
Universidad Nacional del Sur

Tesis de Maestría en Ingeniería

# Técnicas de verificación orientadas a sistemas digitales de procesamiento de señales

Ing. Gabriel H. Pachiana Caba

Bahía Blanca Argentina

2015

Impreso en Argentina.
ISBN - - - -
Abril de 2015

# Prefacio

Esta tesis se presenta como parte de los requisitos para optar al grado académico de Magister en Ingeniería de la Universidad Nacional del Sur y no ha sido presentada previamente para la obtención de otro título en esta Universidad u otra. La misma contiene los resultados obtenidos en investigaciones realizadas en el ámbito del Departamento de Ingeniería Eléctrica y de Computadoras en el período comprendido entre Noviembre del 2012 y Noviembre del 2014, bajo la dirección del Ing. Eduardo E. Paolini y del Dr. Agustín Rodríguez.

Bahía Blanca, 30 de Noviembre de 2015.

Gabriel H. Pachiana Caba
Departamento de Ingeniería Eléctrica y de Computadoras
Universidad Nacional del Sur

Universidad Nacional del Sur
Secretaría General de Posgrado y Educación Continua

La presente tesis ha sido aprobada el ..../..../..... , mereciendo la calificación de .....(.....)

# Resumen

El objetivo de este trabajo es el estudio y el desarrollo de tecnologías de verificación funcional de sistemas digitales especificados a nivel de transferencia entre registros (RTL por sus siglas en inglés, *Register Transfer Level*). Más precisamente, se experimenta con metodologías y herramientas de verificación orientadas a núcleos (*cores*) o bloques aritméticos y de procesamiento digital de señales (PDS).

Se describen conocimientos generales de verificación de sistemas digitales de muy gran escala de integración (VLSI, *Very Large Scale Integration*) de manera de comprender los principales problemas en este área.

Se describen los conceptos teóricos relacionados a la verificación funcional de hardware y la problemática específica de PDS.

Se definen y experimentan los aspectos formales y prácticos de las técnicas de verificación funcional orientadas a unidades de cálculo aritmético y de procesamiento de señales, a través de su aplicación en bloques de distintas complejidades como pueden ser los filtros con respuesta infinita al impulso (IIR, *Infinite Impulse Response*) o transformada rápida de Fourier (FFT, *Fast Fourier Transform*).

# Abstract

The objective of this work is the study and development of functional verification technologies for digital systems at register transfer level (RTL). Also, to experience with verification methodologies and tools oriented to arithmetic and digital signal processing (DSP) cores.

General knowledge of verification for VLSI (Very Large Scale Integration) systems is described to understand the main problems in this area.

Theoretical concepts related to hardware functional verification and the relation with PDS specific systems are addressed.

Functional verification concepts are applied to the definition of formal and practical implementations to address the functional verification of arithmetic units and DSP cores, for example infinite impulse response (IIR) digital filters and fast Fourier transform (FFT) cores.

# Agradecimientos

En primer lugar, quisiera agradecer a mi familia y amigos, soporte y motivación principal en toda mi vida. En especial a mi madre Susana, y mi compañera de vida Mercedes.

Quisiera agradecer a quienes han dirigido este trabajo, Eduardo y Agustín. Que desinteresadamente me han enseñado la teoría, pero más importante aún, han sabido transmitir motivación y calma en los momentos necesarios. Al igual que mis compañeros de oficina, Manuel Soto y Juan Francesconi, amigos con los que he convivido día a día.

A las personas e instituciones que hicieron posible que se realice este trabajo y me han abierto las puertas al mundo de la microelectrónica: los hermanos Soto, Pedro Julián y Pablo Mandolesi, el proyecto proyecto FSTICS 001 TEAC (Plataforma Tecnológica para Sistemas de Tecnología Electrónica de Alta Complejidad), mi entidad apadrinadora, INTI (Instituto Nacional de Tecnología Industrial), el Instituto de Investigaciones en Ingeniería Eléctrica "Alfredo Desages" (IIIE) y el LaPSyC (Laboratorio de Procesamiento de Señales y Comunicaciones).

# Índice

# Capítulo 1

# Introducción

## 1.1. Problemática general de verificación VLSI

Hoy, en la era de los diseños de circuitos integrados de aplicación específica (en inglés, ASIC *Application-Specific Integrated Circuit*) de millones de compuertas, los bloques de propiedad intelectual (del inglés *Intellectual Property*, IP) reutilizables y los sistemas en un chip (SoC, System-on-Chip), se pueden generar sistemas digitales con un gran conjunto de funcionalidades y, debido al crecimiento exponencial de las capacidades de diseño e integración, se podrá continuar agregando más funciones de mayor complejidad.

Este crecimiento de las facilidades de diseño provisto por las herramientas de EDA (*Electronic Design Automation*) y la capacidad de integrar mayor cantidad de transistores en menor área, ha expuesto una necesidad no contemplada, y que por ende no ha acompañado el crecimiento de las anteriores, de demostrar que la intención de diseño se corresponde con la implementación, es decir, la **verificación de sistemas digitales**.

Para poder comprobar esa correspondencia se puede simular el comportamiento del sistema para todas las posibles entradas, pero aún para pequeños sistemas esto se vuelve impracticable. Por ejemplo en un sistema con una entrada de 16 bits se podrian generar $2^{16} = 65536$ valores diferentes de la señal de entrada. Si cada uno de estos valores tarda 1 segundo en ser simulado, entonces simular todos los casos de prueba demandaría poco más de 18 horas. Si la entrada es de 32 bits, la cantidad de vectores crece a $2^{32}$ lo que implica aproximadamente 49710 días y 6 horas de simulación: el *espacio de prueba* crece de forma exponencial en función de la entrada. En consecuencia, los ingenieros en verificación necesitan una heurística para simular los casos de prueba que generen discrepancias, es decir generar la menor cantidad de pruebas posibles sin comprometer los objetivos de la verificación.

Los avances en esta disciplina han resultado en sofisticadas herramientas y técnicas que

ayudan a los ingenieros a verificar sus complejos diseños. Aún así, quedan pendientes preguntas básicas de la temática, tales como ¿cómo asegurar que se generan señales de prueba que sean útiles para detectar fallas?, o, en otras palabras, ¿cómo se genera el *modelo de cobertura*?

El impacto de un proceso de verificación incompleto o erróneo implica una posible implementación con errores que no han sido advertidos antes de fabricación, implicando costos monetarios y temporales relacionados con la re-ingeniería del proceso, o la pérdida de una oportunidad de negocio por no finalizar a tiempo un producto comercial. Y en el peor de los casos, implica pérdidas fatales cuando se desarrollan sistemas críticos (por ejemplo, sistemas médicos o aeronáuticos).

En consecuencia, la etapa de verificación se ha vuelto clave en el éxito de un proyecto de desarrollo de sistemas digitales. Un estudio sobre la utilización de verificación funcional en la industria de desarrollo de hardware [1] indicó que el porcentaje medio de tiempo empleado en verificación en proyectos VLSI con respecto al tiempo total de desarrollo fue del 49 % en el año 2007, mientras en el año 2012 esta medida creció al 56 %. Así mismo, el número de ingenieros empleados en la verificación ha crecido un 75 % en el mismo período, logrando una relación de 1 a 1 en comparación con la cantidad de ingenieros empleados para diseño. Con respecto a las principales causas de errores que implican una re-ingeniería posterior al proceso de fabricación, la misma fuente indica que el 50 % de las veces se debe a fallas funcionales.

## 1.2. La tecnología de verificación y los sistemas de PDS

La tarea de verificación funcional de sistemas de procesamiento digital de señales (PDS), como pueden ser sistemas de procesamiento de audio, video, comunicaciones, aeroespacial, está estrechamente vinculada con el cumplimiento de restricciones en cuanto a precisión, desempeño y otras cuestiones aritméticas tales como resultados fuera de rango (desborde positivo u *overflow* y desborde negativo o *underflow*).

Comúnmente la metodología de diseño de un bloque de PDS, como puede ser una transformada rápida de Fourier (FFT, *Fast Fourier Transform*), filtros digitales (IIR o FIR), CODECs de audio o video, implica una primera etapa de diseño asistido por herramientas matemáticas de alto nivel como Matlab o de programación como C/C++. En esta etapa la representación numérica es aproximada a la aritmética real, con aritmética computacional de punto flotante (de 32 o 64 bits [2]). En la etapa siguiente, generalmente, se opta por utilizar una representación numérica de punto fijo debido a que permite una reducción de área, retardos y consumo de energía en comparación con una implementación en punto flotante. Pero estas mejoras tienen como desventaja la reducción en la precisión y la posibilidad de generar efectos indeseables como resultados fuera de rango e inestabilidades. Estos efectos son los desafíos con los que se enfrenta la verificación de sistemas de PDS [3], y de sistemas de control en general.

Tanto los tipos de datos de punto flotante como los de punto fijo sufren de rango limitado, el cual puede tener consecuencias impensadas como ocurrió durante el despegue del cohete Ariane 5 [4]. En ese sistema se producía una conversión de números de punto flotante de 64 bits a enteros signados en punto fijo de 16 bits, la cual produjo un resultado no representable por la aritmética destino. Esta cadena de errores finalizó en la explosión de la aeronave a 39 segundos de haber sido lanzada, produciendo pérdidas monetarias de 370 millones de dólares.

Otra desventaja es que su precisión es limitada, es decir que producen resultados inexactos. Más aún, los errores se magnifican al continuar operando sobre resultados inexactos. Este fenómeno fue el origen de la falla del sistema militar de defensa "Patriot" cuyo error acumulado de redondeo en el cálculo de tiempo generó que no pueda lograr su cometido de interceptar un misil adversario, resultando en 28 pérdidas fatales [5].

Por ende, la verificación de sistemas de PDS y aritméticos es de gran importancia, más aún en sistemas de seguridad críticos los cuales requieran certificación.

## 1.3. Objetivos

En primer lugar, el objetivo de esta tesis es brindar conocimientos generales de verificación de sistemas digitales VLSI de manera de comprender los principales problemas en este área. Además, experimentar con las herramientas y metodologías de verificación de índole académica y comercial.

En segundo lugar, exponer experimentación significativa en los aspectos formales y prácticos de las técnicas de verificación orientadas a unidades de cálculo aritmético y de procesamiento de señales, a través de su aplicación en bloques de distintas complejidades como pueden ser los filtros digitales o una transformada rápida de Fourier.

Por último, demostrar el desarrollo de módulos experimentales para verificación de bloques de procesamiento digital de señales.

## 1.4. Estructura de la tesis

El capítulo siguiente brinda los conceptos específicos que serán utilizados en el desarrollo del trabajo. A continuación se exponen los casos de estudio abordados y las abstracciones definidas que son de vital importancia para la verificación, es decir, los modelos de cobertura.

Los casos de estudio se verifican mediante la implementación e inclusión de los modelos de cobertura en infraestructuras de prueba (o bancos de pruebas, en inglés *testbenchs*) descritos en el Capítulo 4. Los resultados obtenidos mediante dichas infraestructuras se exponen en el Capítulo 5.

Por último, las conclusiones y posibles vías de continuación de este trabajo se resumen en el Capítulo 6

## 1.5. Contribución

Dos de los casos de estudio abordados en este trabajo han sido publicados en la octava edición de la *Conferencia Argentina de Micro-Nanoelectrónica, Tecnología y Aplicaciones (CAMTA)*, referenciados por la asociación *IEEE*.

El trabajo [6] presenta la verificación funcional de FFT, donde se define un modelo de cobertura, se implementa un ambiente de verificación para generar las señales que cubren el mismo, y se realiza la tarea para distintas FFTs. En forma similar, en [7] se propone un modelo de cobertura para unidades aritméticas de punto flotante y un ambiente de verificación para llevar a cabo la tarea.

# Capítulo 2

# Marco teórico

## 2.1. Introducción

La verificación funcional comprende el proceso de garantizar la correcta correspondencia entre la especificación de un diseño y su implementación [8]. En este contexto, los errores son discrepancias entre el comportamiento previsto del sistema y el comportamiento observado.

Los errores de diseño se deben a especificaciones incompletas o ambiguas, o a la incorrecta interpretación de las especificaciones por parte de los diseñadores, o a una erronea ejecución en las tareas de diseño. Los errores de implementación están frecuentemente relacionados con errores de codificación.

## 2.2. Verificación funcional

La verificación funcional se puede llevar a cabo con métodos estáticos utilizando herramientas formales usando métodos dinámicos por medio de simulaciones, o empleando métodos híbridos [9]. A continuación se describen brevemente cada uno de ellos.

### 2.2.1. Verificación formal

Los métodos formales, también conocidos como métodos estáticos, se emplean para demostrar la correctitud de la implementación basandose en técnicas computacionales/matemáticas tales como verificación de modelos (*Model Checking*), demostración de teoremas (*Theorem Proving*), verificación de equivalencias (*Equivalence Checking*) u otras técnicas analíticas.

*Model Checking* demuestra que nunca se viola una propiedad definida por el usuario para todas las posibles secuencias de entradas. *Theorem Proving* encuentra si un teorema puede ser

probado o no, con la asistencia de un programa demostrador de teoremas. *Equivalence Checking* compara dos modelos para determinar si son lógicamente equivalentes.

Los métodos analíticos utilizan propiedades matemáticas para demostrar propiedades aritméticas. Son herramientas que asisten en la verificación y demostración formal de propiedades aritméticas sobre programas numéricos o hardware que utiliza aritmética computacional. Un ejemplo es la herramienta Gappa (Generación Automática de Pruebas de Propiedades Aritméticas) [11]. Esta herramienta manipula fórmulas lógicas que indican las cotas de expresiones en algunos intervalos, por ejemplo si un sistema que realiza la suma de dos entradas $a$ y $b$ las cuales pueden tomar valores entre 0 y 255 puede generar una prueba formal de la siguiente propiedad:

$$(a \in [0; 255] \wedge b \in [0; 255]) \Rightarrow a + b \in [0; 510].$$

Gappa extiende el paradigma de aritmética de intervalos al campo de la certificación de código numérico. Dada la descripción de una propiedad lógica que involucra cotas de las expresiones matemáticas, la herramienta trata de probar la validez de esta propiedad. Cuando la propiedad contiene expresiones sin acotar, la herramienta computa la cota para la cual la propiedad se sostiene. Gappa está diseñado para manipular fórmulas presentes en la certificación de códigos numéricos. En particular, permite el conocimiento de la cota de error computacional debido a aritmética de punto flotante o fijo [10] [11].

Los métodos formales son robustos, pero la complejidad exponencial de estos algoritmos (o los modelos representados en el lenguaje específico) limita su aplicación a pequeños fragmentos o a sistemas específicos.

### 2.2.2. Verificación funcional basada en simulación

Alternativamente, los métodos basados en simulación pueden ser aplicados sin importar la complejidad del diseño y su correspondiente espacio de verificación. De todas maneras, no proveen una prueba sobre la correctitud de la implementación. Aunque son efectivas para probar la presencia de errores, no es posible asegurar su ausencia a menos que se cubra todo el espacio combinatorio de pruebas. Estimular el diseño bajo prueba (DUV, por sus siglas en inglés *Design Under Verification*) con cada posible vector de entrada no es práctico debido a que el número de vectores de prueba puede volverse inmanejable en tiempo de simulación.

Por lo tanto, los ingenieros en verificación deben seleccionar una muestra representativa del espacio de prueba, probar el correcto funcionamiento mediante su simulación y esperar que esta muestra sea el mejor esfuerzo capaz de exponer cualquier error en el diseño. Aún con estas dificultades, los métodos de verificación mediante simulación son el medio principal de la ingeniería en verificación.

Esta muestra del espacio de prueba puede ser construida en forma manual (pruebas dirigidas) o en forma automática. La construcción manual presenta limitaciones como la gran demanda de tiempo para la escritura de las pruebas o una baja efectividad para encontrar errores no previstos, por lo que sólo se la utiliza para verificar pequeños diseños o pruebas específicas.

La construcción automática se sustenta en un modelo que permite definir los datos de entrada durante el tiempo de simulación. Una de las técnicas automáticas más simples es la verificación aleatoria, en la que el conjunto de estímulos se genera al azar. En este contexto, el avance de la verificación se medirá en cantidad de casos de prueba del espacio total que se han simulado. La desventaja es la probabilidad de que se genere gran cantidad de casos de prueba equivalentes (que estimulen la misma lógica) o no se generen casos que alcancen los límites de la aritmética o la lógica, problema conocido como casos límite o casos de borde (en inglés *corner cases*).

En un enfoque más eficiente se aplican restricciones a la aleatoriedad (denominado en inglés como *Constrained Randomization*) para centrar las pruebas en las posibles áreas de falla y reducir la cantidad casos de simulación.

Otro enfoque, más sofisticado aún, utiliza un modelo de cobertura del espacio de prueba que guiará la elección de los casos de prueba. Este heurística se denomina verificación guiada por la cobertura (CDV, por sus siglas en inglés *Coverage Driven Verification*), la cual es el medio elegido para realizar la verificación de los casos de prueba de este trabajo.

### 2.2.3. Visibilidad

Otra diferencia relevante entre métodos formales y simulaciones está relacionada con la visibilidad del diseño. Mientras que la verificación formal requiere accesibilidad a detalles internos de la implementación, conocido como "prueba de caja blanca" en el campo de la ingeniería en software, la verificación basada en simulación puede ser empleada aún cuando el código de la implementación no esté disponible, verificando la funcionalidad únicamente mediante el análisis de la correctitud de las salidas, también conocido como "prueba de caja negra" [12].

## 2.3. Conceptos de verificación funcional

### 2.3.1. Cobertura

Siempre que se utilice verificación funcional basada en simulación, el proceso necesita una métrica para evaluar la calidad de una colección de casos de pruebas (conocida como *test suite*). Esta medida de calidad, denominada **cobertura**, ayuda a evaluar el progreso del proceso de verificación indicando tanto el tipo como la cantidad de vectores de prueba que son suficientes para alcanzar un determinado valor de calidad.

Existen dos tipos de cobertura. La primera, denominada cobertura estructural, indica qué porciones del diseño han sido ejercitados o no por el banco de pruebas (conocido en inglés como *testbench*). Las métricas estructurales más conocidas son cobertura de líneas (*line-coverage*), cobertura de ramas (*branch-coverage*) y cobertura de cambios (*toggle-coverage*).

Cobertura de líneas identifica las líneas del código fuente que han sido ejecutadas en una simulación, cobertura de ramas indica los caminos o ramificaciones que han sido ejercitados y cobertura de cambios acusa los valores que han tomado o no las variables. Este tipo de métricas de cobertura pueden ser evaluadas dentro de la mayoría de las herramientas de simulación.

El segundo tipo es cobertura funcional, que indica qué funcionalidades han sido ejercitadas y de qué manera. La definición de tal modelo requiere una identificación de la funcionalidad del diseño, modos de operación y configuración de entrada/salida, los cuales pueden ser inferidos directamente de las especificaciones [8].

Existen dos posibles formas de utilización del modelo de cobertura. La forma clásica es de métrica de cubrimiento de un conjunto de pruebas, es decir, se simula una cantidad de estímulos y se mide cuánto se ha cubierto del modelo. La segunda es de fuente de generación de las pruebas. Al conocer el modelo de cobertura se pueden crear directamente aquellos estímulos que cubran el modelo. Cuando la generación de los casos de prueba está "guiado" por el modelo de cobertura, la verificación funcional se denomina *CDV* (del inglés *Coverage Driven Verification*).

Utilizar el modelo de cobertura para medición y generación de pruebas es redundante ya que se mide algo que ya se sabe que cubre.

### 2.3.2. Generación de modelos de cobertura

Hay diversas formas de crear un modelo de cobertura. En [13] y [14], se particiona el espacio de cobertura en forma automática basándose en la definición formal de atributos y sus relaciones. El objetivo es la identificación del espacio de cobertura y la extracción de los casos de prueba límite. Alternativamente en [15] se define el conjunto de posibles escenarios, donde cada escenario es modelado como una función booleana sobre los parámetros del diseño. En una primera aproximación se usa el modelo como métrica para indicar la completitud de la verificación; en otra implementación más sofisticada, se crean los estímulos necesarios para estimular cada escenario en tiempo de simulación y por último, el usuario puede intervenir en el proceso de selección para enfatizar o relajar el estímulo sobre ciertos escenarios.

Por otro lado, una estrategia común para lidiar con la complejidad del espacio de cobertura y así poder estimular el DUV con un número razonable de vectores de prueba, es dividir el espacio en regiones o subconjuntos donde todos los vectores incluidos en un subconjunto son equivalentes desde la perspectiva de verificación. De esta manera, al estimular el diseño con un

vector de cada subconjunto permite una reducción en el tamaño del conjunto de pruebas sin comprometer la calidad del proceso de verificación. Esta técnica es denominada "particiones de equivalencia" en la literatura de verificación de software [12]. En forma complementaria, se deben analizar los casos límite de cada partición (*corner cases*) debido a que la probabilidad de encontrar un error es mayor en estas regiones. Esta técnica complementaria se denomina análisis de valores límite.

### 2.3.3. Particiones de equivalencia y valores límite

El método de particiones de equivalencia involucra los siguientes pasos:

1. Identificar el espacio de posibles entradas.
2. Dividir el espacio en particiones con el criterio definido por las especificaciones y funcionalidad del DUV.
3. Identificar clases válidas e inválidas.
4. Seleccionar muestras representativas de las clases válidas.
5. Estimular el DUV y verificar los resultados.

**Ejemplo 1.**

Para un DUV que implementa la operación de saturación: si la entrada entera de 16 bits es menor a $564_{(10)}$ entonces la salida es igual a la entrada, de lo contrario la salida es igual a $564_{(10)}$. Es decir,

$$salida = \begin{cases} entrada, & \text{si } entrada < 564_{10} \\ 564_{10}, & \text{si } entrada \geq 564_{10} \end{cases}$$

Sea $C_s$ el espacio de cobertura definido por todos los números de 16 bits posibles, entonces $C_s = \{0, \ldots, 65535\}$. Se espera que el DUV ejercite la misma lógica cuando la entrada es menor a $564_{(10)}$ y otra cuando es mayor o igual a $564_{(10)}$. Siguiendo esta suposición, el modelo de cobertura se divide en dos clases, $C_1 = \{0, \ldots, 563\}$ y $C_2 = \{564, \ldots, 65535\}$, y el modo de cobertura $C_m$ se define como un conjunto con dos elementos, $C_m = \{C_1, C_2\}$.

Un posible conjunto de muestras que cubre completamente el modelo es

$$S_{PE} = \left\{24_{(10)}, 1524_{(10)}\right\},$$

donde el primer valor representa a la clase de los enteros de 16 bits menores a 564 y el segundo a la de mayor o igual a 564. En este caso, el método de las particiones de equivalencia redujo el número de 65535 posibles vectores de prueba a 2, cubriendo completamente el modelo. ■

Aunque el método de particiones de equivalencia es un método muy eficiente para reducir el número de vectores de prueba, éste presenta algunas deficiencias: por ejemplo, en un DUV aritmético, los límites superiores pueden causar una condición de "fuera de rango" (conocido como desborde u *overflow*) indeseada. En el Ejemplo 1, un error de implementación al cambiar en la condición de saturación "mayor o igual" por "mayor" hubiese dejado un caso indefinido cuando la entrada es igual a 564, y esta particularidad nunca hubiese sido advertida por los dos vectores de prueba seleccionados previamente por particiones de equivalencia. En este caso, los valores límite o *corner cases* se definen como $S_{CornerCases} = \{0, 563, 564, 65535\}$. Los vectores de prueba identificados por el análisis de valores límite pueden ser agregados a los vectores previamente seleccionados por particiones de equivalencia:

$$SelectedVectors = S_{PE} \cup S_{CornerCases}$$

## 2.4. Antecedentes de verificación en PDS

El estado del diseño e implementaciones de sistemas de PDS está soportado por una amplia teoría que abarca desde el estudio de los efectos de cuantización tanto de la señal de entrada como de los parámetros internos y la aritmética. En [16] se desarrollan estos conceptos teóricos de PDS, donde también se estudian y modelan los efectos de cuantización en determinadas arquitecturas de bloques de PDS tales como filtros digitales o FFT.

Con esta base, se han realizado grandes esfuerzos en el área de la verificación formal, impulsados también por el desarrollo de herramientas y técnicas para lograr la mejor elección de la aritmética a utilizar y la cantidad de bits para representación numérica.

Una mezcla de verificación formal y basada en simulación se aborda en [17]. En este trabajo se analiza el algoritmo en lenguaje C de un sistema DSP para estimar su precisión. Se fija un modelo analítico del error para una dada configuración de longitud de palabra para así obtener una estimación pero los autores demuestran que el modelo no es lo suficientemente preciso ya que se obvian las dependencias entre errores no contempladas cuando el sistema o subsistema es desconocido. Para compensar este error se utilizan métodos de simulación.

Aunque la verificación formal brinda demostraciones irrefutables y son útiles para realizar la tarea, existe una desconexión entre el modelo formal que acepta la herramienta y el diseño o

implementación. Por ejemplo, en el caso expuesto anteriormente [18], se indica que se utilizan técnicas conocidas para modelar diseños de PDS en el lenguaje que acepta el demostrador de teoremas, pero falta una demostración formal del funcionamiento en la traducción, por lo que se pierde el formalismo. Algo similar sucede en [17]: el diseño se expresa algorítmicamente en el lenguaje C, y sobre el mismo se realiza la verificación, pero no hay una correspondencia formal entre el diseño en C y la implementación (por ejemplo a nivel RTL). Por lo que se debe confiar en la capacidad de traducción de modelos de los ingenieros en verificación, o se debe proveer una demostración formal de la correcta construcción del modelo o correspondencia entre los modelos.

Por último, otro factor que dificulta la adopción de los métodos formales es la elevada curva de aprendizaje que poseen estas técnicas y herramientas.

Por otro lado, aunque la verificación basada en simulación para PDS ha evolucionado lentamente, aún así no ha logrado adoptar conceptos tales como cobertura. Los avances de PDS y los métodos de simulación están relacionados en el marco de la búsqueda de optimización de la aritmética.

En este contexto, el conjunto de herramientas de software de Matlab permite el diseño en punto flotante y punto fijo por medio del paquete *Fixed-Point Designer* [1]. La herramienta permite registrar los valores máximos y mínimos de los datos, y resaltar cuando suceden operaciones de punto fijo que producen resultados fuera de rango. Mediante la simulación, se pueden extraer estos datos y proponer nuevas configuraciones de punto fijo o se puede solicitar a la herramienta que proponga uno. Esta capacidad de la herramienta es útil solo si se han simulado las entradas que cubren completamente el rango de las operaciones. También provee la opción de pasaje automático del diseño en punto flotante a punto fijo. La decisión de la selección automática está basada en los mínimos y máximos fijados por el usuario a la entrada y salida de los bloques, o en los valores recolectados de simulaciones.

[1] http://www.mathworks.com/products/datasheets/pdf/fixed-point-designer.pdf

# Capítulo 3

# Casos de estudio y modelos de cobertura definidos

Las especificaciones de sistemas de PDS poseen características similares. Frecuentemente la verificación del correcto funcionamiento está estrechamente ligada al cumplimiento de las restricciones de precisión y/o cotas de error.

Cuando se realiza una verificación basada en simulación se debe intentar estimular el diseño con los casos de prueba que generen los límites del error. Por ende, el modelo de cobertura es la abstracción que debe contemplar estos casos de prueba. Alternativamente y de forma limitada, la verificación formal permite la obtención y demostración de los límites de error de una implementación particular descrita en el nivel de abstracción que permita la herramienta.

Este capítulo presenta tres bloques como casos de estudio: filtro digital, transformada rápida de Fourier y unidad aritmética de punto flotante. Los dos primeros generalmente se encuentran en sistemas de PDS que presentan el desafío de construir modelos de cobertura y casos de prueba que constan de señales, mientras que el tercero se aborda con el objetivo de experimentar la verificación de las operaciones básicas de los bloques de PDS, que determinaran la precisión de los mismos.

En este Capítulo cada bloque se define conceptualmente junto con sus especificaciones de diseño, y se presentan las posibles formas de verificación, modelos de cobertura y las implementaciones sobre las cuales. En el Capítulo 4 se describe la construcción de los ambientes de verificación, y los resultados se presentan en el Capítulo 5.

## 3.1. Filtros

A menudo, en sistemas de PDS se requiere resaltar o disminuir características de las señales, por ejemplo utilizando la operación de **filtrado digital**, frecuentemente implementada en hardware.

Estos tipos de bloques realizan operaciones sobre señales tomadas en tiempo discreto para modificar ciertos aspectos de las mismas. En esencia, un filtro digital es una función sobre una señal de entrada a una señal de salida. Una señal es una secuencia de valores en el tiempo. Para nuestro propósito, un filtro digital es causal, esto es, un valor de las señal de salida en tiempo $t$ es una función de los valores de entrada en tiempo previo o igual $t$ (y los valores previos computados de salidas anteriores).

Un filtro digital está caracterizado por su función transferencia, o en forma equivalente, su ecuación a diferencias. Un filtro digital lineal invariante en el tiempo puede ser expresado como una función transferencia en el campo transformado:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-M}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-N}}, \tag{3.1}$$

donde $z$ representa una variable compleja. La ecuación lineal a diferencias es:

$$y[n] = -\sum_{k=1}^{N} a_k y[n-k] + \sum_{k=0}^{M} b_k x[n-k]. \tag{3.2}$$

El filtro se caracteriza por el tipo de respuesta. Son de respuesta finita al impulso (FIR, *finite impulse response*) si $a_k = 0 \ \forall \ k$ y en en caso contrario, se denominan de respuesta infinita al impulso (IIR, *infinite impulse response*). Los filtros FIR tienen una respuesta impulsiva $h_f(t) = 0 \, \forall \, t > N$, mientras que los filtros IIR pueden tener una respuesta impulsiva que puede ser no nula infinitamente. La respuesta en frecuencia o la respuesta impulsiva son otras caracterizaciones comunes de los filtros. Cualquiera de esas respuestas caracterizan completamente la funcionalidad del sistema (o filtro) digital desde el punto de vista ingenieril.

La construcción de un filtro digital se basa típicamente en un conjunto de plantillas de diseño [16], las cuales indican la arquitectura del filtro a implementar. Cada una posee diferentes características en cuanto a insumo de recursos y precisión Las más comunes son las implementaciones *forma directa I, forma directa II, cascada y paralela*. Para diseñar un filtro, los ingenieros deben seleccionar una plantilla (por ejemplo, filtros *forma directa I*), fijar las características del filtro y luego obtener los coeficientes ($a_k$ y $b_k$ en las Ecuaciones (3.1) y (3.2)) que son utilizados para instanciar esas plantillas.

El diseño de filtros digitales está guiado por una amplia teoría que incluye el entendimiento

del comportamiento en términos de sus propiedades en dominio temporal y frecuencial, tanto como el análisis del impacto de la cuantización de los parámetros [16]. Los diseñadores de filtros se basan en herramientas de aritmética de punto flotante para realizar el diseño y la verificación, por ejemplo mediante la herramienta Matlab.

Luego, el flujo continúa con la etapa de implementación, que puede realizarse utilizando aritmética de punto flotante o de punto fijo, siendo la última la opción preferida para optimizar la relación entre costo y desempeño.

Si bien existen herramientas que asisten a la conversión de un diseño en punto flotante a una implementación en punto fijo, generalmente este proceso es manual generando una desconexión entre diseño e implementación. Existen soluciones "automáticas" para casos sencillos.

La transición de aritmética de punto flotante a punto fijo conlleva efectos indeseables tales como inestabilidades (ej: ciclos límite de pequeña ó de gran amplitud, en donde el filtro ante una entrada nula, retorna valores no nulos que se repiten infinitamente causados por el redondeo ó los desbordes respectivamente) y discrepancia entre las respuestas deseadas y las obtenidas. Esto se debe a (a) errores de redondeo debido a las multiplicaciones y sumas, y (b) la cuantización de los coeficientes del filtro. Por ello, las representaciones en punto fijo necesitan ser lo suficientemente precisas, es decir, tener la adecuada cantidad de bits para representar la parte entera y fraccionaria tal que no se observe ningún efecto indeseable en la implementación.

En este contexto, el principal objetivo del proceso de verificación es que una implementación procese precisa y apropiadamente las señales de entrada. Estas pueden ser verificadas mediante métodos formales o de simulación.

### 3.1.1. Verificación funcional

Para llevar a cabo la verificación funcional basada en simulación de este tipo de bloques se deben conocer los requerimientos de diseño o especificaciones. Estas comprenden información sobre el tipo de señales que debe aceptar (formato, velocidad de procesamiento, interfaz de comunicación, rango de frecuencias); características específicas de diseño (respuesta en módulo y/o fase y sus tolerancias, velocidad de operación, modo de filtrado en línea o fuera de línea); y otras como área o consumo de energía.

#### Especificaciones

Respecto a las características de diseño, frecuentemente se especifican en el dominio frecuencial, y para el caso de los filtros selectivos en frecuencia, estas especificaciones toman la forma de bandas de tolerancia, como muestra la Fig. 3.1 para el caso de un filtro pasabajos. Las zonas grisadas indican los límites de tolerancia. En la banda de paso, el módulo de la respuesta admite

una variación máxima de $\pm\delta_p$ , y en la banda de rechazo se pretende que la ganancia no exceda $\delta_s$. Es frecuente especificar estas cotas en decibeles (dB).

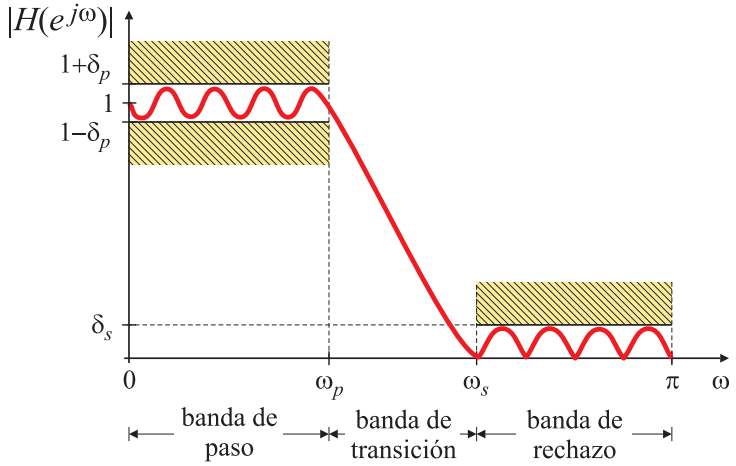


Figura 3.1: Bandas de tolerancia para un filtro pasabajos.

El ancho de la zona de transición determina qué tan abrupto es el filtro. En esta región se espera que el módulo de la respuesta en frecuencia decrezca monótonamente desde la banda de paso a la banda de rechazo. Los principales parámetros de interés son:

- desviación en la banda de paso: $\delta_p$ ,
- desviación en la banda de rechazo: $\delta_s$ ,
- frecuencia de corte de la banda de paso: $\omega_p$ ,
- frecuencia de corte de la banda de rechazo: $\omega_s$.

Las frecuencias de corte o límite se suelen normalizar a la frecuencia de muestreo $f_s$ ($\omega_i = 2\pi f_i/f_s$), pero también son frecuentes las especificaciones en unidades típicas como Hz, kHz, y a menudo son mucho más significativas. Las desviaciones con respecto a la respuesta deseada tanto en la banda de paso como en la banda de rechazo pueden expresarse en valores absolutos o en decibeles (dB), indicando la ondulación máxima ("*ripple*") tolerada en la banda de paso, y la atenuación mínima exigida en la banda de rechazo:

- atenuación en la banda de rechazo: $A_s = -20\log_{10}\delta_s$ ,
- ondulación en la banda de paso: $A_p = 20\log_{10}(1+\delta_p)$ .

En general, la respuesta de fase un filtro no se especifica tan detalladamente como el módulo de la respuesta en frecuencia. En muchos casos es suficiente indicar que importa la distorsión

de fase, o que se pretende una respuesta de fase lineal. Sólo es necesario detallar la respuesta en fase deseada en aquellas aplicaciones donde los filtros se utilizan para compensar o ecualizar la respuesta en fase de un sistema.

La aritmética a utilizar debe estar especificada acorde del desempeño que se quiera lograr, el tipo de señal de entrada y la precisión de la señal resultante. Se debe diseñar cuidadosamente las dimensiones de registros y operaciones con el fin de evitar resultados erróneos (desbordes) o imprecisos.

#### Modelo de cobertura de filtros

En los casos de estudio abordados se espera no tener visibilidad sobre los detalles de implementación del DUV. Por esta razón el espacio de prueba debe ser explorado a conciencia para exponer cualquier error de implementación. De todas maneras, las señales de entrada involucran tantas muestras como se desee con precisión típica de entre 8 y 32 bits, lo que resulta en un número de vectores de prueba infinito a menos que se adopte una metodología para reducirlo, por lo tanto es necesario definir un modelo de cobertura y equivalencias desde la perspectiva de verificación.

El filtro puede ser descrito (o especificado) mediante un gráfico de respuesta en magnitud vs. frecuencia como el de la Fig. 3.1. Como primera aproximación a la construcción de modelos de cobertura para PDS, se seleccionarán los casos de prueba de interés tomando como referencia dicha gráfica.

Con respecto al eje de frecuencias, estas se particionan en función de las bandas especificadas delimitadas por $\omega_p$ y $\omega_s$, resultando (para el caso del filtro pasabajos) en tres regiones de equivalencia: (1) equivalencia banda de paso (2) equivalencia banda de transición y (3) equivalencia banda de rechazo. Dentro de cada equivalencia se deberán seleccionar señales representantes de la clase con valores límite correspondientes a amplitudes máximas y mínimas.

Dentro de los conjuntos de frecuencias, a su vez se seleccionan los casos límites relacionados a la magnitud de la señal de entrada, seleccionando amplitudes máximas para estimular las condiciones de borde y amplitudes mínimas para ejercitar la representación aritmética.

### 3.1.2. Verificación formal de filtros

Una aproximación para verificar formalmente un bloque de filtrado digital es mediante la prueba de propiedades aritméticas. Mediante demostraciones matemáticas sobre intervalos numéricos éstas permiten verificar la correcta elección de la aritmética a implementar. Gappa [11] es una de estas herramientas, donde se debe generar un modelo de referencia (en aritmética real) y otro que plasme la intención de la implementación (en el lenguaje que acepte la herra-

mienta), más las propiedades que se verificarán. Luego se puede consultar sobre propiedades aritméticas que involucren a uno o ambos modelos.

La verificación de condiciones como desbordes se realizan consultando por el rango de los valores internos del diseño. Asimismo, la cota de error se verifica consultando por el rango de la diferencia entre el modelo de referencia y aquel que refleja las intenciones de implementación.

Por otro lado, la herramienta presenta la limitación de no permitir la definición de fórmulas recursivas, como es el caso de los filtros IIR. Solo se pueden verificar propiedades para un cierto número de entradas o instancias del sistema. Por ende, se puede afirmar que la verificación con este tipo de técnica es incompleta.

### 3.1.3. Implementaciones a verificar

Los casos de estudio se obtienen a partir de herramientas automáticas de diseño e implementación de filtros. La primera etapa consiste en el diseño del filtro: la herramienta FDA Tool de Matlab permite obtener los coeficientes del filtro para una dada arquitectura y precisión. La herramienta Simulink de la misma compañía permite implementar la arquitectura y hacer la "síntesis de alto nivel" desde el diagrama en bloques a RTL. Por otro lado, se encuentra la solución del **proyecto Spiral** que pone a disposición la generación de RTL mediante una interfaz web, trabajo generado a partir de [19]. También está a disposición el software que genera la "síntesis de alto nivel" bajo licencia GNU GPL.

#### Generación automática de implementaciones RTL

A partir de los coeficientes obtenidos, se genera la implementación del filtro diseñado. Esta tarea puede realizarse automáticamente con algunas herramientas, entra ellas *Simulink HDL Coder* y la interfaz web de *Spiral Filters*. La primera permite realizar un diagrama en bloques a partir de constructores (bloques) básicos que brinda la herramienta, entre ellos retardos (registros), multiplicadores y sumadores. En cambio, Spiral brinda una interfaz donde se deben especificar ciertos parámetros del diseño para luego obtener la descripción en lenguaje de hardware (HDL, *Hardware Description Language*).

#### Filtros generados

Simulink provee una gran cantidad de bloques básicos y otros de mayor complejidad ya listos para ser generados en HDL, pero esta automatización no está provista para el bloque IIR que se necesita, por ende se debe construir a partir de los bloques básicos. En la Fig. 3.2 se puede observar el diagrama en bloques del filtro IIR en la estructura Forma Directa I correspondiente

a la función de sistema

$$\frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}$$

en el que se utilizan registros (retardos), sumadores, multiplicadores y conexiones.

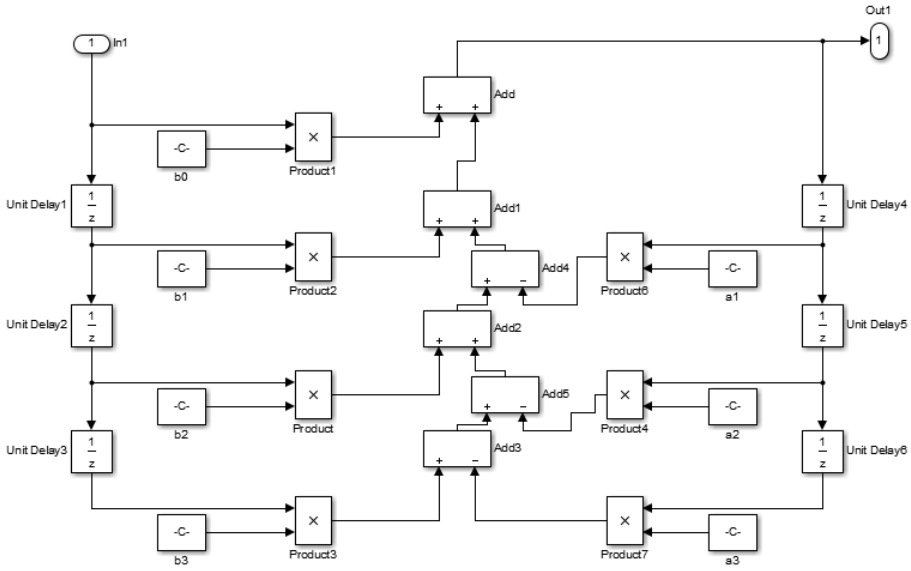


Figura 3.2: Diagrama en bloques IIR en forma directa I en Simulink.

Una vez listo el diagrama en bloques que representa al filtro IIR, se debe especificar la aritmética a utilizar, es decir configurar los bloques aritméticos indicando el ancho de palabra, cantidad de bits utilizados para representar la parte entera y la fraccionaria, y cómo se representará el signo (signo magnitud, complemento a la base o complemento a la base disminuida). Otra opción para la configuración de la aritmética del sistema es indicar el formato de la entrada y automáticamente, por medio del generador HDL, se deduce la aritmética de los bloques y señales intermedias. Así por ejemplo en un bloque de suma de dos números de 16 bits de parte entera, se deduce un resultado con 17 bits de parte entera. Para este caso de estudio generado, se utilizó la segunda opción indicando los formatos numéricos de entrada y salida. En la Fig. 3.3 se puede apreciar el diagrama en bloques con la aritmética. La configuración de entrada se indica con el nombre "sfix32_En24" que significa que la aritmética a utilizar es complemento a 2 de 32 bits con 24 bits de parte fraccionaria.

Finalizado el proceso de implementación se procede a la generación de código HDL. Para esta tarea se lanza un asistente donde en primera instancia se indica el objetivo de la generación, en este caso será sólo de generación de HDL (otras opciones son generación para FPGA

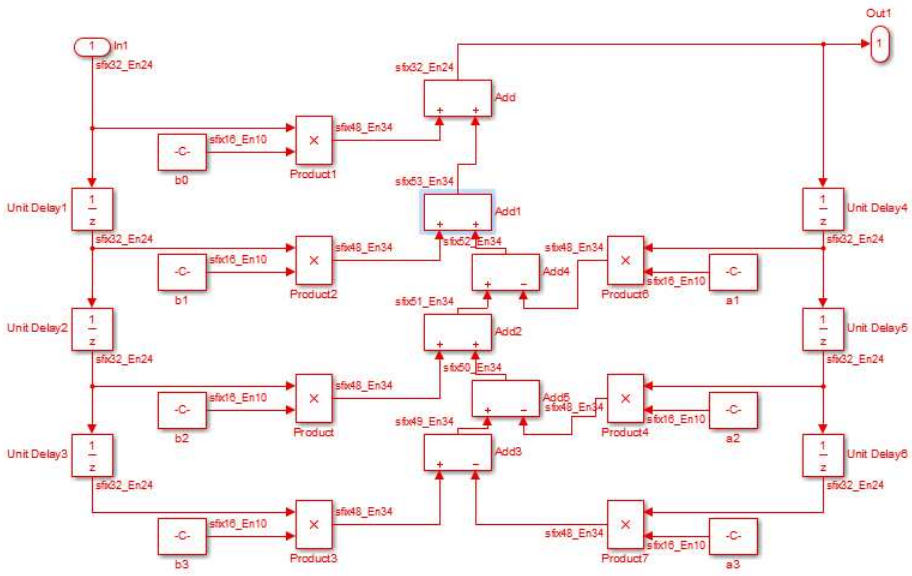


Figura 3.3: Aritmética de IIR en Simulink.

propietaria, archivos de simulación y de síntesis). Luego la herramienta realiza varios controles, como por ejemplo que se hayan usado bloques compatibles con la generación HDL, que no haya recursividad algebraica, etc. Como resultado se obtiene el archivo con el HDL del filtro IIR.

En cambio, Spiral provee una interfaz especifica para la generación de filtros, por lo que solo se deben ingresar los parámetros del filtro. Entre ellos, la cantidad y valores de los coeficientes (en punto flotante), la aritmética a utilizar (cantidad de bits y cuántos de ellos corresponden a la parte fraccionaria), la arquitectura a utilizar (forma directa I o II). A partir de los coeficientes reales y la aritmética especificada, se convierten éstos a enteros. En la Fig. 3.4 se puede observar la interfaz web utilizada[1] para generar los filtros.

Los filtros generados con Matlab y Spiral son filtros pasabajos IIR tipo Butterworth en forma directa I (FD I) de orden 3, con una frecuencia de corte de 10800 Hz y frecuencia de muestreo 48000 Hz. Con respecto a la aritmética, todos poseen 32 bits de ancho de palabra, usando el formato es complemento a dos. La diferencia entre los filtros generados es la cantidad de bits de parte fraccionaria. La Tabla 3.1 muestra las opciones disponibles para los filtros IIR con su aritmética asociada.

Para ambas herramientas de generación automática se debe suponer que se ha realizado correctamente la "traducción" del diseño en alto nivel a la implementación generada en HDL. No se cuenta con la certeza de la traducción correcta ya que carece de una demostración formal

[1] http://www.spiral.net/hardware/filter.html

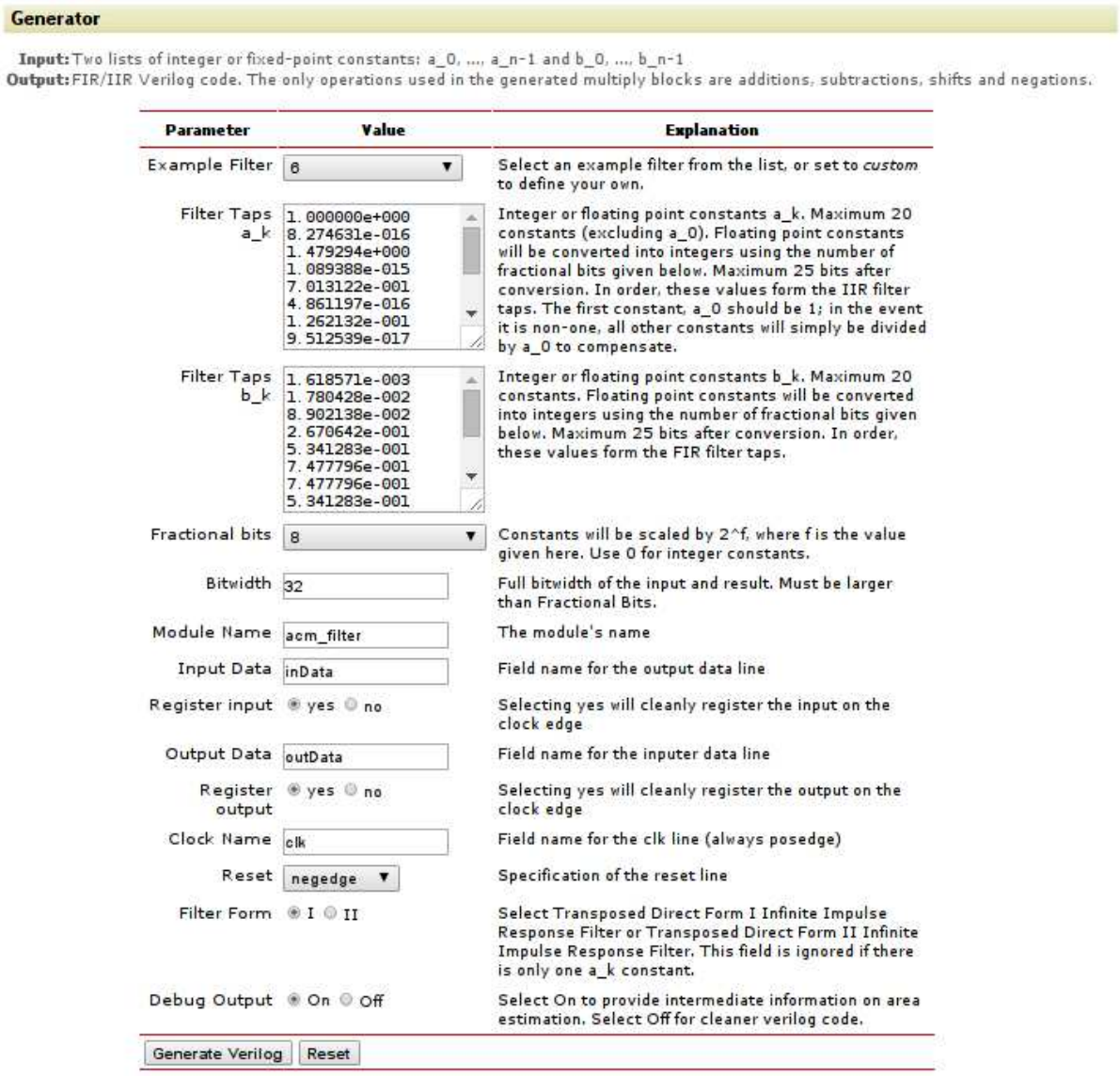


Figura 3.4: Generación de filtros usando la herramienta Spiral.

Tabla 3.1: Filtros generados

| Herramienta Gen. | Implementación | Long. fraccionaria |
|---|---|---|
| Matlab | ML-8 | 8 |
| Matlab | ML-12 | 12 |
| Matlab | ML-16 | 16 |
| Matlab | ML-20 | 20 |
| Matlab | ML-24 | 24 |
| Spiral | Sp-2 | 2 |
| Spiral | Sp-3 | 3 |
| Spiral | Sp-4 | 4 |
| Spiral | Sp-5 | 5 |
| Spiral | Sp-6 | 6 |
| Spiral | Sp-8 | 8 |
| Spiral | Sp-12 | 12 |
| Spiral | Sp-16 | 16 |
| Spiral | Sp-20 | 20 |
| Spiral | Sp-24 | 24 |

de la mecánica utilizada para construir correctamente las implementaciones; en caso de contar con una demostración formal se generaría una implementación correcta por construcción. Este es uno de los motivos por lo que es necesaria la verificación de las implementaciones que se hayan generado en estos casos.

## 3.2. Transformada rápida de Fourier (FFT)

Una de las operaciones más relevantes en PDS, comúnmente implementada en hardware, es la transformada rápida de Fourier (FFT, por sus siglas en inglés *Fast Fourier Transform*).

La FFT es es un algoritmo muy eficiente para realizar la transformada discreta de Fourier. Puede ser implementada utilizando aritmética de punto flotante o de punto fijo, siendo la última la opción usual cuando se requiere un diseño optimizado. En este contexto, el principal objetivo del proceso de verificación funcional, tanto en términos de tiempo de ejecución y consumo de energía, es que una implementación en punto fijo procese precisa y apropiadamente las señales de entrada.

Esta operación está presente en sistemas avanzados de PDS. Por ejemplo, los procesadores auxiliares para los sistemas de telefonía de alta gama usan un DSP de la familia C55x de Texas Instruments que incluye un procesador FFT de 1024 puntos de punto fijo.

El trabajo [18] presenta una metodología de verificación donde la precisión se verifica algebraicamente. El trabajo presentado en [20] se enfoca en la implementación de un bloque FFT, pero también se realiza la verificación con unos pocos casos triviales de prueba que no proveen un alto grado de confianza sobre la correctitud. En [17] se analiza la arquitectura del sistema para estimar su precisión: se fija un modelo analítico del error para una dada configuración de longitud de palabra para así obtener una estimación pero, debido a dependencias entre errores no contempladas por el modelo, se utilizan simulaciones para corregir la estimación inicial.

En [21] se describen en profundidad los algoritmos e implementaciones de FFT y se incluye la verificación mediante señales específicas de prueba, donde el diseño de las mismas está basado en la experiencia previa en evidenciar errores por parte de los autores. El efecto se ejemplifica en una implementación de FFT de tamaño 16 de raíz-4 (de arquitectura tipo "mariposa"). Los cuatro tipos de señales que propone son:

- Pulso unitario
- Constante
- Senoidal simple
- Senoidal doble

Cada señal posee ciertas características específicas para encontrar determinados errores. El pulso unitario es una señal digital donde uno de sus valores complejos es distinto de cero mientras que los demás son todos cero. Su característica principal es que, al activar una única entrada, se puede observar qué salidas estimula. Si el pulso unitario es un vector de longitud $T_s$, con un único valor real $A$ en posición r-ésima, y ceros en el resto de las posiciones, el vector de salida debe estar formado por los valores $Ae^{j\frac{2\pi}{T_s}kr}$, $0 \leq k \leq T_s - 1$. Una posible estrategia, es aplicar ésta señal a cada una de las entradas de la FFT y verificar que la respuesta sea correcta. Luego, al ser la FFT una operación lineal, debería funcionar para cualquier señal arbitraria (sin tener en cuenta los efectos de la aritmética finita).

La señal constante es una señal donde todos sus valores complejos son iguales. Es fácil de generar y es difícil de equivocarse en la inyección de estos valores al bloque, por lo que es una buena señal para hacer una primera prueba básica sobre el funcionamiento de la aritmética, pero solo se estará estimulando una única salida (la primera) ya que todas las demás se cancelan, y se estimulará solo una pequeña porción de la lógica de multiplicación.

En cambio, la señal senoidal simple centrada en la primer frecuencia de salida distinta de cero, tiene exactamente un ciclo durante las $N$ entradas al bloque ($N$ igual al tamaño de la transformada). Generalmente, ésta señal requiere ejercitar todas las multiplicaciones internas

por constantes para proveer la salida correcta. Su desventaja es que también causa que valores intermedios en las operaciones se hagan nulos.

Un enfoque más elaborado es el de la señal de senoidal doble, ésta es una señal que es la suma de dos ondas senoidales. En éste caso de prueba se evita el problema del caso anterior donde las computaciones intermedias se anulan. Pero, este tipo de casos de prueba son más difíciles de generar que los anteriores, y también es más complicado descifrar la fuente de error cuando el comportamiento no es el deseado.

Además, el trabajo ejemplifica un orden en la aplicación de éstas pruebas con el objetivo de reconocer y aislar un error en la implementación de ejemplo. El orden está basado en las características de ejercitar parcial o totalmente los bloques que componen la FFT y el costo asociado a crear las señales.

A continuación se describe el espacio de cobertura y la generación de un modelo para la construcción de un componente de verificación de bloque FFT. La premisa en el desarrollo de este ambiente es proveer un conjunto de prueba sólido para realizar la verificación tipo caja negra de tales diseños. Como caso de estudio, se proponen tres implementaciones diferentes de FFT verificadas con la metodología propuesta.

### 3.2.1. Verificación funcional

La especificación de un diseño define su funcionalidad, parámetros de implementación, precisión de cómputo, restricciones de desempeño en términos de tiempo y consumo de energía y de formato entrada/salida. A partir de estos, es posible determinar las características a verificar.

En el caso de un bloque FFT implementado en punto fijo, los dos parámetros de implementación fundamentales son el tamaño de la transformada y la precisión de los datos. Ambos definen la cardinalidad del espacio de cobertura; esto significa que la cantidad de posibles señales de entrada o posibles secuencias de prueba, así como las salidas, dependen del tamaño de la transformada y la precisión de los datos.

La evaluación de la precisión debería estar especificada como una cota superior sobre el máximo error permitido de la salida. Esta cota del máximo error determina tanto la métrica para evaluar si el bloque computa la transformada, como si la aritmética es lo suficientemente precisa para la aplicación deseada.

Restricciones de desempeño temporal definen el tiempo requerido en computar una transformada, o el número de ciclos de reloj cuando la frecuencia del reloj para el bloque está fijada. Estas también pueden fijarse en términos de número de transformadas por unidad de tiempo o *throughput* y en términos de tiempo requerido para obtener una salida para una determinada entrada o latencia.

Por especificaciones de formato de entrada/salida se entiende la estructura en la que los datos ingresan y egresan del bloque. Particularmente define el número puntos de entradas y salidas paralelas por ciclo de reloj, también referido como ancho de la transferencia (*streaming width*), y el tipo de ordenamiento de los datos asociados a la base (radix) de la FFT. El formato de entrada/salida también debería fijar el factor de escala asociado a los datos para permitir la correcta configuración de las entradas e interpretación de las salidas.

### 3.2.2. Modelo de cobertura de FFT

Dado que el componente de verificación está diseñado para verificación tipo caja negra, se espera no tener visibilidad sobre los detalles de implementación del DUV. Por esta razón el espacio de prueba debe ser explorado a conciencia para exponer cualquier error de implementación. De todas maneras, las secuencias de entrada características del procesamiento de FFT típicamente involucran bloques desde 64 a 4096 ítems con precisión entre 8 y 32 bits, lo que resulta en un número de vectores de prueba impráctico de manejar a menos que se adopte una metodología para reducirlo. Por esta razón, se analiza el espacio de señales para definir una estructura adecuada para aplicar la estrategia introducida en la sección 2.3.3. Entonces, un modelo de cobertura se define como una lista de conjuntos de cobertura.

#### Complejidad del espacio de prueba

Como se introdujo en la sección 3.2.1, la cardinalidad del espacio de prueba depende de dos parámetros, el tamaño de la transformada $T_s$ (donde $T_s$ es potencia de 2) y la precisión del tipo de dato $D_p$. Por ejemplo, para una FFT de longitud $T_s = 4$ y formato de dato punto fijo con $D_p = 4$, que son valores muy pequeños para un uso práctico, el espacio de salida está formado por 4 posibles puntos cada uno representado con 4 bits, lo que resulta en $2^{D_p \times T_s} = 65536$ posibles señales de entrada.

Este espacio de cobertura es un claro ejemplo de la complejidad con que el proceso de verificación debe lidiar, que es inviable de cubrir completamente.

#### Clasificación del espacio de cobertura

La transformada discreta de Fourier transforma una sucesión de $T_s$ muestras de longitud en otra sucesión del mismo largo. Usualmente la sucesión original representa información temporal, y el resultado de la TDF son los coeficientes de la representación en series de Fourier de esa sucesión, aunque con ligeras modificaciones el mismo algoritmo puede utilizarse para realizar la operación inversa.

Para clasificar el espacio de cobertura es conveniente recordar algunas de las características de la transformada discreta de Fourier, que para sucesiones de largo $T_s$, donde $T_s$ es potencia de 2, se calcula con el algoritmo conocido como Transformada Rápida de Fourier o FFT (por las siglas en inglés de *fast Fourier transform*).

Es habitual considerar que cada muestra (o *bin*) $k$ del resultado de la TDF está asociado a una *frecuencia discreta* $\omega_k = (2\pi/T_s)k$, con $0 \leq k \leq T_s - 1$. Los resultados de la TDF son muy diferentes si las sucesiones a transformar son exponenciales complejas (o senos o cosenos) cuya frecuencia coincide con la de alguno de los *bins*, o no.

- Si la sucesión temporal es una exponencial compleja cuya frecuencia coincide con alguno de los *bins*, es decir tiene la forma $Ae^{j(2\pi/T_s)k_0 n}$, para algún $k_0 \in [0, T_s - 1]$ y $0 \leq n \leq T_s - 1$, su TDF será un vector nulo de $T_s$ muestras o *bins*, salvo el $k_0$-ésimo *bin* cuyo valor será $AT_s$. Si la sucesión de entrada al algoritmo es un coseno de frecuencia $(2\pi/T_s)k_0$, la TDF contendrá únicamente dos muestras no nulas, las que corresponden a los *bins* $k_0$ y $T_s - k_0$, cuyo valor será $AT_s/2$. Finalmente, si la sucesión temporal es un seno de frecuencia $(2\pi/T_s)k_0$, la TDF también tendrá sólo dos elementos no nulos (los correspondientes a las mismas muestras $k_0$ y $T_s - k_0$), pero sus valor será complejo, de amplitud $-jAT_s/2$ para el *bin* $k_0$, $jAT_s/2$ para el *bin* $T_s - k_0$.

- Si la sucesión a transformar es una exponencial compleja (o un seno o un coseno) de frecuencia $\omega_0$ que *no* puede escribirse como $(2\pi/T_s)k_0$ para algún $k_0 \in [0, T_s - 1]$, su TDF resultará en un vector de $T_s$ elementos todos distintos de cero, fenómeno que se conoce como *fuga espectral* [16].

Estas características de la TDF muestran que las señales tipo exponenciales complejas (o senos o cosenos) cuya frecuencia coincide con alguno de los *bins* de la TDF, o combinaciones de este tipo de señales, son excelentes para verificar el comportamiento del algoritmo de cálculo de la TDF, ya que permiten verificar no sólo el comportamiento funcional, sino también la precisión de los resultados numéricos. Si se utilizasen señales de prueba cuya frecuencia no coincidiese con alguno de los *bin* de la TDF, la fuga espectral podría enmascarar los errores numéricos que ocurriesen para *bins* alejados de los próximos a la frecuencia de la señal.

La TDF posee varias otras propiedades; por ejemplo, cuando $T_s$ es par, si el conjunto de muestras de entrada son reales, los valores de la TDF en el rango $[1, ..., T_s/2 - 1]$ son conjugados de los valores en el rango $[T_s - 1, ..., T_s/2 + 1]$. Aunque estas propiedades también puede explotarse en la verificación funcional de la TDF, son mucho menos convenientes que las citadas más arriba.

**Ejemplo 2.**

El efecto de este tipo de señales que coinciden con un *bin* o varios puede verse en el estímulo de un implementación radix-2 "decimación en tiempo" con tamaño de transformada 8 (Fig. 3.5).

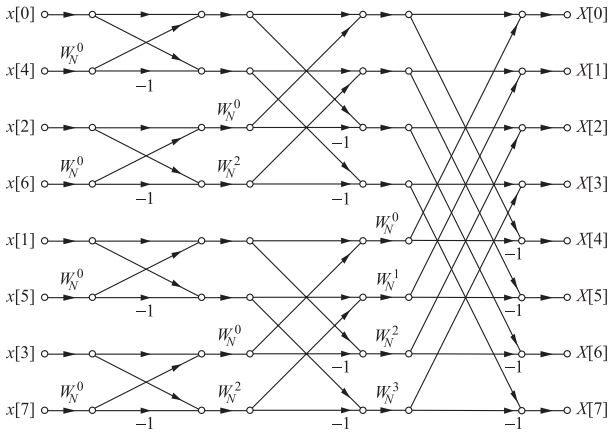


Figura 3.5: Arquitectura FFT tipo mariposa, tamaño 8.

En la Fig. 3.6 una señal constante donde todas las muestras de la señal poseen el mismo valor estimulan la lógica de direccionado (conexiones) y las operaciones aritméticas. Los números sobre los nodos indican los resultados de las operaciones intermedias. En este caso todas las $x[i]$ se estimulan pero solo la salida $X[0]$ toma valor diferente de 0, y puede observarse que menos de la mitad del conexionado propaga los resultados intermedios. Este tipo de señal es simple de generar y efectiva para probar que la correctitud de la primera salida, pero aún así no resulta efectiva en el estímulo de todo el direccionado y la aritmética ya que solo se ejercita la parte real de un conjunto reducido de operadores de la mariposa.

En forma equivalente, si se ejercita con una señal de tipo impulso como la de la Fig. 3.7, se estimula el camino inverso de la señal de ejemplo anterior (por dualidad de la FFT). Por lo que esta señal expone las mismas virtudes y carencias que la de la Fig. 3.6.

En forma equivalente, si se ejercita con una señal tipo impulso como la de la Fig. 3.7, también se estimula sólo una parte de las conexiones: por la dualidad de la TDF, la FFT de un impulso es una constante, y por este motivo los caminos que se ejercitan son en cierta forma complementarios de los de la Fig. 3.6. Por lo que esta señal expone las mismas virtudes y carencias que la del ejemplo anterior.

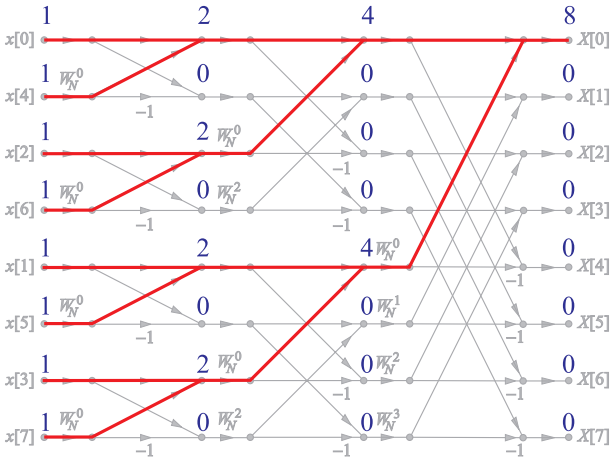


Figura 3.6: Estímulo señal constante.

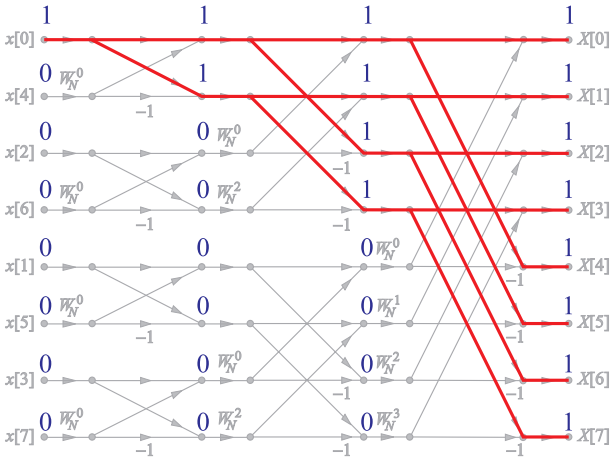


Figura 3.7: Estímulo señal tipo impulso.

Cuando se estimula con una señal exponencial compleja con una componente frecuencial ($k = 2$) como la de la Fig. 3.8 se logra un mejor estímulo de las operaciones aritméticas pero aún así una gran parte de la lógica de direccionado no es ejercitada, y sólo la salida $X[2]$ toma un valor diferente de 0. Lo mismo sucede con una señal exponencial compleja con $k = 4$ como la de la Fig. 3.9, solo que en este caso se ejercita la aritmética compleja del bloque.

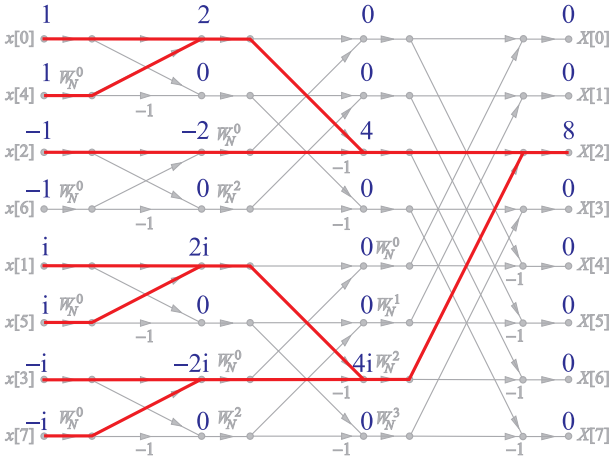


Figura 3.8: Estímulo señal compleja $k = 2$.

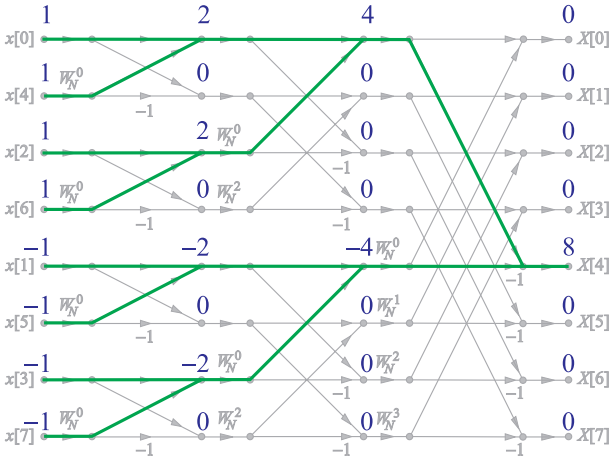


Figura 3.9: Estímulo señal compleja $k = 4$.

Una mejor aproximación para la construcción de una señal de prueba es la composición de las dos anteriores ($k = 2$ y $k = 4$). La suma permite un mejor estímulo de la aritmética y mayor cobertura sobre las salidas del bloque como se expone en la Fig. 3.10. Aunque el estímulo de la lógica de direccionado sigue siendo insuficiente.

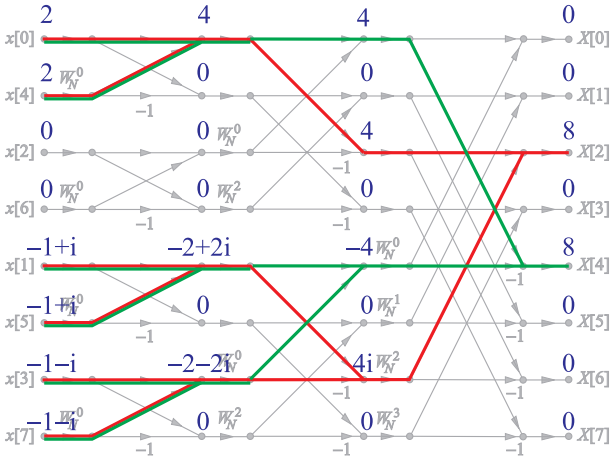


Figura 3.10: Estímulo suma de señales complejas ($k = 2$ y $k = 4$).

Todas las señales anteriores son útiles en verificar alguna características del bloque, pero cada una posee sus carencias. Es importante la elección de aquellas señales efectivas en estimular tanto la aritmética como la lógica de direccionado para así poder reducir el espacio de señales de prueba definiendo equivalencias que faciliten la posibilidad de encontrar errores.

■

Por lo tanto, la metodología propuesta para particionar el espacio de señales está basado en la representación de cada señal mediante tres parámetros:

- la cantidad de componentes frecuenciales de una señal $N_{fc}$,
- el valor de cada componente frecuencial $F[n]$,
- y su amplitud $A[n]$,

donde $n \in \{0 \dots N_{fc} - 1\}$ y $N_{fc} \leq T_s$.

Es útil contar con una estructura que represente la relación entre estos parámetros para poder definir e identificar las clases de equivalencia y los valores límite. Una posible representación es la de árbol, donde los valores de posibles de frecuencias $P_n$ son nodos, de los que se desprenden ramas que representan los posibles valores de esas frecuencias, de los que a su vez se desglosan en hojas con los posibles valores de amplitud.

La Fig. 3.11 muestra la estructura del espacio de cobertura propuesto. Cada nivel del árbol corresponde a cada uno de los tres parámetros. El primer nivel define las posibles configuraciones de $N_{fc}$, el segundo nivel fija los posibles valores para cada elemento del arreglo de frecuencias

$F$, y el tercero fija las posibles amplitudes asociadas a cada elemento de frecuencia en $F$ que son almacenadas en $A$.

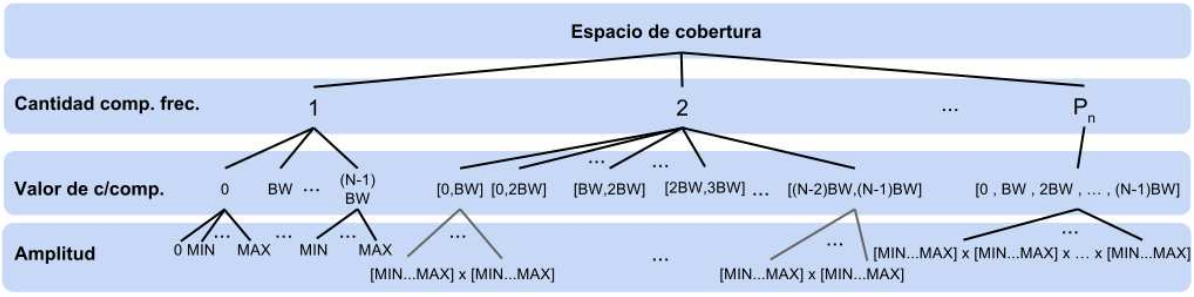


Figura 3.11: Espacio de cobertura en árbol.

Para un dado nodo padre $P_n$ de los posibles valores de *número de frecuencias* ($N_{fc} \leq P_n$), los arreglos $F$ y $A$ tendrán longitud $P_n$. Cada elemento $F_i$ pertenece a alguna de las frecuencias del espacio de salida:

$$F_i \in \{0, f_{bw}, \ldots, (T_s - 1)f_{bw}\}$$

donde $f_{bw}$ es el "ancho" del bin, $f_{bw} = 2\pi/T_s$ y $T_s$ es el tamaño de la implementación de la FFT; y las posibles configuraciones para $F$ están definidas por las combinaciones de $T_s$ elementos tomados de a $P_n$ sin repeticiones.

Con respecto a las amplitudes, cada elemento (hoja) $A[i]$ pertenece a una de las amplitudes posibles entre el un mínimo (el siguiente a cero) y la amplitud máxima (determinada por la precisión $D_p$):

$$A[i] \in \{min, \ldots, max\}$$

donde $min$ es la amplitud mínima diferente de cero y $max = 2^{D_p} - 1$ es la máxima amplitud.

La amplitud cero ($A = 0$) se excluye de la posible selección de amplitudes ya que diferentes hojas modelarían una señal nula independiente del nodo frecuencia. Esta señal de amplitud nula se representa una única vez en la rama con valores $P_n = 1$ y $F[1] = 0$.

Las posibles configuraciones de amplitud que corresponden a cada nodo $F$ son obtenidas mediante el producto vectorial $\{min, \ldots, max\}_1 \times \cdots \times \{min, \ldots, max\}_P$, la cual implica $max^P$ posibilidades.

Esta representación ayuda a analizar las clases de equivalencia y valores límite con el objetivo de producir las señales de entrada para cubrir el espacio de verificación. Más aún, el árbol es una representación computacional que puede ser usada dinámicamente en una *verificación guiada por la cobertura* (CDV, por sus siglas en inglés *Coverage Driven Verification*) para generar los vectores de prueba o medir la calidad de la verificación.

**Conjuntos de Cobertura de la FFT**

En verificación tipo caja negra, la definición de casos de prueba relevantes corresponde principalmente a la experiencia previa relacionada al DUV o a criterios intuitivos. En este caso, los conjuntos de cobertura resultantes deberían implicar suficientes casos de prueba para estimular la lógica de la aritmética con el máximo y mínimo de entradas buscando generar condiciones de *overflow* o *underflow*. También la lógica de direccionado que genera el patrón de acceso a las estructuras de cómputo típicas (conocidas como *mariposas*), debería ser probada exhaustivamente para prever cualquier error.

La aplicación del concepto de clases de equivalencia en la aplicación de la agrupación de los casos de prueba sigue la premisa de que cuando dos vectores de entrada pertenecen a la misma clase, el DUV debería comportarse correcta o incorrectamente para ambos casos. De esta manera, la clasificación asume que los mismos recursos son estimulados por cada vector dentro de la misma clase. Realizando pruebas sobre el diseño con un elemento de cada clase reduciría el número de casos de prueba, pero aún proveería una alta confianza sobre la calidad del proceso de verificación.

En este contexto, la clasificación se hace dividiendo el dominio de los tres parámetros $N_{fc}$, $F$ y $A$. En una primera aproximación $N_{fc}$ puede usarse para particionar, dejando las señales con una componente frecuencial en un conjunto, las señales con $T_s$ componentes en otra y el resto en una tercera clase. El dominio correspondiente a $F$ puede también ser dividido; por ejemplo el subconjunto de señales con dos componentes frecuenciales puede ser dividido en tres regiones: una donde las dos componentes frecuenciales pertenecen a los primeros dos bins, otro con aquellas pertenecientes a las últimos dos bins, y otro con aquellos que pertenecen a cualquier otro par de posiciones. Finalmente, el hipercubo que define las combinaciones de $A$ puede ser particionado en regiones con límites tales como $(min, min, \ldots, min)$ o $(max, max, \ldots, max)$.

El primer conjunto de cobertura es denominado el *conjunto de señales con una frecuencia* el cual es determinado por el límite inferior para el dominio $N_{fc}$, esto es, el subárbol con una componente frecuencial en la Fig. 3.11. Su importancia yace en verificar el comportamiento de cada bin de salida individualmente, por lo tanto, este conjunto contiene un caso de prueba para cada posible valor del parámetro $F$. Con respecto a la amplitud, son considerados equivalentes las amplitudes intermedias, es decir las amplitudes mayores a $min$ y menores a $max$ ($A \in (min, \ldots, max)$), con valores límite $A = max$ y $A = min$. Entonces, el *conjunto de señales con una frecuencia* contiene $3 \times T_s$ casos de prueba: uno correspondiente al valor límite con amplitud mínima, otro al máximo y por último el que agrupa las señales con amplitud entre estas últimas dos (consideradas equivalentes).

Un segundo conjunto de cobertura es el *conjuntos de señales de dos componentes frecuencia-*

*les* identificado por la rama $N_{fc} = 2$. Su intención es exponer cualquier problema en la primera etapa de la mariposa de raíz-2. La equivalencia se aplica al parámetro $F$, por lo que cualquier combinación puede ser elegida. Los límites o casos especiales están definidos por componentes frecuenciales consecutivas $[0, f_{bw}], [2f_{bw}, 3f_{bw}] \dots [(T_s - 2)f_{bw}, (T_s - 1)f_{bw}]$ y frecuencias intercaladas $[0, (T_s/2)f_{bw}], [f_{bw}, (T_s/2 + 1)f_{bw}] \dots [(T_s/2 - 1)f_{bw}, (T_s - 1)f_{bw}]$. Para cada uno de los conjuntos de vectores anteriores definidos por frecuencias relevantes, se aplica la equivalencia sobre $A$ con valores límite $(min, min)$, $(min, max)$ y $(max, max)$. Se puede definir conjuntos de cobertura similares para estimular implementaciones en raíces diferentes.

Un tercer conjunto de cobertura, *conjunto de señales con frecuencia general*, es definido aplicando equivalencias sobre $N_{fc}$ con $3 \leq N_{fc} \leq T_s - 1$. El número de posibles configuraciones para $F$ es definido por:

$$\binom{T_s}{N_{fc}} = \frac{T_s!}{(T_s - N_{fc})! \times N_{fc}!} \tag{3.3}$$

En este caso, la equivalencia se aplica sobre $F$ y la identificación de valores límite o casos especiales puede ser relevante para poner a prueba la implementación de las etapas de mariposa más profundas que la primera.

Finalmente, el último conjunto definido es la *clase de señales de espectro completo* y corresponde a la selección de todas las posibles frecuencias ($P_n = T_s$). En este caso sólo es posible una configuración de $F$ (todas), y los valores límite de $A$ están definidos por todas las $F$ con $A = min$ y todas las $F$ con $A = max$. Por lo tanto el número de vectores de prueba generados será igual a 3, dos para los valores límite y uno para los equivalentes donde $A[n]$ toma valores intermedios entre $min$ y $max$.

### 3.2.3. Implementaciones a verificar

En esta sección se presentan tres implementaciones diferentes de bloques FFT a verificar con el modelo de cobertura propuesto. La primera implementación corresponde al diseño descrito en [22], implementada en VHDL como el lenguaje de descripción de hardware (HDL). La segunda y tercera implementación de FFT fueron generados por el "Generador de bloques IP DFT/FFT" del proyecto Spiral [23].

#### FFT(256) IIIE-CONICET Agustín

Esta implementación tiene las siguientes especificaciones de caja negra. $T_s = 256$, aritmética de punto fijo de 16 bits (con factor de escala 256), con ingreso de 1 punto complejo por ciclo, orden de los datos: ingreso natural / salida invertida, señal de inicialización (*reset*) con flanco

ascendente y protocolo de entrada/salida: señales específicas de sincronización de entrada/salida (*sync_in, sync_out*) que indican a su flanco ascendente el comienzo de entrada o salida de datos (*data_in, data_out*).

#### FFT(256) Spiral

El segundo bloque FFT fue generado con las mismas especificaciones que el anterior, con el ingreso de datos es de 2 puntos complejos por ciclo, el orden de los datos es ingreso natural / salida natural, y el protocolo de entrada/salida es: señales específicas de sincronización de entrada/salida (*sync_in, sync_out*) que indican a su flanco ascendente que en próximo ciclo de reloj comenzará el ingreso o egreso de los datos (*data_in, data_out*).

#### FFT(1024) Spiral

Una tercera FFT fue generada con diferentes características para probar la flexibilidad del modelo y del ambiente de verificación. Sus especificaciones son: $T_s = 1024$, aritmética de punto fijo de 32 bits (factor de escala 1024), con ingreso de 4 palabras complejas por ciclo, orden de los datos entrada natural/salida natural, y protocolo de entrada/salida igual al bloque FFT(256) Spiral.

## 3.3. Unidad aritmética de punto flotante

### 3.3.1. Introducción

Los bloques utilizados en sistemas de procesamiento digital de señales poseen una gran carga computacional aritmética y su correcto funcionamiento depende fuertemente de la correcta implementación de las operaciones matemáticas básicas para la representación aritmética que se haya seleccionado. La representación aritmética más popular en el diseño VLSI es la aritmética de punto fijo por sus beneficios de velocidad y área, pero cuando se requiere mayor rango dinámico en un sistema, se debe utilizar aritmética de punto flotante.

La verificación de unidades de punto flotante (FPU, por sus siglas en inglés *Floating Point Unit*) es una tarea estudiada y desafiante. La complejidad surge del extenso espacio de prueba el cual incluye una gran cantidad de "casos de borde" difíciles de identificar y que deben ser analizados. La verificación ha sido abordada utilizando métodos formales [24] y simulaciones [25], [26].

En [24] se usa un demostrador de teoremas para confirmar condiciones formales para un diseño de división mediante el método SRT, ambos (diseño y condiciones) son modelados formalmente como un conjunto de relaciones algebraicas. La complejidad algorítmica exponencial

de estas técnicas resultantes del problema de explosión de estados, limita sus aplicaciones a pequeños fragmentos del diseño o a implementaciones específicas.

### 3.3.2. Verificación funcional

Al igual que los casos de estudio anteriores, la verificación funcional basada en simulación de la FPU se basa en su modelo de cobertura. Se utiliza la heurística basada en la división del espacio de cobertura en particiones, casos límite y su implementación como generador de casos de prueba, aplicada a la operación de suma de una unidad de punto flotante con dos entradas $A$ y $B$ y un resultado $F$, donde entradas y salidas son interpretadas en un formato particular: *Binary16*.

En este trabajo sólo se verifica la operación de suma, pero el mismo análisis puede realizarse a las operaciones de resta y multiplicación de cualquier FPU.

#### Especificaciones

El formato Binary16 consiste de números de 16 bits de punto flotante, también conocidos como flotantes de media precisión por la comparación entre éste y las representaciones comúnmente adoptadas: simple y doble precisión (32 y 64 bits). Este formato es útil para aplicaciones que necesitan manejar rangos dinámicos menos exigentes y con un menor costo de memoria que simple y doble precisión, pero las computaciones aritméticas no deben ser críticas dado a que su desventaja es la pérdida de precisión y rango.

El formato Binary16 está definido por el estándar IEEE 754-2008 para operaciones en punto flotante [2]. El formato es el siguiente:

- Bit de signo: 1 bit,
- Ancho de exponente: 5 bits (codificados usando una representación con corrimiento binario (offset), con corrimiento a cero igual a 15),
- Precisión de la mantisa: 11 bits (10 almacenados explícitamente).

La parte fraccionaria tiene diez bits almacenados explícitamente, mas un bit implícito al principio (*hidden bit*) con un valor 1 cuando el exponente es mayor a cero o 0 en otro caso.

El formato es almacenado como un triplete (s,e,m) donde $s$ es el signo, $e$ el exponente y $m$ la mantisa:

$$\overbrace{\underbrace{s}_{\text{signo}}\underbrace{e_4e_3e_2e_1e_0}_{\text{exponente}}\underbrace{m_9m_8...m_1m_0}_{\text{mantisa}}}^{\text{Número Binary16}}.$$

El número real se obtiene según la siguiente función $f$:

$$f(s,e,m) = \begin{cases} 0, & \text{si } e = 0_{10} \wedge m = 0_{10}, \\ (-1)^s 2^{-14}(m_9 2^{-1}) \ldots (m_0 2^{-10}), & \text{si } e = 0_{10} \wedge m \neq 0_{10}, \\ (-1)^s 2^{e-15}(m_9 2^{-1}) \ldots (m_0 2^{-10}), & \text{si } 00001_2 \leq e \leq 11110_2, \\ (-1)^s \infty, & \text{si } e = 11111_2 \wedge m = 0_{10}, \\ f = \text{NaN}, & \text{si } e = 11111_2 \wedge m \neq 0_{10}. \end{cases}$$

El cero tiene doble representación, una positiva y otra negativa. Los casos cuando el *hidden bit* o bit oculto es 1 ($00001 \leq e \leq 11110$) son denominados "números normales" y cuando es cero son llamados "números subnormales".

Otra característica importante es el modo de redondeo implementado por el DUV. En este caso se realiza truncamiento.

### Modelo de cobertura de FPU

El espacio de cobertura de entrada son todas las combinaciones posibles de dos números de 16 bits, por lo que la cardinalidad del conjunto es $2 \times 2^{16} - 1 \simeq 4,29 \times 10^9$ puntos. La interpretación apropiada del formato Binary16 de números normales, subnormales, cero, infinito y NaN (evidenciados en $f(s,e,m)$) provee criterio para identificar las clases de entrada y salida, y dividir el espacio de cobertura en subconjuntos disjuntos. El método de particiones de equivalencia permite inferir que estas definiciones guiarán al diseñador sobre el diferente manejo (y diseño de lógica) al sumar números que pertenezcan al mismo o a distintos subconjuntos; por ejemplo, la lógica al sumar dos normales, o al sumar un subnormal con un normal, o al sumar dos subnormales y el resultado es un normal. Consecuentemente, una entrada y salida de un caso de prueba puede pertenecer a uno de los siguiente subconjuntos (clases): $\pm$ normal, $\pm$ subnormal, cero, $\pm$ infinito, y NaN.

Debido a que el DUV tiene dos entradas, el espacio de entrada puede ser particionado por el producto vectorial (o producto cruz) de los anteriores subconjuntos. Pero también se debe tener en cuenta la funcionalidad del DUV, es decir, la salida como una función de las entradas. El caso de sumar dos subnormales con resultado normal expone la necesidad de añadir las clases de salida debido a que infieren diferentes comportamientos para el mismo par de clases de entrada.

Las clases de salida son consideradas en el producto vectorial de las entradas. Pero hay algunos resultados del producto vectorial que no pueden resultar ya que $F$ es igual a $A$ más $B$, por ejemplo, $A$ y $B$ pertenecen a los números normales positivos y $F$ pertenece a los negativos normales. Las particiones resultantes pueden se listan en la tabla 3.2.

Tabla 3.2: Particiones de equivalencia

| Nro. Partición | A | B | F |
|---|---|---|---|
| 1 | subnormal | subnormal | subnormal |
| 2 | subnormal | subnormal | normal |
| 3 | normal | subnormal | normal |
| 4 | subnormal | normal | normal |
| 5 | normal | normal | normal |
| 6 | normal | normal | +inf |

Esta relación también puede mostrarse en un gráfico de dos dimensiones. Para visualizar las particiones, primero se dibujan los límites de cada clase (cero, ± subnormal, etc.) y para cada variable ($A$, $B$, y $F$) en el plano $A, B$. La Fig. 3.12 muestra estas líneas rectas.

Sobre el gráfico se identifican las regiones válidas, puntos y líneas que representan cada partición. Por ejemplo, la región entre las líneas límite de $A$ igual al mínimo número normal ($A = MIN\_NORM$), $B$ igual al mínimo número normal ($B = MIN\_NORM$) y $F$ igual al máximo número normal ($F = MAX\_NORM$), el punto ($A = 0, B = 0, F = 0$), y la línea $B = 0$ entre los $A$ normales ($A \epsilon Normales$) y $F$ normales ($F \epsilon Normales$) son todas válidas. La posición de la línea $F = MAX\_NORM$ es la representación de precisión infinita de la operación de suma. En este caso esta operación tiene un redondeo que mueve hacia arriba la recta ya que añadir un número $x$ en la vecindad de los máximos normales con un número relativamente pequeño $s$ da como resultado el mismo $x$. Finalmente, al colorear estos espacios, líneas y puntos, y eliminando las líneas de los límites, se obtiene un diagrama más detallado de las clases de equivalencia que puede observarse en la Fig. 3.13.

La Fig. 3.13 representa gráficamente las particiones obtenidas como las equivalencias definidas por el formato Binary16 y la operación de suma del DUV. Los números dentro de las particiones de la Fig. 3.13 corresponden con los valores de la columna “Nro. Partición” de la tabla 3.2.

El próximo paso es seleccionar una muestra de cada partición (puntos en la figura 3.13).

El análisis de valores límite y los casos resultantes son definidos por los límites entre los conjuntos. Los límites pueden inferirse del modelo de cobertura, los cuales son fáciles de identificar gráficamente.

Los límites entre conjuntos consisten en general de un valor como cota superior y otro como inferior. En este caso particular, los límites son regiones formadas por más de un vector de prueba y por la tanto se debe seleccionar una muestra de cada región de valores límite. Por ejemplo, las zonas más oscuras dentro de las equivalencias (rectas azules) en la Fig. 3.14 muestran las

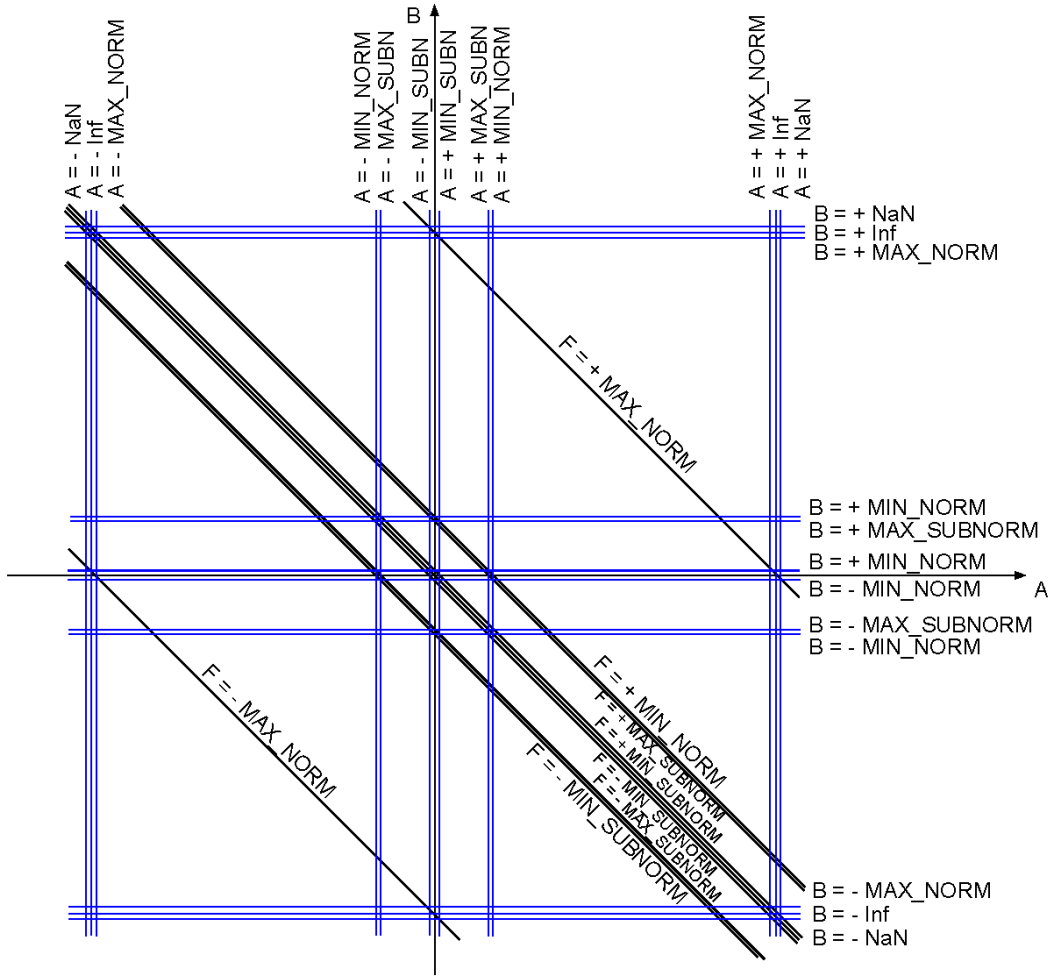


Figura 3.12: Límites inferiores y superiores en el plano $A$,$B$.

regiones de valores límite encontradas para el primer cuadrante.

Como resultado, el modelo de cobertura completo define 105 subconjuntos definidos por particiones de equivalencia y 204 regiones de valores límite añadidas por el análisis de valores límite.

### Detalle del subconjunto en el primer cuadrante

A modo de ejemplo, el primer cuadrante del modelo de cobertura tiene 70 subconjuntos. Particiones de equivalencia define 26 y análisis de valores límite añade 44 subconjuntos, como puede observarse en la Fig. 3.14.

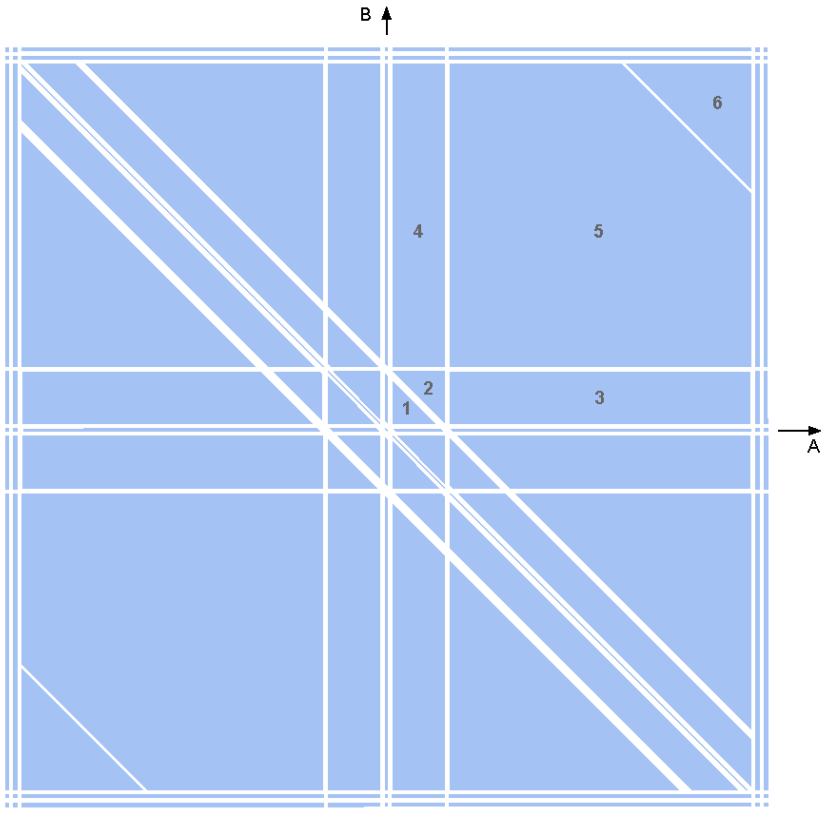


Figura 3.13: Clases de equivalencia.

### Impacto de las clases de equivalencia en el espacio de cobertura

Para comprender el impacto de verificar una muestra de una partición, es importante conocer el número de vectores de prueba representados por un vector de prueba dado (sus equivalentes), y el porcentaje cubierto por cada partición.

La cardinalidad del modelo de cobertura para el primer cuadrante es

$$|CM_{primer\_cuadrante}| = 1\,073\,741\,842$$

Cada $|Particion_i|$ se muestra en la Tabla 3.3. Por lo tanto, un vector $t_i$ cubre el porcentaje definido por la división entre la cardinalidad de la partición y la cardinalidad del espacio de entrada (en este caso de estudio sólo se considera el primer cuadrante).

$$\%Cubierto_{t_i} = \frac{100 \times |Particion_i|}{|CM_{primer\_cuadrante}|}$$

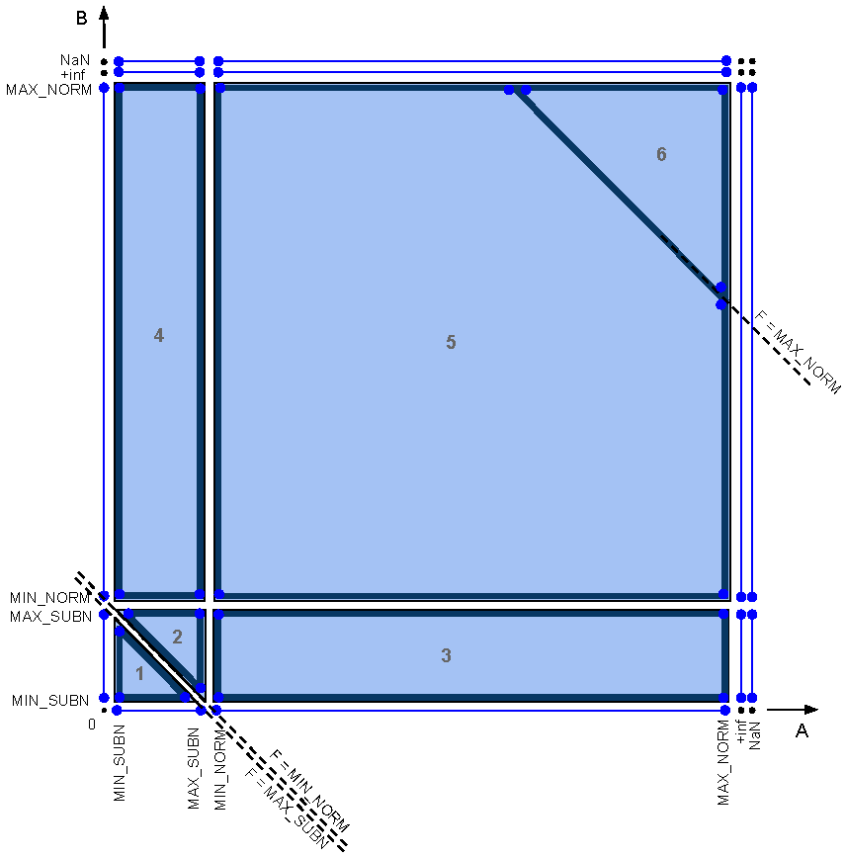


Figura 3.14: Modelo de cobertura para el primer cuadrante.

Este porcentaje es el mismo para todos los vectores equivalentes, y por ende el porcentaje cubierto por cada partición (tercer columna de la Tabla 3.3). La Tabla 3.3 muestra la cardinalidad ($|Part.|$) y el porcentaje que representa del primer cuadrante para los primeros seis subconjuntos (excluyendo el subconjunto del 0).

Tabla 3.3: Cardinalidad de las particiones de equivalencia.

| Nro. Partición | $\|Part.\|$ | Porcentaje representado |
|---|---|---|
| 1 | 522.753 | 0,0487 % |
| 2 | 523.776 | 0,0488 % |
| 3 | 31.426.560 | 2,9268 % |
| 4 | 31.426.560 | 3,9268 % |
| 5 | 939.537.408 | 87,5012 % |
| 6 | 4.180.992 | 0,3894 % |

# Capítulo 4

# Ambientes de verificación implementados

Para llevar a cabo la verificación de los casos de estudio definidos en el Capítulo 3 se construyen programas contenedores de los componentes de verificación, definidos como *ambientes de verificación*.

A continuación se describen los ambientes de verificación específicos para cada caso. Éstos comprenden tanto la verificación funcional basada en simulación como también experiencias en verificación formal realizadas con herramientas analíticas.

## 4.1. Filtros

La verificación funcional para este caso de estudio es abordado con ambos tipos de verificación, tanto basada en simulación como formal.

La verificación basada en simulación consiste en un ambiente desarrollado en el lenguaje SystemVerilog que contiene los componentes necesarios para generar las señales, estimular el DUV, recolectar las respuestas (en aritmética de punto fijo) y comparar con un modelo de referencia generado en este mismo lenguaje (en precisión real).

Por otro lado, la verificación formal se aborda con una herramienta analítica. Esta será al nivel de abstracción que permita la herramienta, dejando a responsabilidad de quien implementa la correcta correspondencia entre el diseño codificado en la herramienta formal y la implementación en HDL.

### 4.1.1. Verificación basada en simulación

La implementación de verificación propuesta comprende un ambiente de verificación en SystemVerilog el cual es responsable de la creación de las señales de prueba, estimulación del DUV y recolección de los resultados. Además contiene un modelo de referencia mediante el cual se indica cúal es la respuesta ideal (en precisión real) al mismo estímulo.

Por otro lado, la comparación entre la salida del DUV y la salida en precisión real se realiza una vez finalizada la simulación mediante un programa matemático. Este permite efectuar diferentes análisis de los resultados tales como cálculo de error y reportes gráficos.

#### Ambiente de verificación

La tecnología adoptada para la construcción del ambiente de verificación es SystemVerilog. Este es un lenguaje de programación específico para la tarea de verificación, es decir, HVL por sus siglas en inglés *Hardware Verification Language*. Permite la codificación de programas con el paradigma orientado a objetos (POO, programación orientada a objetos), y entre sus características se encuentra la facilidad para construir programas en forma modular con componentes reutilizables.

Un ambiente de verificación consta de dos partes principales: el DUV a estimular y la unidad contenedora de los objetos de verificación [27]. Además, debe contar con un componente encargado de la comunicación entre estos últimos dos, llamado *Interfaz*. La Fig. 4.1 muestra dicha arquitectura básica, donde *Top* es la entidad superior que contiene los componentes de verificación, y *Test* es el contenedor de los objetos dinámicos de verificación.

El *Top* consiste en la instanciación del DUV (el filtro digital a verificar), instanciación de la *Interfaz*, instanciación del *Test* y creación del reloj necesario para el funcionamiento del DUV. Una vez realizada esta tarea se da inicio al *Test*, que a su vez crea los componentes:

- *Generador* para la generación de las señales a inyectar,
- *Monitor* para sensar entradas y salidas del DUV, y
- *Report* que reporta en archivos externos las entradas/salidas evidenciadas al DUV y la salida ideal ante las entradas. Para conocer la salida ideal, contiene el *Modelo de Referencia IIR*.

El *Test* comienza con la creación y conexionado de los objetos que contiene. Su primera tarea relacionada a la comunicación con el DUV es la inicialización (*reset*) mediante un método provisto por la interfaz. A continuación lanza el *Monitor* para que comience a sensar las entradas/salidas. Luego se configura el *Generador* (comúnmente denominado *Driver*) indicándole la señal a inyectar y el comienzo de la inyección.

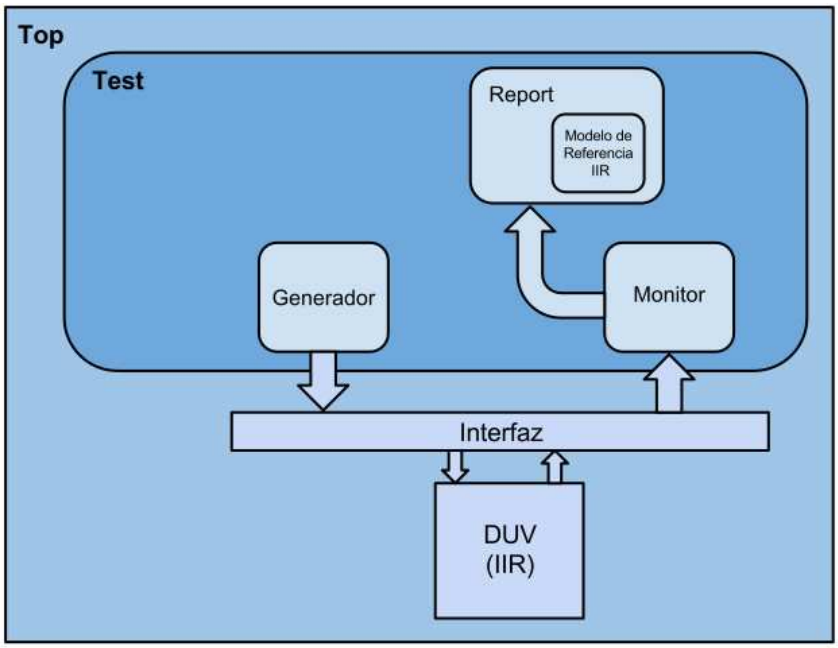


Figura 4.1: Ambiente de verificación de filtros.

Las señales configuradas corresponden a aquellos casos de prueba especificados por el modelo de cobertura de filtros definido en la sección 3.1.1. El *Generador* está provisto de métodos para crear señales sinusoidales discretas, y para configurar dicha señal mediante su frecuencia $f$, su amplitud $A$, frecuencia de muestreo $f_s$ y la cantidad de puntos que se quieren generar $i = \{0, \ldots, cant\_puntos\}$.

Al llamar al método *start* del *Generador*, este genera la i-ésima muestra de la señal en punto flotante, la discretiza, y se inyecta al DUV mediante la interfaz. Es decir:

$$punto\_real_i = A\ sen(2\pi i f / f_s)$$
$$punto\_fix_i = float\_to\_fix(punto\_real_i, BIT\_WIDTH, FRACTIONAL\_BITS),$$

donde $BIT\_WIDTH$ y $FRACTIONAL\_BITS$ son parámetros globales que se deben definir en el paquete ("*package*") del entorno.

Mientras se inyectan las señales, el *Monitor* sensa tanto entradas como salidas que reporta inmediatamente al objeto *Report*. Este último, al recibir una entrada en formato punto fijo, la convierte a real y consulta al modelo de referencia cuál es el resultado ante esa entrada real; estos datos se almacenan internamente y en archivos para ser leídos por herramientas externas. Al finalizar la inyección y recolección se puede solicitar un reporte donde se indica el error absoluto promedio.

La *Interfaz* es el componente clave en la flexibilidad del ambiente de verificación. Al dar soporte a las operaciones de alto nivel de lectura, escritura e inicialización del DUV, los componentes del *Test* se abstraen de su implementación. De esta manera, tanto los casos de estudio generados con Matlab como los generados con Spiral pueden ser verificados con el mismo entorno pero distinta interfaz, debido a que las implementaciones tienen distintas formas de comunicarse con el exterior: por ejemplo, los filtros generados con Matlab tienen señales de sincronización de entradas y salidas mientras, que están ausentes en los generados con Spiral.

Una vez finalizado el proceso de simulación de los casos de prueba, se procede a analizar los datos con un programa matemático, el cual lee los archivos de texto con las entradas y salidas del modelo de referencia y las salidas del DUV. Con estos datos, el programa reporta para cada frecuencia el error obtenido. La elección de medida de error es la raíz del error medio cuadrático o RMSD (del inglés root-mean-square deviation) definido como:

$$RMSD = \sqrt{\frac{\sum_0^{T_s}(out_{DUV} - out_{ref\_model})^2}{A^2}},$$

y grafica las tres señales leídas en dominio tiempo. En la Fig. 4.2 se puede observar dicha dinámica.

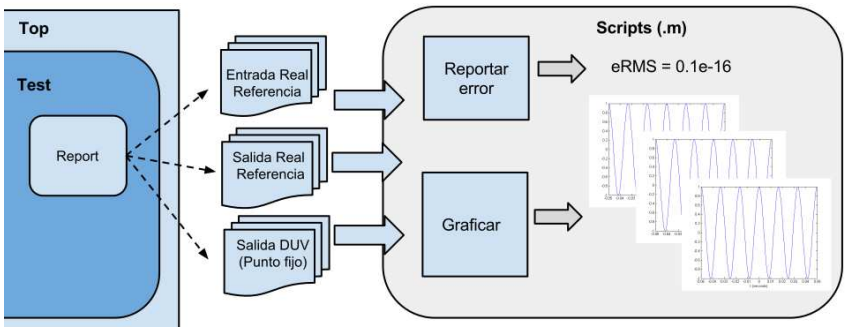


Figura 4.2: Script matemático.

Una alternativa a la verificación basada en simulación con ambientes generados con SystemVerilog, es el conjunto de facilidades que provee la herramienta Simulink de Matlab. Para este caso de estudio, se puede generar un conjunto de pruebas con los bloques básicos que provee la herramienta, tanto de generación de señales de prueba como de análisis de la salida. De todas maneras, esta verificación sería del modelo de referencia (que luego será generado en HDL), pero una característica interesante es la capacidad de conectar (co-simular) con simuladores de HDL como ModelSim. De esta manera, se pueden ingresar los mismos casos de prueba al modelo de referencia y a la implementación en HDL, luego capturar ambos resultados bajo el entorno

Simulink y comparar los resultados.

### Implementación del modelo de cobertura

La finalidad del ambiente de verificación es dar soporte a la simulación de los casos de prueba definidos por el modelo de cobertura. Es por ello que se deben implementar las señales acorde al modelo.

Para tal fin se modelan parámetros que permiten realizar un "barrido de frecuencias" sobre una banda de interés. A su vez, sobre ésta se define una banda interna donde se podrá definir una granularidad de barrido diferente.

Para el caso de un filtro pasa bajos (o pasa altos) el parámetro que actúa como centro de la banda de interés es la freciencia de corte del filtro $f_c$. A partir de este se configura un parámetro $f_{be}$ para indicar los limites de la banda $f_c \pm f_{be}$ y otro $f_{bi}$ para indicar una banda interna $f_c \pm f_{bi}$ (donde $f_{be} > f_{bi}$). Para ambas bandas se debe definir la granularidad del barrido de frecuencias, $f_{gbi}$ para la banda interna y $f_{gbe}$ para el barrido externo.

A partir de los parámetros especificados se comienza a generar los casos de prueba desde la frecuencia mas baja a la frecuencia mas alta, es decir, se comenzara con una señal de $f_c - f_{be}$ y se avanzara de a $f_{gbe}$ hasta llegar a $f_c - f_{bi}$, donde se cambia la granularidad de avance a $f_{gbi}$ hasta llegar a $f_c + f_{bi}$, para luego volver a avanzar de a $f_{gbe}$ hasta el final de las pruebas $(f_c + f_{be})$, como se representa en la Fig. 4.3.

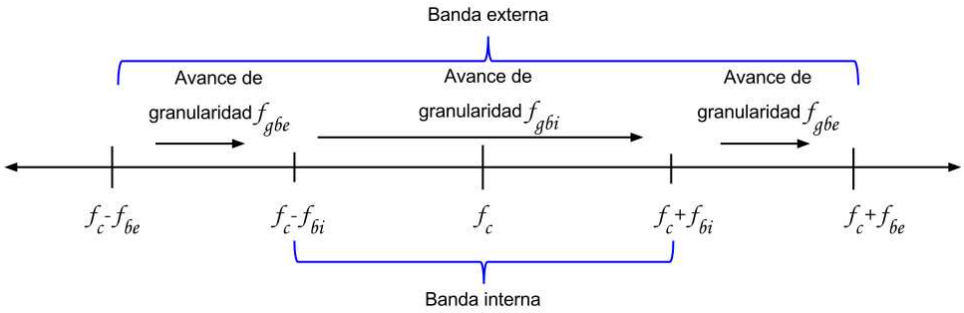


Figura 4.3: Bandas de prueba y granularidad de avance.

De esta manera, al centrar la banda de interés $(f_c)$ en la frecuencia de corte de la banda de paso $(\omega_p)$ y configurando los parámetros de las bandas correctamente, se permite realizar las pruebas sobre las tres bandas especificadas por el modelo de cobertura.

Para los filtros pasabanda ó eliminabanda, se utiliza una representación similar pero basandose en las dos frecuencias esquinas típicas de estos filtros.

Para los casos de estudio abordados, la frecuencia de corte es de $f_c = 10{,}8$ kHz, se opta por definir $f_{be} = 5$ kHz para la banda exterior con una granularidad de $f_{gbe} = 1$ kHz y $f_{bi} = 2$ kHz para la banda interior con granularidad $f_{gbi} = 0,1$ kHz.

Por último, también se debe configurar la amplitud máxima ($A_{max}$) y mínima ($A_{min}$) de las señales, que serán aplicadas a todas las frecuencias. Primero se correrán los casos de señales de amplitud máxima y luego las de amplitud mínima. Las amplitud mínima para todos los casos de prueba es de $A = 10$ mientras que la amplitud máxima depende de cada caso de prueba ya que la aritmética específica de cada uno permite aplicar amplitudes máximas diferentes determinadas por la cantidad de bits destinados a la parte entera de la representación de punto fijo.

### 4.1.2. Verificación formal

La verificación formal se realiza mediante la herramienta de prueba de propiedades aritméticas *Gappa*. Esta permite modelar los filtros digitales en forma ideal y la implementada en punto fijo, para poder consultar propiedades de cada uno de los modelos y entre estos dos.

#### Secciones de un programa Gappa

Un programa de Gappa contiene tres partes. La primera se usa para declarar expresiones con el objetivo de simplificar la escritura de las partes siguientes. La segunda es la fórmula lógica que el usuario quiere probar con la herramienta. La tercera son ayudas o "hints" a Gappa sobre cómo realizar la prueba.

La sección de definiciones contienen las operaciones de redondeo que serán usadas más de una vez y expresiones matemáticas de sub términos, es decir resultados intermedios y final.

Luego se debe generar la fórmula lógica. Esta es la parte fundamental del programa, contiene la fórmula lógica que se espera que Gappa pruebe. Esta es escrita entre corchetes y puede contener implicaciones, disyunciones, conjunciones, y negaciones de las propiedades de expresiones matemáticas. Las propiedades pueden ser intervalos, desigualdades, errores relativos, igualdades, y expresiones de precisión. Cualquier identificador sin definición es asumido como cuantificado universalmente sobre el conjunto de los números reales.

La propiedad que se busca probar definir estará relacionada en este caso con:

- rangos o expresiones para prevenir desbordes (*overflow*).
- rangos de error absolutos y/o relativos para caracterizar la precisión de los resultados

Por ultimo, el programa se finaliza con ayudas o *hints*. Éstas se presentan a la herramienta cuando no es capaz de probar la fórmula lógica por sí sola, y pueden ser reglas de reescritura o directivas de bisección.

Código 4.1: Definiciones para `Forma Directa I` .

```
# DEFINICIONES
@rndf = fixed<-16,dn>; # punto fijo lsb = 2^-16, redondeo a menos infinito

# coeficientes reales
b0_full = 0.130063011757311;
b1_full = 0.390189035271933;
b2_full = 0.190189035271933;

a0 = 0.130063011757311;
a1 = 0.390189035271933;
a2 = 0.190189035271933;

# entradas al sistema
x0 = rndf(xx0); # x_i son numeros en aritmetica rnd
x1 = rndf(xx1);
x2 = rndf(xx2);
```

La limitación de la herramienta (mencionada en el Capítulo 2), no permite la verificación de una ejecución infinita de los filtros IIR, por lo que para estos filtros se verificará solo una determinada cantidad de iteraciones.

A modo de ejemplo, a continuación se define un programa de Gappa para verificar una iteración de un filtro IIR forma directa I de orden 2, con el objetivo de introducir al lector a la dinámica de verificación para luego poder inferir la aplicación a los casos de estudio definidos.

Los scripts de verificación formal de los casos de estudio pueden encontrarse en el Apéndice A.

### Ejemplo Forma Directa I

El primer paso es la codificación de la sección de definiciones. En ella se definen los valores de los coeficientes cuantizados y reales, el formato de los parámetros de entrada y un alias de la aritmética a utilizar. Esta última, cuando es en punto fijo, se especifica mediante dos parametros: el peso del bit menos significativo y el modo de redondeo. En Código 4.1 se encuentra dicha definición.

A continuación, también en la sección de definiciones, se especifican las computaciones a realizar por el sistema, tanto en la aritmética implementada como en la real. Se definen paso a paso con el objetivo de indicar cada operación realizada por el sistema, donde quedan explicitadas las operaciones de redondeo realizadas por la aritmética.

La operaciones de filtrado IIR (forma directa I) para los tres primeros valores ($x_0$, $x_1$ y $x_2$) pueden encontrarse en Código 4.2.

Por último se especifica la propiedad lógica, en la que se indica los rangos de la entrada y se consulta por los rangos de la salida en este caso de estudio. Se debe indicar cúal es el rango

Código 4.2: Definicion de operaciones `Forma Directa I` .

```
# IIR DF1
# Resultado real
 z0 = x0*b0;
 z1_aux = x0*b1 + z0*(-a1);
 z1 = x1*b0 + z1_aux;
 z2_sub1 = x0*b2 + z0*(-a2);
 z2_sub2 = z2_sub1 + x1*b1 + z1*(-a1);
 z2 = z2_sub2 + x2*b0;
 z = z2;

# Resultado en aritmetica rndf
 y0 rndf= x0*b0_quant;
 y1_aux rndf= x0*b1_quant + y0*(-a1_quant);
 y1 rndf= x1*b0_quant + y1_aux;
 y2_sub1 rndf= x0*b2_quant + y0*(-a2_quant);
 y2_sub2 rndf= y2_sub1 + x1*b1_quant + y1*(-a1_quant);
 y2 rndf= y2_sub2 + x2*b0_quant;
 y rndf= y2;
```

Código 4.3: Propiedad lógica `Forma Directa I` .

```
 # PROPIEDAD LOGICA
{   x0 in [-MAX_PRES, MAX_PRES-1] /\    # Rango de las entradas
    x1 in [-MAX_PRES, MAX_PRES-1] /\
    x2 in [-MAX_PRES, MAX_PRES-1]
        ->
     y in ? /\   # Rango de la salida fxp?
            z in ? /\   # Rango de la salida real?
            z - y in ?  # Rango del error?
}
```

de las entradas. En este ejemplo se deja genérico representando el menor y mayor número que permite el formato numérico seleccionado $[-MAX\_PRES, MAX\_PRES - 1]$ (Código 4.3).

El programa completo, acorde a las implementaciones de filtros IIR contempla las cuatro etapas y especifica siete ingresos de datos, a diferencia del programa simplificado que es de tres etapas y con tres datos de entrada. El *script* completo puede encontrarse en el Apéndice A, donde puede observarse una de las configuraciones de punto fijo seleccionada; las demás han sido comentadas. Solo se debe cambiar esas líneas y el rango $[-MAX\_PRES, MAX\_PRES-1]$ para probar las distintas configuraciones de punto fijo.

## 4.2. Transformada rápida de Fourier (FFT)

El ambiente de verificación de FFT reusa la estructura básica de verificación de filtros digitales, adaptada a la verificación específica de FFT con su modelo de cobertura asociado. Esto se debe a la facilidad que provee un diseño e implementación en el lenguaje orientado a objetos SystemVerilog.

El ambiente de verificación de FFT implica dos componentes principales: (1) un ambiente en lenguaje SystemVerilog que incluye los recursos para generar casos de prueba, estimular el DUV y recolectar el resultado de las simulaciones; y (2) un modelo de referencia implementado en Matlab que computa la FFT en precisión infinita y compara este resultado con el obtenido con el DUV.

### 4.2.1. Ambiente de verificación

Al igual que el caso de estudio anterior, SystemVerilog es la tecnología adoptada para desarrollar el ambiente de FFT. La Fig. 4.4 ilustra la arquitectura del ambiente para el componente de verificación de caja negra para FFT. El módulo *Top* es la entidad principal del programa de verificación. Ésta instancia el bloque FFT bajo prueba y los recursos de verificación funcional, los conecta por medio de un objeto de interfaz y define la señal de reloj.

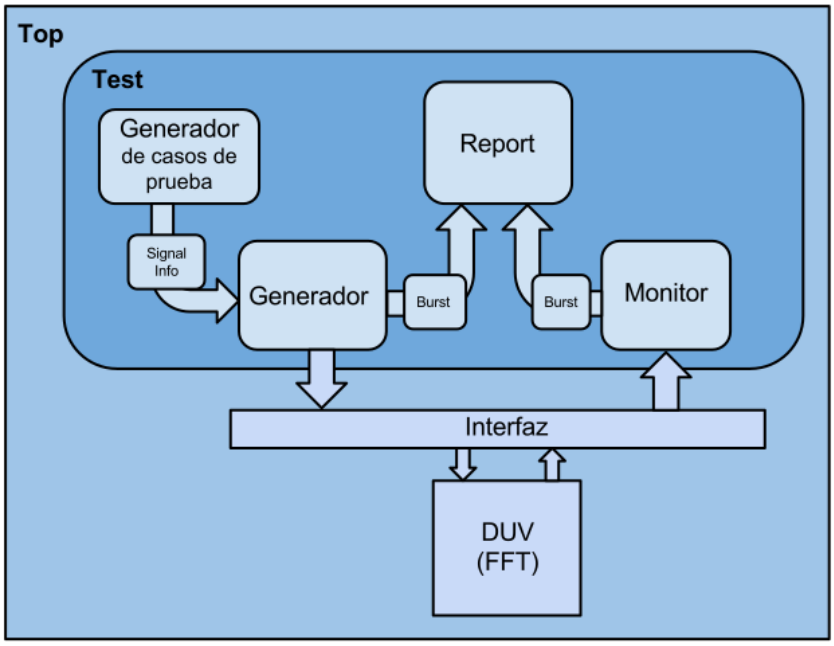


Figura 4.4: Ambiente de verificación FFT.

La entidad *Interfaz* facilita la comunicación entre unidades, definiendo la estructura asociada a las señales de entrada/salida y elevando el nivel de abstracción para comunicarse entre ellos; para este último propósito, los métodos tales como “reset”, “enviar_frame” o “recibir_frame” permiten a estimular el diseño y recolectar las entradas sin estar pendiente de los detalles a nivel de señales o sincronizaciones.

Los recursos de prueba involucran un *Generador de casos de prueba*, un *Generador*, un

*Monitor* y una unidad *Report*.

El generador provee tareas para generar señales de entrada en relación a los conjuntos de cobertura definidos en la sección 3.2.2. Cada caso de prueba es modelado por el objeto *Signal Info* el cual contiene la información requerida por el *Generador* para sintetizar una señal discreta de entrada. Por ejemplo, en la Fig. 4.5, el método "test_all_bins" crea una secuencia de instancias de *Signal Info* del "conjunto de señales con una componente frecuencial" y lo envía al *Generador*. Las líneas de código entre 17 y 21 crean estos casos de prueba y definen sus parámetros.

```
10 // Test 1: 256 burst. One frequency signals, frequency = bin_i.
11 task testAllBins();
12         SignalInfo input_signal;
13         // foreach bin
14         for (int i = 0; i < FFT_SIZE; i++)
15           begin
16                 // create a Signal Info object
17                 input_signal = new();
18                 // Set input signal information
19                 input_signal.setNumFreqs(1);
20                 input_signal.setFreqComp(0,BIN_WIDTH*i);
21                 input_signal.setAmplitudComp(0, FFT_MAX_AMP / 2);
22                 // Drive signals to the DUT
23                 fft_driver.driveSignal(input_signal);
24           end
25 endtask
```

Figura 4.5: Código SystemVerilog para verificar cada bin.

Los parámetros de *Signal Info* pueden ser configurados con valores específicos o pueden ser aleatorios usando las funciones de aleatorización nativas de SystemVerilog; esta característica es de ayuda para generar múltiples muestras para un dado subconjunto de equivalencias del modelo de cobertura. La aleatorización se aplica, por ejemplo, en el "conjunto general de componentes frecuenciales" para definir valores de frecuencia o amplitud.

El Driver toma cada ítem *Signal Info*, genera la correspondiente señal tal como lo requiere el DUV y lo estimula a través de los métodos provistos por la interfaz. La correspondencia que debe hacer *Signal Info* involucra la evaluación de las $T_s$ muestras de la señal y su escalado a la precisión $D_p$ requerida por el bloque FFT, y la evaluación en punto flotante para luego ser usado por el modelo de referencia.

Ambos *Generador* y *Monitor* usan el objeto *Burst* para enviar los valores al objeto *Report*. Pero los tipos de datos de puntos complejos están en diferentes formatos: la entrada está en punto flotante y la salida debe ser el del formato del DUV, por ejemplo punto fijo de 32 bits. Este comportamiento es modelado por medio de la herencia y el polimorfismo en programación orientada a objetos (POO); en este caso, la clase *Burst* contiene una colección de objetos *Complex Point*, que es una clase abstracta implementada por *Real Complex Point* (para modelar entradas

de 32 bits en punto flotante) y *Quant Complex Point* (para modelar las salidas del DUV). La herencia en *Complex Point* y el polimorfismo en *Burst* y el diseño de *Complex Point* permiten al objeto *Burst* contener una colección de tipo de dato de entrada o salida en tiempo de ejecución. La Fig. 4.6 ilustra este concepto de diseño en lenguaje UML (por sus siglas en inglés, Unified Modeling Language).

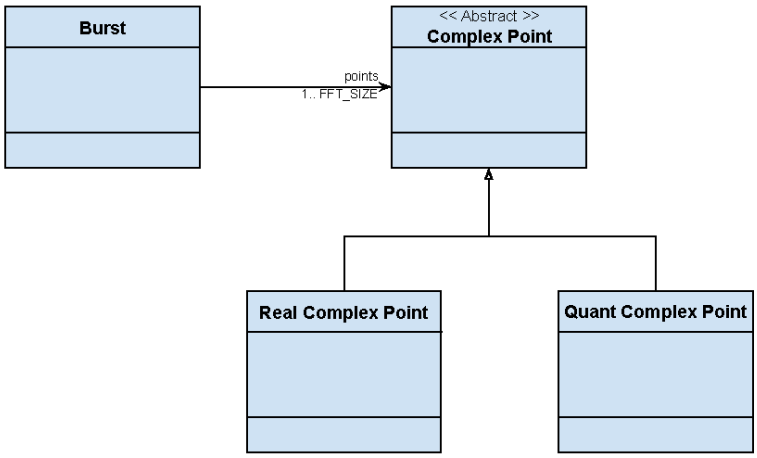


Figura 4.6: UML de Burst y Complex Point.

Las restricciones de desempeño son calculadas por el *Monitor* solo en el primer registro de ráfagas de entrada y salida como el número de ciclos de reloj entre el flanco positivo de las señales "sync_in" y "sync_out". El *Monitor* sensa los flancos positivos de reloj y los ciclos mediante la *Interfaz*, ya que es solo visible a través de ésta. El resultado es enviado al objeto *Report* que mostrará el resultado por consola.

### 4.2.2. Programa de comparación en Matlab

Una vez que finaliza la etapa de simulación, todos los archivos con las ráfagas de entrada y salida son procesados por el modelo de referencia. El programa de Matlab lee los archivos de ráfagas de entrada, calcula la FFT con precisión infinita y compara los resultados con los archivos de ráfagas de salida. El modelo de referencia está basado en una implementación de FFT exacta y probada. Al igual que el caso de estudio de filtros, la comparación se hace por medio del cálculo de la medida de error entre los resultados del DUV y del modelo de referencia, con la diferencia de que en éste caso se la divide por el tamaño de la transformada $T_s$ en vez de la amplitud de la señal $A$, ya que las sinusoidales que componen las señales de prueba tienen

amplitudes distintas. La elección de medida de error es el error medio cuadrático o RMSD (del inglés root-mean-square deviation) definido como:

$$RMSD = \sqrt{\frac{\sum_0^{T_s}(out_{DUV} - out_{ref\_model})^2}{T_s}}$$

Además, el *script* permite seleccionar entre dos funcionalidades: el análisis de error de todos los casos o mostrar las gráficas de la señal de entrada, la magnitud resultante de la FFT del DUV y la magnitud del modelo de referencia para un determinado caso de prueba. Estas gráficas permiten permiten realizar la verificación mediante inspección de la salida del DUV y el resultado esperado para este caso de prueba particular.

La Fig. 4.7 ilustra el procedimiento para finalmente corroborar el resultado con el modelo de referencia.

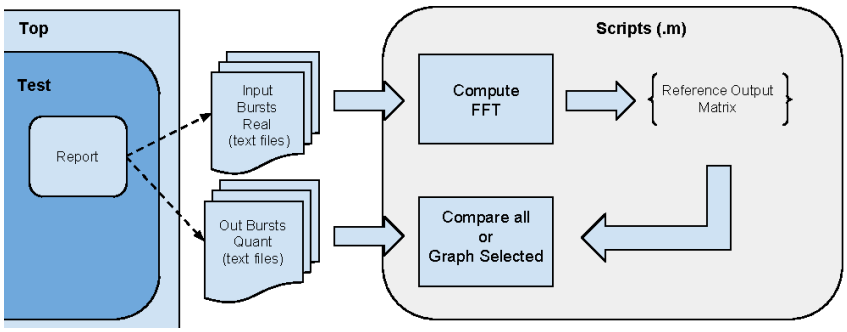


Figura 4.7: Programa matemático.

Cuando se ejecuta el programa para calcular RMSD de todas las señales, se debe especificar un límite superior para que solo se muestren aquellos casos que exceden este valor y por lo tanto considerarlos como una falla. Entonces, si se desea, el ingeniero en verificación puede correr el programa de inspección para aquellos casos que han fallado.

### 4.2.3. Configuración y mecánica de la verificación

La verificación de un bloque FFT bajo el ambiente propuesto requiere su configuración e implementación de la interfaz. Los parámetros de la configuración están agrupados en un paquete de SystemVerilog, que contiene valores específicos para la implementación particular de la FFT así como los parámetros relacionados a la especificación del DUV (restricciones de tiempo/desempeño, formato de entrada/salida y límite superior de error deseado) y también el rango de valores de amplitud considerados como mínimo y máximo.

## 4.3. Unidad de punto flotante para formato Binary16

El ambiente de verificación de la unidad aritmética de punto flotante FPU es diferente a las descritas anteriormente, debido a la naturaleza de los estímulos que reciben: los anteriores reciben señales y la FPU recibe operandos. De todas maneras los ambientes comparten la estructura jerárquica básica *Top-Test*, *Interfaz* y DUV.

El objetivo de esta implementación de ambiente de verificación es el uso del modelo de cobertura (definido en la Sección 3.3.2) como generador de vectores de prueba. Por simplicidad, solo se implementa la generación de vectores de prueba para el primer cuadrante del modelo de cobertura, pero la misma lógica puede ser aplicada a los tres restantes.

### 4.3.1. Ambiente de verificación y generación de vectores de prueba

El ambiente de verificación se centra en la generación de los casos de prueba, e incluye los recursos para generar los vectores de prueba en forma aleatoria (dirigido por el modelo de cobertura) y resolver las restricciones (*constraints*) asociadas. Además se cuenta con la comparación con el modelo de referencia (Fig. 4.8).

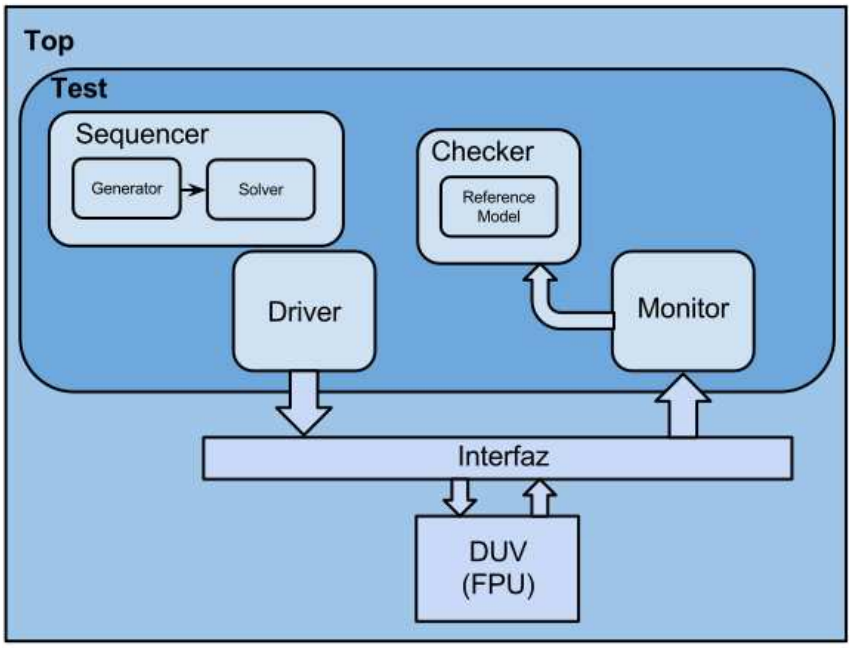


Figura 4.8: Arquitectura del ambiente de verificación.

La Fig. 4.8 ilustra la arquitectura del ambiente de verificación para un componente de verificación tipo caja negra. El módulo *Top* es la entidad principal del programa de verificación. Éste

instancia a la unidad de punto flotante (FPU) a verificar y los recursos de verificación funcional, los conecta por medio de un objeto de interfaz y define la señal de reloj.

Los recursos de verificación para implementar el modelo de cobertura como un generador de vectores de prueba involucran componentes de secuenciador (*Sequencer*), generador (*Generator*) y un asistente de generación (*Solver*).

Cada subconjunto de equivalencia tiene un número de conjunto asociado en el modelo. El secuenciador provee métodos para obtener los vectores de prueba, de menor a mayor en número de subconjunto. Cuando un vector de entrada se solicita al secuenciador, este pide al generador un vector que pertenezca a un dado número de subconjunto. Luego, el generador habilita las restricciones necesarias del asistente de generación para obtener un vector aleatorio dentro del subconjunto deseado, aleatoriza los valores y obtiene el vector resultante.

El asistente de generación (*Solver*) contiene las variables de tipo aleatorias y todas las restricciones necesarias para generar los casos de prueba. Debido a que todas las restricciones están agrupadas en este objeto, el generador debe habilitar y deshabilitar las restricciones apropiadas para obtener el resultado deseado.

La aleatorización para este formato numérico específico expone un necesidad particular para resolverlos. SystemVerilog provee constructores para aleatorización y definición de restricciones solo para los tipos de datos básicos, debido a que las operaciones (matemáticas y lógicas) que fijan los límites de las restricciones son definidos únicamente por estos tipos. El caso de estudio abordado, una unidad aritmética de punto flotante (FPU) en formato Binary16, tiene una representación especial de los números no soportada por este lenguaje, y mucho menos sus operaciones. Por ejemplo, en el caso de $A$,$B$ (entradas) y $F$ (salida) pertenecen a los números normales, sería ideal poder escribir el código de la fig. 4.9.

```
rand float16 A, B, F;

constraint ABFnormals{
  A >= MIN_NORM; A <= MAX_NORM;
  B >= MIN_NORM; B <= MAX_NORM;
  F >= MIN_NORM; F <= MAX_NORM;
  F = A + B;
}
```

Figura 4.9: Restricciones ideales.

La Fig. 4.9 cuenta con tres variables $A$, $B$ y $F$ con valores aleatorios Binary16, y con una sentencia de restricción donde $MIN\_NORM$ y $MAX\_NORM$ son los límites superior e inferior de los números normales (en formato Binary16), y la operación de suma definida para el formato Binary16.

Pero esto no es posible ya que el tipo de dato Binary16, la comparación ("mayor o igual" y "menor o igual"), y la operación de suma no están soportadas para el formato de punto flotante del lenguaje.

La solución propuesta es tratar a $A$, $B$ y $F$ como enteros de 16 bits. Esto permite resolver la generación aleatoria para una de las entradas (por ejemplo, $A$), y el resultado ($F$) para el primer cuadrante del modelo, con la restricción de que la entrada debe ser menor al resultado ($A$ menor que $F$). Luego, la otra entrada ($B$) puede ser calculada mediante la resta en Binary16 del resultado y la otra entrada ($B = F - A$), en Binary16). Por ende, el código para la generación de $A$, $B$ y $F$ normales resulta en la figura de código que se muestra en la Fig. 4.10.

```
rand bit[15:0] A_randomized;
rand bit[15:0] F_randomized;

constraint ABFnormals{
  A >= MIN_NORM; A < MAX_NORM;
  F >= MIN_NORM_X2; F <= MAX_NORM;
  A < F;
}

function getABforSusetN(output bit[15:0] A_res,B_res);
  A_res = A_randomized;
  F_res = F_randomized;
  subtractHalfPrecisionNum(F,A,B_res);
endfunction
```

Figura 4.10: Restricciones implementadas y función para obtener A y B.

# Capítulo 5

# Resultados

Los modelos de cobertura y ambientes de verificación de las secciones 3 y 4 definen los casos de prueba para estimular los DUVs. En este capítulo se presentan los resultados obtenidos de las verificaciones realizadas para los tres casos de estudio.

## 5.1. Filtros

Los resultados de las pruebas se presentan en tres partes. La primera corresponde a aquellos obtenidos por la simulación del ambiente en SystemVerilog, la segunda a los resultados de los programas de la herramienta Gappa (métodos formales), y la última corresponde a un análisis comparativo de los resultados de ambas estrategias de verificación.

### 5.1.1. Verificación funcional basada en simulación

La configuración del ambiente de filtros digitales ha resultado en la simulación de 46 señales de prueba. En las Tablas 5.1 y 5.2 se puede observar la frecuencia que ha resultado en la mayor diferencia obtenida para cada implementación para cada amplitud máxima y mínima. Los tablas de los resultados completos pueden encontrarse en el Apéndice C.

Las tablas muestran que el mayor error se produce para frecuencias próximas a la frecuencia de corte, pero el error relativo es suficientemente pequeño, lo que permite concluir que las implementaciones satisfacen los criterios de prueba. Naturalmente el error relativo disminuye para las implementaciones numéricas de mayor precisión.

En las Figuras 5.1 y 5.2 se encuentran capturas de la simulación correspondientes a la simulación de un bloque generado con Simulink y al generado con Spiral, respectivamente.

Con respecto al reporte de cobertura de código obtenido por la herramienta de simulación, la Tabla 5.3 muestra el porcentaje cubierto por sentencia (*statement*), rama (*branch*) y cambios

Tabla 5.1: Máximo Error para las distintas implmentaciones de filtros IIR en Simulink

| Filtro | Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|---|
| ML-8 | 10 | 12,7 | $3,80 \times 10^{-2}$ | $3,80 \times 10^{-3}$ |
| | $2^{23} - 1$ | 12 | $2,67 \times 10^{4}$ | $3,18 \times 10^{-3}$ |
| ML-12 | 10 | 10,5 | $3,31 \times 10^{-3}$ | $3,31 \times 10^{-4}$ |
| | $2^{19} - 1$ | 11,5 | $1,56 \times 10^{2}$ | $2,97 \times 10^{-4}$ |
| ML-16 | 10 | 9,5 | $1,79 \times 10^{-4}$ | $1,79 \times 10^{-5}$ |
| | 32767 | 9,5 | $5,63 \times 10^{-1}$ | $1,72 \times 10^{-5}$ |
| ML-20 | 10 | 10,7 | $9,69 \times 10^{-6}$ | $9,68 \times 10^{-7}$ |
| | 2047 | 9,9 | $0,02 \times 10^{-1}$ | $9,77 \times 10^{-7}$ |
| ML-24 | 10 | 12,7 | $6,24 \times 10^{-7}$ | $6,24 \times 10^{-8}$ |
| | 127 | 12,5 | $7,07 \times 10^{-6}$ | $5,57 \times 10^{-8}$ |

Tabla 5.2: Máximo Error para las distintas implmentaciones de filtros IIR en Spiral

| Filtro | Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|---|
| Sp-8 | 10 | 10,5 | $1,04 \times 10^{-1}$ | $10,4 \times 10^{-3}$ |
| | $2^{23} - 1$ | 12 | $2,67 \times 10^{4}$ | $3,18 \times 10^{-3}$ |
| Sp-12 | 10 | 12,7 | $3,23 \times 10^{-3}$ | $3,23 \times 10^{-4}$ |
| | $2^{19} - 1$ | 5,8 | $1,45 \times 10^{2}$ | $2,76 \times 10^{-4}$ |
| Sp-16 | 10 | 5,8 | $2,14 \times 10^{-4}$ | $2,14 \times 10^{-5}$ |
| | 32767 | 5,8 | $8,86 \times 10^{-1}$ | $2,72 \times 10^{-5}$ |
| Sp-20 | 10 | 5,8 | $1,89 \times 10^{-5}$ | $1,89 \times 10^{-6}$ |
| | 2047 | 5,8 | $4,84 \times 10^{-3}$ | $2,36 \times 10^{-6}$ |
| Sp-24 | 10 | 5,8 | $1,11 \times 10^{-6}$ | $1,11 \times 10^{-7}$ |
| | 127 | 5,8 | $1,61 \times 10^{-5}$ | $1,27 \times 10^{-7}$ |

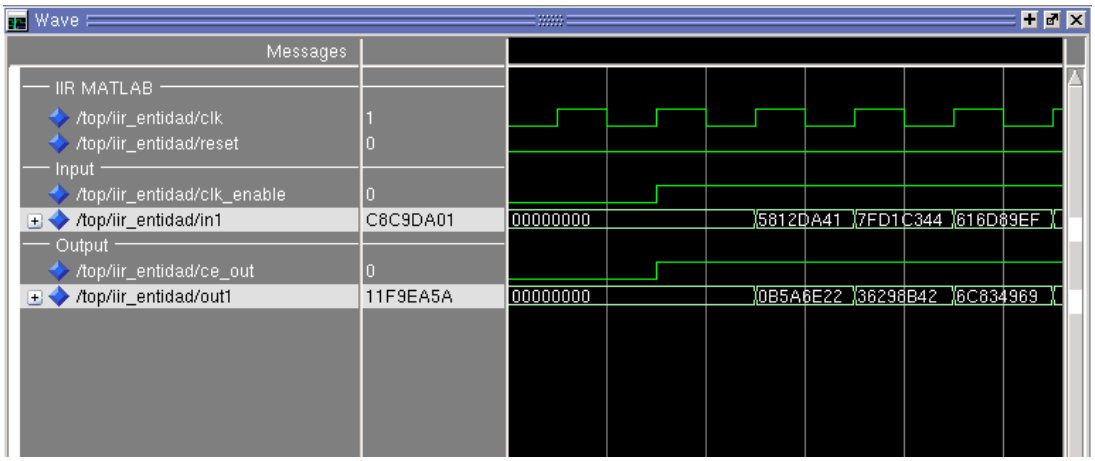


Figura 5.1: Simulación eb ModelSim del filtro IIR generado en Simulink.

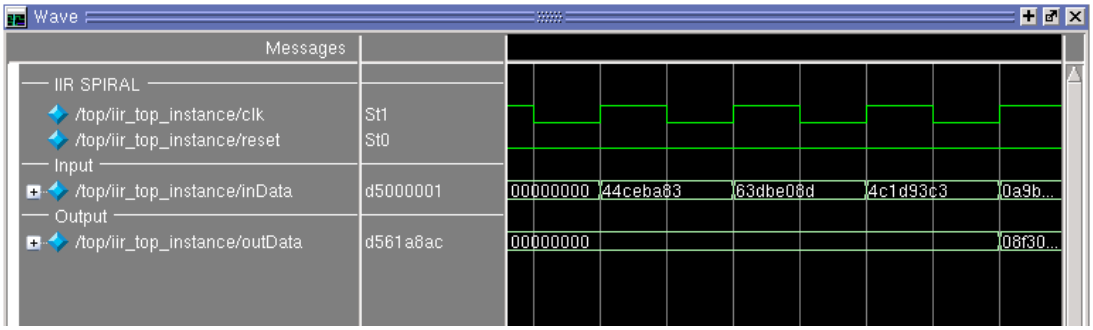


Figura 5.2: Simulación eb ModelSim del filtro IIR generado en Spiral.

(*toggle*) para cada bloque. No se reporta la cobertura para cada amplitud por separado ya que han resultado en la misma cobertura. Lo mismo sucede con los distintos bloques Spiral.

Tabla 5.3: Reporte de Cobertura de Código de las distintas implementaciones de filtros IIR en Simulink y Spiral

| Filtro | Sentencia (%) | Rama (%) | Toggle (%) |
|---|---|---|---|
| ML-8 | 100 | 100 | 89,3 |
| ML-12 | 100 | 100 | 89,4 |
| ML-16 | 100 | 100 | 88,7 |
| ML-20 | 100 | 100 | 89,7 |
| ML-24 | 100 | 100 | 89,2 |
| Sp-X | 100 | 100 | 99,5 |

### 5.1.2. Verificación formal

Las propiedades aritméticas de los filtros se realizaron con la herramienta Gappa y los programas del Apéndice A, los cuales fueron introducidos en la sección 4.1.2.

Para poder realizar estas pruebas se debe tener visibilidad del diseño o implementación ya que se deben conocer todas las operaciones que se realizan para que la herramienta pueda analizar el rango de las salidas a partir del rango de las entradas. Para este caso de estudio, se define la misma secuencia de operaciones, con aritmética variable en la configuración de cantidad de bits.

#### Rangos obtenidos

En los programas generados se define la máxima precisión (el valor del bit menos significativo) y el rango de las entradas, ambas son determinadas por la configuración de punto fijo. Por ejemplo, la configuración de 8 bits en total de los cuales 1 es destinado a la parte entera y el resto fraccionaria implica que la máxima precisión es $2^{-7}$ y el rango de entradas (en complemento a dos) es $[-1\ ,\ 1-2^{-7}] = [-1\ ,\ 0,999969482]$, con estos datos el programa permite obtener el rango de la salida con máxima precisión $2^{-7}$ y el rango de la diferencia entre la salida con la precisión $2^{-7}$ y la precisión real.

Por simplicidad, se establece una convención para notar las configuraciones a verificar. Ésta está formada por la cantidad de bits total, seguido por la cantidad de bits de la parte entera "punto" la cantidad de bits de parte fraccionaria entre paréntesis. En el caso del ejemplo anterior de 8 bits, la notación correspondiente es *8(1.7)*.

Entonces, se definen seis configuraciones a verificar: *8(1.7)*, *8(2.6)*, *16(1.15)*, *16(2.14)*, *32(1.31)* y *32(2.30)*.

Gappa obtuvo en todos los casos los rangos solicitados de la salida real (*Salida (Real)*), la salida en la configuración en punto fijo (*Salida (Fxp)*) y el rango de la diferencia entre las anteriores (*Error*). En la Tabla 5.4 se muestran los rangos obtenidos.

Tabla 5.4: Rangos para distintas configuraciones de punto fijo para un IIR DFI

| Conf. | Rango Entrada | Rango Salida Real | Rango Salida (Fxp) | Rango Error |
|---|---|---|---|---|
| 8(1.7) | $[-1; 0,9921875]$ | $[-2,54502; 2,53711]$ | $[-2,55469; 2,50781]$ | $[-0,03977; 0,0923655]$ |
| 8(2.6) | $[-2; 1,984375]$ | $[-5,09003; 5,07423]$ | $[-5,10938; 5,01562]$ | $[-0,0795399; 0,184731]$ |
| 16(1.15) | $[-1; 0,999969482]$ | $[-2,55101; 2,55097]$ | $[-2,55106; 2,55084]$ | $[-0,000155351; 0,000360803]$ |
| 16(2.14) | $[-2; 1,999938965]$ | $[-5,10201; 5,10195]$ | $[-5,10211; 5,10168]$ | $[-0,000310703; 0,000721606]$ |
| 32(1.31) | $[-1; 0,999999999534339]$ | $[-2,55103; 2,55103]$ | $[-2,55103; 2,55103]$ | $[-2,37047E-9; 5,50541E-9]$ |
| 32(2.30) | $[-2; 1,99999999906868]$ | $[-5,10206; 5,10206]$ | $[-5,10206; 5,10206]$ | $[-4,74095E-9; 1,10108E-8]$ |

Tal como se espera, en la Tabla 5.4 se puede observar que a medida que se aumenta la cantidad de bits de la parte fraccionaria el rango de *Salida Fxp* se asemeja más al rango de la *Salida Real*, y el rango del error disminuye.

Otro resultado notorio es que el rango de salida (ya sea real o Fxp) al menos duplica el rango de la entrada, por lo que se evidencia que se necesitan más bits para representar la salida o adoptar una estrategia diferente de escalado. Por ejemplo, es el caso de la configuración *16(1.15)* con rango de entrada $[-1; 0,999969482]$ y $[-2,55106; 2,55084]$ de salida (Fxp) el cuál necesita 3 bits para representar la parte entera a diferencia de 1 bit de parte entera de la entrada. Entonces, siguiendo el ejemplo, para un filtro IIR implementado con ésta estructura y con la misma configuración de punto fijo tanto de entrada como de salida, se puede encontrar un conjunto de entradas que produzca un desborde, como es el caso de ingresar los tres primeros valores con el máximo positivo $x[0] = x[1] = x[2] = 1 - 2^{(-15)} = 0,999969482$ que resulta en la segunda salida con el valor $y[2] = 1,0239$ que sobrepasa el límite superior representable por *16(1.15)*.

Por otro lado, estos mismos rangos pueden obtenerse analíticamente, es decir en forma manual o con una herramienta matemática como *Matlab/Octave/Mathematica*. Se puede desarrollar la función para las primeras 7 entradas. La función completa de modo tal que solo quede en función se expone en el Código 5.1.

En forma resumida, reemplazando los valores de los coeficientes, se obtiene el Código 5.2.

Al reemplazar $x_i$ por el valor mínimo y máximo según corresponda, por ejemplo $-1$ y $1-2^{-15}$ correspondiente al formato *16(1.15)*, se obtiene como resultado mínimo el valor $1,38112139137878$ y como máximo $-1,38115225902710$. Pero aún no es equivalente al valor obtenido por Gappa

Código 5.1: Función para 7 valores de entrada del IIR.

```
# Ecuacion Completa de y7
  y7_completa = -a1*a1*a1*a1*a1*a1*a1*b0*x0 + a3*a3*b1*x0 + a3*a3*b0*x1 - a3*b3*x1 +
    a1*a1*a1*a1*a1*a1*(b1*x0 + b0*x1) - a2*a2*a2*(b1*x0 + b0*x1) - a3*b2*x2 +
    a1*a1*a1*a1*a1*(6*a2*b0*x0 - b2*x0 - b1*x1 - b0*x2) - a3*b1*x3 +
    a2*a2*(-3*a3*b0*x0 + b3*x0 + b2*x1 + b1*x2 + b0*x3) +
    a1*a1*a1*a1*(-5*a3*b0*x0 + b3*x0 + b2*x1 - 5*a2*(b1*x0 + b0*x1) + b1*x2 + b0*x3)-
    a3*b0*x4 + b3*x4 -
    a1*a1*a1*(10*a2*a2*b0*x0 + b3*x1 - 4*a3*(b1*x0 + b0*x1) + b2*x2 -
       4*a2*(b2*x0 + b1*x1 + b0*x2) + b1*x3 + b0*x4) +
    b2*x5 -
    a2*(b3*x2 - 2*a3*(b2*x0 + b1*x1 + b0*x2) + b2*x3 + b1*x4 + b0*x5) +
    a1*a1*(6*a2*a2*(b1*x0 + b0*x1) + b3*x2 -
    3*a3*(b2*x0 + b1*x1 + b0*x2) + b2*x3 +
    3*a2*(4*a3*b0*x0 - b3*x0 - b2*x1 - b1*x2 - b0*x3)
     + b1*x4 + b0*x5) +
    b1*x6 +
    a1*(4*a2*a2*a2*b0*x0 - 3*a3*a3*b0*x0 -
        3*a2*a2*(b2*x0 + b1*x1 + b0*x2) - b3*x3 +
        2*a3*(b3*x0 + b2*x1 + b1*x2 + b0*x3) - b2*x4 +
        2*a2*(b3*x1 - 3*a3*(b1*x0 + b0*x1) + b2*x2 + b1*x3 + b0*x4) -
        b1*x5 - b0*x6)
```

Código 5.2: Función resumida para 7 valores de entrada del IIR.

```
  y7_res =  0.0274148 * x0 + 0.0277707 * x1 - 0.0637264 * x2 - 0.121117 * x3 +
     0.1161970 * x4 + 0.4672040 * x5 + 0.4276650 * x6 + 0.130063 * x7;
```

en forma real ($[-2{,}55101; 2{,}55097]$).

El paso siguiente es introducir la función completa (Código 5.1) y la función resumida (Código 5.2 en el programa Gappa para poder corroborar que las cotas obtenidas sean idénticas a las del programa desarrollado en el Apéndice A. Al consultar por las cotas de *y_7completa* y *y7_res*, se obtienen los valores de la Tabla 5.5, que ademas contiene el rango analítico y el obtenido en la Tabla 5.4 referido como y7_Gappa. En ésta puede observarse que para y7_res se obtiene la misma cota superior que en forma analítica, lo mismo sucede con la equivalencia entre los resultados de y7_completa e y7_Gappa, mientras que lo que se espera es que los rangos de las cuatro funciones sean equivalentes.

Tabla 5.5: Resultados comparativos de rangos obtenidos por diferentes funciones

| Función | Rango |
|---|---|
| Analítico | $[-1,38115; 1,38112]$ |
| y7_Gappa | $[-2,55101; 2,55097]$ |
| y7_res | $[-1,38115; 1,38112]$ |
| y7_completa | $[-2,55101; 2,55097]$ |

Lo que sucede es que la herramienta formal Gappa encuentra un rango mucho más conservador que lo que se obtiene en forma analítica. Evidentemente la ésta no interpreta de la misma manera los coeficientes $a_i$ y $b_i$, que los números $x_i$. Otra alternativa es que no encuentre la relación adecuada entre los términos de la función completa. En el manual de Gappa esta situación es habitual, por lo que se guía a la herramienta estructurando las funciones y dando pistas sobre la resolución de los rangos.

#### Demostración

Otra funcionalidad adicional es la generación de un archivo que contiene los pasos para probar la formula lógica. En el Apéndice B se encuentra la demostración de la formula lógica correspondiente al script del caso del filtro IIR en forma directa I.

### 5.1.3. Comparación de ambos enfoques

En la verificación funcional basada en simulación se ha estimulado cada DUV con 46 casos de prueba donde cada caso contiene una señal de 100 puntos, lo que resulta en 4600 entradas simuladas. Los resultados obtenidos no han mostrado funcionamiento erróneo de los mismos. Por otro lado, en la verificación formal se obtuvieron los rangos para 7 entradas que han sido suficientes para demostrar que el rango de salida es superior a la entrada, por lo que podrían producirse desbordes.

Ambos enfoques de verificación funcional tienen ventajas y desventajas. Aquel basado en simulación permite crear casos de prueba de manera rápida y sencilla, pero es impracticable simular todos los posibles casos de prueba por lo que no se puede demostrar el correcto funcionamiento del DUV. Por ejemplo, los casos de prueba definidos en la verificación funcional basada en simulación del caso de estudio IIR son todos sinusoides de un tono, lo que implica que no se han simulado posibles casos extremos o inusuales para este tipo de sistemas, como puede ser una señal continua de amplitud máxima, o mínima.

En cambio, mediante el método formal se puede demostrar la correcta elección de la aritmética, pero la complejidad se encuentra en la correcta construcción del programa que lo demuestra, que en caso de no dominar la herramienta se pueden obtener rangos muy conservadores o erróneos. Además, al tener la limitación de no poder representar sistemas recursivos como filtros IIR se debe representar la función para una dada cantidad de entradas, proceso que es propenso a errores en función de la longitud de la recursividad a probar. En el caso de 7 entradas la complejidad es aceptable como para escribirla manualmente, pero en caso de querer representar mayor cantidad de entradas se debe optar por la generación de un algoritmo realice la misma.

## 5.2. Transformada rápida de Fourier (FFT)

La implementación del modelo de cobertura resulta en un conjunto de pruebas que contiene 792 casos para la FFT con $T_s = 256$ y 3096 casos para la FFT con $T_s = 1024$. Los resultados de latencia obtenidos mediante la verificación de cada bloque se muestran en la Tabla 5.6, también se reporta el *throughput* obtenido por inspección de las simulaciones. La FFT-Agustín resulta ser la mejor en términos de latencia, ya que responde 1,33 veces más rápido que el diseño FFT(256) Spiral. Sin embargo, debido al reducido ancho de *stream* la FFT-Agustín muestra un reducido *throughput*.

Tabla 5.6: Desempeño de los núcleos FFT

| FFT Core | Latencia (ciclos) | Throughput (transformadas/ciclo) |
|---|---|---|
| FFT(256) Agustín | 271 | 1/258 |
| FFT(256) Spiral | 403 | 1/128 |
| FFT(1024) Spiral | 777 | 1/256 |

El correcto comportamiento se verifica mediante el programa de Matlab con la computación del valor RMSD. La mecánica general del programa es especificar una cota superior y reportar solo los casos de prueba que han excedido este valor, pero en este caso de estudio se calculan y muestran todos los RMSD para poder reportar y comparar los resultados de los bloques. Los resultados de los valores RMSD se resumen en la Tabla 5.7 como el promedio de cada conjunto de cobertura y el máximo RMSD observado. Todos los bloques se comportan correctamente y exhiben valores similares de RMSD para cada subconjunto de cobertura.

Aunque los valores RMSD obtenidos son similares para todos los bloques, la FFT *256 Agustín* es la que exhibe el mayor RMSD promedio y el mayor RMSD máximo. también puede notarse que para todos los bloques se incrementa el error en las pruebas *Middle-freq* y *Full-freq*, esto se debe a que las señales que pertenecen a estos conjuntos tienen mayor cantidad de componentes frecuenciales, que implica mayor estimulo sobre los distintos *bins* de la FFT. Pero para las señales del conjunto *Full-freq* el error es menor a aquellas del conjunto *Middle-freq*, este fenómeno puede deberse a las cancelaciones que ocurren entre los cómputos intermedios del bloque.

Con respecto al reporte de cobertura de código obtenido por la herramienta de simulación, la Tabla 5.8 muestra el porcentaje cubierto por sentencia (*statement*), rama (*branch*) y cambios (*toggle*) para cada bloque.

Tabla 5.7: Errores medios cuadráticos para los casos de prueba de la FFT

| FFT | Cov. SubConj. | Casos Prueba | RMSD Prom. | RMSD Max. |
|---|---|---|---|---|
| 256 Agustín | 1-freq | 768 | 0,6891 | 1,3094 |
| | 2-freq | 12 | 0,8102 | 0,9846 |
| | Middle-freq | 9 | 1,1319 | 1,2792 |
| | Full-freq | 3 | 1,1179 | 1,2039 |
| 256 Spiral | 1-freq | 768 | 0,5942 | 1,2480 |
| | 2-freq | 12 | 0,7160 | 0,8844 |
| | Middle-freq | 9 | 0,9635 | 1,2592 |
| | Full-freq | 3 | 0,8324 | 0,9083 |
| 1024 Spiral | 1-freq | 3072 | 0,6791 | 1,0842 |
| | 2-freq | 12 | 0,7179 | 0,8907 |
| | Middle-freq | 9 | 0,8314 | 1,1031 |
| | Full-freq | 3 | 0,8012 | 0,9056 |

Tabla 5.8: Reporte de Cobertura de Código para la FFT

| FFT Core | Sentencia (%) | Rama (%) | Toggle (%) |
|---|---|---|---|
| FFT(256) Agustín | 99,5 | 99,5 | 98,8 |
| FFT(256) Spiral | 99,3 | 99,1 | 97,8 |
| FFT(1024) Spiral | 99,7 | 97,8 | 98,8 |

## 5.3. Unidad aritmética de punto flotante (FPU)

Los resultados se presentan en dos partes. La primera es una prueba de confianza sobre el modelo de cobertura propuesto, comparándolo contra un modelo externo. El segundo expone la efectividad del conjunto de pruebas en encontrar errores.

### 5.3.1. Prueba de confianza del modelo de cobertura

FPGen (v1.1) es un conjunto de pruebas para el cumplimiento del estándar IEEE-754R descrito en [28]. Este conjunto de pruebas se genera siguiendo los modelos de cobertura definidos en su documentación; cada modelo de cobertura es cubierto por una serie de vectores de prueba definidos en archivos de texto. La herramienta provee modelos de cobertura para varias operaciones y formatos de punto flotante, pero no ha sido desarrollado un conjunto de pruebas para el modelo de cobertura Binary16. A pesar de la falta, provee vectores de prueba para la operación de suma en punto flotante de 32 bits de precisión (precisión simple) la cual, mediante

previa adaptación, puede ser usada para medir la cobertura utilizando como métrica el modelo propuesto en este trabajo.

De hecho, es muy directa la adaptación de nuestro modelo de cobertura para la operación de suma en formato de precisión simple: sólo se deben cambiarse los valores de las rectas que delimitan las fronteras de las particiones. Las facilidades de medición en SystemVerilog permiten crear fácilmente dicho modelo. Registrando cada vector se obtendrán los resultados de cobertura. Los pasos para registrar la cobertura son: primero se deben extraer los vectores de prueba de los archivos (extraer operandos y resultado), luego convertir los valores al formato de precisión simple (ya que están codificados con una sintaxis específica) y finalmente analizar a qué región de cobertura pertenece el vector de prueba usado en cada caso. El número de vectores de prueba extraídos y registrados es 19.067. Los resultados muestran que el conjunto de vectores de FPGen cubre el 100 % del modelo de cobertura propuesto (particiones más valores límite). La Tabla 5.9 indica el número de vectores que han cubierto las primeras 6 particiones (incluyendo la partición 0-ésima).

Tabla 5.9: Aciertos de cobertura de la herramienta FPGen

| Nro. Partición | Aciertos |
|---|---|
| 1 | 47 |
| 2 | 21 |
| 3 | 67 |
| 4 | 225 |
| 5 | 16.441 |
| 6 | 44 |

Esto quiere decir que para todas las regiones definidas por nuestro modelo de cobertura ($CM_{own}$) existe un vector de prueba ($v$) generado por el modelo de cobertura de la suma de FPGen ($CM_{FGen}$) tal que $v$ cubre una partición $P_i$ de $CM_{own}$.

Sin embargo no se puede asegurar que el conjunto de pruebas generado por nuestro modelo de cobertura cubre el modelo de FPGen dado que no hay una descripción detallada del mismo. Basado en la documentación, sólo se puede asegurar que los vectores propuestos cubren los sub-modelos “Basic Types” y “Near FP Base Values”.

Aún así, la cobertura completa de FPGen sobre nuestro modelo, brinda confianza que la heurística propuesta es sensata para verificar la operación de suma en unidades de punto flotante. Las principales discrepancias entre ambos modelos se deben a la consideración de modos de redondeo y operandos intermedios definidos por el estándar; los vectores de prueba de FPGen cubren estos casos. En nuestro caso de estudio solo se ha tomado en cuenta el modo de redondeo

tipo truncación debido a las necesidades del proceso de verificación de un diseño con estas características.

### 5.3.2. Casos de prueba

Una unidad de punto flotante fue diseñada e implementada para realizar cálculos en un bloque dedicado al procesamiento digital de señales. La unidad de punto flotante descrita en el lenguaje de descripción de hardware VHDL, fue verificada mediante la implementación de verificación y conjunto de pruebas descritos en la Sección 3.3. Las simulaciones se han efectuado con la herramienta VCS de Synopsys.

El diseño de la FPU ha pasado por un proceso de mejoras debido a los errores encontrados por la herramienta de verificación. La mayor cantidad de errores se detectaron en la incorrecta implementación de suma de números subnormales y en la suma de números normales cercanos al límite inferior. Específicamente, todos los casos de prueba correspondientes a la partición 2 y el caso de valores límite MIN_NOR + MIN_NORM de la partición número 5 han sido reportados como fallas y corregidos en consecuencia.

# Capítulo 6

# Conclusiones

Este trabajo describe la problemática general de verificación de sistemas digitales y su aplicación particular al campo de procesamiento digital de señales. Se aborda la verificación funcional basada en simulación de tres bloques comunes en PDS, *filtros digitales*, *transformada rápida de Fourier (FFT)* y *unidad aritmética de punto flotante (FPU)*. Además, se experimenta con herramientas de verificación formales sobre propiedades aritméticas de los filtros digitales.

El estudio del espacio de prueba y la necesidad de reducirlo en este tipo de bloques ha resultado en la construcción de abstracciones que permiten identificar los casos de prueba relevantes. Éstos son la clave en la verificación funcional basada en simulación. Se demuestra en este trabajo su necesidad y beneficios en su construcción y uso como referencia para la generación de los casos de prueba. Más aún, los modelos de cobertura generados pueden ser extendidos para construir modelos más sofisticados, o su heurística puede ser aplicada a operaciones similares.

En base a los modelos de cobertura generados y los casos de prueba que estos inducen, se describe la construcción de ambientes de verificación que sirven como plataforma para simular estos casos. La heurística de generación de las abstracciones de cobertura, así como los ambientes de verificación, forman las bases de una heurística integral o plataforma de verificación para abordar bloques de similares características dentro de las aplicaciones de PDS. Los componentes de verificación permiten su aplicación a implementaciones de diversos bloques con diferentes características realizando mínimas configuraciones y esfuerzos de codificación. En consecuencia, se demostró que los ambientes de verificación o IP (por sus siglas en ingles *Intellectual Property*) de verificación para transformada rápida de Fourier (FFT), filtros y unidad aritmética de punto flotante (FPU) propuestos son capaces de realizar la verificación funcional basada en simulación para tres bloques diferentes. Es decir, la implementación de verificación propuesta puede ser usada como un marco de trabajo (*framework*) de verificación para reducir el tiempo de proyecto en un flujo de diseño de hardware.

Se han logrado resultados diversos en la verificación de bloques de PDS. El objetivo primordial de la verificación es la detección de errores en el DUV, lo que se logró satisfactoriamente en la implementación de la FPU ya que el flujo de desarrollo fue conjunto con la verificación. No es el caso de los demás bloques ya que han sido abordados para verificar una vez que han sido implementados por terceros.

Como resultado de la experiencia en verificación de bloques FFT se han generado los conjuntos de cobertura que permiten identificar grupos de señales de entrada que idealmente estimulan cada aspecto de la lógica de una implementación de FFT. Se ha logrado un avance significativo en esta dirección; de todas maneras la variedad de posibles implementaciones así como su complejidad, requiere un mayor análisis para definir los mejores conjuntos de cobertura. Llevando esta estrategia al límite, se puede obtener una definición formal generalizada de las secuencias de prueba para bloques FFT.

Otro resultado interesante es el logrado en la aplicación de métodos formales y de simulación en los filtros digitales y su comparación. Se han evidenciado algunas de las limitaciones de los métodos formales, tal como la aplicación a pequeñas porciones o instancias del diseño al verificar solo una limitada cantidad de entradas que han resultado ser suficientes para probar aspectos que no han sido alcanzados por los métodos de simulación. Aún así, la verificación mediante simulación de los filtros ha permitido de manera rápida probar el correcto funcionamiento en una gran cantidad de casos definidos por el modelo de cobertura.

Por ende, se ha demostrado la aplicación de los conceptos y herramientas utilizados por la industria en la actualidad.

Como trabajo futuro se propone ahondar en la aplicación de los métodos formales de propiedades aritméticas para otras operaciones de PDS o aquellas que no han sido abordas en este trabajo con dicha técnica, como es el caso de los bloques FFT. Por otro lado, la verificación mediante simulación provee dos caminos. Uno para continuar con la generalización del ambiente de verificación para poder instanciar cualquier bloque de procesamiento de señales y realizar su verificación, y otro de investigación y desarrollo de modelos de cobertura de mayor complejidad para los bloques aquí abordados u otros de mayor complejidad. La clave está en la caracterización de las señales y las equivalencias desde el punto de vista de verificación.

# Apéndice

## A. Script Gappa para filtros IIR

Código 1: Script `Forma Directa I` .

```
# TEST IIR
# Arithmetic definitions:
# fixed point, lsb = 0, rounding truncation (toward minus infinity)
# Por ejemplo, aritmetica con 24 bits de parte fraccionaria
@rndf = fixed<-24,dn>;

# DEFINITIONS
# coeficientes sin cuantizar
# Sin cuantizacion de los coeficientes
b0 = 0.130063011757311;
b1 = 0.390189035271933;
b2 = 0.390189035271933;
b3 = 0.130063011757311;

a0 = 1;
a1 = -0.288140506601264;
a2 = 0.355310921356962;
a3 = -0.026666320697210;

# coeficientes cuantizados
b0_quant rndf= b0;
b1_quant rndf= b1;
b2_quant rndf= b2;
b3_quant rndf= b3;

a0_quant rndf= a0;
a1_quant rndf= a1;
a2_quant rndf= a2;
a3_quant rndf= a3;

# x_i are fixed point numbers
x0 = rndf(xx0);
x1 = rndf(xx1);
x2 = rndf(xx2);
x3 = rndf(xx3); # 1st register filling

x4 = rndf(xx4);
x5 = rndf(xx5);
x6 = rndf(xx6);
```

```
40 x7 = rndf(xx7); # 2nd register filling
41
42 # REAL RESULT
43  z0  = x0*b0;
44  z1_aux = x0*b1 - z0*(a1);
45  z1  = x1*b0 + z1_aux;
46
47  z2_sub1= x0*b2 - z0*(a2);
48  z2_sub2= z2_sub1 + x1*b1 - z1*(a3);
49  z2  = z2_sub2 + x2*b0;
50
51  z3_sub1 =       x0*b3 + z0*(-a3);
52  z3_sub2 = z3_sub1 + x1*b2 + z1*(-a2);
53  z3_sub3 = z3_sub2 + x2*b1 + z2*(-a1);
54  z3   = z3_sub3 + x3*b0;
55
56  z4_sub1 =       x1*b3 + z1*(-a3);
57  z4_sub2 = z4_sub1 + x2*b2 + z2*(-a2);
58  z4_sub3 = z4_sub2 + x3*b1 + z3*(-a1);
59  z4   = z4_sub3 + x4*b0;
60
61  z5_sub1 =       x2*b3 + z2*(-a3);
62  z5_sub2 = z5_sub1 + x3*b2 + z3*(-a2);
63  z5_sub3 = z5_sub2 + x4*b1 + z4*(-a1);
64  z5   = z5_sub3 + x5*b0;
65
66  z6_sub1 =       x3*b3 + z3*(-a3);
67  z6_sub2 = z6_sub1 + x4*b2 + z4*(-a2);
68  z6_sub3 = z6_sub2 + x5*b1 + z5*(-a1);
69  z6   = z6_sub3 + x6*b0;
70
71  z7_sub1 =       x4*b3 + z4*(-a3);
72  z7_sub2 = z7_sub1 + x5*b2 + z5*(-a2);
73  z7_sub3 = z7_sub2 + x6*b1 + z6*(-a1);
74  z7   = z7_sub3 + x7*b0;
75  z = z7;
76
77 # FIXED POINT
78 # IIR DF1
79  y0 rndf= x0*b0_quant;
80
81  y1_aux rndf= x0*b1_quant + y0*(-a1_quant);
82  y1  rndf= x1*b0_quant + y1_aux;
83
84  y2_sub1 rndf= x0*b2_quant + y0*(-a2_quant);
85  y2_sub2 rndf= y2_sub1 + x1*b1_quant + y1*(-a1_quant);
86  y2   rndf= y2_sub2 + x2*b0_quant;
87
88  y3_sub1 rndf=    x0*b3_quant + y0*(-a3_quant);
89  y3_sub2 rndf= y3_sub1 + x1*b2_quant + y1*(-a2_quant);
90  y3_sub3 rndf= y3_sub2 + x2*b1_quant + y2*(-a1_quant);
91  y3   rndf= y3_sub3 + x3*b0_quant;
92
93  y4_sub1 rndf=    x1*b3_quant + y1*(-a3_quant);
94  y4_sub2 rndf= y4_sub1 + x2*b2_quant + y2*(-a2_quant);
95  y4_sub3 rndf= y4_sub2 + x3*b1_quant + y3*(-a1_quant);
96  y4   rndf= y4_sub3 + x4*b0_quant;
97
98  y5_sub1 rndf=    x2*b3_quant + y2*(-a3_quant);
```

```
  y5_sub2 rndf= y5_sub1 + x3*b2_quant + y3*(-a2_quant);
  y5_sub3 rndf= y5_sub2 + x4*b1_quant + y4*(-a1_quant);
  y5   rndf= y5_sub3 + x5*b0_quant;

  y6_sub1 rndf=    x3*b3_quant + y3*(-a3_quant);
  y6_sub2 rndf= y6_sub1 + x4*b2_quant + y4*(-a2_quant);
  y6_sub3 rndf= y6_sub2 + x5*b1_quant + y5*(-a1_quant);
  y6   rndf= y6_sub3 + x6*b0_quant;

  y7_sub1 rndf=    x4*b3_quant + y4*(-a3_quant);
  y7_sub2 rndf= y7_sub1 + x5*b2_quant + y5*(-a2_quant);
  y7_sub3 rndf= y7_sub2 + x6*b1_quant + y6*(-a1_quant);
  y7   rndf= y7_sub3 + x7*b0_quant;

y rndf= y7;

# LOGICAL PROPERTY
# Especificar rango de las entradas
{ x0 in [-128,127] /\
  x1 in [-128,127] /\
  x2 in [-128,127] /\
  x3 in [-128,127] /\
  x4 in [-128,127] /\
  x5 in [-128,127] /\
  x6 in [-128,127] /\
  x7 in [-128,127]
      -> y in ? /\ z in ? /\ z - y in ? }

# HINTS
# No need
```

# B. Resultado Prueba Formal Gappa IIR DF1

Under the following hypotheses

$$(((x0 \in [0,1] \wedge x1 \in [0,1]) \wedge x2 \in [0,1]) \wedge (\neg(y \in [-14925 \cdot 2^{-16}, 47855 \cdot 2^{-16}]) \vee \neg (z - y \in [-784422508792937463 \cdot 2^{-75}, 742517115061416033 \cdot 2^{-73}]))), \tag{1}$$

one can deduce the following properties:

$$(\neg(y \in [-14925 \cdot 2^{-16}, 47855 \cdot 2^{-16}]) \vee \neg(z - y \in [-784422508792937463 \cdot 2^{-75}, 742517115061416033 \cdot 2^{-73}])) \tag{2}$$

by using (1), and selecting a component.

$$((x0 \in [0,1] \wedge x1 \in [0,1]) \wedge x2 \in [0,1]) \tag{3}$$

by using (1), and selecting a component.

$$(x0 \in [0,1] \wedge x1 \in [0,1]) \tag{4}$$

by using (3), and selecting a component.

$$x0 \in [0,1] \tag{5}$$

by using (4), and selecting a component.

$$a2\_quant \in [1 \cdot 2^{-3}, 109636514352720843 \cdot 2^{-59}] \tag{6}$$

by using theorem `constant10`.

$$x0 \times a2\_quant \in [0, 24929 \cdot 2^{-17}] \tag{7}$$

by using (5), (6), and theorem `mul_pp`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x0 \times a2\_quant) \in [0, 779 \cdot 2^{-12}] \tag{8}$$

by using (7), and theorem `fixed_round,dn`.

$$a0\_quant \in [1 \cdot 2^{-3}, 68191 \cdot 2^{-19}] \tag{9}$$

by using theorem `constant10`.

$$z0 \in [0, 68191 \cdot 2^{-19}] \tag{10}$$

by using (5), (9), and theorem `mul_pp`.

$$y0 \in [0, 8523 \cdot 2^{-16}] \tag{11}$$

by using (10), and theorem `fixed_round,dn`.

$$-a2\_quant \in [-109636514352720843 \cdot 2^{-59}, -1 \cdot 2^{-3}] \tag{12}$$

by using (6), and theorem `neg`.

$$y0 \times -a2\_quant \in [-1621 \cdot 2^{-16}, 0] \tag{13}$$

by using (11), (12), and theorem `mul_pn`.

$$\text{fixed}\langle -16, \text{dn}\rangle(y0 \times -a2\_quant) \in [-1621 \cdot 2^{-16}, 0] \tag{14}$$

by using (13), and theorem `fixed_round,dn`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x0 \times a2\_quant) + \text{fixed}\langle -16, \text{dn}\rangle(y0 \times -a2\_quant) \in [-1621 \cdot 2^{-16}, 779 \cdot 2^{-12}] \tag{15}$$

by using (8), (14), and theorem `add`.

$$y2\_sub1 \in [-1621 \cdot 2^{-16}, 779 \cdot 2^{-12}] \tag{16}$$

by using (15), and theorem `fixed_round,dn`.

$$x1 \in [0, 1] \tag{17}$$

by using (4), and selecting a component.

$$a1\_quant \in [1 \cdot 2^{-2}, 56232166203351385 \cdot 2^{-57}] \tag{18}$$

by using theorem `constant10`.

$$x1 \times a1\_quant \in [0, 51143 \cdot 2^{-17}] \tag{19}$$

by using (17), (18), and theorem `mul_pp`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x1 \times a1\_quant) \in [0, 25571 \cdot 2^{-16}] \tag{20}$$

by using (19), and theorem `fixed_round,dn`.

$$y2\_sub1 + \text{fixed}\langle -16, \text{dn}\rangle(x1 \times a1\_quant) \in [-1621 \cdot 2^{-16}, 38035 \cdot 2^{-16}] \tag{21}$$

by using (16), (20), and theorem `add`.

$$\text{fixed}\langle -16, \text{dn}\rangle(y2\_sub1 + \text{fixed}\langle -16, \text{dn}\rangle(x1 \times a1\_quant)) \in [-1621 \cdot 2^{-16}, 38035 \cdot 2^{-16}] \tag{22}$$

by using (21), and theorem `fixed_round,dn`.

$$x1 \times a0\_quant \in [0, 2131 \cdot 2^{-14}] \tag{23}$$

by using (17), (9), and theorem `mul_pp`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x1 \times a0\_quant) \in [0, 2131 \cdot 2^{-14}] \tag{24}$$

by using (23), and theorem `fixed_round,dn`.

$$x0 \times a1\_quant \in [0, 6393 \cdot 2^{-14}] \tag{25}$$

by using (5), (18), and theorem `mul_pp`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x0 \times a1\_quant) \in [0, 6393 \cdot 2^{-14}] \tag{26}$$

by using (25), and theorem `fixed_round,dn`.

$$-a1\_quant \in [-56232166203351385 \cdot 2^{-57}, -1 \cdot 2^{-2}] \tag{27}$$

by using (18), and theorem `neg`.

$$y0 \times -a1\_quant \in [-1663 \cdot 2^{-15}, 0] \tag{28}$$

by using (11), (27), and theorem `mul_pn`.

$$\text{fixed}\langle -16, \text{dn}\rangle(y0 \times -a1\_quant) \in [-1663 \cdot 2^{-15}, 0] \tag{29}$$

by using (28), and theorem `fixed_round,dn`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x0 \times a1\_quant) + \text{fixed}\langle -16, \text{dn}\rangle(y0 \times -a1\_quant) \in [-1663 \cdot 2^{-15}, 6393 \cdot 2^{-14}] \tag{30}$$

by using (26), (29), and theorem `add`.

$$y1\_aux \in [-1663 \cdot 2^{-15}, 6393 \cdot 2^{-14}] \tag{31}$$

by using (30), and theorem `fixed_round,dn`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x1 \times a0\_quant) + y1\_aux \in [-1663 \cdot 2^{-15}, 2131 \cdot 2^{-12}] \tag{32}$$

by using (24), (31), and theorem `add`.

$$y1 \in [-1663 \cdot 2^{-15}, 2131 \cdot 2^{-12}] \tag{33}$$

by using (32), and theorem `fixed_round,dn`.

$$y1 \times -a1\_quant \in [-1663 \cdot 2^{-13}, 10383 \cdot 2^{-19}] \tag{34}$$

by using (33), (27), and theorem `mul_on`.

$$\text{fixed}\langle -16, \text{dn}\rangle(y1 \times -a1\_quant) \in [-1663 \cdot 2^{-13}, 1297 \cdot 2^{-16}] \tag{35}$$

by using (34), and theorem `fixed_round,dn`.

$$\text{fixed}\langle -16, \text{dn}\rangle\text{fixed}\langle -16, \text{dn}\rangle(y1 \times -a1\_quant) \in [-14925 \cdot 2^{-16}, 9833 \cdot 2^{-14}] \tag{36}$$

by using (22), (35), and theorem `add`.

$$y2\_sub2 \in [-14925 \cdot 2^{-16}, 9833 \cdot 2^{-14}] \tag{37}$$

by using (36), and theorem `fixed_round,dn`.

$$x2 \in [0, 1] \tag{38}$$

by using (3), and selecting a component.

$$x2 \times a0\_quant \in [0, 68191 \cdot 2^{-19}] \tag{39}$$

by using (38), (9), and theorem `mul_pp`.

$$\text{fixed}\langle -16, \text{dn}\rangle(x2 \times a0\_quant) \in [0, 8523 \cdot 2^{-16}] \tag{40}$$

by using (39), and theorem `fixed_round,dn`.

$$y2\_sub2 + \text{fixed}\langle -16, \text{dn}\rangle(x2 \times a0\_quant) \in [-14925 \cdot 2^{-16}, 47855 \cdot 2^{-16}] \tag{41}$$

by using (37), (40), and theorem `add`.

$$y \in [-14925 \cdot 2^{-16}, 47855 \cdot 2^{-16}] \tag{42}$$

by using (41), and theorem `fixed_round,dn`.

$$\neg\left(z - y \in [-784422508792937463 \cdot 2^{-75}, 7425171150614160 33 \cdot 2^{-73}]\right) \tag{43}$$

by using (2), (42), and discarding contradictory literals.

$$\bot \tag{44}$$

by using (43), (42), and discarding contradictory literals.

# C. Tabla Resultados IIR

Tablas simplificadas de resultados de simulación de verificación de filtros IIR.

Tabla 1: IIR Ml-24

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-127,+127] | 5,8 | 0,0000064423 | 0,00000064423 |
| [-127,+127] | 8,8 | 0,00000557575 | 0,000000557575 |
| [-127,+127] | 9,1 | 0,00000405267 | 0,000000405267 |
| [-127,+127] | 9,4 | 0,0000006578 | 0,00000006578 |
| [-127,+127] | 9,7 | 0,00000321444 | 0,000000321444 |
| [-127,+127] | 10 | 0,00000529451 | 0,000000529451 |
| [-127,+127] | 10,3 | 0,00000488601 | 0,000000488601 |
| [-127,+127] | 10,6 | 0,00000165079 | 0,000000165079 |
| [-127,+127] | 10,9 | 0,00000257759 | 0,000000257759 |
| [-127,+127] | 11,2 | 0,00000602874 | 0,000000602874 |
| [-127,+127] | 11,5 | 0,00000681994 | 0,000000681994 |
| [-127,+127] | 11,8 | 0,00000464685 | 0,000000464685 |
| [-127,+127] | 12,1 | 0,000000212819 | 0,0000000212819 |
| [-127,+127] | 12,4 | 0,00000427182 | 0,000000427182 |
| [-127,+127] | 12,7 | 0,00000722787 | 0,000000722787 |
| [-127,+127] | 14,8 | 0,0000019193 | 0,00000019193 |
| [-10,+10] | 5,8 | 0,000000555179 | 0,0000000555179 |
| [-10,+10] | 8,8 | 0,000000546073 | 0,0000000546073 |
| [-10,+10] | 9,1 | 0,000000274206 | 0,0000000274206 |
| [-10,+10] | 9,4 | 0,0000000741346 | 0,00000000741346 |
| [-10,+10] | 9,7 | 0,000000258256 | 0,0000000258256 |
| [-10,+10] | 10 | 0,000000347993 | 0,0000000347993 |
| [-10,+10] | 10,3 | 0,000000425979 | 0,0000000425979 |
| [-10,+10] | 10,6 | 0,000000110429 | 0,0000000110429 |
| [-10,+10] | 10,9 | 0,000000128754 | 0,0000000128754 |
| [-10,+10] | 11,2 | 0,000000554607 | 0,0000000554607 |
| [-10,+10] | 11,5 | 0,00000051312 | 0,000000051312 |
| [-10,+10] | 11,8 | 0,000000420544 | 0,0000000420544 |
| [-10,+10] | 12,1 | 0,00000000499008 | 0,000000000499008 |
| [-10,+10] | 12,4 | 0,000000270514 | 0,0000000270514 |
| [-10,+10] | 12,7 | 0,00000062452 | 0,000000062452 |
| [-10,+10] | 14,8 | 0,00000020311 | 0,000000020311 |

Tabla 2: IIR Ml-20

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-2047,+2047] | 5,8 | 0,000577009 | 0,0000577009 |
| [-2047,+2047] | 8,8 | 0,000901682 | 0,0000901682 |
| [-2047,+2047] | 9,1 | 0,001686 | 0,0001686 |
| [-2047,+2047] | 9,4 | 0,00195928 | 0,000195928 |
| [-2047,+2047] | 9,7 | 0,00161899 | 0,000161899 |
| [-2047,+2047] | 10 | 0,000764832 | 0,0000764832 |
| [-2047,+2047] | 10,3 | 0,000311058 | 0,0000311058 |
| [-2047,+2047] | 10,6 | 0,00125842 | 0,000125842 |
| [-2047,+2047] | 10,9 | 0,00177534 | 0,000177534 |
| [-2047,+2047] | 11,2 | 0,00172964 | 0,000172964 |
| [-2047,+2047] | 11,5 | 0,00117788 | 0,000117788 |
| [-2047,+2047] | 11,8 | 0,000352272 | 0,0000352272 |
| [-2047,+2047] | 12,1 | 0,00046094 | 0,000046094 |
| [-2047,+2047] | 12,4 | 0,000998503 | 0,0000998503 |
| [-2047,+2047] | 12,7 | 0,00113823 | 0,000113823 |
| [-2047,+2047] | 14,8 | 0,000244219 | 0,0000244219 |
| [-10,+10] | 5,8 | 0,00000264317 | 0,000000264317 |
| [-10,+10] | 8,8 | 0,00000430575 | 0,000000430575 |
| [-10,+10] | 9,1 | 0,00000867571 | 0,000000867571 |
| [-10,+10] | 9,4 | 0,00000887678 | 0,000000887678 |
| [-10,+10] | 9,7 | 0,00000861681 | 0,000000861681 |
| [-10,+10] | 10 | 0,00000282402 | 0,000000282402 |
| [-10,+10] | 10,3 | 0,000000894563 | 0,0000000894563 |
| [-10,+10] | 10,6 | 0,0000060885 | 0,00000060885 |
| [-10,+10] | 10,9 | 0,00000711119 | 0,000000711119 |
| [-10,+10] | 11,2 | 0,00000940333 | 0,000000940333 |
| [-10,+10] | 11,5 | 0,00000522435 | 0,000000522435 |
| [-10,+10] | 11,8 | 0,0000031786 | 0,00000031786 |
| [-10,+10] | 12,1 | 0,00000328267 | 0,000000328267 |
| [-10,+10] | 12,4 | 0,00000411019 | 0,000000411019 |
| [-10,+10] | 12,7 | 0,00000622882 | 0,000000622882 |
| [-10,+10] | 14,8 | 0,00000239822 | 0,000000239822 |

Tabla 3: IIR Ml-16

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-32767,+32767] | 5,8 | 0,273941 | 0,0273941 |
| [-32767,+32767] | 8,8 | 0,410414 | 0,0410414 |
| [-32767,+32767] | 9,1 | 0,103686 | 0,0103686 |
| [-32767,+32767] | 9,4 | 0,249163 | 0,0249163 |
| [-32767,+32767] | 9,7 | 0,503418 | 0,0503418 |
| [-32767,+32767] | 10 | 0,55258 | 0,055258 |
| [-32767,+32767] | 10,3 | 0,375097 | 0,0375097 |
| [-32767,+32767] | 10,6 | 0,0458002 | 0,00458002 |
| [-32767,+32767] | 10,9 | 0,295362 | 0,0295362 |
| [-32767,+32767] | 11,2 | 0,503059 | 0,0503059 |
| [-32767,+32767] | 11,5 | 0,490089 | 0,0490089 |
| [-32767,+32767] | 11,8 | 0,265148 | 0,0265148 |
| [-32767,+32767] | 12,1 | 0,0709447 | 0,00709447 |
| [-32767,+32767] | 12,4 | 0,370482 | 0,0370482 |
| [-32767,+32767] | 12,7 | 0,503997 | 0,0503997 |
| [-32767,+32767] | 14,8 | 0,0439005 | 0,00439005 |
| [-10,+10] | 5,8 | 0,0000766743 | 0,00000766743 |
| [-10,+10] | 8,8 | 0,00010381 | 0,000010381 |
| [-10,+10] | 9,1 | 0,0000415449 | 0,00000415449 |
| [-10,+10] | 9,4 | 0,000095177 | 0,0000095177 |
| [-10,+10] | 9,7 | 0,000149876 | 0,0000149876 |
| [-10,+10] | 10 | 0,000172945 | 0,0000172945 |
| [-10,+10] | 10,3 | 0,000107365 | 0,0000107365 |
| [-10,+10] | 10,6 | 0,0000220829 | 0,00000220829 |
| [-10,+10] | 10,9 | 0,000109712 | 0,0000109712 |
| [-10,+10] | 11,2 | 0,000133166 | 0,0000133166 |
| [-10,+10] | 11,5 | 0,000173605 | 0,0000173605 |
| [-10,+10] | 11,8 | 0,0000602719 | 0,00000602719 |
| [-10,+10] | 12,1 | 0,0000104708 | 0,00000104708 |
| [-10,+10] | 12,4 | 0,000135927 | 0,0000135927 |
| [-10,+10] | 12,7 | 0,000144669 | 0,0000144669 |
| [-10,+10] | 14,8 | 0,000000316233 | 0,0000000316233 |

Tabla 4: IIR Ml-12

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-524287,+524287] | 5,8 | 23,5306 | 2,35306 |
| [-524287,+524287] | 8,8 | 61,0108 | 6,10108 |
| [-524287,+524287] | 9,1 | 1,98087 | 0,198087 |
| [-524287,+524287] | 9,4 | 75,5522 | 7,55522 |
| [-524287,+524287] | 9,7 | 129,063 | 12,9063 |
| [-524287,+524287] | 10 | 137,021 | 13,7021 |
| [-524287,+524287] | 10,3 | 91,1902 | 9,11902 |
| [-524287,+524287] | 10,6 | 5,87873 | 0,587873 |
| [-524287,+524287] | 10,9 | 86,6516 | 8,66516 |
| [-524287,+524287] | 11,2 | 149,192 | 14,9192 |
| [-524287,+524287] | 11,5 | 155,554 | 15,5554 |
| [-524287,+524287] | 11,8 | 102,323 | 10,2323 |
| [-524287,+524287] | 12,1 | 10,9366 | 1,09366 |
| [-524287,+524287] | 12,4 | 81,1897 | 8,11897 |
| [-524287,+524287] | 12,7 | 136,681 | 13,6681 |
| [-524287,+524287] | 14,8 | 38,5188 | 3,85188 |
| [-10,+10] | 5,8 | 0,000210127 | 0,0000210127 |
| [-10,+10] | 8,8 | 0,000806507 | 0,0000806507 |
| [-10,+10] | 9,1 | 0,000322455 | 0,0000322455 |
| [-10,+10] | 9,4 | 0,00177985 | 0,000177985 |
| [-10,+10] | 9,7 | 0,00250145 | 0,000250145 |
| [-10,+10] | 10 | 0,00281452 | 0,000281452 |
| [-10,+10] | 10,3 | 0,00149939 | 0,000149939 |
| [-10,+10] | 10,6 | 0,000163409 | 0,0000163409 |
| [-10,+10] | 10,9 | 0,00186554 | 0,000186554 |
| [-10,+10] | 11,2 | 0,00266347 | 0,000266347 |
| [-10,+10] | 11,5 | 0,00325801 | 0,000325801 |
| [-10,+10] | 11,8 | 0,00160686 | 0,000160686 |
| [-10,+10] | 12,1 | 0,000425269 | 0,0000425269 |
| [-10,+10] | 12,4 | 0,0018427 | 0,00018427 |
| [-10,+10] | 12,7 | 0,00238819 | 0,000238819 |
| [-10,+10] | 14,8 | 0,000449034 | 0,0000449034 |

Tabla 5: IIR Ml-8

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-8388607,+8388607] | 5,8 | 5033,78 | 503,378 |
| [-8388607,+8388607] | 8,8 | 12754,1 | 1275,41 |
| [-8388607,+8388607] | 9,1 | 7535,55 | 753,555 |
| [-8388607,+8388607] | 9,4 | 2956,58 | 295,658 |
| [-8388607,+8388607] | 9,7 | 13801,6 | 1380,16 |
| [-8388607,+8388607] | 10 | 19328,4 | 1932,84 |
| [-8388607,+8388607] | 10,3 | 15973,3 | 1597,33 |
| [-8388607,+8388607] | 10,6 | 4283,4 | 428,34 |
| [-8388607,+8388607] | 10,9 | 11009,9 | 1100,99 |
| [-8388607,+8388607] | 11,2 | 23004,5 | 2300,45 |
| [-8388607,+8388607] | 11,5 | 25840,5 | 2584,05 |
| [-8388607,+8388607] | 11,8 | 17584,9 | 1758,49 |
| [-8388607,+8388607] | 12,1 | 1357,06 | 135,706 |
| [-8388607,+8388607] | 12,4 | 15923,8 | 1592,38 |
| [-8388607,+8388607] | 12,7 | 26642,7 | 2664,27 |
| [-8388607,+8388607] | 14,8 | 7245,85 | 724,585 |
| [-10,+10] | 5,8 | 0,0133781 | 0,00133781 |
| [-10,+10] | 8,8 | 0,0176222 | 0,00176222 |
| [-10,+10] | 9,1 | 0,00531673 | 0,000531673 |
| [-10,+10] | 9,4 | 0,000572886 | 0,0000572886 |
| [-10,+10] | 9,7 | 0,0195464 | 0,00195464 |
| [-10,+10] | 10 | 0,0182401 | 0,00182401 |
| [-10,+10] | 10,3 | 0,0188424 | 0,00188424 |
| [-10,+10] | 10,6 | 0,00394146 | 0,000394146 |
| [-10,+10] | 10,9 | 0,0128259 | 0,00128259 |
| [-10,+10] | 11,2 | 0,0320091 | 0,00320091 |
| [-10,+10] | 11,5 | 0,0276575 | 0,00276575 |
| [-10,+10] | 11,8 | 0,0282162 | 0,00282162 |
| [-10,+10] | 12,1 | 0,0011418 | 0,00011418 |
| [-10,+10] | 12,4 | 0,0159867 | 0,00159867 |
| [-10,+10] | 12,7 | 0,037954 | 0,0037954 |
| [-10,+10] | 14,8 | 0,0135374 | 0,00135374 |

Tabla 6: IIR Sp-24

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-127,+127] | 5,8 | 0,0000160883 | 0,00000160883 |
| [-127,+127] | 8,8 | 0,00000849094 | 0,000000849094 |
| [-127,+127] | 9,1 | 0,00000351712 | 0,000000351712 |
| [-127,+127] | 9,4 | 0,00000345922 | 0,000000345922 |
| [-127,+127] | 9,7 | 0,00000781242 | 0,000000781242 |
| [-127,+127] | 10 | 0,00000907021 | 0,000000907021 |
| [-127,+127] | 10,3 | 0,0000054852 | 0,00000054852 |
| [-127,+127] | 10,6 | 0,000000196958 | 0,0000000196958 |
| [-127,+127] | 10,9 | 0,00000570746 | 0,000000570746 |
| [-127,+127] | 11,2 | 0,00000839558 | 0,000000839558 |
| [-127,+127] | 11,5 | 0,00000826004 | 0,000000826004 |
| [-127,+127] | 11,8 | 0,00000429384 | 0,000000429384 |
| [-127,+127] | 12,1 | 0,000000153215 | 0,0000000153215 |
| [-127,+127] | 12,4 | 0,0000049073 | 0,00000049073 |
| [-127,+127] | 12,7 | 0,0000064216 | 0,00000064216 |
| [-127,+127] | 14,8 | 0,00000129935 | 0,000000129935 |
| [-10,+10] | 5,8 | 0,00000111375 | 0,000000111375 |
| [-10,+10] | 8,8 | 0,000000407601 | 0,0000000407601 |
| [-10,+10] | 9,1 | 0,00000044105 | 0,000000044105 |
| [-10,+10] | 9,4 | 0,000000491367 | 0,0000000491367 |
| [-10,+10] | 9,7 | 0,000000457 | 0,0000000457 |
| [-10,+10] | 10 | 0,000000903704 | 0,0000000903704 |
| [-10,+10] | 10,3 | 0,000000229672 | 0,0000000229672 |
| [-10,+10] | 10,6 | 0,00000012799 | 0,000000012799 |
| [-10,+10] | 10,9 | 0,000000526897 | 0,0000000526897 |
| [-10,+10] | 11,2 | 0,000000458672 | 0,0000000458672 |
| [-10,+10] | 11,5 | 0,000000857787 | 0,0000000857787 |
| [-10,+10] | 11,8 | 0,000000175503 | 0,0000000175503 |
| [-10,+10] | 12,1 | 0,000000124199 | 0,0000000124199 |
| [-10,+10] | 12,4 | 0,000000563951 | 0,0000000563951 |
| [-10,+10] | 12,7 | 0,000000269549 | 0,0000000269549 |
| [-10,+10] | 14,8 | 0,0000000242957 | 0,00000000242957 |

Tabla 7: IIR Sp-20

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-2047,+2047] | 5,8 | 0,00483612 | 0,000483612 |
| [-2047,+2047] | 8,8 | 0,00135849 | 0,000135849 |
| [-2047,+2047] | 9,1 | 0,000647638 | 0,0000647638 |
| [-2047,+2047] | 9,4 | 0,00219683 | 0,000219683 |
| [-2047,+2047] | 9,7 | 0,0026096 | 0,00026096 |
| [-2047,+2047] | 10 | 0,00185682 | 0,000185682 |
| [-2047,+2047] | 10,3 | 0,000420731 | 0,0000420731 |
| [-2047,+2047] | 10,6 | 0,000934082 | 0,0000934082 |
| [-2047,+2047] | 10,9 | 0,00164263 | 0,000164263 |
| [-2047,+2047] | 11,2 | 0,00151286 | 0,000151286 |
| [-2047,+2047] | 11,5 | 0,000842953 | 0,0000842953 |
| [-2047,+2047] | 11,8 | 0,0000969083 | 0,00000969083 |
| [-2047,+2047] | 12,1 | 0,000284834 | 0,0000284834 |
| [-2047,+2047] | 12,4 | 0,000216478 | 0,0000216478 |
| [-2047,+2047] | 12,7 | 0,000118748 | 0,0000118748 |
| [-2047,+2047] | 14,8 | 0,000120241 | 0,0000120241 |
| [-10,+10] | 5,8 | 0,0000188556 | 0,00000188556 |
| [-10,+10] | 8,8 | 0,00000335207 | 0,000000335207 |
| [-10,+10] | 9,1 | 0,00000104632 | 0,000000104632 |
| [-10,+10] | 9,4 | 0,0000149651 | 0,00000149651 |
| [-10,+10] | 9,7 | 0,00000854933 | 0,000000854933 |
| [-10,+10] | 10 | 0,0000133884 | 0,00000133884 |
| [-10,+10] | 10,3 | 0,00000101279 | 0,000000101279 |
| [-10,+10] | 10,6 | 0,000000587216 | 0,0000000587216 |
| [-10,+10] | 10,9 | 0,0000110086 | 0,00000110086 |
| [-10,+10] | 11,2 | 0,00000394811 | 0,000000394811 |
| [-10,+10] | 11,5 | 0,00000717342 | 0,000000717342 |
| [-10,+10] | 11,8 | 0,00000222493 | 0,000000222493 |
| [-10,+10] | 12,1 | 0,00000137532 | 0,000000137532 |
| [-10,+10] | 12,4 | 0,00000447288 | 0,000000447288 |
| [-10,+10] | 12,7 | 0,00000432147 | 0,000000432147 |
| [-10,+10] | 14,8 | 0,00000525924 | 0,000000525924 |

Tabla 8: IIR Sp-16

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-23767,+32767] | 5,8 | 0,885589 | 0,0885589 |
| [-23767,+32767] | 8,8 | 0,236357 | 0,0236357 |
| [-23767,+32767] | 9,1 | 0,0464907 | 0,00464907 |
| [-23767,+32767] | 9,4 | 0,256594 | 0,0256594 |
| [-23767,+32767] | 9,7 | 0,310669 | 0,0310669 |
| [-23767,+32767] | 10 | 0,220595 | 0,0220595 |
| [-23767,+32767] | 10,3 | 0,0698298 | 0,00698298 |
| [-23767,+32767] | 10,6 | 0,0437231 | 0,00437231 |
| [-23767,+32767] | 10,9 | 0,0681737 | 0,00681737 |
| [-23767,+32767] | 11,2 | 0,0213695 | 0,00213695 |
| [-23767,+32767] | 11,5 | 0,0272599 | 0,00272599 |
| [-23767,+32767] | 11,8 | 0,00937329 | 0,000937329 |
| [-23767,+32767] | 12,1 | 0,0940791 | 0,00940791 |
| [-23767,+32767] | 12,4 | 0,233415 | 0,0233415 |
| [-23767,+32767] | 12,7 | 0,317216 | 0,0317216 |
| [-23767,+32767] | 14,8 | 0,0280141 | 0,00280141 |
| [-10,+10] | 5,8 | 0,000214003 | 0,0000214003 |
| [-10,+10] | 8,8 | 0,0000122576 | 0,00000122576 |
| [-10,+10] | 9,1 | 0,0000568037 | 0,00000568037 |
| [-10,+10] | 9,4 | 0,000110436 | 0,0000110436 |
| [-10,+10] | 9,7 | 0,0000278054 | 0,00000278054 |
| [-10,+10] | 10 | 0,0000966508 | 0,00000966508 |
| [-10,+10] | 10,3 | 0,0000299646 | 0,00000299646 |
| [-10,+10] | 10,6 | 0,0000373417 | 0,00000373417 |
| [-10,+10] | 10,9 | 0,0000639355 | 0,00000639355 |
| [-10,+10] | 11,2 | 0,0000499391 | 0,00000499391 |
| [-10,+10] | 11,5 | 0,0000362761 | 0,00000362761 |
| [-10,+10] | 11,8 | 0,0000312808 | 0,00000312808 |
| [-10,+10] | 12,1 | 0,0000658232 | 0,00000658232 |
| [-10,+10] | 12,4 | 0,0000166608 | 0,00000166608 |
| [-10,+10] | 12,7 | 0,000145248 | 0,0000145248 |
| [-10,+10] | 14,8 | 0,0000460926 | 0,00000460926 |

Tabla 9: IIR Sp-12

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-524287,+524287] | 5,8 | 144,969 | 14,4969 |
| [-524287,+524287] | 8,8 | 17,254 | 1,7254 |
| [-524287,+524287] | 9,1 | 16,3612 | 1,63612 |
| [-524287,+524287] | 9,4 | 26,8 | 2,68 |
| [-524287,+524287] | 9,7 | 12,1287 | 1,21287 |
| [-524287,+524287] | 10 | 12,1732 | 1,21732 |
| [-524287,+524287] | 10,3 | 24,0552 | 2,40552 |
| [-524287,+524287] | 10,6 | 9,6303 | 0,96303 |
| [-524287,+524287] | 10,9 | 26,9463 | 2,69463 |
| [-524287,+524287] | 11,2 | 64,5624 | 6,45624 |
| [-524287,+524287] | 11,5 | 76,8447 | 7,68447 |
| [-524287,+524287] | 11,8 | 48,137 | 4,8137 |
| [-524287,+524287] | 12,1 | 15,4627 | 1,54627 |
| [-524287,+524287] | 12,4 | 86,6043 | 8,66043 |
| [-524287,+524287] | 12,7 | 129,379 | 12,9379 |
| [-524287,+524287] | 14,8 | 25,6109 | 2,56109 |
| [-10,+10] | 5,8 | 0,00167497 | 0,000167497 |
| [-10,+10] | 8,8 | 0,000414197 | 0,0000414197 |
| [-10,+10] | 9,1 | 0,000810736 | 0,0000810736 |
| [-10,+10] | 9,4 | 0,00129157 | 0,000129157 |
| [-10,+10] | 9,7 | 0,000672382 | 0,0000672382 |
| [-10,+10] | 10 | 0,000373109 | 0,0000373109 |
| [-10,+10] | 10,3 | 0,00118615 | 0,000118615 |
| [-10,+10] | 10,6 | 0,00065169 | 0,000065169 |
| [-10,+10] | 10,9 | 0,000156551 | 0,0000156551 |
| [-10,+10] | 11,2 | 0,00221934 | 0,000221934 |
| [-10,+10] | 11,5 | 0,000648239 | 0,0000648239 |
| [-10,+10] | 11,8 | 0,00181111 | 0,000181111 |
| [-10,+10] | 12,1 | 0,00140183 | 0,000140183 |
| [-10,+10] | 12,4 | 0,000354564 | 0,0000354564 |
| [-10,+10] | 12,7 | 0,00322705 | 0,000322705 |
| [-10,+10] | 14,8 | 0,00125995 | 0,000125995 |

Tabla 10: IIR Sp-8

| Amplitud | Frecuencia (kHz) | Error absoluto | RMSD |
|---|---|---|---|
| [-8388607,+8388607] | 5,8 | 72834,2 | 7283,42 |
| [-8388607,+8388607] | 8,8 | 47491,6 | 4749,16 |
| [-8388607,+8388607] | 9,1 | 720,079 | 72,0079 |
| [-8388607,+8388607] | 9,4 | 47485,1 | 4748,51 |
| [-8388607,+8388607] | 9,7 | 77067,4 | 7706,74 |
| [-8388607,+8388607] | 10 | 76322,8 | 7632,28 |
| [-8388607,+8388607] | 10,3 | 46405,4 | 4640,54 |
| [-8388607,+8388607] | 10,6 | 165,287 | 16,5287 |
| [-8388607,+8388607] | 10,9 | 43818,5 | 4381,85 |
| [-8388607,+8388607] | 11,2 | 69146,5 | 6914,65 |
| [-8388607,+8388607] | 11,5 | 67984,6 | 6798,46 |
| [-8388607,+8388607] | 11,8 | 43395,9 | 4339,59 |
| [-8388607,+8388607] | 12,1 | 7034,39 | 703,439 |
| [-8388607,+8388607] | 12,4 | 26378,4 | 2637,84 |
| [-8388607,+8388607] | 12,7 | 45169,5 | 4516,95 |
| [-8388607,+8388607] | 14,8 | 13945,1 | 1394,51 |
| [-10,+10] | 5,8 | 0,0686532 | 0,00686532 |
| [-10,+10] | 8,8 | 0,048784 | 0,0048784 |
| [-10,+10] | 9,1 | 0,0142145 | 0,00142145 |
| [-10,+10] | 9,4 | 0,0697396 | 0,00697396 |
| [-10,+10] | 9,7 | 0,0742036 | 0,00742036 |
| [-10,+10] | 10 | 0,102854 | 0,0102854 |
| [-10,+10] | 10,3 | 0,0436576 | 0,00436576 |
| [-10,+10] | 10,6 | 0,00777729 | 0,000777729 |
| [-10,+10] | 10,9 | 0,0613928 | 0,00613928 |
| [-10,+10] | 11,2 | 0,0656472 | 0,00656472 |
| [-10,+10] | 11,5 | 0,0973425 | 0,00973425 |
| [-10,+10] | 11,8 | 0,0381901 | 0,00381901 |
| [-10,+10] | 12,1 | 0,0245793 | 0,00245793 |
| [-10,+10] | 12,4 | 0,042607 | 0,0042607 |
| [-10,+10] | 12,7 | 0,0362648 | 0,00362648 |
| [-10,+10] | 14,8 | 0,0059938 | 0,00059938 |

# Bibliografía


[1] H. D. Foster, "Why the design productivity gap never happened," en *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, San Jose, California, EE.UU., 18 al 21 de noviembre de 2013. pp. 581–584.

[2] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, agosto 2008.

[3] A. Cox, S. Sankaranarayanan, y B.-Y. Chang, "A bit too precise? Verification of quantized digital filters," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 2, p. 175–190, 2014.

[4] G. Le Lann, "The Ariane 5 Flight 501 Failure - A Case Study en System Engineering for Computing Systems," Research Report RR-3079, Rocquencourt, Yvelines, France, 1996. Projet REFLECS.

[5] M. Blair, S. Obenski, y P. Bridickas, "GAO/IMTEC-92-26 Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia," tech. rep., Washington, D.C., EE.UU., febrero 1992.

[6] G. Pachiana, J. Agustín Rodríguez, y E. Paolini, "Functional verification for FFT cores," en *2014 Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, Mendoza, Argentina, 19 al 26 julio de 2014. pp. 95-100.

[7] G. Pachiana y J. Agustín Rodríguez, "Coverage modeling for verification of floating point arithmetic units," en *2014 Argentine Conference on Micro-Nanoelectronics, Technology and Applications (EAMTA)*, Mendoza, Argentina, 19 al 26 julio de 2014. pp. 83-88.

[8] A. Meyer, *Principles of Functional Verification*. Amsterdam, The Netherlands, The Netherlands: Elsevier Science, 2003.

[9] A. Piziali, *Functional Verification Coverage Measurement and Analysis*. New York, New York, EE.UU.: Springer Publishing Company, Incorporated, 1st ed., 2007.

[10] M. Daumas y G. Melquiond, "Generating formally certified bounds on values and round-off errors," en *6th Conference on Real Numbers and Computers (RNC6)*, Dagstuhl, Alemania, 15 al 17 de noviembre de 2004. pp. 55-70, RR-5259 RR-5259.

[11] G. Melquiond, *User's Guide for Gappa*. Rocquencourt, Yvelines, France: INRIA, 2013.

[12] G. J. Myers y C. Sandler, *The Art of Software Testing*. Hoboken, New Jersey, EE.UU.: John Wiley & Sons, 2004.

[13] C. Castro, M. Strum, y W. J. Chau, "Automatic generation of a parameter-domain-based functional input coverage model," en *11th Latin American Test Workshop (LATW)*, Punta del Este, Uruguay, 28 al 31 de marzo de 2010. pp. 1-6.

[14] V. Jerinic, J. Langer, U. Heinkel, y D. Muller, "New methods and coverage metrics for functional verification," en *Proceedings of the Design, Automation and Test in Europe. DATE 06.*, 6 al 10 de marzo de 2006. vol. 1, pp. 1-6.

[15] S. Yang, R. Wille, D. Grosse, y R. Drechler, "Coverage-driven stimuli generation," *15th Euromicro Conference on Digital System Design (DSD)*, Cesme, Izmir, 5 al 8 de setiembre de 2012. pp. 525-528.

[16] A. V. Oppenheim, R. W. Schafer, y J. R. Buck, *Discrete-time Signal Processing (2Nd Ed.)*. Upper Saddle River, New York, EE.UU.: Prentice-Hall, Inc., 1999.

[17] D. Novo, S. El Alaoui, y P. Ienne, "Accuracy vs speed tradeoffs in the estimation of fixed-point errors on linear time-invariant systems," *Procedings of the 2013 Conference on Design, Automation Test in Europe Conference Exhibition (DATE 13)*, Grenoble, Francia, 18 al 22 de marzo de 2013. pp. 15-20.

[18] B. Akbarpour y S. Tahar, "An approach for the formal verification of DSP designs using Theorem proving," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, agosto 2006. vol. 25, pp. 1441-1457.

[19] A. Gacic, M. Püschel, y J. M. F. Moura, "Fast automatic implementations of fir filters," en *Procedings of the 2003 International Conference on Acoustics, Speech, and Signal Processing (ICASSP 03)*, Hong Kong, China, 6 al 10 de abril de 2003. vol. 2, pp. 541–544.

[20] M. Sun, L. Tian, y D. Dai, "Radix-8 FFT processor design based on FPGA," *5th International Congress on Image and Signal Processing (CISP'12)*, Guwahati, India, 2 al 3 de marzo de 2012. pp. 1453-1457.

[21] W. W. Smith y J. M. Smith, *Handbook of real-time fast Fourier transforms : algorithms to product testing*. New York, New York, EE.UU.: IEEE Press, 1995.

[22] J. Agustín Rodríguez, P. Julián, y A. Andreou, "Frame and arithmetic pipelining for a radix-4 FFT streamed core," *Argentine School of Micro-Nanoelectronics Technology and Applications (EAMTA)*, Montevideo, Uruguay, 1 al 10 de octubre de 2010. pp. 112-116.

[23] P. Shenoy, "Universal FFT core generator," Master's thesis, Computer Science, Drexel University, 2007.

[24] E. Clarke, S. German, y X. Zhao, "Verifying the srt division algorithm using theorem proving techniques," *Formal Methods in System Design*, vol. 14, no. 1, pp. 7–44, 1999.

[25] O. Goni, E. Todorovich, y O. Cadenas, "Generic construction of monitors for floating point unit designs," *VIII Southern Conference on Programmable Logic (SPL2012)*, Bento Gonçalves, Brasil, 20 al 23 de marzo de 2012. pp. 1-8.

[26] E. Guralnik, M. Aharoni, A. Birnbaum, y A. Koyfman, "Simulation-based verification of floating-point division," *IEEE Transactions on Computers*, febrero 2011. vol. 60, pp. 176-188.

[27] C. Spear, *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. New York, New York, EE.UU.: Springer Publishing Company, Incorporated, 2nd ed., 2008.

[28] IBM, "Floating-point test generator (FPgen)," 2010. `https://research.ibm.com/haifa/projects/verification/fpgen`.