0175, 0.0133, -0.0103, -0.0406, -0.0482, -0.0053, 0.0900, 0.2034, 0.2804, 0.2804, 0.2034, 0.0900, -0.0053, -0.0482, -0.0406, -0.0103, 0.0133, 0.0175, 0.0084, -0.0018, -0.0055, -0.0037, -0.0005, 0.0013, 0.0016]$ .
- **Hardware Oversampling Specification > Sample Format:** Sample\_period.
  - **Sample Period:** 1.
- **Implementation > Filter Architecture:** Distributed\_Arithmetic.
  - **Coefficient Options > Coefficient Structure:** Inferred.
  - **Coefficient Type:** Signed.
  - **Quantization:** Quantize\_Only.
  - **Coefficient Width:** 16.
  - Checar la opción **Best Precision Fraction Length**.



Figura 6.10: Configuración del bloque FIR Compiler.

10. Una vez teniendo estos elementos en el nuevo modelo de Simulink, se deben conectar como se muestra en la Figura 6.11.

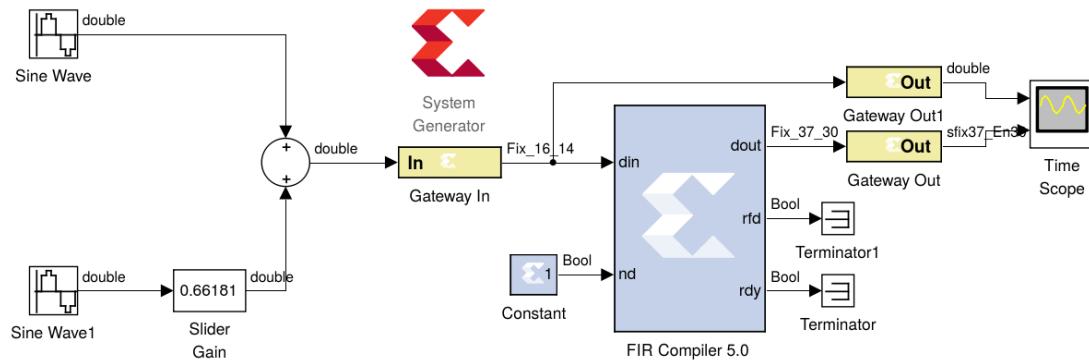


Figura 6.11: Interconexión de los bloques de Simulink/Sysgen para la elaboración del filtro FIR.

11. Debido a que la señal de interés en este sistema, es la  $f_{senBase}$  con 1KHz, para poder observar 10 muestras en el osciloscopio de simulink, pondremos como valor en el *Simulink stop time* = 0.010, que equivale a multiplicar el periodo de la señal de 1KHz por 10.

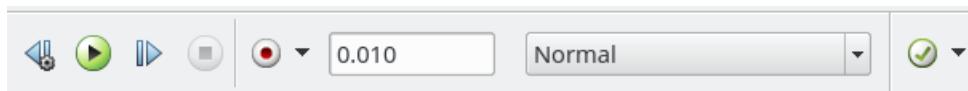


Figura 6.12: Parámetro de tiempo de simulación para este ejercicio.

12. Ejecutar el sistema. El scope deberá mostrar dos señales, en la parte superior la señal con ruido inducido y en la posterior, la señal filtrada libre de cualquier deformación, como se muestra en la Figura 6.3.

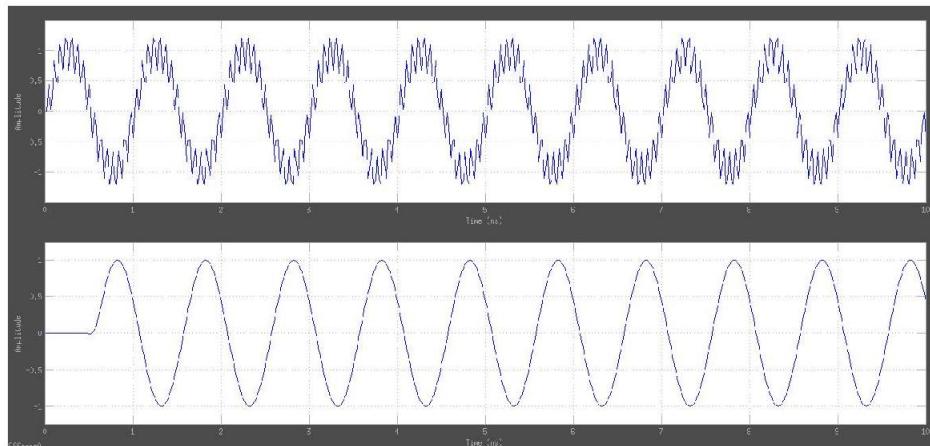


Figura 6.13: Resultado de la simulación del sistema utilizando los bloques de Sysgen.

## 6.4. Implementación del sistema en el kit Atlys.

El siguiente proceso describe como utilizar el kit Atlys como co-procesador, mediante la compilación y generación de una biblioteca que contiene el modelo que se describió con anterioridad. Mediante el uso de la herramienta Sysgen, se generarán los archivos de HDL necesarios así como la síntesis e implementación en segundo plano. Al finalizar, La tarjeta de desarrollo será capáz de comunicarse con Simulink para enviar y recibir los datos correspondientes a este modelo.

1. Al dar doble clic sobre '*Sysgen token*', existe un parámetro de usuario donde se define el tipo de compilación. Para este caso, se utilizarán las siguiente configuración:

- **Compilation:** Hardware Co-Simulation > Atlys > JTAG.

2. Dar clic en '*Generate*'. Esto ejecutará a PlanAhead en segundo plano, con la intención de generar el *bitfile* o ejecutable que se descargará en la FPGA. Al finalizar, aparecerá en una nueva ventana la biblioteca generada por Sysgen, que no es más que la representación del algoritmo en términos de hardware.

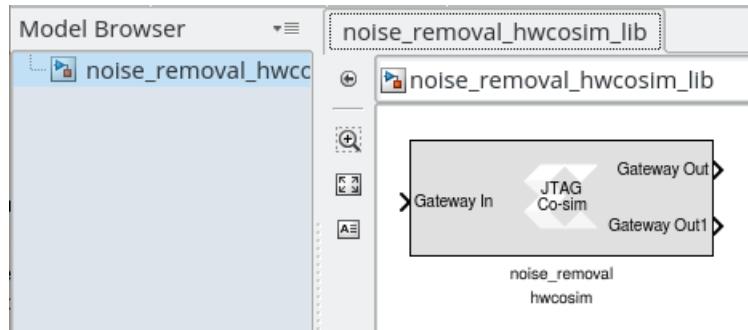


Figura 6.14: Generación de la representación del kit Atlys en Simulink, el cual contiene los recursos de hardware necesario para ejecutar el algoritmo del filtro FIR.

3. Con el nuevo bloque generado, se deben actualizar las conexiones de la siguiente forma.

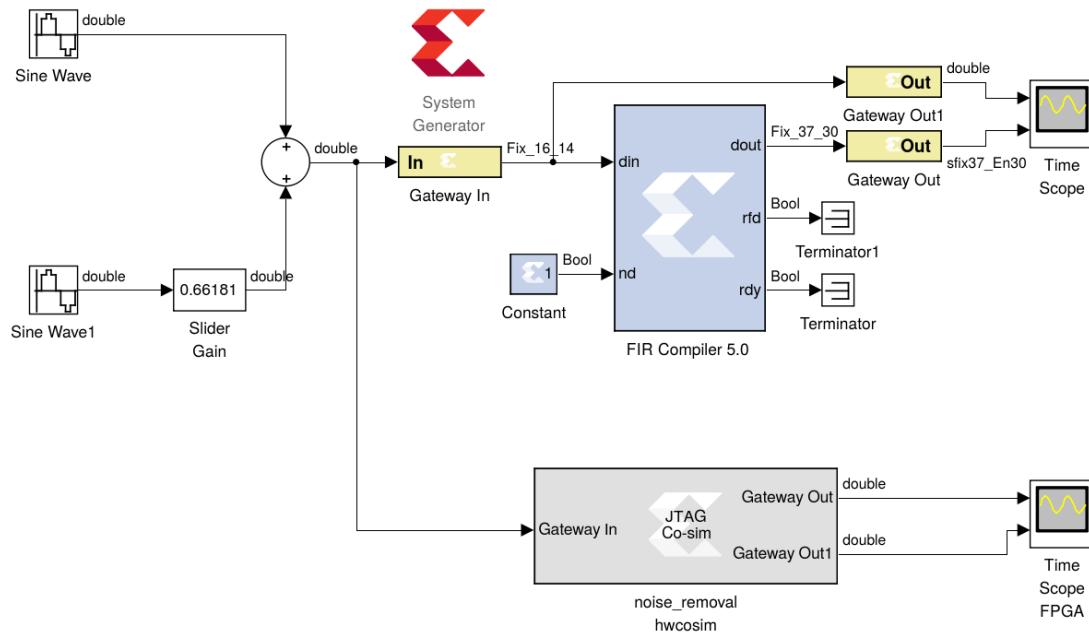


Figura 6.15: Conexión del bloque de Co-simulación.

4. Al integrar el bloque generado por Sysgen, es necesario conectar el kit Atlys a la PC antes de volver a ejecutar la simulación, puesto que ahora, el bitfile se descargará en el kit y podremos observar los resultados en el segundo osciloscopio, como se muestra en la Figura 6.16.

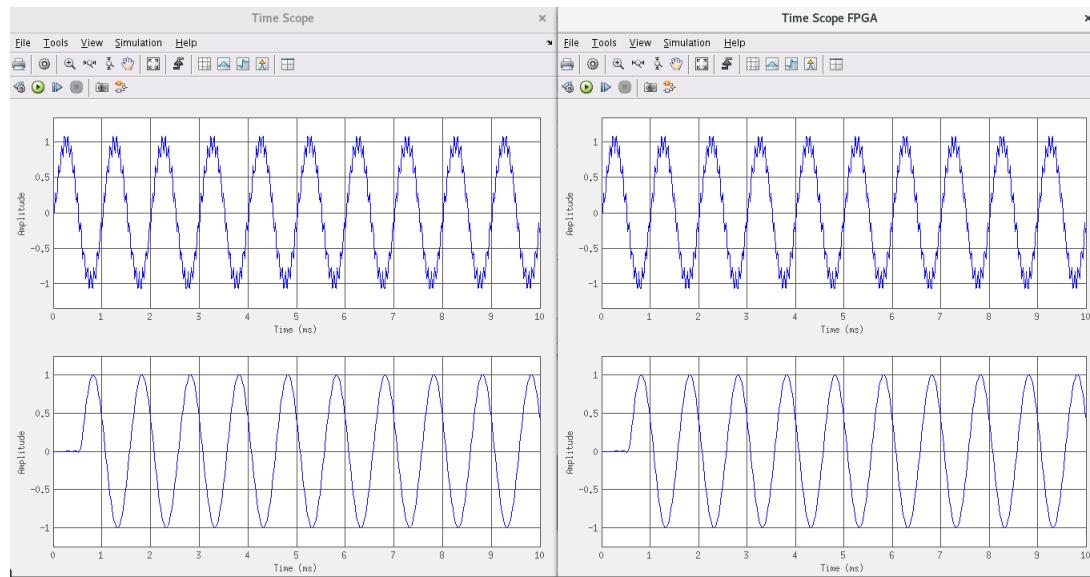


Figura 6.16: Comparación entre el resultado de la simulación en contraste con la ejecución en FPGA como co-procesador.

5. La traducción de este filtro FIR a hardware se hizo mediante la utilización de BRAMs para alma-

cenar los coeficientes, un contador y un par de elementos multiplicadores y acumuladores. Todo esto se puede analizar abriendo el proyecto generado que se encuentra en *netlist > hwcosim > nombre\_del\_proyecto.ppr* con PlanAhead.

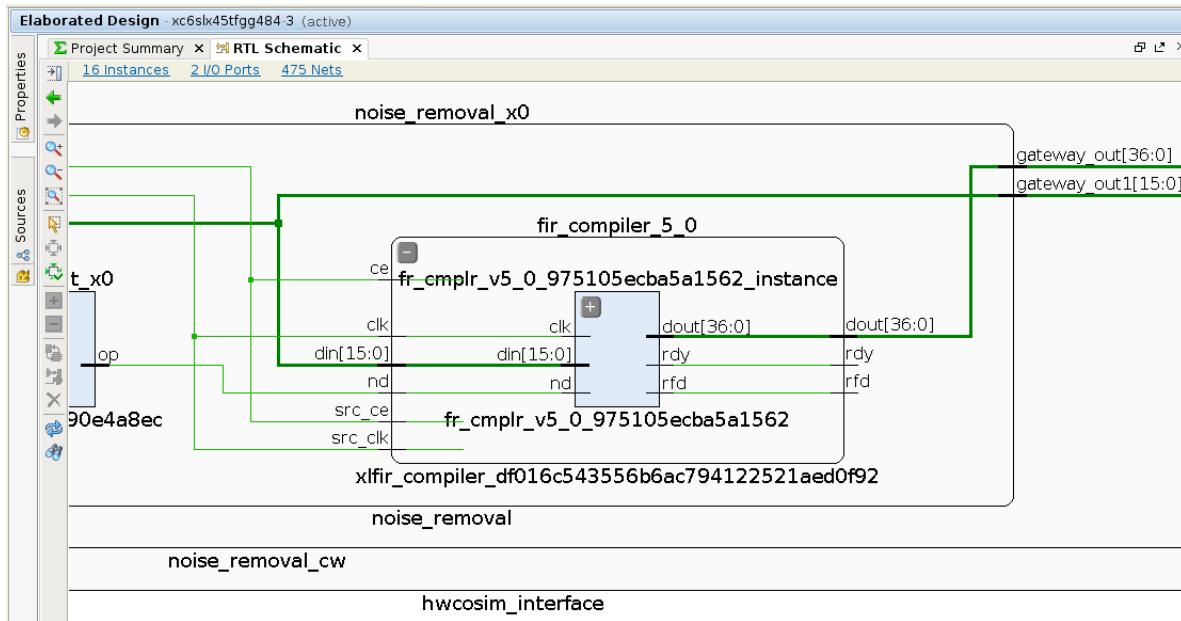


Figura 6.17: Diagrama esquemático de la implementación del modelo del filtro FIR en FPGA.

6. La utilización del sistema y frecuencia máxima a la que se puede ejecutar este modelo se muestra en la Figura 6.18.

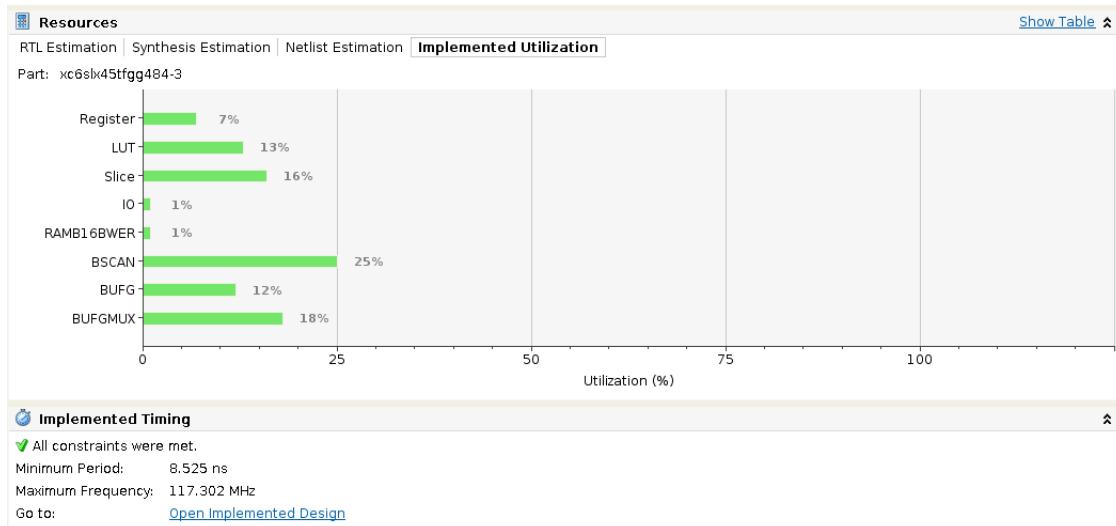


Figura 6.18: Utilización de recursos de la FPGA para la implementación del filtro FIR.

## 7. Procesamiento de Audio - Algoritmo de eco.

### 7.1. Filtros FIR peine o *comb filter*.

El efecto de eco es uno de los más utilizados en el procesamiento de señales de audio. De este se pueden derivar otros más como la reverberación, vibrato, coro y otras transformaciones basadas en retrasos.

Esta implementación está basada en la utilización del filtro FIR conocido como **peine** o en inglés como *fir comb filter*. La entrada del filtro es retrasada por un tiempo definido y el efecto sólo será audible cuando la señal procesada sea combinada con la original (de ahí el nombre de *comb filter*).

La ecuación diferencial del filtro peine está dada por:

$$y_n = x_n + g * x_{n-M} \quad (7.1)$$

Si trasladamos la ecuación 7.1 a un diagrama de bloques, nos queda como se muestra en la Figura 7.1.

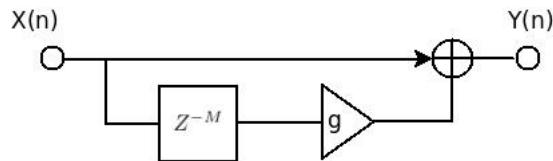


Figura 7.1: Diagrama a bloques del filtro peine o *comb filter*.

A partir de esto podemos observar que, la estructura se conforma de elementos básicos como bloques de retraso ( $Z^{-M}$ ), multiplicadores que sirven como ganancia y sumadores, los cuales fácilmente pueden ser implementados en un FPGA.

### 7.2. Implementación en FPGA del códec de audio AC97.

El códec de audio disponible en la tarjeta Atlys es el AC97 LM4550<sup>9</sup> de National Semiconductor; AC97 es un standard desarrollado por Intel en 1997, el cual es utilizado en tarjetas madre de ordenadores, módems y tarjetas de sonido.

<sup>9</sup>Para obtener mayor información técnica a cerca del códec, visitar: <http://www.xilinx.com/products/boards/ml510/datasheets/lm4550.pdf>

El formato de datos del LM4550 es serial, de 18 bits con una frecuencia de muestreo a 48KHz. Es importante mantener este dato en consideración a la hora de realizar cualquier algoritmo de procesamiento, puesto que, si la magnitud de la trama enviada al códec es mayor o menor a la requerida, será necesario alinear los datos para que la señal de audio no sea deformada, haciéndola parecer como una señal de ruido.

Puesto que el objetivo de este documento no es detallar el desarrollo del controlador del códec, se provee un netlist pre-sintetizado (Verilog) el cual tiene toda las siguientes características (véase Figura 7.2):

- Frecuencia de trabajo a 100MHz, provenientes del oscilador externo al FPGA.
- Reset activado por flanco de bajada (activo en bajo).
- Reloj de 12.288MHz desde el puerto *bit clock* del códec.
- Selector de 3 bits para escoger la fuente de entrada (Mic, LineIn, headphone) y de 5 bits para volumen. Estos se pueden mapear a los switches de la tarjeta Atlys para controlar dichos parámetros.
- Parámetros seleccionados para la mayor calidad de audio posible: 48KHz de frecuencia de sampleo, con una palabra de 16 bits en lugar de 18 (por compatibilidad con la mayoría de los formatos de audio disponibles).

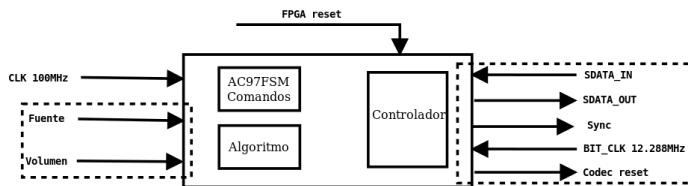


Figura 7.2: Diagrama a bloques de la implementación del controlador para el códec AC97 en Verilog.

### 7.3. Desarrollo del modelo en Sysgen.

Los pasos para llevar a cabo la implementación del algoritmo de eco se muestran a continuación:

1. A partir de tener el nuevo espacio de trabajo en Simulink (se pueden seguir los pasos 1 a 4 de la subsección **6.3 Simulación del sistema en Sysgen**), el bloque *Sysgen Token* deberá tener las siguientes configuraciones:
  - **Compilation:** HDL Netlist.

- **Part:** Sparnta6 xc6lx45-3csg324
- **Synthesis tool:** XST, **Language:** Verilog.
- **Project Type:** PlanAhead.
- **Synthesis Strategy:** TimingWithIOBPacking
- **Implementation Strategy:** ParHighEffort.
- **Clocking Tab > FPGA clock period (ns):** 10.
- **Multirate implementation:** Clock Enables.
- **Simulink system period:**  $1/48KH\zeta = 2.0833e-05$ .

2. Añadir el bloque **DSP System Toolbox > From Multimedia File** y configurarlo como sigue:

- **File name:** En este documento se proporciona el archivo partita\_e\_major.wav obtenida libremente de <http://www.music.helsinki.fi/tmt/opetus/uusmedia/esim/index-e.html>. Este archivo de audio tiene la frecuencia de muestreo a 48KHz, stereo a 16 bits.
- Seleccionar el recuadro **Inherit sample time from file**.
- **Number of times to play the file:** 1.
- **Outputs > Samples per audio channel:** 1.
- **Outputs > Audio output sampling mode:** Sample based.

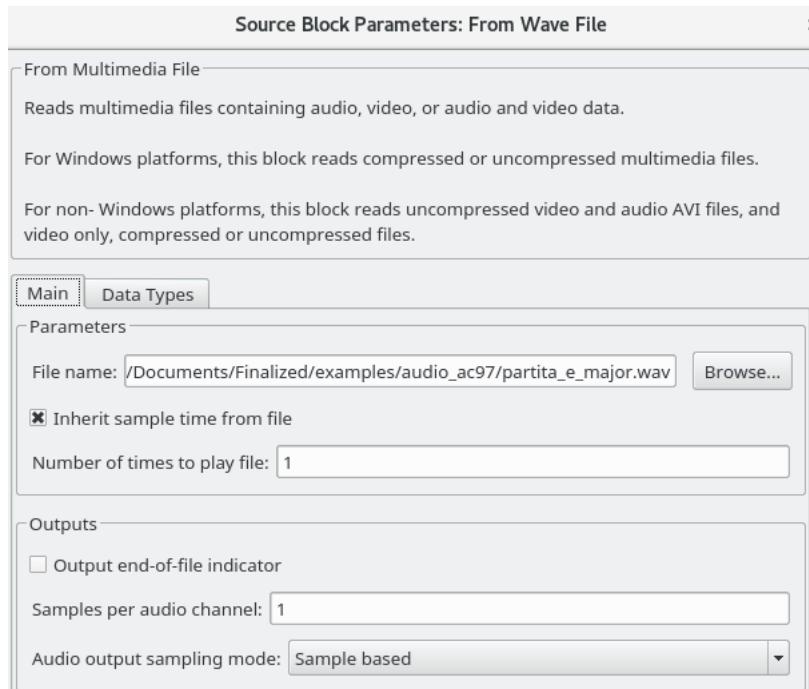


Figura 7.3: Configuración del bloque *From multimedia File* para servir como fuente de sonido.

3. El archivo de audio representa una matriz de 1x1 (datos, frecuencia de muestreo). Es necesario convertir esta matriz a un vector de un solo elemento mediante el uso de los bloques '*Reshape*' y '*Unbuffer*', ubicados en **Simulink > Math Operations** y **DSP System Toolbox > Signal Management > Buffers** respectivamente.
4. Para manejar los canales de audio dentro de la FPGA, se utilizarán dos '*Gateway In*' con los parámetros siguientes:

- **Output type:** Fixed point.
- **Arithmetic type:** Signed (2's comp).
- **Fixed-point Precision:** Number of bits = 16, Binary point = 14.
- **Overflow:** Saturate.
- **Sample period:** 2.0833e-05.
- Es necesario nombrar ambos puertos como **AudioLeftIn** y **AudioRightIn** respectivamente, cuidando de siempre mantener los mismos nombres o de lo contrario, el archivo NCG del controlador del códec marcará un error al intentar utilizar el netlist de Sysgen. Se deben configurar como se muestra en la Figura 7.4.

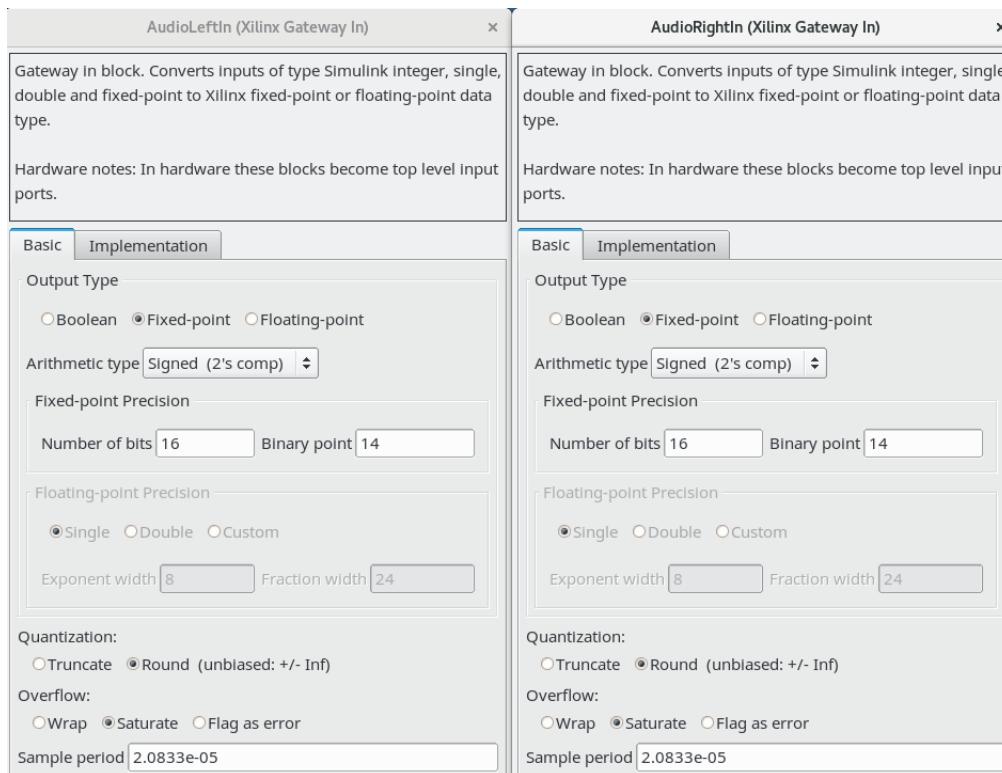


Figura 7.4: Configuración del tipo de dato usado por los puertos de entrada al filtro peine.

5. Como se mostró al principio de la **subsección 7.1**, ahora será necesario construir el filtro comb, siguiendo el bloque de la Figura 7.1 con algunas alteraciones necesarias para adaptarlo al kit Atlys, mismas que se explicarán posteriormente. El bloque debe quedar como se muestra en la Figura 7.5.

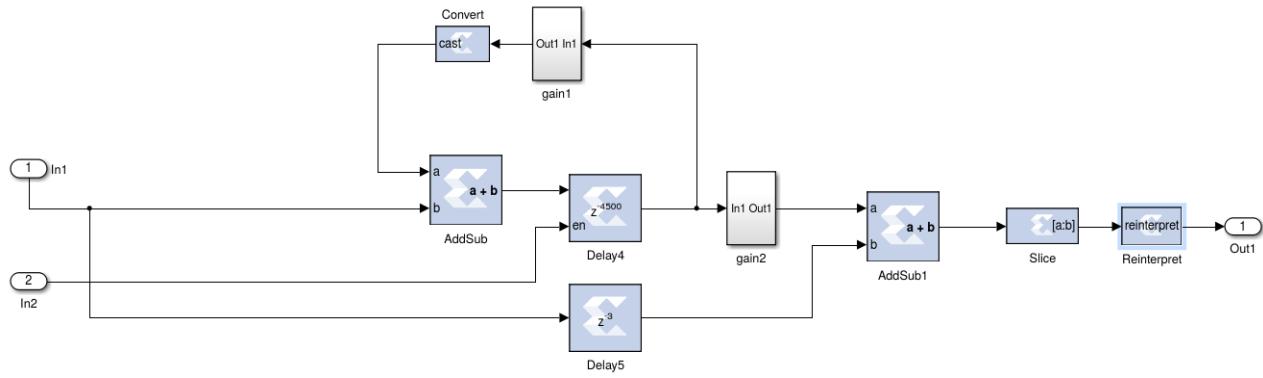


Figura 7.5: Filtro comb (FIR) con retroalimentación realizado en Sysgen.

El filtro se compone de un bloque *delay* de 4500 unidades de retraso (cada una de estas unidades representa un flipflop en términos de recursos de FPGA) y un subsistema de ganancia nombrado **gain2**, el cual se muestra a más detalle en la Figura 7.6.

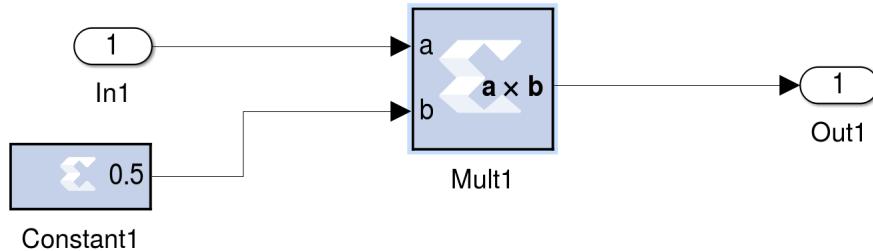


Figura 7.6: Parámetros de la ganancia del filtro comb.

Los bloques que le preceden al sumador **AddSub1** son necesarios debido a que, durante el flujo de información a través de los multiplicadores que actúan como ganancia, los datos se ven afectados sufriendo incrementos debido a la magnitud que ahora deben representar (aumenta el número de bits). Como el controlador del AC97 trabaja sólo con palabras de 16 bits, es necesario alinear y transformar la salida de dicho sumador, a que cumpla con los 16 bits requeridos por el códec.

La ganancia de retroalimentación es utilizada para estabilizar la salida del filtro, con el fin de remover lo más posible, ruido inferido por las modificaciones aplicadas a la señal de audio. Dicha ganancia

esta elaborada por el subsistema **gain1**, el cual se muestra a mayor detalle en la Figura 7.7.

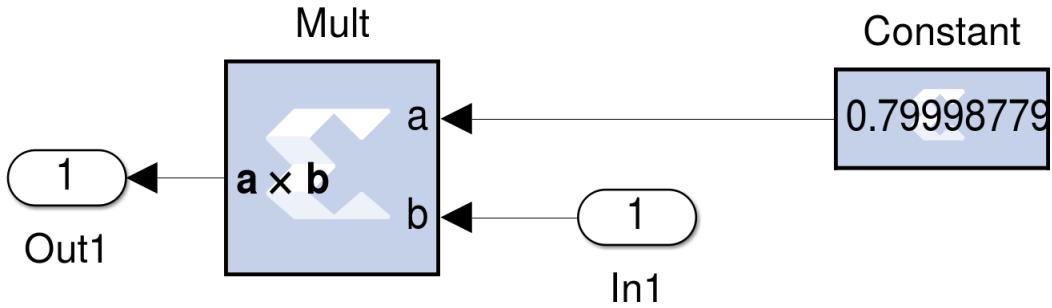


Figura 7.7: Ganancia de retroalimentación para estabilizar el sistema.

Por último, el bloque **Delay5** es utilizado para sincronizar los sumadores **AddSub** y **AddSub1** y así evitar problemas de timing debido al *cruce de reloj cruzado* (*CDC* por sus siglas en Inglés). Se pueden modificar los valores de retraso y ganancias con el fin de obtener efectos mucho más notables.

- Los elementos mostrados en el paso 5, pueden ser convertidos a subsistema para mantener la sencillez del sistema (seleccionar los bloques y ejecutar CTRL+G, los subsistemas pueden ser nombrados a gusto del usuario) y replicados para así, tener los dos canales del formato estéreo, como se muestra en la Figura 7.8.

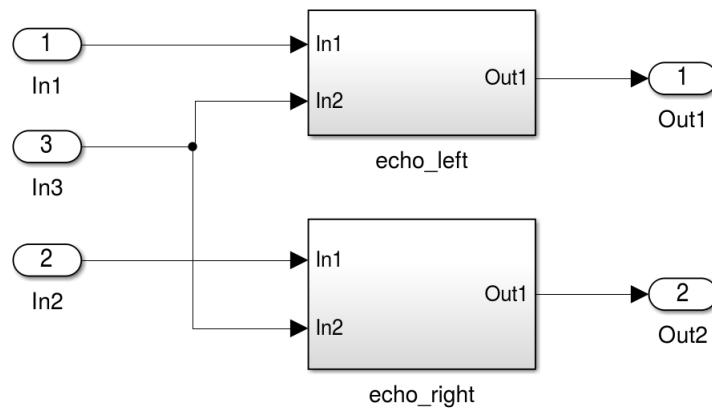


Figura 7.8: Conversión del filtro comb en subsistemas y replicación para formato estéreo.

- Como paso final, se deberán arrastrar dos '*Gateway Out*' al espacio de trabajo de simulink, y nombrarlos **AudioLeftOut** y **AudioRightIn** respectivamente, así como un bloque **Mux** (**Simulink > Commonly Used Blocks**) y un bloque **To Multimedia File** (**DSP System Toolbox > Sinks**), y conectarlos como se muestra en la Figura 7.9. El recuadro gris nombrado **Audio\_algoritmo** representa ambos filtros desarrollados.

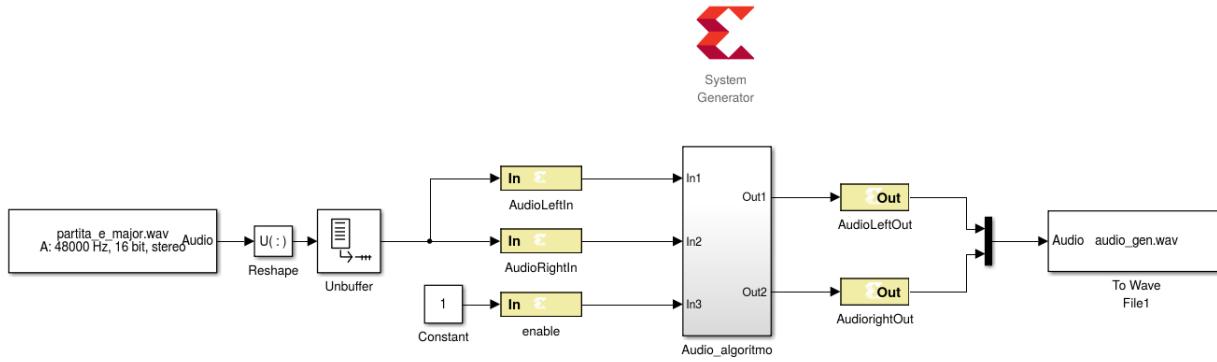


Figura 7.9: Interconexión del sistema de eco finalizado.

8. Para hacer la simulación de este modelo, utilizaremos un valor menor a la duración en segundos del archivo de audio como '*stop time*' en Simulink. El efecto es perfectamente audible utilizando el valor de **10.0(s)**. El efecto de eco puede ser analizado reproduciendo el archivo generado por el bloque '*To Wave File*' en la ruta especificada dentro del mismo, o utilizando el '*Time Scope*' para observar los datos con un ligero desfase, como se muestra en la Figura 7.10.

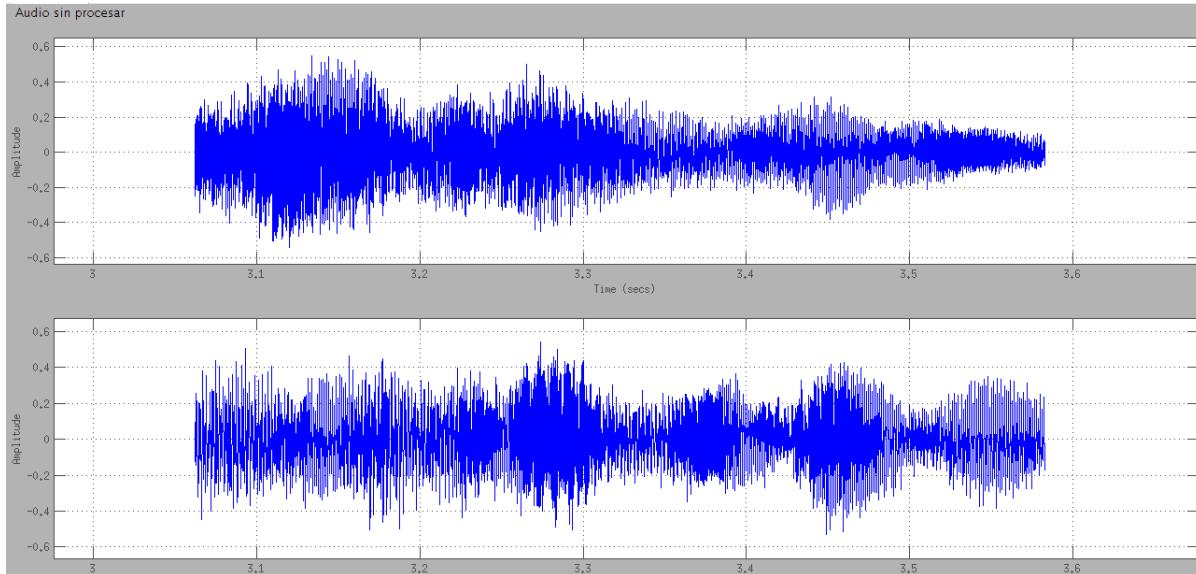


Figura 7.10: Gráfica resultante del eco aplicado al archivo de audio. La señal de la parte superior representa el audio sin procesar, mientras que la señal inferior muestra el audio procesado, el cual se compone de un desfase y una amplitud diferente a la original.

## 7.4. Implementación en el kit Atlys.

1. Una vez que obtenemos el resultado esperado, es necesario generar un '*Netlist*' (archivos HDL en Verilog o VHDL) para poder utilizar este modelo dentro de la interface del codec de audio. Esto se logra dando doble clic sobre el '*Sysgen Token*' > '*Generate*'.
2. Al terminar el proceso anterior, lanzar PlanAhead y crear un nuevo proyecto RTL<sup>10</sup> (véase **Apéndice D: Creación de proyectos usando PlanAhead** para más detalles). Se deben agregar primeramente los archivos **ac97cmd.v**, **ac97cmd.ngc**, **ac97.v**, **ac97.ngc**, **ac97\_constraints.ucf** y **rt\_audio\_controller.v**<sup>11</sup>, este último es el top module o módulo principal., que se conforma de las siguientes estructuras:
  - En primer lugar se definen los puertos de la interfáz principal entre el códec de audio y la FPGA. La directiva de compilación de la línea 1, '*NO\_LOOPBACK*', al estar comentada, hace que las salidas del códec se conecten a sus entradas (*loopback*). Al momento de instanciar el modelo generado en Sysgen, esta directiva debe ser descomentada.

### Puertos del módulo principal

```

1  `define NO_LOOPBACK
2  module rt_audio_controller( input      clk,
3                               input      n_reset,
4                               input      sdata_in,
5                               input      bitclk,
6                               input [4:0] volume,
7                               input [2:0] source,
8                               output     sync,
9                               output     codec_n_reset,
10                             output    sdata_out );

```

- La instanciación de la IP de audio se hace como se muestra a continuación. Se necesitan los archivos **ac97cmd.v**, **ac97cmd.ngc**, **ac97.v** y **ac97.ngc** para ser utilizada. Además, sólo funciona para la familia de FPGAs utilizada en el kit Atlys.

<sup>10</sup>Se añade un proyecto previamente configurado y funcionando con este documento, para referencia.

<sup>11</sup>Estos archivos se encuentran adjuntos en el proyecto de referencia.

### Instanciación de la IP de audio AC97

```

1      ac97cmd controller( .clk(gclk),
2                          .ac97_ready_sig(ac97_ready_sig_w),
3                          .cmd_addr(cmd_addr_w),
4                          .cmd_data(cmd_data_w),
5                          .latching_cmd(latching_cmd_w),
6                          .volume(volume),
7                          .source(source) );
8
9      ac97 datapath( .n_reset(n_reset),
10                      .clk(gclk),
11                      .ac97_sdata_out(sdata_out),
12                      .ac97_sdata_in(sdata_in),
13                      .ac97_sync(sync),
14                      .ac97_bitclk(bitclk),
15                      .ac97_n_reset(codec_n_reset),
16                      .ac97_ready_sig(slot_status_signal),
17                      .latching_cmd(latching_cmd_w),
18                      .cmd_addr(cmd_addr_w),
19                      .cmd_data(cmd_data_w),
20  `ifdef NO_LOOPBACK
21                      .L_out(left_audioin),
22                      .R_out(right_audioin),
23                      .L_in(left_audioout),
24                      .R_in(right_audioout)
25  `else
26                      .L_out(left_loopback),
27                      .R_out(right_loopback),
28                      .L_in(left_loopback),
29                      .R_in(right_loopback)
30  `endif
31      );

```

- Por último, el módulo generado por Sysgen está instanciado como se muestra en el siguiente cuadro. Si se utilizara otro módulo con un nombre diferente, sólo basta con especificar el nombre de este y hacer las conexiones necesarias.

**Instanciación del algoritmo de eco generado por Sysgen**

```

1  `ifdef NO_LOOPBACK
2      algodev_cw algorithm_from_sysgen( .clk(gclk),
3                                         .audioleftin(left_audioout),
4                                         .audiorightin(right_audioout),
5                                         .audioleftout(left_audioin),
6                                         .audiorightout(right_audioin),
7                                         .enable(slot_status_signal) ); // synthesis black_box
8  `endif

```

3. Al terminar de configurar el nuevo proyecto, se deberá ver como se muestra en la Figura 7.11.

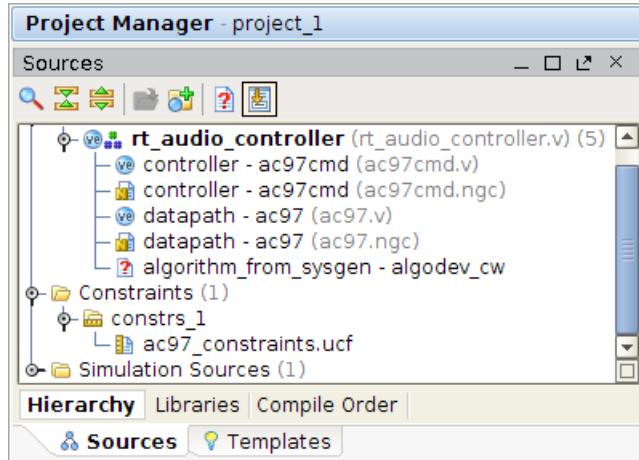


Figura 7.11: Vista jerárjica de las fuentes utilizadas para el proyecto.

Como se puede observar, el netlist del algoritmo de eco aparece como faltante. Para agregar estas dependencias, basta con dar clic sobre **Add sources > Add or Create Design Sources**, en la columna **Project Manager**, y seleccionar **Add Files**. En la ventana que aparece, seleccionar el directorio **netlist/sysgen** para añadir los archivos Verilog **algodev.v** y **algodev\_cw.v**, como se muestra en la Figura 7.12.

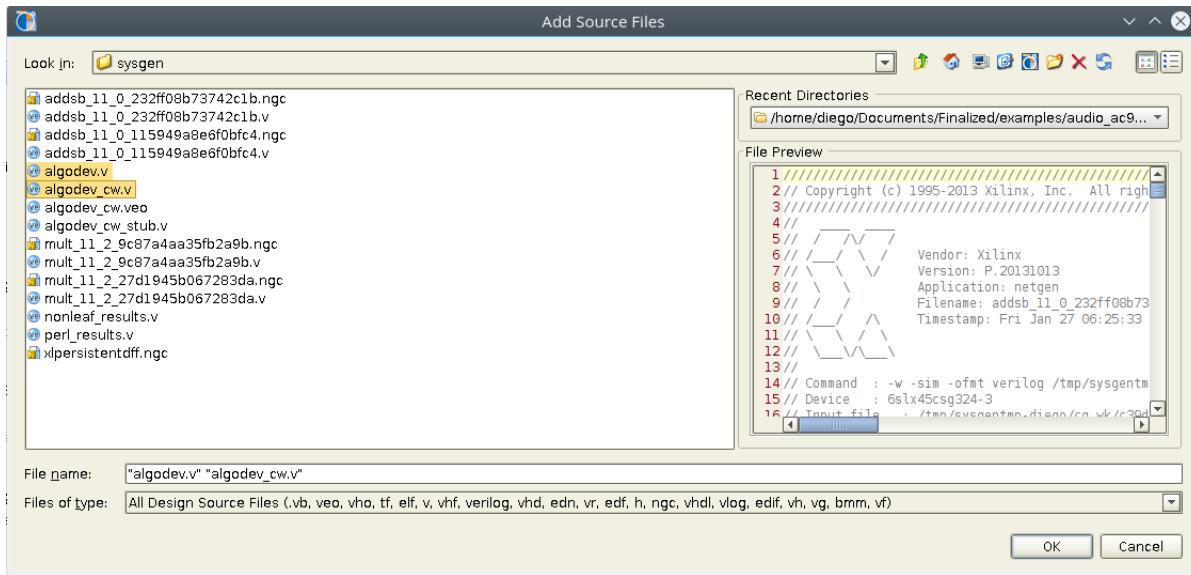


Figura 7.12: Archivos de Verilog que contienen el hardware realizado en Sysgen para implementar el algoritmo de procesamiento de audio.

4. Ahora, se puede ir a través de todo el flujo de implementación en FPGA dando clic sobre **Program and Debug > Generate Bitstream**. Esto ejecutará el proceso de Síntesis, Place and Route y generará el *bitfile* o archivo de configuración que se deberá descargar al kit Atlys.

El esquemático de módulo principal se muestra en la Figura 7.13.

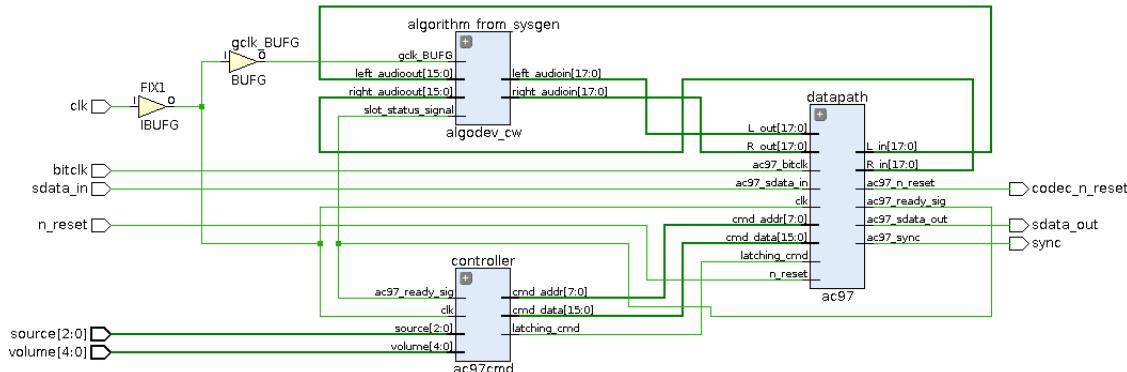


Figura 7.13: Esquemático completo de la implementación del modelo de procesamiento de audio.

5. La utilización total de recursos y frecuencia máxima de trabajo se muestra en la Figura 7.14.

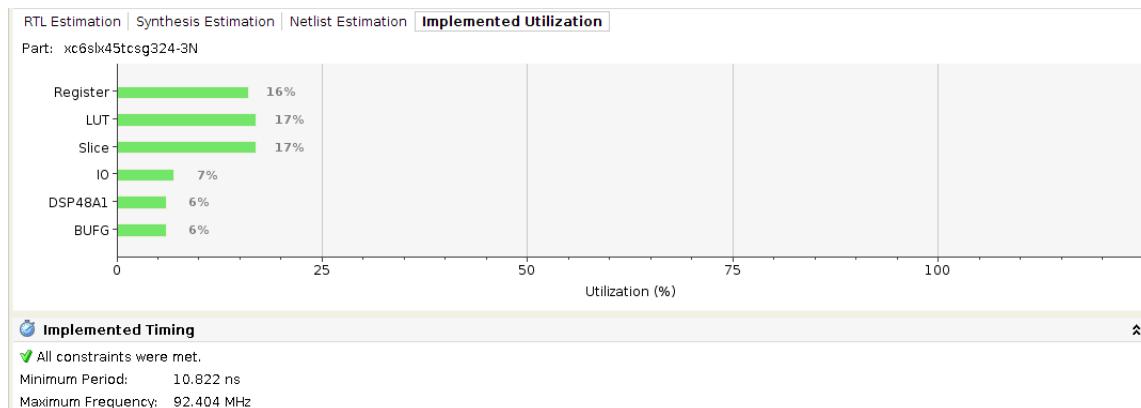


Figura 7.14: Utilización y frecuencia máxima de trabajo de la implementación final.

- Para ejecutar este diseño en la tarjeta Atlys, se debe conectar una fuente de audio (micrófono, reproductor MP3, etc) en el plug de color azul (Line In) y unos audífonos o bocina en el plug verde (Line Out), como se muestra en la Figura 7.15.

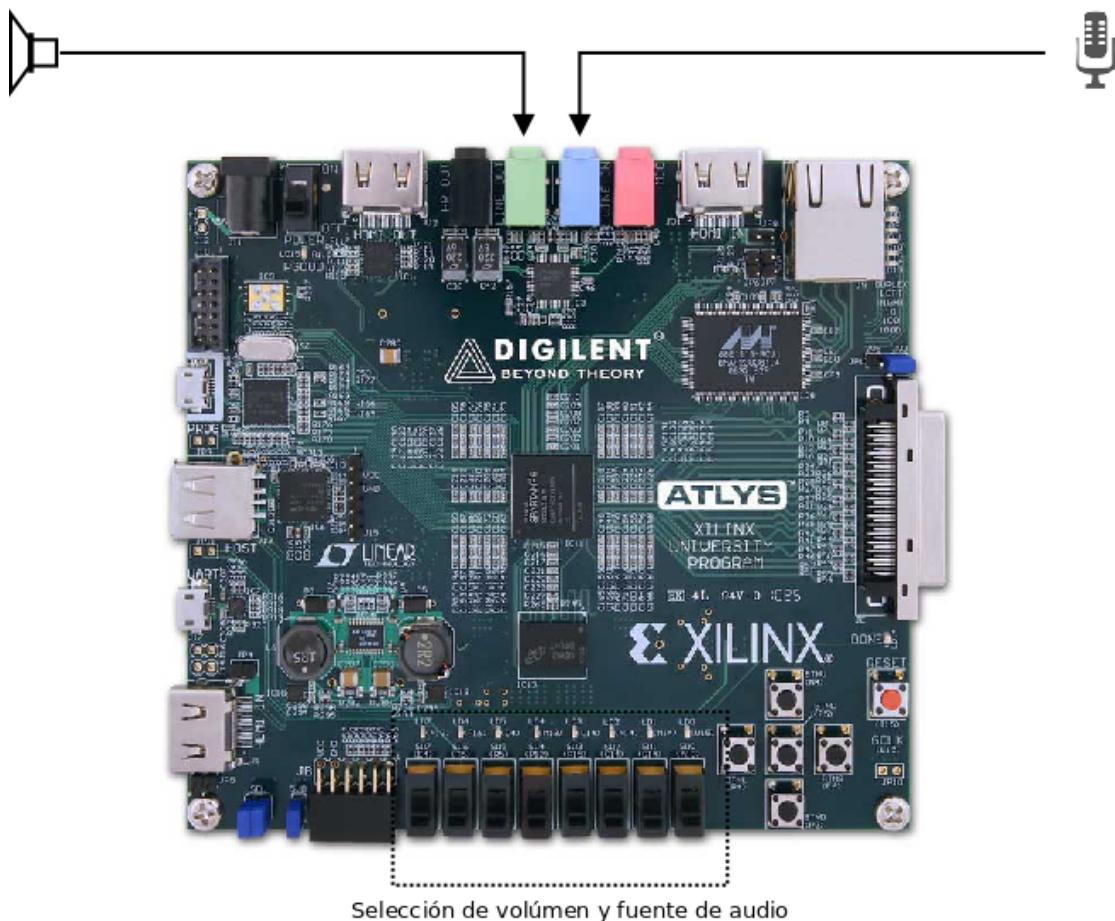


Figura 7.15: Conexiones a la tarjeta Atlys para ejecutar el algoritmo desarrollado.

## 8. Procesamiento de imágenes - Detección de bordes.

### 8.1. Características técnicas y arquitectura del modelo a implementar.

El flujo fundamental del diseño de algoritmos para el procesamiento digital de señales, se basa en la simulación y refinamiento del prototipo a implementar. Una vez que se tiene el resultado esperado del filtro en la imagen, se puede implementar en un modelo más complejo.

Para esta sección, se ilustrará paso a paso el desarrollo del operador Sobel para detección de bordes sobre imágenes en escala de grises, el cual debe cumplir las siguientes características:

- La transformación de RGB a escala de grises se debe hacer directo en la FPGA, de esta manera se puede reutilizar en sistemas mucho más complejos o en algoritmos diferentes.
- La frecuencia (periodo) del modelo de Simulink debe ser de 5.0863ns.
- Para términos de simplicidad, el modelo será ejecutado en el kit Atlys como co-simulación.
- La imagen a procesar es la que se muestra en la Figura 8.1, con una dimensión de 512\*384.



Figura 8.1: Imagen utilizada para la implementación del filtro Sobel.

### 8.2. Algoritmo del filtro Sobel.

La detección de bordes mediante la implementación del filtro Sobel es una práctica fundamental en el área del procesamiento de imágenes y video. Este algoritmo es utilizado para la extracción de características

de imágenes en 2D a escala de grises<sup>12</sup>, mediante el énfasis de las regiones de alta frecuencia espacial, que corresponden a los bordes.

El principio de implementación se basa en utilizar dos operadores de convolución matriciales o **kernels**, mostrados en la Ecuación 8.1. Estos operadores son matrices de diferente tamaño a la imagen (generalmente más pequeñas) pero de igual dimensión que dan como resultado, la estimación de la primera derivada vertical y horizontalmente con respecto a la cuadrícula de pixeles.

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad (8.1)$$

Después de aplicar la convolución usando dichos kernels, es necesario obtener la magnitud absoluta de las componentes obtenidas en ambos cálculos, utilizando la fórmula mostrada en la Ecuación 8.2:

$$|G| = \sqrt{G_x^2 + G_y^2} \quad (8.2)$$

Así mismo, se puede obtener el ángulo de orientación del borde (para usos más complejos) aplicando la Ecuación 8.3:

$$\theta = \arctan \frac{G_x}{G_y} \quad (8.3)$$

Este algoritmo, a diferencia de otros como el *Canny* o *Roberts Cross*, es menos sensible al ruido lo que significa que es más estable, aunque es más lento en cuanto a términos de computación de resultados.

### 8.3. Descomposición del espacio de colores RGB.

El proceso mostrado a continuación muestra como separar las componentes R | G | B de la imagen a procesar:

---

<sup>12</sup>Se utilizan imágenes a escala de grises y de dimensiones pequeñas para diseñar operaciones sobre ellas, puesto que simplifican la potencia computacional necesaria para obtener un resultado satisfactorio.

1. En un nuevo modelo de Simulink, instanciar un objeto '*Image From File*' que se encuentra en **Computer Vision System > Sources**, con las configuraciones mostradas en la Figura 8.2.

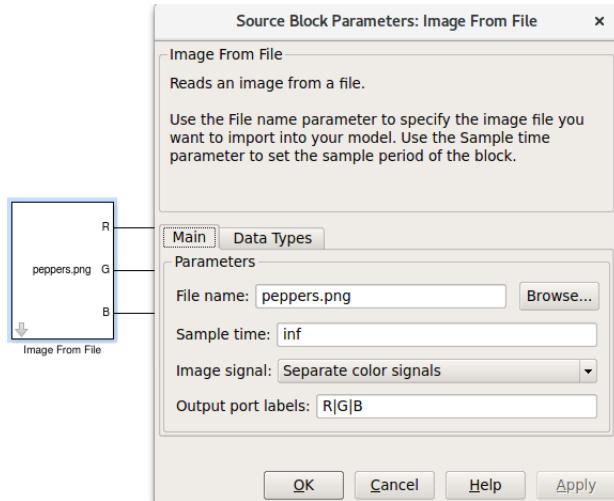


Figura 8.2: Parámetros de configuración del bloque '*Image From File*'.

El tipo de dato se deja en **double**, en la pestaña '*Data types*'.

2. Se necesitan también los bloques de '*Transpose*', '*Reshape*', '*Frame Conversion*' y finalmente '*Unbuffer*', con los parámetros mostrados en la Figura 8.3.

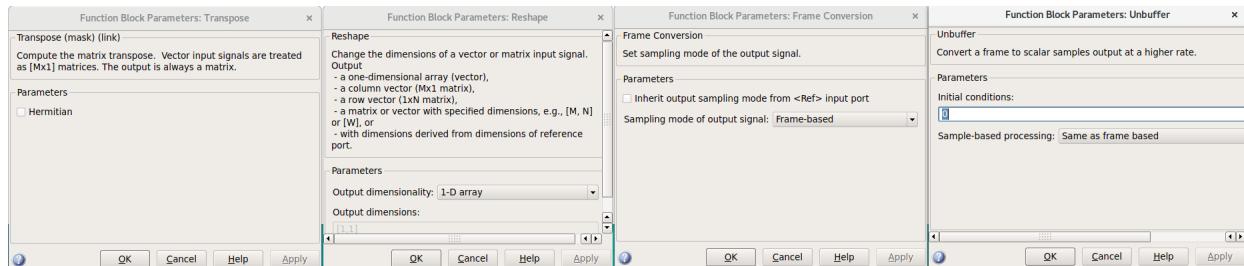


Figura 8.3: Configuración de los bloques '*Transpose*', '*Reshape*', '*Frame Conversion*' y '*Unbuffer*', respectivamente.

3. Se deben conectar dichos bloques como se muestra en la Figura 8.4, y repitiendo la estructura para cada uno de los espacios de colores RGB.

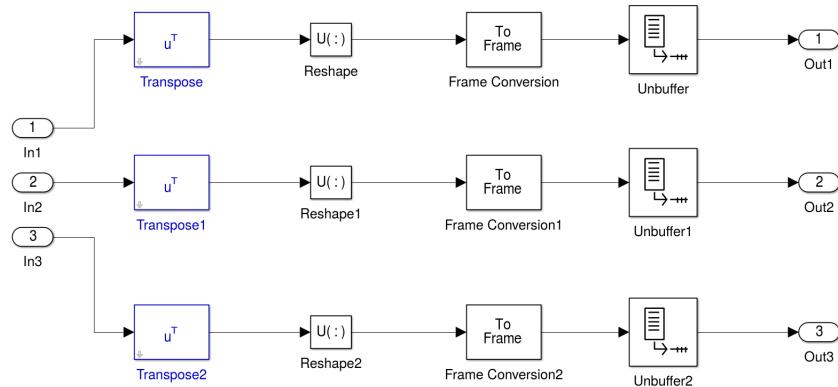


Figura 8.4: Conexión de los bloques necesarios para convertir la matriz 2D en un vector de 1D necesario para cada una de las componentes del espacio de color.

Las entradas a esta estructura vienen del bloque '*Image From File*', y a las salidas se puede conectar ya, el modelo de conversión RGB a escala de grises. La Figura 8.5 muestra los mismos bloques del Paso 3, convertidos en un subsistema.

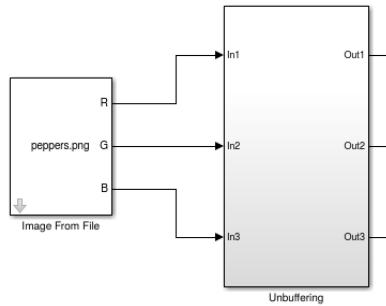


Figura 8.5: Conexión entre la fuente de imagen hacia el bloque de conversión de la matriz 2D a vectores por espacio de color.

#### 8.4. Modelo de conversión RGB a escala de grises.

Para llevar a cabo esta conversión, se utilizará el método de luminosidad, el cual consiste en ponderar específicamente a cada uno de los espacios de color. Dicho de otra forma, se debe aplicar la expresión mostrada en la Ecuación 8.4:

$$Y_{i,j} = 0.2989Y_{i,j,1} + 0.58709Y_{i,j,2} + 0.1140Y_{i,j,3} \quad (8.4)$$

Siguiendo los lineamientos dados en las especificaciones técnicas, la construcción de este algoritmo se logra siguiendo los pasos a continuación.

1. Agregar el '*Sysgen Token*' con los siguientes parámetros:

- **Compilation:** Hardware Co-Simulation > Atlys > JTAG.
- **Synthesis tool:** XST.
- **Hardware Description Language:** Verilog.
- **Project type:** PlanAhead.
- **Synthesis strategy:** PlanAhead Defaults.
- **Implementation Strategy:** ISE Defaults.
- **Clocking > Simulink system period:** 5.0863e-06.

2. Añadir un '*Gateway In*' por cada espacio de color, con la configuración mostrada en la Figura 8.6.

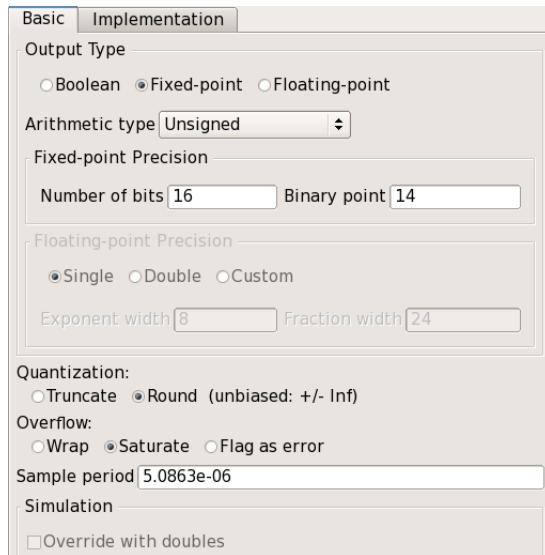


Figura 8.6: Configuración para las entradas RGB al sistema en FPGA.

3. Para efectos de rendimiento, se sugiere añadir un bloque '*Register*' a cada una de las entradas, como se muestra a continuación. Esto agrega un flujo de **cañería**, mejor conocido como **pipeline**, que transforma un sólo proceso en varias fases para evitar problemas de **timing**.

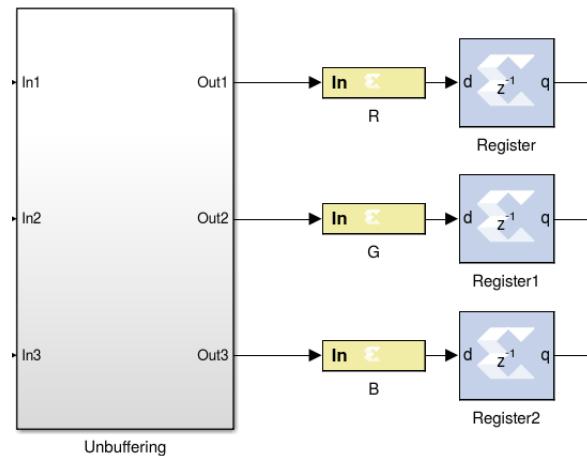


Figura 8.7: Entradas registradas usando bloques de Sysgen, con sus configuraciones originales.

4. La Ecuación 8.4 utiliza operadores como la suma y la multiplicación por cada uno de los valores constantes. Esto se traduce en hardware de la misma manera, utilizando un bloque '*CMult*' (**Xilinx Blockset > Floating-Point**) para cada uno de los canales de color, como se muestra en la Figura 8.8.

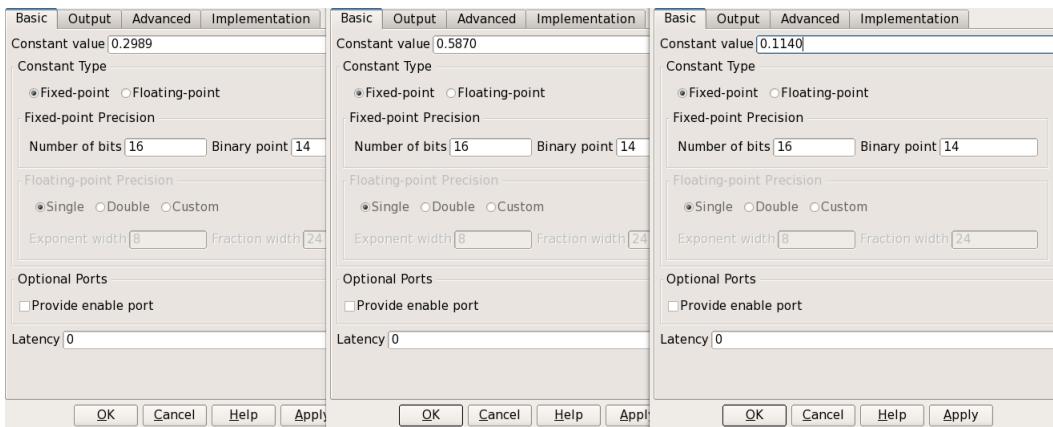


Figura 8.8: Configuración de los multiplicadores por el valor constante requerido para R, G y B respectivamente.

5. Así mismo, dos bloques '*AddSub*' (**Xilinx Blockset > Floating-Point**) son necesarios para obtener la suma de los valores calculados en el Paso 4. Los parámetros de estos dos bloques no se modifican.
6. La conexión a realizar utilizando los bloques antes mencionados se muestra en la Figura 8.9.

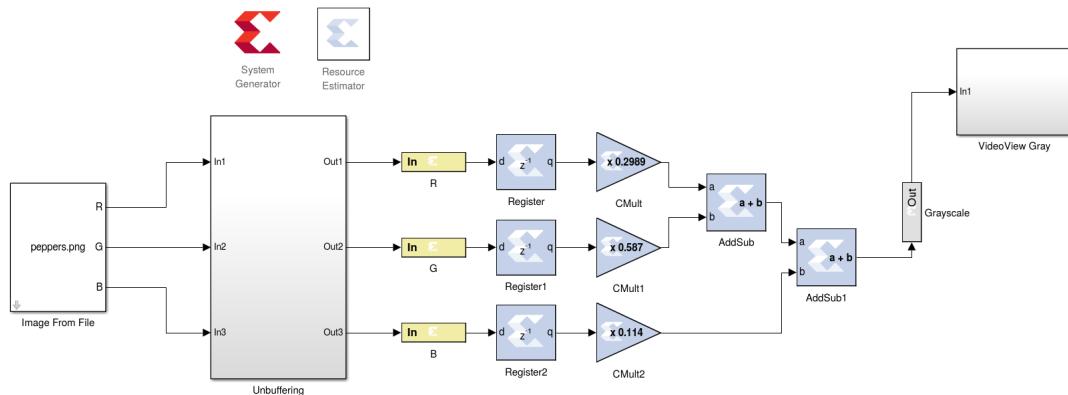


Figura 8.9: Modelo completo que realiza la conversión RGB a escala de grises de la imagen obtenida por el bloque '*Image From File*'.

7. El subsistema '*VideoView Gray*' se compone de un '*buffer*' con capacidad de 196608 unidades (512\*384), un bloque '*Frame conversion*' con su configuraciones sin modificar, un bloque '*Convert 1-D to 2-D*' con los parámetros mostrados en la Figura 8.10, y un bloque '*Transpose*' sin modificar. Todos conectados como se muestra en la Figura 8.11. El principal objetivo de este bloque es, reconstruir los datos multiplicados por el operador de luminosidad mostrada en la Ecuación 8.4 y al mismo tiempo, ensamblar esos datos en una matriz 2D que representa la imagen obtenida.

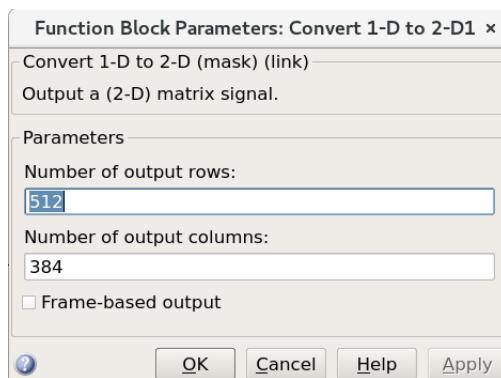


Figura 8.10: Parámetros necesarios para el bloque '*Convert 1-D to 2-D*'.

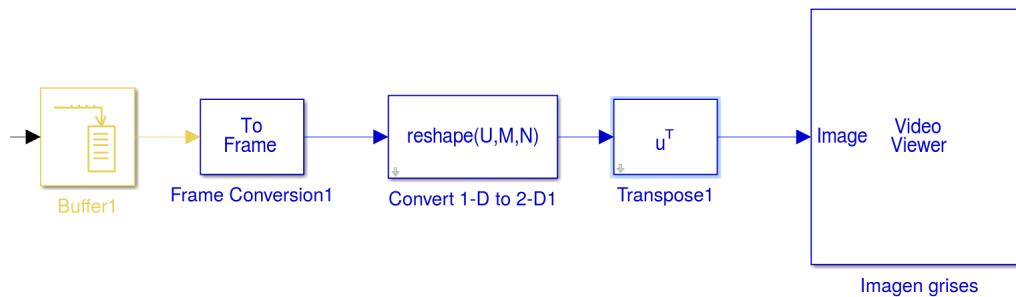


Figura 8.11: Reconstrucción de la imagen procesada por los bloques FPGA.

8. Como tiempo de simulación, se recomienda generalmente utilizar el número de filas por el número de columnas, en este caso  $512 \times 384$ . Al ejecutar la simulación, el resultado debe ser como se muestra en la Figura 8.12.



Figura 8.12: Resultado esperado de la conversión RGB a escala de grises.

Si se compara la salida de este modelo, con la popular función de Matlab, '*rgb2gray*', las matrices tendrán los mismos valores puesto que dicha función está basada en el mismo principio que el modelo desarrollado en esta sección.

## 8.5. Operador Sobel usando filtros FIR.

Como se mencionó en la subsección **8.2 Algoritmo del filtro Sobel**, serán necesarios dos kernels que realicen el filtrado de la imagen construida en la subsección **8.4 Modelo de conversión RGB a escala de grises**.

Para ello, se utilizarán dos bloques '*Fir Compiler 5.0*' que realizarán el filtrado en el eje X y Y de la imagen, utilizando los valores mostrados en la Ecuación 8.1 como coeficientes del filtro. Los pasos a seguir

se ilustran a continuación:

1. Se deben arrastrar dos bloques '*Fir Compiler 5.0*' al espacio de trabajo donde se generó el modelo de conversión RGB a escala de grises. Dichos bloques se encuentran en **Xilinx Blockset > DSP**.
2. El vector de coeficientes para el filtro del eje X tiene los valores de  $[-1, 0, +1, -2, 0, +2, -1, 0, +1]$ , como se muestra en la Figura 8.13.

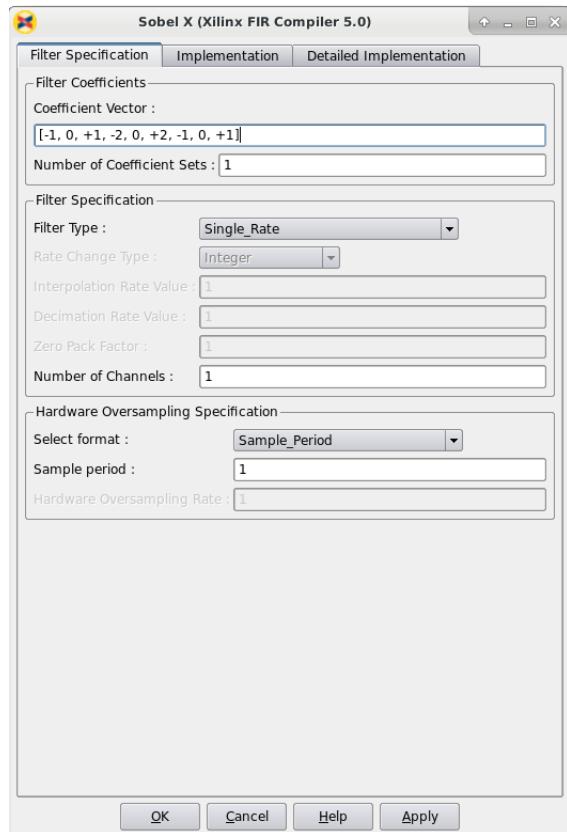


Figura 8.13: Coeficientes del filtro FIR para el eje X de la imagen.

3. Adicionalmente, se deben especificar los siguientes parámetros del tipo de datos usados para los coeficientes del filtro.

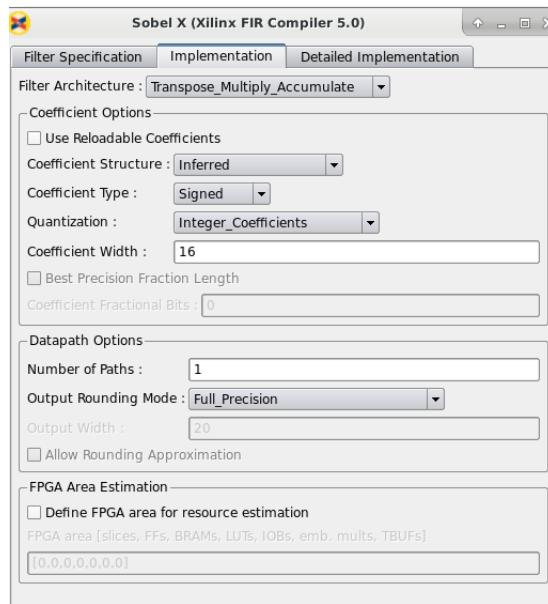


Figura 8.14: Configuración del tipo de dato de los coeficientes para el filtro FIR.

4. Se debe hacer lo mismo para el filtro del eje Y, en este caso el vector de coeficientes debe ser  $[+1, +2, +1, 0, 0, 0, -1, -2, -1]$ .

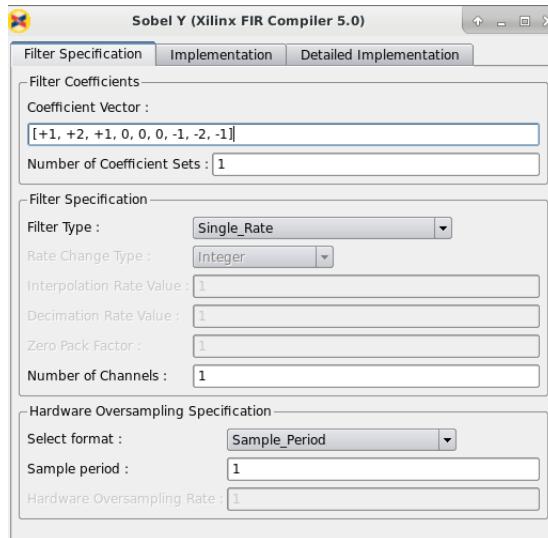


Figura 8.15: Coeficientes del filtro FIR para el eje Y de la imagen.

5. Se debe conectar una constante con un valor de 1 booleano, así como '*'Terminators'*' en las salidas no utilizadas de los filtros, como se muestra en la Figura 8.16.

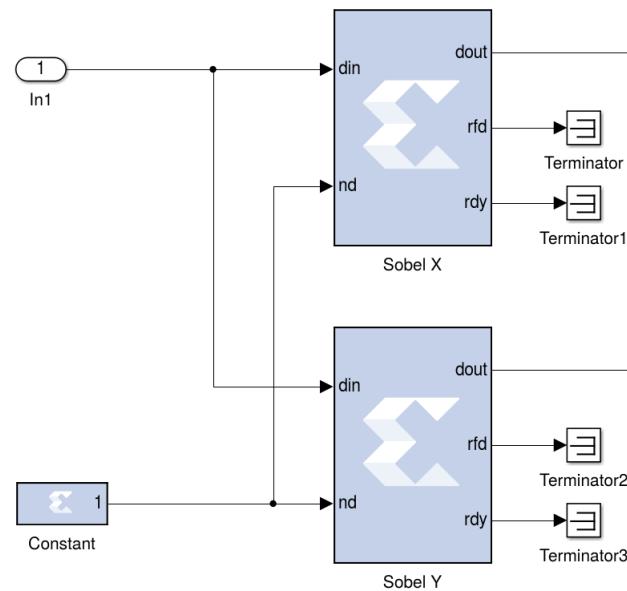


Figura 8.16: Conexión de los filtros FIR y sus terminales.

6. Continuando con las especificaciones técnicas, ahora es necesario implementar la Ecuación 8.2. Utilizar raíces cuadradas es muy costoso en términos de hardware, por esa razón se modelará dicha ecuación de una forma menos compleja pero matemáticamente equivalente.

En primer lugar, se deben sumar ambas salidas **dout** de los filtros FIR, lo que supone utilizar el bloque '*AddSub*', sin modificación alguna.

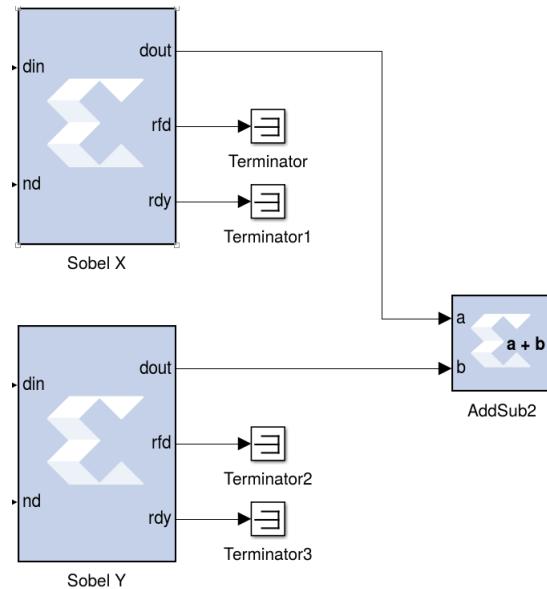


Figura 8.17: Adición de los valores calculados por ambos filtros FIR.

7. Para obtener al valor absoluto requerido por el algoritmo, basta con analizar el signo de los datos a la salida del bloque 'AddSub' mostrado en la Figura 8.17, e invertir el signo de todo aquel valor negativo, mediante el uso de un selector, en donde el puerto de **sel** sea el bit más significativo del dato (el cual es el signo), el puerto **d0** se active cuando el dato sea positivo (sin ningún cambio), pero el puerto **d1** por el contrario, cuando el dato sea negativo. A la entrada del puerto **d1** se agregará un bloque 'Negate' el cual se puede encontrar en **Xilinx Blockset > Math**. Todo esto se ilustra en la Figura 8.18.

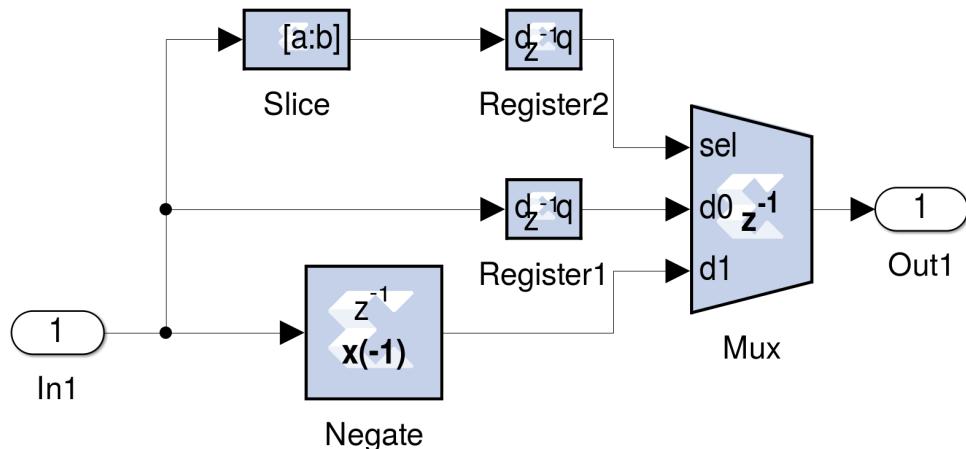


Figura 8.18: Modelo para obtener el valor absoluto de los datos arrojados por los filtros FIR.

8. El modelo completo del filtro Sobel se muestra a continuación.

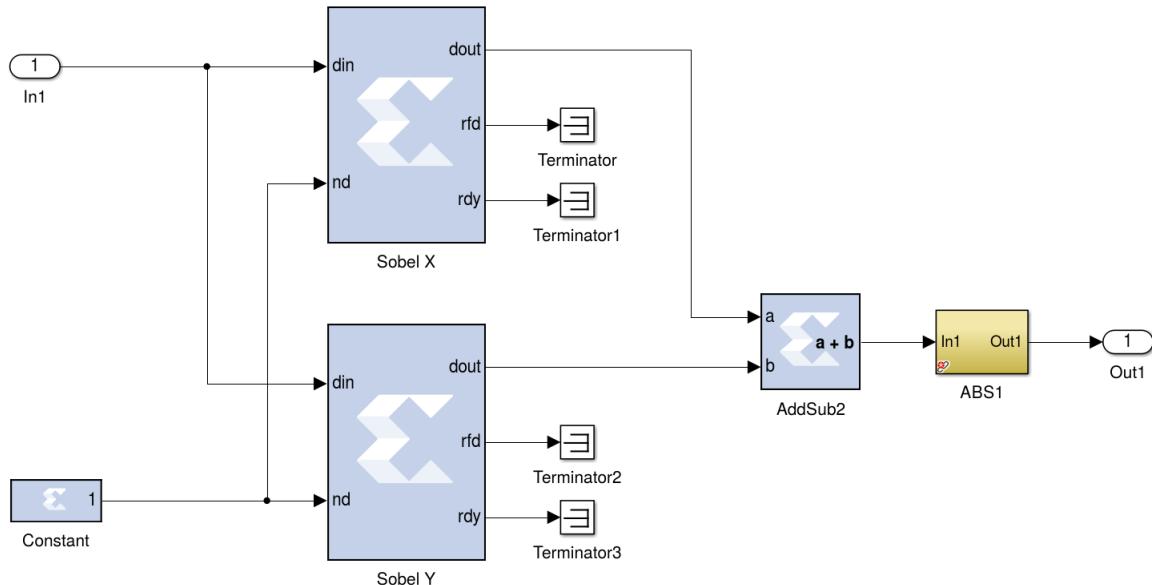


Figura 8.19: Modelo completo del filtro Sobel.

9. Este modelo se convirtió en subsistema, para conectarse como se muestra en la Figura . La conversión

a subsistema es opcional, sólo le da más legibilidad a modelos complejos con gran cantidad de bloques.

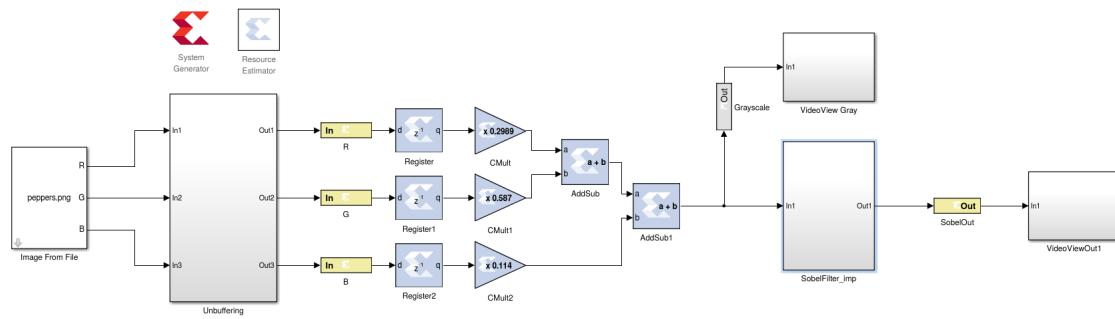


Figura 8.20:

## 8.6. Implementación y resultados.

- Al simular el modelo de Sysgen, el resultado se debe observar como en la Figura 6.3.



Figura 8.21: Resultados de la simulación del modelo del filtro Sobel.

- Para generar el modelo de Co-simulación, basta con abrir el 'Sysgen Token' y dar clic en 'Generate'.

Una nueva biblioteca se abrirá, la cual contiene el modelo compilado para poder ejecutarse en FPGA (véase Figura 8.22).

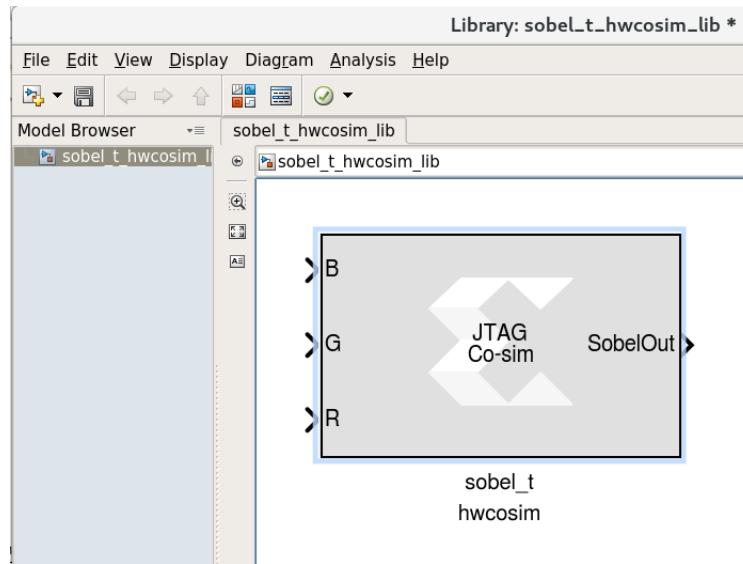


Figura 8.22: Bloque de Co-Simulación del filtro Sobel, que se ejecuta en FPGA.

3. El bloque generado en el Paso 2, se conecta de la siguiente manera:

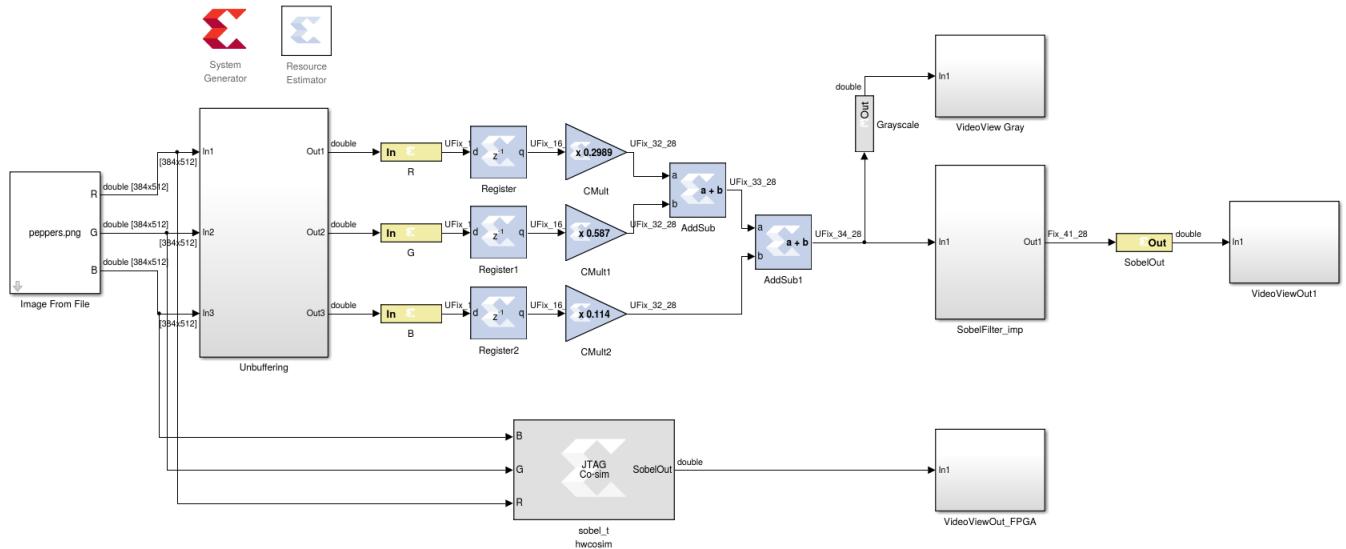


Figura 8.23: Conexión del modelo completo para FPGA, el cual ejecuta el resultado simulado y el obtenido por el hardware reconfigurable.

4. El resultado obtenido al ejecutar el modelo de la Figura 8.23 se ilustra a continuación.

TBD

Figura 8.24: Resultado de la ejecución en FPGA del modelo del filtro Sobel.

5. Las estadísticas de recursos utilizados y frecuencia máxima de trabajo de este prototipo, se muestran

en la Figura 8.25.

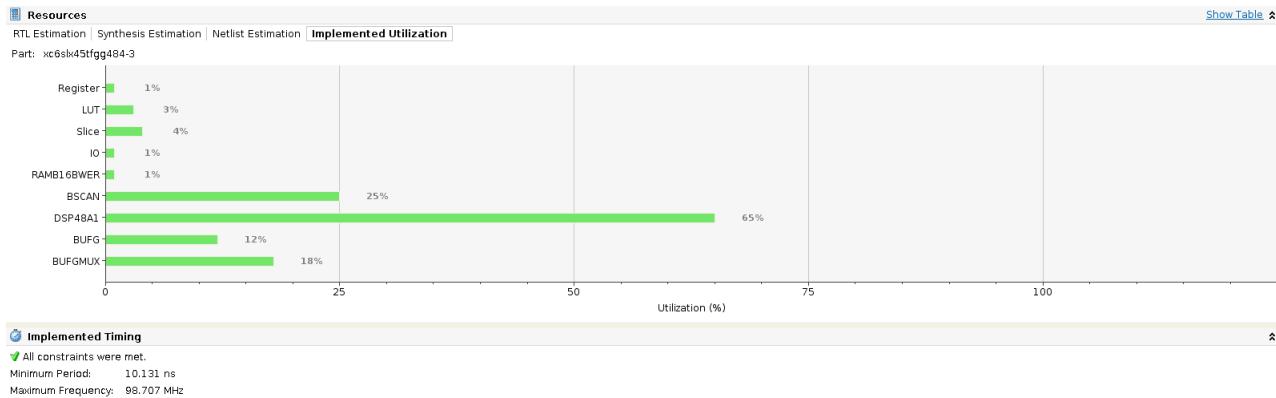


Figura 8.25: Gráfica de utilización de recursos en la FPGA al implementar el modelo del operador Sobel.

Como se puede observar, la mayoría de los recursos utilizados son DSP48A1, los cuales ejecutan las funciones matemáticas de los dos filtros FIR utilizados en este prototipo. Esto asegura que las operaciones con mayor flujo de datos se hacen utilizando primitivas especiales para esto, aumentando el paralelismo de una manera significativa, puesto que el modelo puede ejecutarse a frecuencias de hasta 98 MHz a diferencia de la mayoría de los DSP comerciales, que funcionan a frecuencias máximas de

## **Part IV**

### **Conclusiones.**

## Apéndice A: Conversión Analógico-Digital.

## Apéndice B: Conversión Digital-Analógico.

Para convertir una señal digital a una analógica después de que se ha procesado por el sistema DSP, se utiliza un conversor D/A (o DAC, por sus siglas en inglés). La manera en que este proceso se lleva a cabo es interpolando los datos de las señales entre las muestras tomadas, es decir, aplicando una aproximación sucesiva al valor de dichas muestras.

La manera más sencilla de convertir una señal de analógico a digital, es tomando las muestras de dicha señal desde la memoria donde el DSP las almacena, y transformarlas en un tren de impulsos, como se muestra en la **Figura .26**. Después, este mismo tren de pulsos, se hacer pasar por un filtro pasa bajas, con la frecuencia de corte igual a la mitad de la frecuencia de muestreo, cumpliendo el *Teorema de Nyquist*.

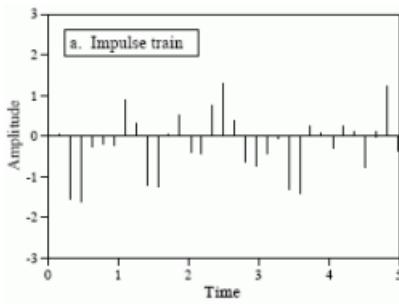


Figura .26: Información digital convertida en un tren de impulsos .[18]

En otras palabras, la señal original y el tren de impulsos tendrán espectros de frecuencia idénticos, lo cual cumple con la *frecuencia de Nyquist* anteriormente descrita. En frecuencias más altas, el tren de impulsos contiene una duplicación de la señal, mientras que la señal analógica original no contiene ninguna información, suponiendo que el *aliasing*<sup>13</sup> no ocurrió.

Mientras que este método es matemáticamente correcto, es difícil generar esos trenes de pulsos tan estrechos entre si, utilizando componentes electrónicos. Para poder manejar esta dificultad, la mayoría de los ADC operan manteniendo el último valor de entrada hasta que se recibe otra muestra proveniente del DSP. A esto se le conoce como *retención de orden cero*, el equivalente del proceso de *muestra y retención* del DAC. La *retención de orden cero* produce una señal con apariencia de escalera, como se muestra en la **Figura .27**<sup>14</sup>.

<sup>13</sup>Múltiples señales en tiempo continuo pueden producir series de muestras idénticas. A este fenómeno se le conoce como *Aliasing*. Cuando esto ocurre, el DAC no es capaz de regenerar la señal de salida a menos que haya un filtro *anti-aliasing* de por medio, muestreando a una frecuencia mayor a la actual[1].

<sup>14</sup>Información obtenida de [19].

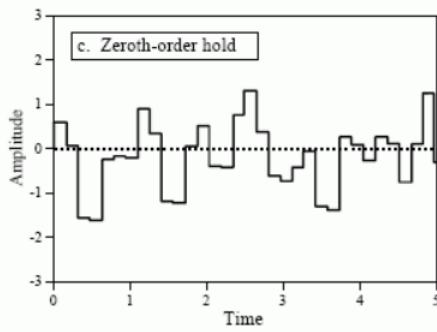


Figura .27: Representación gráfica de una señal producida por el efecto de *retención de orden cero* en el proceso de transformación A/D [19].

Hay diferentes formas de implementar estas conversiones D/A utilizando componentes electrónicos discretos y circuitos integrados, por ejemplo<sup>15</sup>

- **Ponderación binaria:** Este DAC basado en resistores en modo voltaje es usualmente la implementación más simple utilizada como referencia en los libros de texto (véase **Figura .28**). No es inherentemente monolítico y es muy difícil de fabricar de forma eficaz en grandes masas. Además, la salida del DAC utilizando el método de ponderación de voltaje binaria, cambia con el código de entrada.

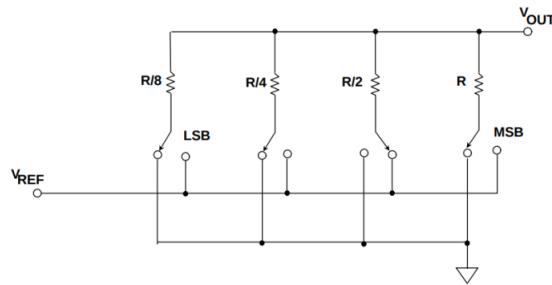


Figura .28: Estructura de un DAC resistivo de ponderación binaria. Imagen adaptada de: [17]

- **Ponderación binaria capacitiva en aproximación sucesiva:** El uso de una redistribución de carga capacitiva ofrece la ventaja de comportarse como un circuito de muestreo y retención (SHA, por sus siglas en inglés), por lo cual, ningún circuito SHA externo o incluso, alguna construcción monolítica SHA dentro del circuito integrado, es requerido al utilizar esta estructura (véase **Figura .29**).

<sup>15</sup>Para obtener más información sobre este tema, se puede referir a [10], de donde se extrajo información de esta sección.

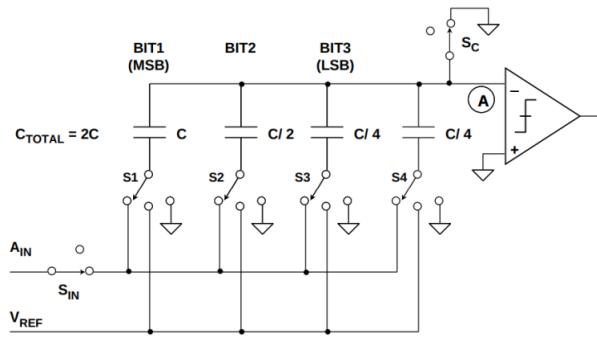


Figura .29: Estructura de un DAC capacitivo de ponderación binaria.

- **R-2R:** Una de las estructuras más comunes de DACs es la muy conocida escalera R-2R, la cual consiste en una red de resistores de sólo dos diferentes valores, en una proporción de 2:1. Un DAC de N bits requiere  $2N$  resistores. El voltaje de salida se mantiene siempre con la misma impedancia, (véase **Figura .30**).

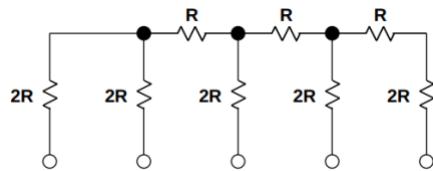


Figura .30: DAC R-2R red en escalera.

El voltaje salida de un DAC ideal se muestra en la **Figura .31**.

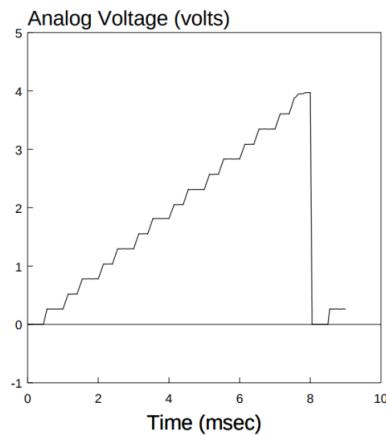


Figura .31: Modelo en NGSpice de un DAC ideal de 4 bits.

# Apéndice C: Script en Matlab de la Convolución de dos vectores de forma interactiva.

**Algoritmo 3** Ilustración interactiva de la convolución, usada en la subsección 3.2.1 Convolución.

```

1 clear all; close all;
2 t = [-4 -3 -2 -1 0 1 2 3 4];
3 x = [ 0 0 0 0 1 2 3 0 0];
4 h = [ 0 0 0 0 2 1 0 0 0];
5 y = [ 0 0 0 0 0 0 0 0 0];
6 yc = 1;
7 for n=min(t):max(t),
8     pause(3);
9     % flip h
10    ht = flipud(h);
11    if n<0,
12        % shift to the left
13        ht = [ht(-n+1:length(ht)) zeros(1,-n)];
14    else
15        % shift to the right
16        ht = [zeros(1,n) ht(1:length(ht)-n)];
17    end
18
19    y(yc) = sum(x.*ht);
20    yc = yc + 1;
21
22    subplot(2,1,1);
23    stem(t,x);
24    hold on;
25    stem(t,ht, 'filled', 'r');
26    hold off;
27    xlabel('t');
28    legend('x[n]', 'h[n-k]', 0);
29    title(['n= ' num2str(n)]);
30
31    subplot(2,1,2);
32    stem(t,y);
33    xlabel('t'); ylabel('y[n]');
34 end

```

# Apéndice D: Creación de proyectos usando PlanAhead.

1. Abrir PlanAhead y seleccionar '*Create New Project*'.
2. Esto invocara al '*Project wizard*', en la primer ventana de bienvenida, dar clic en '*Next >*'.
3. En la ventana '*Project name*', seleccionar el nombre del proyecto y la ruta donde se guardará, estos parámetros son a gusto del usuario.
4. En '*Project Type*', seleccionar '*RTL Project*' y marcar la casilla '*Do not specify sources at this time*'.
5. Se abrirá una nueva ventana llamada '*Default Part*', donde se debe especificar la matrícula de la FPGA, en el caso del kit Atlys es **XC6SLX45CSG324-3**. Se puede poner esta matrícula en el apartado '*Search*' para que sea más sencillo y libre de errores dado que, si se selecciona cualquier otra parte distinta, el proyecto no funcionara en el kit.
6. Dar clic en '*Finish*', el proyecto está configurado en este punto.
7. En la columna '*Project Manager*', seleccionar '*Add sources*' y después '*Add or Create DSP Sources*' como se muestra a continuación:



Figura .32: Selección del tipo de proyecto en PlanAhead.

8. Seleccionar la opción '*Create Sub-Design*' y definir el nombre del módulo principal, por ejemplo **lab1\_sysgen**.
9. La interfáz de Matlab se desplegará en la pantalla, con el toolbox de Sysgen añadido a las librerías de Simulink. En este paso hay que esperar un par de minutos hasta que aparezca el workspace principal de Simulink, como se muestra en la Figura .33.

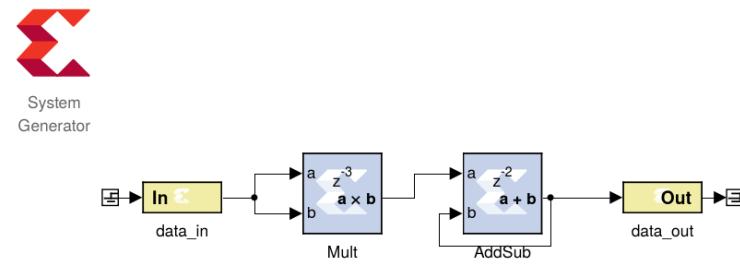


Figura .33: Modelo de ejemplo en Sysgen.

## **Apéndice E: Instalación del plugin de Sysgen para el kit Atlys.**

# Apéndice F: Código Verilog completo del módulo de procesamiento de audio con el códec AC97.

```

1  `define NO_LOOPBACK
2  module rt_audio_controller( input      clk,
3                               input      n_reset,
4                               input      sdata_in,
5                               input      bitclk,
6                               input [4:0] volume,
7                               input [2:0] source,
8                               output     sync,
9                               output     codec_n_reset,
10                              output     sdata_out );
11
12  wire      ac97_ready_sig_w;
13  wire [7:0] cmd_addr_w;
14  wire [15:0] cmd_data_w;
15  wire      latching_cmd_w;
16  wire      slot_status_signal;
17 `ifdef NO_LOOPBACK
18  wire [17:0] left_audoin;
19  wire [17:0] right_audoin;
20  wire [17:0] left_audioout;
21  wire [17:0] right_audioout;
22 `else
23  wire [17:0] left_loopback;
24  wire [17:0] right_loopback;
25 `endif
26
27 /* FIX1: Puesto que los dos archivos NCG utilizan un IBUF para el puerto CLK, es necesario forzar a la herramienta a
28    eliminarlos y utilizar un IBUG (global) en su lugar, de otra forma el proceso de MAP fallara con un enrutamiento
29    ilegal entre clock buffers, resultado esperado:
30    [Opt 31-35] Removing redundant IBUF, controller/clk_IBUF, from the path connected to top-level port: clk
31    [Opt 31-35] Removing redundant IBUF, datapath/clk_IBUF, from the path connected to top-level port: clk
32 */
33 IBUG FIX1 (.I(clk), .O(gclk));
34
35 ac97cmd controller( .clk(gclk),
36                      .ac97_ready_sig(ac97_ready_sig_w),
37                      .cmd_addr(cmd_addr_w),
38                      .cmd_data(cmd_data_w),
39                      .latching_cmd(latching_cmd_w),
40                      .volume(volume),
41                      .source(source) );
42
43 ac97 datapath( .n_reset(n_reset),

```

```

44      .clk(gclk),
45      .ac97_sdata_out(sdata_out),
46      .ac97_sdata_in(sdata_in),
47      .ac97_sync(sync),
48      .ac97_bitclk(bitclk),
49      .ac97_n_reset(codec_n_reset),
50      .ac97_ready_sig(slot_status_signal),
51      .latching_cmd(latching_cmd_w),
52      .cmd_addr(cmd_addr_w),
53      .cmd_data(cmd_data_w),
54 `ifdef NO_LOOPBACK
55      .L_out(left_audioin),
56      .R_out(right_audioin),
57      .L_in(left_audioout),
58      .R_in(right_audioout)
59 `else
60      .L_out(left_loopback),
61      .R_out(right_loopback),
62      .L_in(left_loopback),
63      .R_in(right_loopback)
64 `endif      );
65
66 `ifdef NO_LOOPBACK
67   algodev_cw algorithm_from_sysgen( .clk(gclk),
68      .audioleftin(left_audioout),
69      .audiorightin(right_audioout),
70      .audioleftout(left_audioin),
71      .audiorightout(right_audioin),
72      .enable(slot_status_signal) ); // synthesis black_box
73 `endif
74   assign ac97_ready_sig_w = slot_status_signal;
75 /* assign source = 3'b000; Ejemplo para forzar una fuente de audio, en este caso el microfono */
76 endmodule // rt_audio_controller

```

## Bibliografía

- [1] *Sampling and Aliasing*, number 8 in AD/DSP Lecture 8, May 2002.
- [2] *Spartan-6 FPGA Configurable Logic Block User Guide*, February 2010.
- [3] Analog Devices. A Beginner's Guide to Digital Signal Processing (DSP), May 2015.
- [4] James Barnes et al. *IIR Filter Representations – Difference Equations*,  $h[n]$ ,  $H(z)$ , chapter 1, page 21.  
Number Lecture 7 - IIR Filters in ECE423. 1 edition, March 2014.
- [5] Ali Bashashati. *EECE 359*, 1 edition, October 2009.
- [6] Digilent et al. Atlys FPGA Board Reference Manual. Technical Report Atlys FPGA Board Reference Manual, April 2016.
- [7] Piraj Fozoonmayeh. *A practical approach to DSP algorithms using FPGA devices*. PhD thesis, Simon Fraser University, April 2011.
- [8] Michael Francis et al. Infinite Impulse Response Filter Structures in Xilinx FPGAs. Technical Report 330, August 2009.
- [9] Nasser Kehtarnavaz and Sidharth Mahotra. *Digital Signal Processing Laboratory: LabVIEW-Based FPGA Implementation*. Universal-Publishers, 2010. ISBN 978-1-59942-550-4.
- [10] Walt Kester. *Basic DAC Architectures II: Binary DACs*. Analog Devices, rev a edition, October 2008.
- [11] Shashikala Narasimha Murthy. *Implementation of unmanned vehicle control on FPGA based platform using System Generator*. PhD thesis, University of South Florida, October 2007.
- [12] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Filter Design Techniques*. Signal processing - Mathematics. Prentice-Hall, 2nd edition, October 1999. ISBN 0-13-754920-2.
- [13] Sophocles J. Orfanidis. *Introducción al procesamiento de Señales*. Pearson Education, 1 edition, March 2009. ISBN 0-13-209172-0.
- [14] Xilinx Inc. Peter Alfke. *Xilinx Virtex-6 and Spartan-6 FPGA Families*, Aug 2009.
- [15] Prof. Jan Van der Spiegel. Introduction to Xilinx ISE 8.2i. volume I, pages 1–5, May 2015.
- [16] D. Schlichttharle. *Digital Filters, Basics and Design*. Springer, 2nd edition, August 2000. ISBN 978-3-642-14324-3.

- [17] B. D. Smith. Coding by Feedback Methods. Technical report, August 1993.
- [18] Steven W. Smith. The Scientist and Engineer's Guide to Digital Signal Processing, May 2014.
- [19] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing, chapter 3.* 2nd edition, May 2014. ISBN The Scientist and Engineer's Guide to Digital Signal Processing.
- [20] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* 2nd edition, May 2014. ISBN The Scientist and Engineer's Guide to Digital Signal Processing.
- [21] Evgeni Stavinov. *100 Power Tips for FPGA Designers.* I edition, April 2011. ISBN 978-1-4507-7598-4.
- [22] Arvind Sundararajan and Sanjay Churiwala. *SysGen for DSP, chapter VIII - SysGen for DSP,* page 257. I edition, October 2017. ISBN 978-3-319-42438-5.
- [23] Xilinx Technical Documentation. Spartan-6 Family Overview. Technical Report DS160 (v2.0), October 2011.
- [24] The Mathworks Company. MATLAB Getting Started Guide. Technical report, October 2008.
- [25] Xilinx. *Hardware Design Using System Generator.* Xilinx, inc., October 2012.