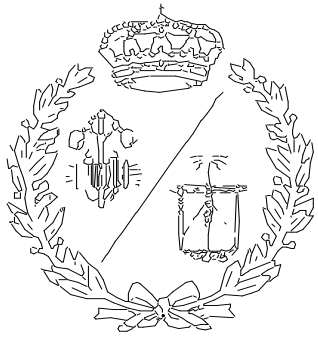
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Grado



Implementación de la metodología de verificación UVM en Vivado para diseños digitales.
Implementation of Universal Verification Methodology UVM on Vivado for digital desings.

Para acceder al Título de

GRADUADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: Juan Camilo Martínez Uribe

Septiembre – 2025

Resumen

El proyecto aborda el aprendizaje e implementación de la metodología estándar de verificación UVM (Universal Verification Methodology) en el entorno Vivado para realizar la verificación funcional de un módulo digital generador de ondas de radiofrecuencia. Para ello fue necesario también aprender el lenguaje de descripción hardware SystemVerilog que permite la programación orientada a objetos en la que se apoya UVM. El sistema de verificación permitió ejecutar pruebas configurables con las que se detectaron errores varios de precisión en el módulo y su caracterización, comprobando que UVM es un acercamiento robusto y eficaz para la verificación de diseños digitales.

Abstract

The project addresses the learning and implementation of the standard verification methodology UVM (Universal Verification Methodology) in the Vivado environment to perform the functional verification of a digital radiofrequency waveform generator module. For this purpose, it was also necessary to learn the SystemVerilog hardware description language, which allows the object-oriented programming that UVM relies on. The verification system allowed to perform configurable tests that revealed several precision errors on the module and its characterization, demonstrating that UVM is a robust and effective approach to digital designs verification.

Índice General

1 MEMORIA 4

2 PRESUPUESTO 81

3 CODIGO 86

1 MEMORIA

Índice de Memoria

1 MEMORIA 4

1.1 INTRODUCCIÓN 7
1.1.1 Objetivos 7
1.1.2 Alcance 8
1.1.3 Estructura del documento 8
1.2 ESTADO DEL ARTE 8
1.2.1 Verificación de diseños hardware 9
1.2.2 Métodos de verificación anteriores 10
1.3 CONOCIMIENTOS PREVIOS 12
1.3.1 SystemVerilog 12
1.3.2 UVM 13
1.4 IMPLEMENTACIÓN DEL SISTEMA DE VERIFICACIÓN 20
1.4.1 Esquema general 20
1.4.2 DUT 20
1.5 IMPLEMENTACIÓN DEL ENTORNO 26
1.5.1 Top 26
1.5.2 Entorno 26
1.5.3 Agente puerto de entrada 27
1.5.4 Agente puerto de salida 29
1.5.5 Agente reloj 29
1.5.6 Modelo de referencia 30
1.5.7 Marcador 33
1.5.8 Cobertura 39
1.6 IMPLEMENTACIÓN DEL BANCO DE PRUEBAS 41
1.6.1 Planteamiento del test 42
1.6.2 Configuración de test. 43
1.6.3 Secuencia de agente de puerto de entrada 44
1.6.4 Secuencia de agente reloj 49
1.6.5 Llamada a test configurado 52
1.6.6 Parámetros de cobertura 54
1.6.7 Modulabilidad de los componentes. 56
1.6.8 Simulación Vivado 57
1.6.9 Tras la simulación 59
1.6.10 Test compuesto 63
1.7 RESULTADOS Y ANALISIS 65
1.7.1 Errores detectados durante el proceso de implementación 65
1.7.2 Resultados de la prueba final 71
1.8 CONCLUSIONES 78
1.9 BIBLIOGRAFÍA 79

Índice de figuras

Figura 1 - Fases de UVM 14
Figura 2 - Árbol jerárquico de las clases de UVM 15
Figura 3 - Puertos TLM 16
Figura 4 - Resumen del sistema de mensajes de UVM 17
Figura 5 - Estructura básica de un sistema UVM 17
Figura 6 - Estructura del sistema de verificación implementado 20
Figura 7 - Intervalo de repetición de pulsos 23
Figura 8 - Chirp en modulación IQ 23
Figura 9 - Tono en modulación IQ 24
Figura 10 - Secuencia de rampas 25
Figura 11 - Estructura de la interfaz del puerto de entrada del DUT 27
Figura 12 - Fase de ejecución del controlador del agente de entrada 28
Figura 13 - Fase de ejecución del monitor del agente de entrada 28
Figura 14 - Manejo de estados del modelo de referencia 31
Figura 15 - Estado WAIT_FOR_PARAM del modelo de referencia 32
Figura 16 - Estado WAVE_GEN del modelo de referencia 32
Figura 17 - Cálculo de las ondas en el modelo de referencia 33
Figura 18 - Fase de ejecución del marcador 35
Figura 19 - Detección de desincronización 35
Figura 20 - Manejo de estados del acumulador de errores 36
Figura 21 - Estructura del acumulador de errores 37
Figura 22 - Informe final de la verificación por terminal 38
Figura 23 - Conteo de errores para histograma 38
Figura 24 - Manejo del muestreo para cobertura 41
Figura 25 - Diferentes retrasos de señal aleatorizados 43
Figura 26 - Restricción de aleatorización condicionada 44
Figura 27 - Varios rangos de aleatorización para un mismo registro 45
Figura 28 - Mensaje de resumen de la configuración seleccionada previa al test 45
Figura 29 - Mensaje de resumen de configuración distinta 45
Figura 30 - Gestión de restricciones contradictorias 46
Figura 31 - Visión general de simulación de formas de onda en Vivado 47
Figura 32 - Aleatorización de intervalo entre resets 48
Figura 33 - Restricciones de aleatorización de periodo del reloj generado 50
Figura 34 - Discrepancia entre frecuencia generada e indicada 50
Figura 35 - Generación del reloj y su recalculación 52
Figura 36 - Fase de construcción del test 53
Figura 37 - Fase de ejecución del test 53
Figura 38 - Rangos de aleatorización junto a los selectores en el objeto configuración 55
Figura 39 - Ejemplo de estructura de los puntos de cobertura. 56
Figura 40 - Paquete de lista de tests 56
Figura 41 - Paquete del entorno 57
Figura 42 - Configuración de elaboración de la simulación en Vivado 58
Figura 43 - Jerarquía de fuentes de simulación en Vivado 58
Figura 44 - Interfaces a las que se tiene acceso en la simulación de Vivado 59
Figura 45 - Registros del DUT y modelo real 59
Figura 46 - Resumen de mensajes generados por entorno UVM tras la simulación 60
Figura 47 - Generación y resumen de cobertura de código 61
Figura 48 - Generación de cobertura funcional 61
Figura 49 - Informe de un grupo de cobertura 62
Figura 50 - Informe de puntos de cobertura con bins individuales 63

Figura 51 - Estructura del test largo 64
Figura 52 - Fase de ejecución del test largo..... 65
Figura 53 - Desincronización entre DUT y modelo de referencia 66
Figura 54 - No transmisión del reloj al modelo de referencia para resincronizar el comportamiento 66
Figura 55 - Error de número de muestras en la generación de rampas..... 67
Figura 56 - Histograma de errores en una prueba con los 3 tipos de onda 68
Figura 57 - Errores altos en una prueba con los 3 tipos de onda 68
Figura 58 - Error acumulativo en el cálculo de la fase del chirp..... 69
Figura 59 - Histograma de errores de solo chirps tras la corrección del error de fase 70
Figura 60 - Histograma de solo tonos final 71
Figura 61 - Simulación de verificación completa 72
Figura 62 - Histograma de la prueba larga en escala logarítmica..... 72
Figura 63 - Resumen del informe de cobertura funcional 73
Figura 64 - Puntos de cobertura muestreados cada ciclo de reloj 73
Figura 65 - Punto de cobertura de la señal sRun 74
Figura 66 - Informe del grupo de cobertura sync_sClk_cg completo 74
Figura 67 - Punto de cobertura de ciclos de generación de señal completados 75
Figura 68 - puntos de cobertura muestreados cada reconfiguración de señal..... 75
Figura 69 - Resumen de puntos de cobertura de sync_sRun_cg 76
Figura 70 - Bins no cubiertos del punto de cobertura asociados a f_signal 76
Figura 71 - Bins no cubiertos del punto de cobertura asociados a sRun_delay..... 77
Figura 72 - Bins no cubiertos del punto de cobertura asociado a f_sample..... 77

1.1 INTRODUCCIÓN

La verificación es una parte importante en el diseño de sistemas digitales complejos, que progresivamente ha ido suponiendo una parte más significativa del presupuesto de un sistema digital. Su propósito es comprobar que el diseño opere de la forma esperada dentro de los parámetros para los que fue diseñado. Conforme un diseño se hace más complejo, el tratar de comprobar todas sus partes se hace inviable desde un punto de vista económico, por la necesidad de tener que implementar un banco de pruebas específico para cada funcionalidad del diseño a probar. Para ello se han ido desarrollando diferentes metodologías para simplificar el proceso de verificación optimizando todo el proceso. En este proyecto se implementa la metodología UVM, desarrollada en 2011 por Accellera basándose ampliamente en la metodología VMM/OVM, en la que ya estaba definida buena parte de la estructura que implementa la metodología (ambiente, agentes, monitores…) estandarizándola como una librería de clases base de SystemVerilog para facilitar su implementación. Si bien la metodología sigue recibiendo soporte hoy en día, siendo la última versión la UVM v2020.3.1 lanzada en agosto de 2024, en el proyecto se usará la versión UVM v1.2 lanzada en 2014 ya que es la versión de la librería incluida en la suite de Vivado v2024.2 utilizada en el proyecto.

1.1.1 Objetivos

El objetivo final del proyecto es comprobar el correcto funcionamiento del módulo generador de ondas de radiofrecuencia.

Para ello se implementará un entorno de verificación que incluirá también un modelo de referencia con el que se comparará no solo el comportamiento general del módulo si no también la precisión de las ondas generadas frente a un modelo más cercano a lo ideal.

Se implementarán diferentes pruebas orientadas a diferentes comportamientos del módulo para orientar cada prueba individual de manera más precisa, esto es importante considerando que en una implementación básica de la metodología los estímulos son aleatorios, por lo que controlar la aleatoriedad es vital si, como es el caso, se trabaja principalmente con pruebas cortas individuales.

1.1.2 Alcance

Se implementó un sistema de verificación que consta del manejo de señales a Nivel de Transferencia de Registros, un registro de errores respecto al modelo de referencia y un registro de diferentes secuencias de estímulos de entrada para comprobar que el objetivo de cada prueba individual ha sido realmente cubierto.

Para flexibilizar las pruebas individuales se implementó una prueba configurable que permitió realizar diferentes pruebas orientadas a comportamientos específicos, esto facilitó la repetibilidad de los diferentes errores encontrados para así poder aislar su causa.

Una vez implementado el sistema de verificación, junto a la prueba configurable, se hicieron múltiples pruebas orientadas a la detección de diferentes posibles errores. Se usaron diferentes métricas de verificación con la que se caracterizaron los límites del diseño hardware.

1.1.3 Estructura del documento

En este documento se cubre en una primera parte: un breve repaso de la historia de la verificación funcional de diseños hardware junto a su importancia, una explicación de la metodología de verificación UVM y la necesidad del uso del lenguaje SystemVerilog para la implementación de la metodología. Una vez considerados los conocimientos previos necesarios para la implementación del sistema, se hará una explicación de cada componente del sistema considerando dos partes fundamentales: el entorno, que es fijo entre pruebas y el banco de pruebas, que define el comportamiento de cada prueba individual. Por último, se tratará con los resultados de diferentes pruebas en los que se comentará diferentes errores hallados por el sistema de verificación que pudieron ser corregidos del todo o mejorados dentro de un rango.

1.2 ESTADO DEL ARTE

La verificación es un proceso necesario que idealmente se produce de manera concurrente al diseño del módulo a verificar. Para un diseño muy sencillo se puede realizar un banco de pruebas que compruebe todas sus posibles interacciones y esto sería suficiente para considerarlo verificado. Pero una vez la complejidad de los diseños comienza a incrementar se hace insostenible este acercamiento a la verificación y se hace evidente la necesidad de herramientas especializadas para este proceso.

Un caso famoso que muestra la importancia de una verificación robusta es el del **Error de División del Intel Pentium**. Este afectó a los primeros procesadores Pentium en 1994. El error sucedía por el algoritmo usado para aumentar la velocidad de cálculo de divisiones en coma flotante, en el que ciertos valores en una look-up table no estaban correctamente definidos por lo que los resultados que daba eran erróneos. Este error acabo por costarle a Intel un total de 475 millones de dólares en varias demandas y el reemplazo de los procesadores afectados. Tras esta resolución Intel se vio forzado a implementar una verificación más robusta, para esto colaboraron con academia en lo que sería la implementación de Verificación Formal, ámbito puramente matemático, a la verificación de circuitos. Iterando sobre esta idea es que finalmente se llegaría a la creación de un lenguaje de especificación formal llamado ForSpec, el cual sería cedido a Accellera e influenciaría la creación de SystemVerilog Assertions (SVA), un subset de SystemVerilog centrado en la detección de eventos bajo diferentes secuencias en las señales de un módulo.

Si bien ese error no es más que un incidente ocasionado por una verificación insuficiente, es uno que afectó de manera masiva a un público más generalizado y sirvió como una motivación en la inversión al desarrollo del campo de la verificación.

1.2.1 Verificación de diseños hardware

La verificación de diseños hardware podría decirse que comenzó con la posibilidad de simulación de los diseños en sí, el primer estándar al respecto fue la tecnología SPICE (Simulation Program with Integrated Circuits Emphasis en inglés) en la década de los 70, en la que se implementaron varías tecnologías para diferentes tipos de simulaciones en un software libre que fue rápidamente adoptado por la industria.

Para diseños más complejos se necesitaba simulaciones a nivel de puertas lógicas. Cada gran fabricante llegó a desarrollar sus propios simuladores específicos. La tecnología siguió avanzando hasta llegar a la creación del lenguaje Verilog, que permitía describir hardware con diferentes niveles de abstracción en 1985, el cual no llegaría a ser un estándar IEEE hasta 1995. Durante esa década el foco del desarrollo de técnicas de verificación fue cambiando de ser únicamente la simulación a la metodología implementada.

Los componentes ya habían aumentado su complejidad lo suficiente como para que no resultase viable una simulación directa de toda su funcionalidad por lo que se exploraron diferentes formas de acelerar el proceso. Entre ellas la orientación de la simulación o la consideración de métricas de cobertura, de entre las cuales destacó la cobertura de código,

especialmente útil si el diseño es una caja blanca o la cobertura funcional, aplicable a cualquier diseño.

1.2.2 Métodos de verificación anteriores

Una vez establecido el lenguaje SystemVerilog en 2005 y centrándose únicamente en los predecesores inmediatos a la metodología UVM, es notable que UVM fue desarrollado a raíz de múltiples tecnologías en uso en industria. Adopta conceptos de AVM (Advanced Verification Methodology) de Siemens EDA, OVM (Open Verification Methodology) de Siemens EDA y Cadence, VMM-RAL (Verification Metodology Manual – Register Abstraction Layer) y tecnologías sueltas de Mentor como lo pueden ser TLM2 (Transaction Level Model) y Phasing (Fases de verificación).

De entre ellos podemos destacar dos de esas influencias.

VMM-RAL

Verification Metodology Manual surgió como un intento en estandarizar el proceso de verificación en 2005. Propuso estandarizar diversos componentes de verificación en pos de la reutilización de entornos usando el reciente lenguaje estándar de SystemVerilog, cuando hasta el momento era común que cada proveedor utilizase un lenguaje concreto que se ajustase a sus necesidades específicas.

Al igual que lo estaría UVM, estaba desarrollado como una librería de clases SystemVerilog, si bien no compartía la estructura del sistema que propone UVM, los principios básicos son los mismos: Un entorno reutilizable en el que verificar el funcionamiento de un diseño hardware de manera automatizada.

Tiene especial foco en el uso de aserciones en buena parte de la estructura del sistema de verificación y propone el uso de la cobertura en la prueba no solo como parte del análisis tras la prueba si no para orientar los estímulos generados durante la prueba. Dicho esto, su principal aportación a lo que más tarde sería UVM fue el diseño de componentes del sistema basado en interfaces, lo que significó un paso hacia la modularización del diseño de sistemas de verificación.

OVM

Open Verification Methodology fue lanzado en 2008, siendo el precursor directo a UVM, ya integra varias de las características que acabarían definiendo UVM: su acercamiento a

verificación a través de la programación orientada a objetos, la posibilidad de parametrizar dichos objetos para facilitar cambios en el entorno de verificación, la ejecución por fases para estandarizar la estructura de la programación, etc.

En una primera versión de UVM, la metodología que propone es prácticamente idéntica a la versión en el momento de OVM. El cambio de nombre vino por cuestión de propiedad de la metodología. OVM pertenecía a Cadence como proyecto de código abierto y era necesario fijar las características de la metodología para poder establecer un estándar que pudiese ser usado en la industria.

Durante años OVM fue usada como la metodología por defecto para verificación, pero tras la estandarización de UVM y la adquisición de la propiedad de OVM por Accellera, el soporte tras OVM, es decir, sus recursos de aprendizaje online y foros de discusión han sido sustituidos por UVM.

1.3 CONOCIMIENTOS PREVIOS

Para la implementación de la metodología es necesario conocer tanto la metodología en si como el lenguaje en el que se apoya.

1.3.1 SystemVerilog

SystemVerilog es un lenguaje de descripción y verificación hardware estandarizado en la IEEE 1800 por Accellera en 2005, misma organización que estandarizó la metodología UVM.

Sus características principales pueden ser agrupadas en dos roles

- SystemVerilog para Transferencia a Nivel de Registros (RTL en inglés), que es una extensión de Verilog para diseños sintetizables.
- SystemVerilog para Verificación, incluye características de programación orientada a objetos (OOP en inglés) que, si bien no son necesariamente sintetizables, son útiles para el desarrollo de bancos de prueba.

Para su uso en la metodología UVM se utiliza principalmente sus características de verificación. Algunas de sus características más relevantes a la metodología son:

- Técnicas de programación orientada a objetos que permiten la creación de entornos complejos de verificación dividiendo el proceso en diferentes módulos que están orientados a ser reutilizados.
- Acceso a diferentes métricas de cobertura como lo pueden ser las aserciones (mediante SVA), que puede comprobar que sucedan, o no, eventos definidos por el usuario durante toda la duración de la prueba y los grupos de cobertura que funcionan de manera similar, pero tienen que ser muestreados explícitamente dentro del código.
- Aleatorización condicionada, lo que permite un gran número de posibles casos de prueba además de permitir la creación de pruebas más orientadas para hacer más hincapié en posibles casos limite.
- Posibilidad de agrupar señales en interfaces para asegurar que la transmisión de datos entre registros se produzca de manera sincronizada, este resultó ser un punto particularmente importante de cara a la simulación en vivado.

1.3.2 UVM

UVM, Universal Verification Methodology en inglés, es un estándar establecido por Accellera Systems Initiative para unificar los diferentes acercamientos a la verificación de diseños hardware en uso en industria en el momento.

La metodología en sí está implementada como una librería de clases base con las que se instancia el banco de pruebas, estas clases aseguran la ejecución en un orden especifico de las diferentes fases y subfases de la prueba. La utilidad radica en lo modular de su diseño que permite la reutilización, con mínimos cambios, de algunas de estas partes y en la flexibilidad para implementar diferentes comportamientos.

Fundamentos de UVM

Una simulación de verificación completa se compone de tres partes principales, el entorno, que es fijo entre diferentes pruebas, el banco de pruebas, que establece el comportamiento que se busca obtener en la prueba y el diseño sobre el que se realiza la prueba. La metodología UVM describe cómo tiene que estar creadas las primeras dos partes y la conexión con el diseño.

Fases UVM

Una de las formas en las que la metodología UVM estandariza el proceso de implementación de un sistema de verificación es a través de su sistema de fases. La metodología considera 3 fases principales: construcción, ejecución y limpieza, ilustradas en la figura 1. El manejo de fases ocurre principalmente de manera interna en el sistema con la excepción de la fase de ejecución para que sea el usuario el que controle el final de la prueba.

En la fase de construcción se inicializa, configura y conecta todo el entorno. Este proceso ocurre en orden top to bottom, está orientado para que sean los componentes en capas exteriores los que apliquen las configuraciones necesarias a sus respectivos componentes internos. En esta fase también se crean todos los puertos TLM (Transaction Level Modeling) con los que están conectados los componentes que ocurre en orden bottom-up. Todo esto ocurre en tiempo 0 de simulación.

En la fase de ejecución comienza la comunicación con el módulo. Consiste en un reset del módulo, configuración de este de ser necesario, la generación de estímulos de la prueba y un corto periodo para asegurar que los efectos de los estímulos generados se han llegado a propagar por todo el sistema. La metodología también integra ganchos antes y después de cada una de estas subfases para facilitar la comunicación con el módulo en determinados

momentos de la prueba en caso de ser necesario, como también permite la inclusión de fases creadas por el usuario, aunque no es recomendable porque limita la reutilización del entorno.

Para esta fase es relevante conocer que el manejo de fases se realiza con objeciones, que son esencialmente banderas que indican si una fase ha acabado. En una implementación básica el usuario tiene que levantar explícitamente una de estas banderas al iniciar la secuencia de estímulos y bajarla al acabar la secuencia.

Finalmente está la fase de limpieza, en la que se extrae la información recolectada en la prueba, se establece si la prueba ha sido superada y se crea un informe de la verificación.

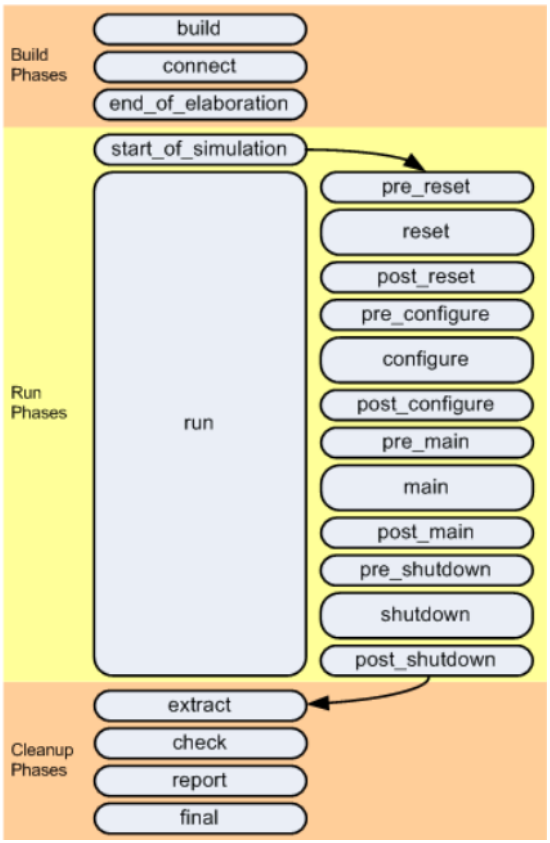


Figura 1 - Fases de UVM

Jerarquía de clases

Los componentes UVM tienen definidos todos los métodos necesarios para el manejo interno del entorno junto a la comunicación entre componentes. Para usarlos basta con crear las clases necesarias para la implementación del entorno de verificación y, respetando las funciones que separan las fases, definir las funcionalidades específicas. Para implementar un entorno básico basta con extender dichas clases de los niveles más exteriores de la jerarquía, ilustrada en la Figura 2.

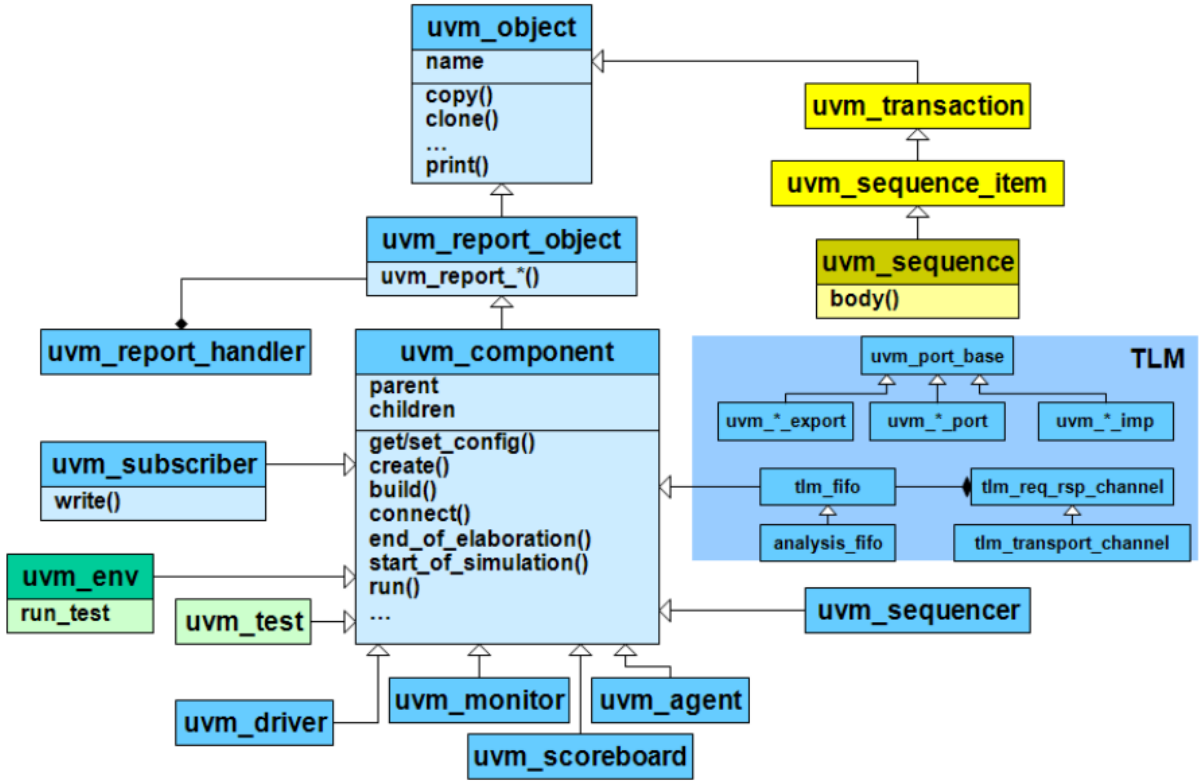


Figura 2 - Árbol jerárquico de las clases de UVM

TLM

La metodología UVM utiliza el modelado a nivel de transacciones para la comunicación entre componentes, que a diferencia de una comunicación RTL, Register Transfer Level, tradicional, empaqueta cada transmisión de datos en una única transmisión en lugar de cada uno de sus registros por separado lo cual podría llegar a ser mucho más demandante desde un punto de vista de recursos de computación. Esto puede llegar a ser importante de tener en cuenta dependiendo de la velocidad con la que tenga que interactuar con el módulo, en

concreto si llega a ser necesaria la implementación de FIFOS u otro método de arbitraje de transmisión de datos.

Este estándar establece el uso de dos tipos de puertos, los que inician transacciones, port, y los que no, export, representados comúnmente en diagramas como círculos y cuadrados respectivamente tal como se muestra en la Figura 3.

Tienen métodos orientados a usarse en la comunicación con el módulo, que son bloqueantes, y también métodos orientados a comunicaciones de análisis que son no bloqueantes.

Para utilizar las lecturas no bloqueantes es necesario poner las transacciones de análisis en puertos de análisis, además estos puertos permiten la comunicación con varios componentes a la vez.

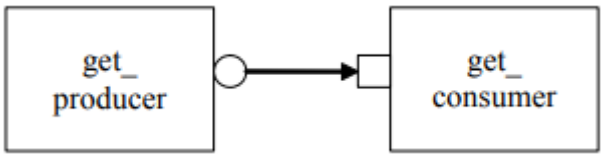


Figura 3 - Puertos TLM

Mecanismos de configuración

UVM permite dos mecanismos de configuración, factory y base de datos. El mecanismo de factory permite de definición de componentes en la raíz de UVM que después pueden ser reemplazados entre sus equivalentes sobrescribiendo la definición dentro de otro componente, este acercamiento permite la instanciación de distintos tests con cambios mínimos en el código, utilizando handlers genéricos.

La base de datos permite configuración de un componente a otro sin importar si está dentro de su alcance. Son datos tanto de lectura como escritura, por lo que se puede cambiar la configuración durante la ejecución haciendo así una prueba dinámica. Si se utiliza como un configurador estático, su valor se sobrescribe al que haya sido inicializado dentro del propio componente, lo que permite la parametrización de una prueba.

Gestión de mensajes

La metodología UVM maneja la gestión de mensajes a usuario dividiéndolos en 4 tipos de mensaje: info, warning, error y fatal. El sistema tiene algunas condiciones en las que imprime un mensaje en caso de que la construcción o conexión del entorno haya fallado, pero está orientado que sea el usuario el que defina los mensajes que considere necesario.

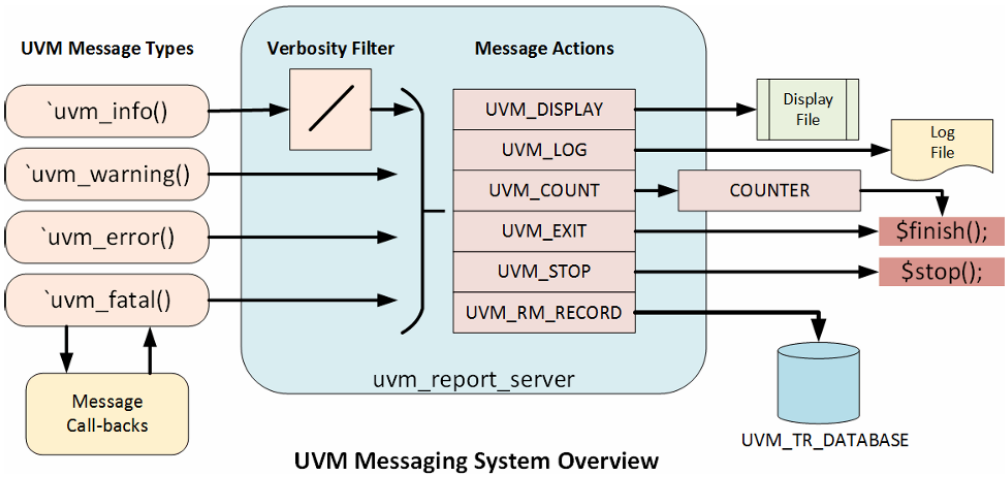


Figura 4 - Resumen del sistema de mensajes de UVM

Por defecto, un mensaje de fatal termina la simulación, un error la detiene, un warning solo imprime el aviso con una connotación de fallo y un info imprime cualquier otro tipo de aviso. Aunque la acción que realiza cada tipo de mensaje puede ser configurada.

El tipo de mensaje info incluye también un filtro de verbosidad para orientar la información que se muestra durante una prueba. Con ello se pueden separar, por ejemplo, avisos únicamente orientados a hacer depuración del sistema y los que dan información sobre la verificación. Además, tiene un sistema de etiquetas con la que se puede informar en el mensaje desde donde se está emitiendo que es uno de los datos que ofrece por defecto el informe final de UVM.

Estructura básica

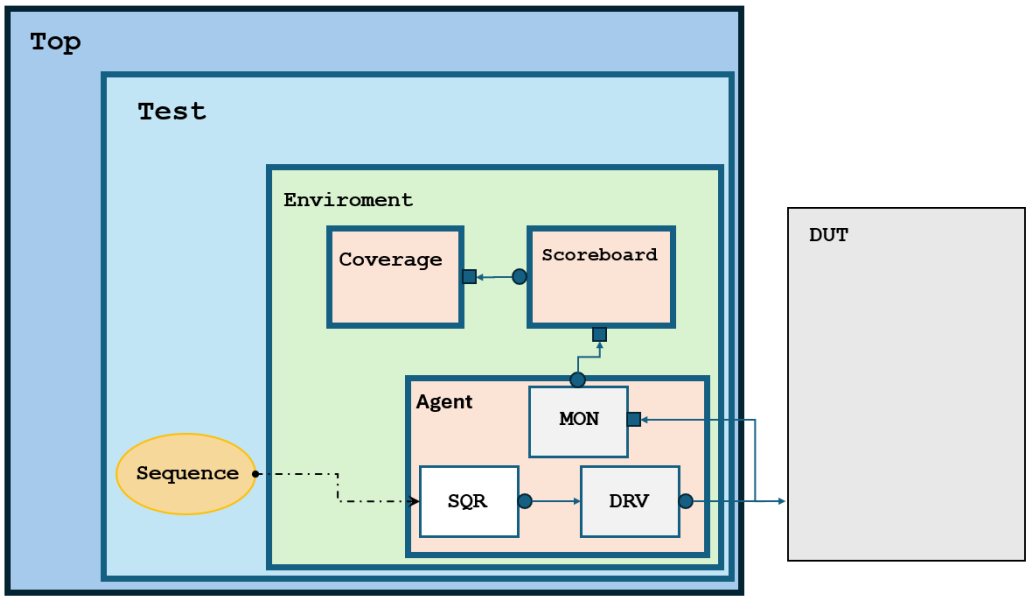


Figura 5 - Estructura básica de un sistema UVM

Desde el punto de vista del código, desde el *top* se instancian las conexiones al diseño, de aquí en adelante DUT (Desing Under Test) considerando su jerarquía dentro del sistema UVM, como también se hace una llamada al test que tiene que estar previamente definido en factory. Es común también definir señales que consumen tiempo real o que deben tener un comportamiento que no se tiene que ver afectado por la prueba en sí, como lo puede ser la señal de reloj del DUT. Ya que la documentación del estándar está toda en inglés, se darán los nombres originales junto a los nombres traducidos para facilitar la consulta de la documentación.

A nivel de test se instancia el entorno, la configuración del test y las secuencias que tendrá, siendo las dos últimas a lo que en este documento me refiero como el banco de pruebas. Según como se quiera plantear la verificación podría no ser siquiera necesario una configuración del test, valdría con crear diferentes secuencias que generen los diferentes comportamientos. Ambos acercamientos son perfectamente válidos, pero siempre es buena práctica parametrizar si se busca flexibilizar la verificación.

A nivel de entorno (Enviroment) se instancian los componentes y las conexiones entre ellos.

Un Agente (Agent), que es el componente que comúnmente termina conectado al DUT que a su vez instancia en su interior:

- un secuenciador (sequencer), que recibe los valores para cada señal de entrada que es generado en la secuencia,
- un controlador (driver), que transmite las señales del secuenciador al DUT
- un monitor, que recibe la respuesta del DUT.

En este ejemplo se considera solamente un agente, pero esto no tiene por qué ser necesariamente el caso, se podría añadir tantos agentes como secuencias se quieran controlar por separado, cabe también destacar que un agente no necesita estar conectado directamente al DUT pero sí es recomendable para simplificar la implementación del entorno.

Un Marcador (Scoreboard), que compara el funcionamiento del DUT con un modelo de referencia y registre los errores, ya sea con un simple conteo o un aviso de las condiciones con las que se produjo el error. Según el diseño a verificar, la complejidad al implementar un modelo de referencia puede variar, con un diseño simple podría bastar con un par de líneas describiendo su comportamiento llegando a ser el caso más simple un diseño combinacional como lo es un operador matemático. Modelos de referencia comunes pueden ser códigos no sintetizables, pero con la misma funcionalidad que el DUT en cuestión, pudiéndose utilizar la funcionalidad de DPI (Direct Programing interface) de SystemVerilog para comunicar el

sistema en SystemVerilog con código escrito en otro lenguaje, comúnmente C o C++ (pero no limitado a esos) y así tratar el modelo como una caja negra de cara a UVM.

Un componente de Cobertura (Coverage), que registra los estímulos que se ha transmitido al DUT como también los rangos de valores que ha generado el DUT durante la prueba. Cabe destacar que la inclusión de un componente de Cobertura implica el uso de grupos de cobertura para su manejo, siendo la otra alternativa la cobertura que ofrece SVA que puede llegar a ser más versátil ya que se instancia dentro de cada componente que se busca registrar. Este último acercamiento fue descartado de cara a este proyecto porque la herramienta Vivado v2024.2 no es compatible con la cobertura que ofrece SVA.

1.4 IMPLEMENTACIÓN DEL SISTEMA DE VERIFICACIÓN

En esta sección se explicará la implementado el sistema de verificación completo, pormenorizando paso a paso sus componentes junto a la justificación de su estructura final.

1.4.1 Esquema general

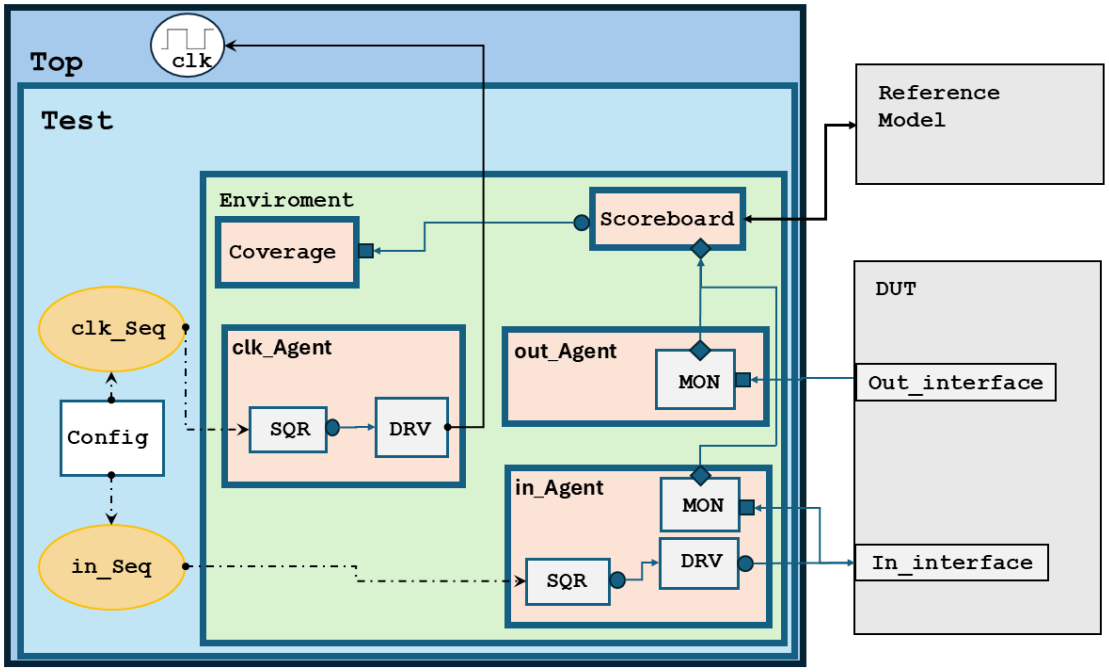


Figura 6 - Estructura del sistema de verificación implementado

En la Figura 6 está ilustrado el esquema general del sistema, se puede ver que a grandes rasgos cumple con la estructura de un entorno básico con una serie de componentes añadidos debido a la complejidad del diseño a verificar.

1.4.2 DUT

Antes de entrar en detalle en cada módulo del sistema es necesario conocer el diseño a verificar. Cabe destacar que hasta el momento en este documento no ha sido relevante, ya que todo lo explicado anteriormente es común entre entornos UVM.

Considerando que el diseño a verificar será tratado como una caja negra, lo que necesitamos saber son solo dos cosas: su interfaz, eso son sus entradas y salidas, y su comportamiento esperado según sus entradas.

Interfaz

Entradas:

`reset`: 1 bit

`sClk`: 1 bit

`sRun`: 1 bit

`vita_time`: 64 bits sin signo – número de muestras

`vita_time_trigger`: 64 bits sin signo – número de muestras

`len_PRI`: 32 bits sin signo – número de muestras

`duration_wave`: 32 bits sin signo – número de muestras

`type_wave`: 3 bits

`gain_percent`: 8 bits sin signo – porcentaje

`f_sample`: 32 bits sin signo – hercios

`bandwidth`: 32 bits sin signo – hercios

`f_signal`: 32 bits sin signo – hercios

`t_ramp`: 32 bits sin signo – número de muestras

Salidas:

`o_tdataI`: 16 bits con signo – amplitud

`o_tdataQ`: 16 bits con signo – amplitud

`o_tvalid`: 1 bit

`system_ready`: 1 bit

De cara a la verificación se hace una subdivisión en los registros de entrada, entre los que rigen el **comportamiento temporal**: `sClk`, `vita_time` y `vita_time_trigger`. Los registros de **control**: `sRun` y `reset`. Por último, los registros de **configuración** de la onda:

`f_sample`, `len_PRI`, `duration_wave`, `gain_percent`, `type_wave`, `f_signal`, `bandwidth` y `t_ramp`.

Comportamiento:

El diseño es un generador de ondas que puede generar 3 tipos de onda diferentes, chirp lineal, tono y rampa según el valor en `type_wave`.

La señal `sClk` es una señal de reloj generada a la misma frecuencia que indica el registro de frecuencia de muestreo `f_sample`, es importante que estas coincidan lo máximo posible para que se calcule correctamente la onda.

La señal `vita_time` es un contador de tiempo en muestras que inicia al "encender" el sistema y no se reinicia. La señal `vita_time_trigger` es un registro que indica cuándo se tiene que empezar a generar muestras en referencia a `vita_time`, tiene que ser mayor que `vita_time`.

El registro `sRun` funciona como una señal run/stop, el sistema se para si está a 0 e inicializa una onda, es decir configuración y generación, cuando se conmuta a 1. Tiene que mantenerse accionada durante toda la generación de la onda. La salida de la generación de la onda se produce cuando `vita_time` llega o supera a `vita_trigger_time`.

El registro `reset` pone el sistema en un estado inicial conocido previo a su funcionamiento. Se acciona cuando está a 1 y permite su funcionamiento cuando está a 0.

Los registros `len_PRI` y `duration_wave` se corresponden al intervalo de repetición del pulso (que consta de un tiempo en el que existe pulso y un tiempo en el que no hay señal) y la duración del pulso, ambos en número de muestras, por lo que su duración temporal depende de la frecuencia de muestreo.

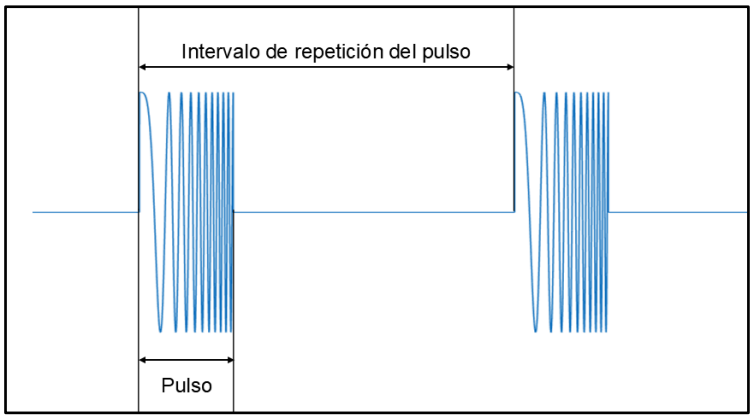


Figura 7 - Intervalo de repetición de pulsos

El registro `gain_percent` es una ganancia que cambia el valor de amplitud máximo que tendrá la onda, a 100% la amplitud máxima es de ±2^13. El registro se lee como numero entero por lo que el paso es de 1%.

Siguiente, están los registros de cada onda:

Chirp lineal:

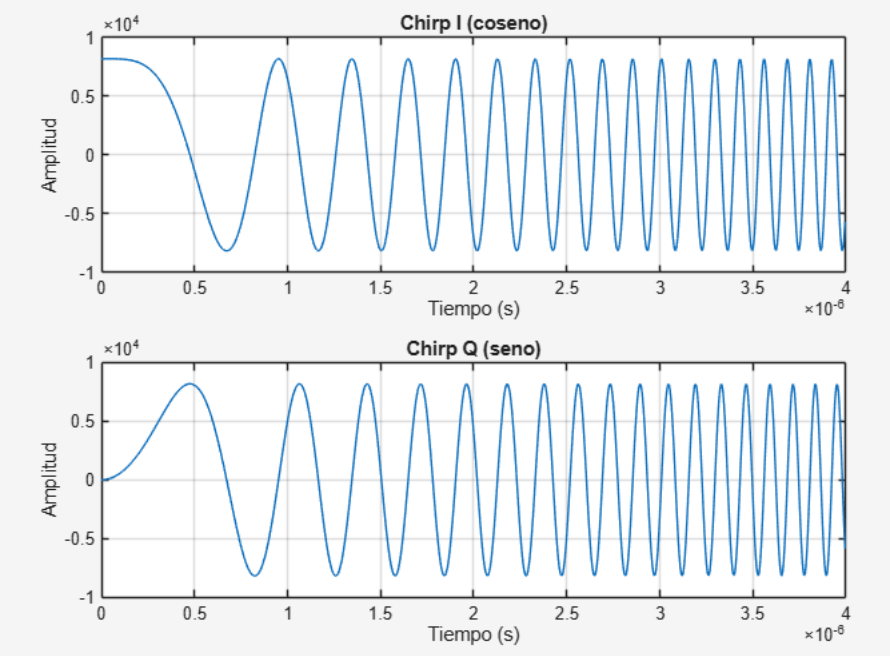


Figura 8 - Chirp en modulación IQ

Es una onda sinusoidal que varía su frecuencia de manera lineal respecto al tiempo. La fase instantánea se rige por la ecuación:

$$\varphi(\mathrm{t}) = \varphi_0 + 2\pi\left(\frac{\mathrm{c}}{2}t^2 + f_0 t\right)$$

En la que φ_0 es desfase inicial, c es el ancho de banda y f_0 es la frecuencia inicial. Como no hay desfase y la frecuencia inicial siempre es 0Hz la expresión se simplifica a:

$$\varphi(\mathrm{t}) = \pi \mathrm{c}\, t^2$$

Siendo c el valor de `bandwidth`. La fase se calcula desde $t = 0$ hasta $t = duration\_wave * (\frac{1}{f\_sample})$. Es decir, el barrido de frecuencia del chirp dura lo que dure el pulso entero.

Tono:

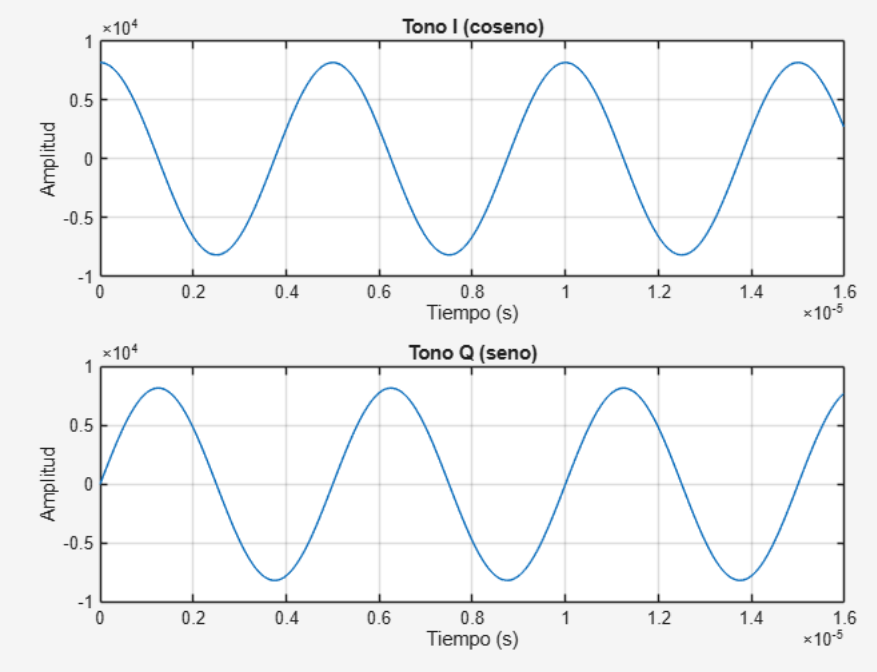


Figura 9 - Tono en modulación IQ

Es una onda sinusoidal simple cuya fase instantánea se calcula con la expresión:

$$\varphi(\mathrm{t}) = 2\pi \cdot \mathrm{t} \cdot \mathrm{f}$$

Siendo f el valor de `f_signal`.

Antes de continuar con el ultimo tipo de onda es adecuado aclarar que tanto para la generación del chirp como el tono se utiliza la modulación I/Q, es decir se generan 2 ondas simultáneamente, una con el seno y otra con el coseno de la misma fase instantánea. Esta es una técnica común en la generación de ondas de radiofrecuencia ya que ayuda a quitar

ambigüedades en la demodulación de la onda. Es por esto por lo que tanto la Figura 8 y Figura 9 están ilustradas 2 ondas con un desfase de 90º.

Rampa:

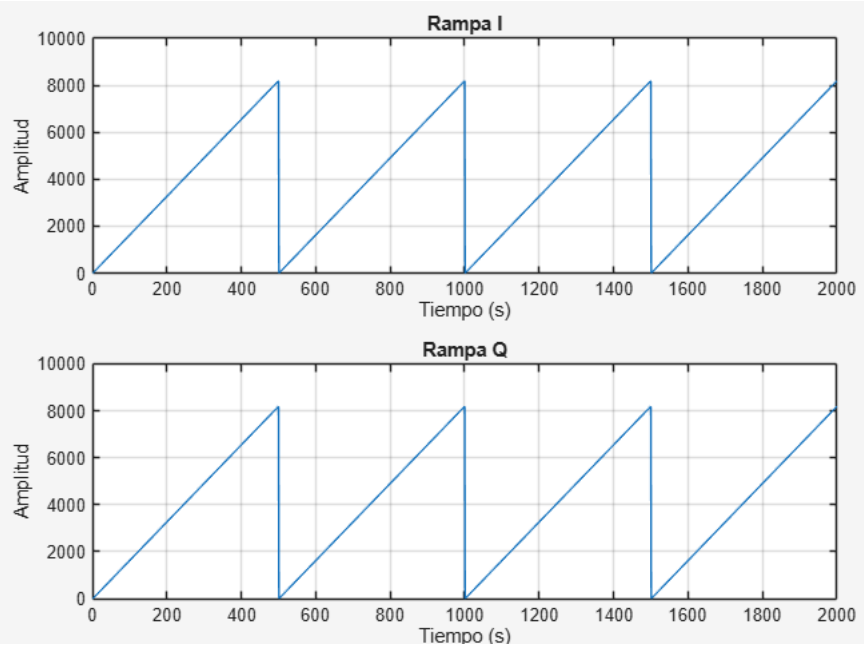


Figura 10 - Secuencia de rampas

A diferencia de las anteriores, no es sinusoidal, es una onda con un incremento lineal que se corresponde al incremento necesario para llegar a la amplitud máxima en un periodo indicado, ese periodo se corresponde al registro `t_ramp` en número de muestras.

Finalmente, están los registros output:

En los registros `o_tdataI` y `o_tdataQ` se escriben las amplitudes instantáneas de las ondas generadas. Estos registros solo tendrán valores válidos cuando el registro `o_tvalid` sea igual a 1. El registro `system_ready` es accionado por el DUT cuando la onda ha sido configurada y el sistema está listo para generar valores de onda.

1.5 IMPLEMENTACIÓN DEL ENTORNO

En el siguiente apartado, detallaré componente a componente como fue implementado el entorno de simulación. Puesto que el código completo está anexado al documento únicamente incluiré fragmentos que considere relevantes para la implementación del entorno UVM.

1.5.1 Top

El documento `wave_gen_tb_top` es la cabecera de toda la simulación, en él se instancian todas las interfaces de simulación que cabe destacar serán los registros visibles desde la simulación de vivado. Debido a esto, se consideró apropiado para la visualización instanciar el modelo de referencia como un módulo SystemVerilog al igual que el propio DUT.

En un entorno más simple es aquí donde se instanciaría el reloj que controlaría todo el sistema, pero en este caso es necesario una generación de señal de reloj dinámica y acceso a su frecuencia real dentro del entorno UVM, es por esto por lo que se hace necesario el agente reloj implementado más adelante. El top únicamente se encarga de conectar el reloj generado al resto de interfaces.

En el top se inician las fases de ejecución del sistema UVM dando inicio al test concreto ya definido en factory de UVM, que se explicará en la sección de implementación del banco de pruebas.

Por último, se asignan los interfaces virtuales a las interfaces correspondientes dentro del sistema UVM.

1.5.2 Entorno

La clase `wave_gen_env` extiende de la clase `uvm_env`, en ella se instancian todos los componentes del sistema UVM como objetos, que en este caso son agentes de puerto de entrada y salida, agente reloj, marcador y cobertura.

En la fase de construcción se crean todos los componentes y en la fase de conexión se conectan los puertos de análisis de los monitores de los agentes de entrada y salida con el marcador y el puerto de análisis del marcador con el de cobertura.

1.5.3 Agente puerto de entrada

El agente `wave_gen_in_agent` se extiende de la clase `uvm_agent`, en ella de declaran su secuenciador, controlador y monitor, que se crean en fase de construcción y se une el secuenciador con el controlador en fase de conexión.

Interfaz

Se declaran los registros que tiene la interfaz de entrada al DUT, junto a los clocking blocks para actualizar los datos a flanco de reloj positivo en el controlador y a flanco de reloj negativo en el monitor.

```
clocking dr_cb@(posedge clk);
   output reset;
   output sClk;
   output vita_time;
   output vita_time_trigger;
   .
   .
   .
   output f_sample;
   output bandwidth;
   output t_ramp;
   output gain_percent;
endclocking
modport DRV (clocking dr_cb);

clocking rc_cb@(negedge clk) ;
   input  reset;
   input  sClk;
   input  vita_time;
   input  vita_time_trigger;
   .
   .
   .
   input  f_sample;
   input  bandwidth;
   input  t_ramp;
   input  gain_percent;
endclocking
modport RCV (clocking rc_cb);
```

Figura 11 - Estructura de la interfaz del puerto de entrada del DUT

Nótese que cada clocking block especifica si cada registro es entrada o salida para el sistema de verificación.

Secuenciador

La clase `wave_gen_in_sequencer` se extiende de la clase `uvm_secuencer#(uvm_secuence_item)` parametrizada por el ítem de secuencia del

agente, en este caso es `wave_gen_in_transaction`, que será explicado en la sección de implementación del banco de pruebas. Únicamente se crea a sí misma, pero su función es de recibir lo que se genera en la secuencia correspondiente y transmitir el ítem secuencia al controlador.

Controlador

Se extiende de la clase `uvm_driver#(uvm_secuence_item)` parametrizada por el mismo ítem secuencia, en ella se instancia la interfaz virtual correspondiente, que se conecta a la interfaz declarada en el top mediante la base de datos de UVM en la fase de construcción.

En la fase de ejecución, el controlador se encarga de volcar el ítem secuencia en la interfaz virtual cumpliendo con el protocolo de handshake entre la secuencia y el controlador.

```
virtual task run_phase(uvm_phase phase);
  forever begin
      seq_item_port.get_next_item(req);
      drive();
      @(vif.dr_cb);
      $cast(rsp,req.clone());
      rsp.set_id_info(req);
      seq_item_port.item_done();
      seq_item_port.put(rsp);
   end
endtask : run_phase
```

Figura 12 - Fase de ejecución del controlador del agente de entrada

Monitor

La clase `wave_gen_in_monitor` se extiende de la clase `uvm_monitor`. En ella se instancia y conecta la interfaz virtual al puerto de la misma manera que en el controlador. También se crea el puerto de análisis que se usa para conectar el agente con el marcador.

En la fase de ejecución lee la interfaz que acaba de ser escrita en el controlador y vuelca la lectura en el puerto de análisis.

```
virtual task run_phase(uvm_phase phase);
  forever begin
    collect_trans();
    mon2sb_port.write(act_trans);
  end
endtask : run_phase
```

Figura 13 - Fase de ejecución del monitor del agente de entrada

1.5.4 Agente puerto de salida

El puerto de salida fue originalmente pensado para que manejase una secuencia en paralelo a la del agente del puerto de entrada, durante el proceso de implementación se descartó una señal de control, lo que hizo que dejase de ser necesario un controlador y un secuenciador, pero se mantuvo el agente en caso de haber sido necesario reintegrar la señal descartada.

Al igual que el agente de puerto de entrada, la clase `wave_gen_out_agent` se extiende de `uvm_agent` pero en este caso solo se encarga de instanciar únicamente el monitor.

Interfaz

En la interfaz se declaran los registros de salida siendo únicamente necesario un clocking block para el monitor, con la misma estructura que el mostrado en el agente de entrada.

Monitor

En el monitor está instanciado una interfaz virtual al puerto de salida a través de la base de datos de UVM. Está creado también el puerto de análisis que conecta al marcador.

En fase de ejecución lee los registros de la interfaz y los envía al marcador.

1.5.5 Agente reloj

En un principio no se estaba considerando la posibilidad de que el reloj fuese dinámico, por lo que los agentes de los puertos de entrada y salida recibían una señal de reloj generada en top. Para ampliar la funcionalidad del sistema de verificación se implementó un agente que generase un reloj de frecuencia variable, pero respetando todo lo que estaba implementado.

Antes de que el agente reloj fuese implementado, se generaba un reloj a una frecuencia concreta en el top y era este reloj el que accionaba cada ítem de secuencia, este es el reloj del test. Al accionarse a cada flanco de reloj positivo, resultaba en un reloj a la mitad de la frecuencia, la señal `sClk`, este es el reloj del DUT. Ya que todo estaba implementado con esta diferencia de frecuencia en mente, el agente tuvo que ser implementado para este comportamiento.

Su clase `wave_gen_clk_agent` se extiende de la clase `uvm_agent` y se encarga de crear el secuenciador, controlador y la conexión entre ambos.

Interfaz

La interfaz del agente reloj es una interfaz interna, es decir no está conectada directamente al DUT, en su lugar está pensada para que se lea desde la secuencia del agente del puerto de entrada. únicamente consta del bit de reloj y la frecuencia en hercios.

Secuenciador

La clase `wave_gen_clk_sequencer` se extiende de la clase `uvm_sequencer#(uvm_sequence_item)`, Recibe el ítem generado en su secuencia y únicamente sirve de conexión entre la secuencia y el controlador.

Controlador

La clase `wave_gen_clk_driver` se extiende de la clase `uvm_driver#(uvm_sequence_item)`, en la fase de construcción conecta su interfaz virtual con la interfaz declarada en top, que es lo que permite el acceso del reloj a nivel de top.

En su fase de ejecución se ejecutan dos hilos de manera concurrente hasta el fin de la simulación con la sentencia `fork…join_none`.

Uno de los hilos genera la señal de reloj del test directamente en la interfaz virtual a una frecuencia determinada y el otro hilo calcula el semiperiodo que tiene que respetar la señal de reloj. El semiperiodo se inicializa por defecto a 5ns y se mantiene así hasta que reciba un cambio válido desde su secuencia. Por último, calcula cual es la frecuencia correspondiente al semiperiodo calculado para que se transmita al agente del puerto de entrada. El cálculo se explicará más adelante junto a la secuencia del agente.

1.5.6 Modelo de referencia

El modelo de referencia se implementó como un módulo de SystemVerilog conectado a nivel de top, esto da acceso no solo a la interfaz del modelo sino también a las variables internas, lo que facilitó la depuración de su código para que se ajuste correctamente al módulo real frente a los mismos estímulos.

El modelo de referencia no necesita limitarse a tener únicamente los puertos del módulo real, en este caso se incluyó en su interfaz una salida que facilitaría el posterior análisis y comparación que es el registro `o_phase` de tipo real (equivalente a un tipo coma flotante de doble precisión en C) que da acceso a la fase instantánea en cada muestra al sistema UVM sin realizar un cálculo redundante de por medio.

Para su funcionamiento se implementó una máquina de estados simple de dos estados, un estado en el que espera los parámetros de onda y un estado para generar la onda.

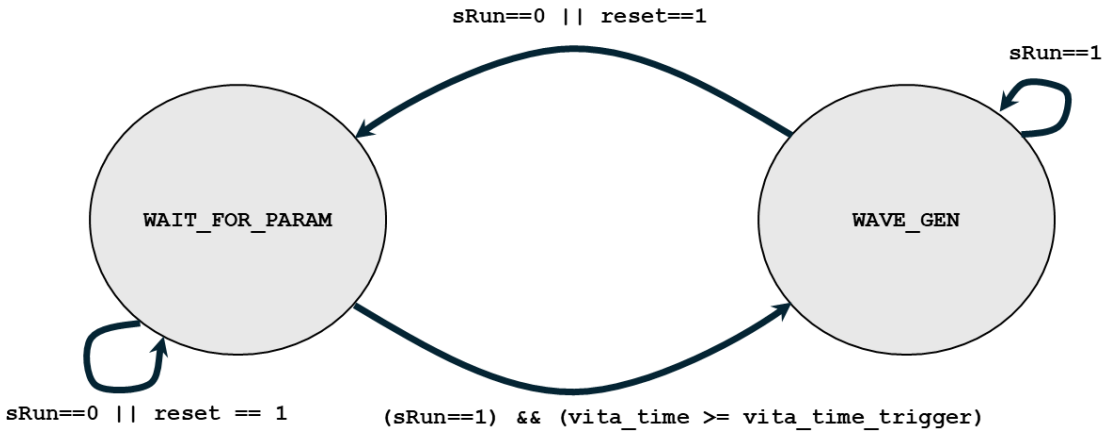


Figura 14 - Manejo de estados del modelo de referencia

El manejo de los estados se hace de acuerdo con lo explicado del comportamiento del modelo real. Es notable recalcar que el modelo de referencia no está limitado de la misma forma en la que lo está el módulo real, por lo que en este caso puede pasar de estar en espera sin leer la configuración en su puerto de entrada a generar onda de un ciclo al siguiente. Lo que esto permite es la detección de errores de desfase de ciclo cuando el módulo real no ha tenido suficiente tiempo para tener datos listos para transmitir en su puerto de salida.

En el estado `WAIT_FOR_PARAM` el modelo guarda los valores en los registros de entrada ya que tiene que ignorar todos los registros de **configuración** durante el estado `WAVE_GEN`, y hace los ajustes para compartir la lógica de generación de onda entre los 3 tipos de onda.

```
if( sRun && (vita_time >= vita_time_trigger)) begin // vuelca parametros de entrada a
registros antes de empezar a generar onda cuando vita_time_trigger se alcance
    system_ready <= 1;
    typeWave<=type_wave;
    lenPRI<=len_PRI;
    durationWave<=duration_wave;
    o_tvalid <= 0;
    i_f_signal = f_signal;
    i_f_sample = f_sample;
    i_t_ramp = t_ramp;

    i_gain_percent = gain_percent;

    real_gain = (gain) * real'(i_gain_percent)/100;
```

```
    i_bw = bandwidth;
    signalT = 1/real'(i_f_signal);
    sampleT = 1/real'(i_f_sample);
    if(type_wave==2) begin // arreglo para rampa
    signalT = i_t_ramp * sampleT;
    i_f_signal = 1/real'(signalT);
    real_gain = (gain-1) * real'(i_gain_percent)/100; // El val max que puede llegar
la rampa real es 8191 no 8192
    end else if (type_wave==0) begin // arreglo para chirp

    signalT = (duration_wave)*sampleT;// Un barrido dura un duration_wave entero
    i_f_signal = 1/sampleT;// Tomo la periodicidad de la señal como el barrido entero
    end
end
```

Figura 15 - Estado WAIT_FOR_PARAM del modelo de referencia

En el estado `WAVE_GEN` se controla el pulso de generación de la onda, cumpliendo con lo especificado en el comportamiento del propio DUT.

```
WAVE_GEN: begin
    o_tvalid <= 1;
    system_ready <= 1;
    sampleCounter <= sampleCounter+1;

    //-------- Verifica que no haya llegado al fin del periodo
    if(sampleCounter <= durationWave) begin
    ////////////// Genera la onda
    . . .
    end
    //--------
    /////////// Actualiza la salida
    if(sampleCounter<durationWave) begin // si es menor al duration_wave leido antes,
    pasa dato
        o_tdataI <= I;
        o_tdataQ <= Q;
        o_tvalid <= 1;
        o_phase <= $realtobits(r_data);
        end else begin // si sobrepasa duration_wave, vuelca 0 a la salida
        o_tdataI <= 0;
        o_tdataQ <= 0;
        o_tvalid <= 1;
        o_phase <= 0;
        signalCounter <=0;
        end

        ///////////////// Verifica que no haya pasado lenPRI
        if (sampleCounter > lenPRI - 2 ) begin
            sampleCounter<=0;
            signalCounter<=0;
        end
    end
```

Figura 16 - Estado WAVE_GEN del modelo de referencia

Por último, la generación de los 3 tipos de onda. Nuevamente, como no necesita ser sintetizable, todos los cálculos se realizan en variables de tipo real y se hace una conversión implícita al final de cada onda en la definición de `Q` e `I`, que es lo que se vuelca en los registros de salida directamente. El truncamiento que realiza implícitamente SystemVerilog es al entero más cercano, en caso de ser necesario o verse apropiado para una tendencia del modelo real se podría forzar la orientación del truncamiento, se decidió dejar el truncamiento al número entero más cercano para que el modelo difiriera lo mínimo respecto a un cálculo exacto.

```
if(sampleCounter <= durationWave) begin
////////////// Genera la onda
   if(typeWave==0) begin // chirp
      r_data = pi * ( ((i_bw) * (sampleCounter*sampleT))/ (signalT) ) *
(sampleCounter*sampleT);

      auxSinQ=$sin(r_data);
      auxCosI=$cos(r_data);

      Q = auxSinQ * (real_gain);
      I = auxCosI * (real_gain);
   end else if(typeWave==1) begin // tono
      r_data=2*pi*sampleCounter * (sampleT*i_f_signal);
      auxSinQ=$sin(r_data);
      auxCosI=$cos(r_data);

      Q = ($sin(r_data) * (gain) * real'(i_gain_percent)/100.0);
      I = ($cos(r_data) * (gain) * real'(i_gain_percent)/100.0);
   end else begin // rampa
      r_data = signalCounter * (real_gain) / real'(i_t_ramp-1);

      I = (r_data); // real a logic
      Q = (r_data);
      if(signalCounter < i_t_ramp - 1) begin
         signalCounter <= signalCounter+1;
      end else begin
         signalCounter<=0;
      end
   end
end
```

Figura 17 - Cálculo de las ondas en el modelo de referencia

1.5.7 Marcador

La clase `wave_gen_scoreboard` se extiende de la clase `uvm_scoreboard`, la clase está orientada a recibir transacciones de los diferentes monitores dentro de los agentes conectados al DUT y compararlos con un modelo de referencia.

Se declaran las interfaces tanto del módulo real como el modelo de referencia, los puertos necesarios junto a una cola fifo para las transacciones que van entrando. Todo esto se crea

en la fase de construcción de UVM. En la fase de conexión se conectan los puertos con la fifo que se asegura que los datos de ambas interfaces correspondan al mismo instante.

Es notable destacar que la fase de ejecución del marcador está controlada por la disponibilidad de transacciones en la fifo, esto es así porque como componente de análisis, el marcador no tiene prioridad en la ejecución del sistema, solo se ejecuta cuando el sistema UVM no está activamente comunicándose con el DUT.

El manejo de errores en el marcador entra también en un amplio espectro de posibilidades, desde una comparación simple entre registros hasta aplicar métricas de error para registros concretos. Se consideró tratar con todos los errores de sincronización y errores en los valores de la señal en el marcador, cada manejo de error será explicado en el orden en la que se ejecutan en cada ciclo en la fase de ejecución, siendo la métrica principal del error el cálculo de la Raíz del Error Cuadrático Medio (Root-Mean-Square Error en inglés).

RMSE

La raíz del error cuadrático medio es una métrica usada para comparar el comportamiento de una variable frente un valor de referencia, su cálculo corresponde a la expresión

$$RMSE = \sqrt{\frac{\sum_{t=0}^{T}(\hat{x}_t - x_t)^2}{T}}$$

En el que t es la muestra actual, T el numero total de muestras, $\hat{x}$ es el valor de referencia y x el valor muestreado.

Es una métrica de precisión en las mismas unidades de la muestra muy sensible a valores de error altos por lo que es adecuada para detectar rápidamente si ha sucedido o no un error a lo largo de la simulación. Idealmente si la referencia y los datos muestreados coinciden totalmente su valor es 0, pero es prácticamente imposible que este sea el caso.

Para su análisis hay que considerar que las unidades de su resultado son las mismas que las de la señal en cuestión, ya que lo que se está calculando es un error absoluto.

Fase de ejecución

```
task run_phase(uvm_phase phase);
     forever begin   // hay lógica extra para resincronizar en caso de que haya desfase
     // Esta corrección está para detectar un desfase sin desechar la onda entera
         input_port_fifo.get(in_trans);
         output_port_fifo.get(out_trans);
         build_actual();
```

```
            build_actual_out();
            if(!ignoreNext) begin
                send_to_ref_model();
                @(vif.rc_sb);
                get_from_ref_model();
                if(actual.sClk) begin
                    compare_data();
                    if(!ignoreNext) begin
                        rmse_sum();
                        sb2cov_port.write(actual);
                    end
                end
            end else begin
                ignoreNext=0;
            end
        end
    endtask
```

Figura 18 - Fase de ejecución del marcador

La primera comparación es una comparación simple entre los registros. Para los registros de entrada no hay una razón por la que puedan ser diferentes, por lo que si lo son se puede asumir que hay un fallo con el sistema de verificación, por lo que salta directamente una advertencia con el tipo de mensaje `uvm_warning.

La segunda comparación realizada es que la señal de o_tvalid, que indica que el valor que está en la salida corresponde a la onda configurada, coincida entre el módulo real y el modelo. Como la única posibilidad de desfase es que el modelo se adelante, se acciona una bandera que indica que está adelantado (el bit ignoreNext) que detiene el modelo de referencia hasta que la señal de o_tvalid del DUT se accione. Además, se hace un conteo del número de desfases en una simulación.

Con esta verificación se logran dos cosas, la primera es la detección de desfases, lo que permitió caracterizar el número de ciclos que necesita el módulo real para empezar a transmitir la onda que genera y la resincronización de la señal para no tener que descartar una onda completa para hacer la comparación.

```
    if((actual.o_tvalid != expected.o_tvalid)) begin
       `uvm_warning("SCOREBOARD", $sformatf("o_tvalid FAILED at vita=%d"
       ,actual.vita_time))
       ignoreNext =1;
       desyncFail=desyncFail+1;
     end
```

Figura 19 - Detección de desincronización

Las siguientes comparaciones se realizan en la llamada a la tarea `rmse_sum`, que comenzó como un acumulador de errores en cada `o_tdata`. Para ello se implementó una máquina de estados para únicamente contabilizar las muestras que correspondan a la parte activa del pulso de generación de onda.

Se consideraron 4 estados:

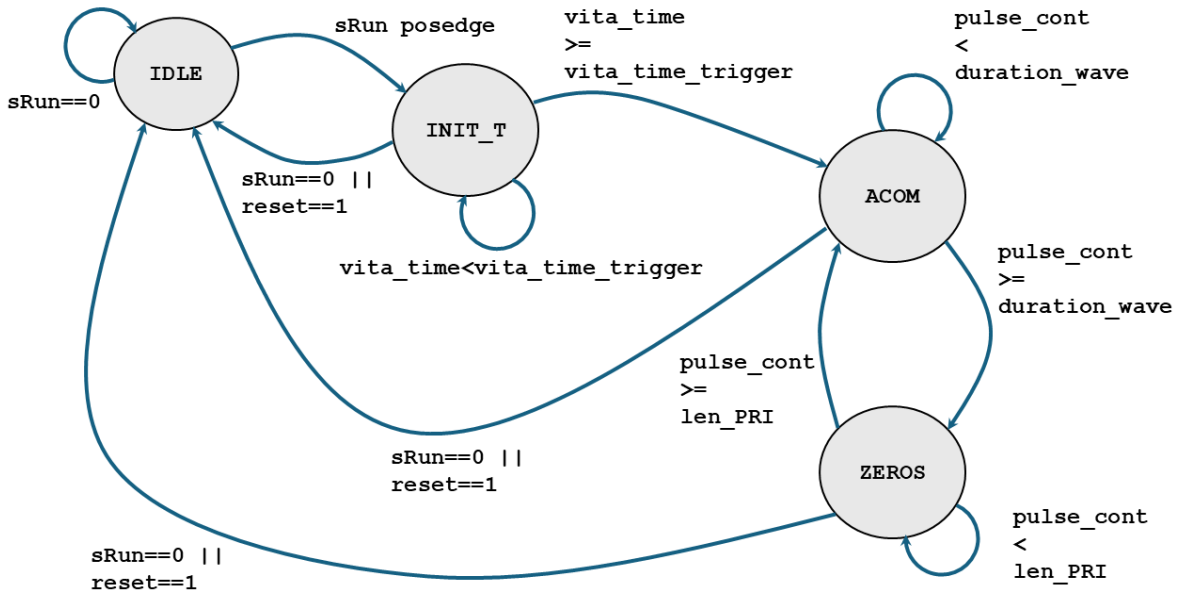


Figura 20 - Manejo de estados del acumulador de errores

`IDLE`, que corresponde a la espera de configuración de la onda.

`INIT_T`, que corresponde con el periodo entre la configuración y la generación de datos.

`ACOM`, que corresponde con el periodo activo del pulso de generación.

`ZEROS`, que corresponde con la generación de ceros en el resto de pulso de generación.

Para controlar los estados fue necesario un contador auxiliar `pulse_cont` para saber si se están generando valores de onda o ceros. Es necesario establecer esta distinción para únicamente tomar en cuenta los valores de onda generados para la acumulación del error, de lo contrario, las muestras en las que la salida son ceros, que siguen contando como muestras válidas, disminuirían artificialmente el error.

```
case(state)
   IDLE: begin ; end
   INIT_T: begin  pulse_cont<=0; end
   ACOM: begin pulse_cont<=pulse_cont+1;
      if(actual.o_tvalid) begin

      err_I = actual.o_tdataI - expected.o_tdataI;
      err_Q = actual.o_tdataQ - expected.o_tdataQ;
      if( err_I>41 || err_Q>41 || err_I<-41 || err_Q<-41) begin
         // si el error es más del 0.5%
         `uvm_info("SCOREBOARD", $sformatf("Readings \n actual.I=%d actual.Q=%d \n
         expected.I=%d expected.Q=%d \n errI = %d errQ=%d \n at vita: %d",
         actual.o_tdataI,actual.o_tdataQ,expected.o_tdataI,expected.o_tdataQ
         ,err_I,err_Q,actual.vita_time), UVM_HIGH)
      end

      //  `uvm_info("SCOREBOARD", $sformatf("\n [Precalc] com_err_I:%d com_err_Q:%d -- at
           vita: %d ",com_error_I,com_error_Q,actual.vita_time ), UVM_HIGH)
      com_error_I= com_error_I + ((err_I)**2);
      com_error_Q= com_error_Q + ((err_Q)**2);

      valid_samp=valid_samp+1;
      track_peak_error();
      histogram_error(err_I, err_Q);
      //$fdisplay(csv_chirps,"%d, %d, %d, %d,%0.12f",
        actual.o_tdataI,actual.o_tdataQ,expected.o_tdataI,expected.o_tdataQ,
        $bitstoreal(expected.o_phase));
      //`uvm_info("SCOREBOARD", $sformatf("\n com_err_I:%d com_err_Q:%d -- at vita:%d ",
         com_error_I,com_error_Q,actual.vita_time ), UVM_HIGH)
        end
     end
  ZEROS: begin
     if(pulse_cont>=actual.len_PRI) begin
        pulse_cont=0;
     end else begin
        pulse_cont=pulse_cont+1;
     end
  end
endcase
```

Figura 21 - Estructura del acumulador de errores

Por orden de aparición en el código, lo que sucede en el estado ACOM es:

Un mensaje por terminal para depuración que avisa que se ha excedido un error del 0.5% respecto al valor máximo de 8191 (2^13 - 1) junto al tiempo en `vita_time` para encontrarlo fácilmente.

El sumatorio de los errores cuadrados para el posterior cálculo del RMSE con el conteo de muestras consideradas en el cálculo.

Un rastreo de los picos de error positivos y negativos por cada registro `o_tdata`, que también guarda el tiempo en el que sucedió y la fase a la que corresponden dichos valores. Se

esperaba que los errores máximos correspondieran al chirp o al tono por la dificultad añadida que es el cálculo de funciones trigonométricas en hardware.

Un conteo de rangos de error para realizar un histograma de los errores una vez hubiese acabado la simulación. Con este se podrían encontrar tendencias en los errores de manera visual.

Fase de extracción

En el marcador se usa también la fase de extracción que se ejecuta una vez han acabado las secuencias de entrada del sistema.

En la fase de extracción se realizan los cálculos necesarios para mostrar las métricas consideradas en la prueba para mostrarlas por terminal junto al informe general de UVM. De manera opcional también crea un archivo csv con todas las métricas orientado a facilitar su almacenamiento y análisis.

```
UVM_INFO C:/Xilinx/Vivado/2024.2/data/system_verilog/uvm_1.2/xlnx_uvm_package.sv(19968) @ 2704000000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO ../../../../../uvm_srcs/wave_gen_scoreboard.svh(437) @ 2704000000: uvm_test_top.env.sb [SCOREBOARD] times desync:                  0
UVM_INFO ../../../../../uvm_srcs/wave_gen_scoreboard.svh(460) @ 2704000000: uvm_test_top.env.sb [SCOREBOARD]
RMSE I: 13.119, RMSE Q: 15.428, valid samples: 104146
UVM_INFO ../../../../../uvm_srcs/wave_gen_scoreboard.svh(463) @ 2704000000: uvm_test_top.env.sb [SCOREBOARD]
Error [max I: 1 phase: 3.261211 rad time:               274 ] | [min I: -34 phase: 4.325574 rad time:              3434 ]
Error [max Q: 58 phase: 3.157454 rad time:              3262]  | [min Q: -58 phase: 3.109432 rad time:              3261]
```

Figura 22 - Informe final de la verificación por terminal

```
[SCOREBOARD] Bin 35 [ -10:  -9] I:4505 Q:2175
[SCOREBOARD] Bin 36 [  -8:  -7] I:4626 Q:3928
[SCOREBOARD] Bin 37 [  -6:  -5] I:4922 Q:4888
[SCOREBOARD] Bin 38 [  -4:  -3] I:5990 Q:5720
[SCOREBOARD] Bin 39 [  -2:  -1] I:23592 Q:15525
[SCOREBOARD] Bin 40 [   == 0    ] I:15959 Q:20242
[SCOREBOARD] Bin 41 [   1:   2] I:2911 Q:8983
[SCOREBOARD] Bin 42 [   3:   4] I:0 Q:6290
[SCOREBOARD] Bin 43 [   5:   6] I:0 Q:5786
[SCOREBOARD] Bin 44 [   7:   8] I:0 Q:2870
[SCOREBOARD] Bin 45 [   9:  10] I:0 Q:1704
```

Figura 23 - Conteo de errores para histograma

Código opcional para análisis pormenorizado

De manera opcional, estando comentado en el código anexado, están incluidas las funciones para guardar en un csv los datos necesarios para el cálculo de cada muestra de una onda, además de las propias muestras generadas por el DUT y el modelo de referencia. Sirve para

facilitar un análisis pormenorizado de una onda en específico en otro entorno como lo puede ser una hoja de cálculo Excel o Matlab.

Está comentado por su consumo innecesario de recursos de computación durante una simulación por lo que es una funcionalidad reservada a pruebas cortas más dirigidas, pero resultó ser útil una vez encontrada una fuente de error para analizar a fondo.

1.5.8 Cobertura

La clase `wave_gen_coverage` se extiende de la clase `uvm_subscriber#(uvm_sequence_item)`, está orientada a recibir transacciones de un único puerto con la estructura con la que se parametriza y a ejecutarse cada que reciba una transacción. Para agrupar tanto entradas como salidas en una sola transacción utilizo la estructura creada para el modelo de referencia, ya que tiene todos los registros en una sola interfaz.

Antes de comenzar con la explicación del contenido del componente, es conveniente recalcar que, si bien vivado soporta gran parte de las funcionalidades que ofrece SystemVerilog, una de las que no, es la cobertura mediante SVA, lo que deja como único método compatible para la cobertura funcional a los grupos de cobertura.

Grupos y puntos de cobertura

Es un constructo de SystemVerilog que permite la cobertura funcional. Lo fundamental de esta forma de controlar la cobertura es que es muestreada, es decir, no considera todos los eventos de los registros consideradas si no solo el estado de los registros en un momento determinado y su cambio respecto a la muestra o muestras anteriores.

La pieza central de esta metodología son los puntos de cobertura: cada punto de cobertura está asociado a un registro en concreto. Cuando dicho registro es muestreado el punto de cobertura acumula en un contador la coincidencia dentro de unos rangos definidos por el usuario. Cada uno de estos rangos se corresponde a un contador llamado bin y pueden ser definidos de manera implícita o explicita. Si no se especifica nada se crea un bin por cada posible valor de una variable. Por ejemplo, a una variable de 8 bits se le asociaría de manera implícita 256 contadores, uno por cada valor que puede tener, por lo que es conveniente especificar de manera explícita los rangos que se esperan.

Como se mencionó, un bin no se limita al valor actual, si no que también permite capturar el cambio de un valor o grupo de valores a otro valor o grupo de valores específicos, como

también permite condicionar el conteo a otro registro por separado. Lo primero es útil para, por ejemplo, contar solo cuando la señal de control conmuta, el lugar de contar cada ciclo de reloj en el que está activado y lo segundo resulta útil cuando un valor solo se tiene en cuenta en una condición especifica.

Un acercamiento al conteo condicionado que no llegó a incluirse en la implementación son los cruces de bins, en el que se crean bins adicionales correspondientes a las combinaciones de cumplimiento de los bins indicados, que, si bien acortaría el proceso de definición de los bins considerados, se optó por una declaración de los condicionales explicita para facilitar la legibilidad en código.

Los puntos de cobertura se encapsulan en grupos de cobertura para asignar opciones de cara al informe de cobertura compartida entre puntos de cobertura y para unir los que tienen que ser muestreados a la vez.

En la implementación se optó por crear 2 grupos de cobertura, uno que sería muestreado cada ciclo de reloj del DUT y otro que solo sería muestreado al momento de configurar la onda a generar.

Para gestionar el instante de muestreo utilizo el contador de tiempo del DUT y varias banderas, que es una manera alternativa para saber en qué estado se encuentra el DUT que la utilizada en el marcador. Se optó por esta forma porque facilita el acceso a los diferentes retardos entre señales que se generan en la secuencia del puerto de entrada.

El siguiente fragmento se corresponde a la última llamada de función en el marcador (`sb2cov_port.write(actual);`) definida en la clase de cobertura, lo que marca el fin de un ciclo en fase de ejecución dentro del entorno de UVM.

```
function void write(T t);
    this.cov_trans = t;
    if(prev_sRun && !cov_trans.sRun) begin
        v_dropsRun=cov_trans.vita_time;
    end
    if(!prev_sRun && cov_trans.sRun) begin
        vita_delay = cov_trans.vita_time_trigger - cov_trans.vita_time;
        v_sRun_delay = cov_trans.vita_time - v_prev_sRun;
        v_prev_sRun = cov_trans.vita_time;
        v_config=v_prev_sRun;
        completeCycles=0;
        v_dropsRun = cov_trans.vita_time - v_dropsRun;
        sync_sRun_cg.sample();
    end
    // Control de interrupciones a traves de flags
    if(( cov_trans.vita_time - v_config )<= cov_trans.duration_wave) begin
        midPulse=1;
```

```
        midZeros=0;
        sync_sClk_cg.sample();
    end else if((cov_trans.vita_time-v_config) > cov_trans.duration_wave) begin
        midPulse=0;
        midZeros=1;
        if((cov_trans.vita_time-v_config)>=cov_trans.len_PRI) begin
            completeCycles=completeCycles+1;
            sync_sClk_cg.sample();
            v_config=cov_trans.vita_time;
        end else begin
            sync_sClk_cg.sample();
        end
    end
    prev_sRun=cov_trans.sRun;
endfunction
```

Figura 24 - Manejo del muestreo para cobertura

Como los puntos de cobertura dependen de un comportamiento esperado, veo conveniente especificar cuales fueron considerados una vez esté explicada el comportamiento que se busca provocar en la secuencia.

1.6 IMPLEMENTACIÓN DEL BANCO DE PRUEBAS

Una vez establecido un entorno funcional, es necesario definir la serie de estímulos con las que se busca verificar el comportamiento del DUT. Un posible comportamiento podría ser aleatorizar todas las variables en todo su posible rango de valores, pero esto sería poco productivo por una razón de peso; el módulo ha sido diseñado con una serie de parámetros de funcionamiento en mente, por lo que estos parámetros se deben tener en cuenta al limitar la aleatorización.

El proceso de implementación de una prueba versátil fue un proceso iterativo, del cual el documento se centrará solo en la estructura final de la prueba, ya que todas las versiones previas de la prueba fueron encapsuladas en un solo test.

A modo de contexto, las primeras pruebas tomaban algún comportamiento, como lo puede ser el periodo activo y frecuencia de los pulsos de generación de onda para dejarlos fijos frente al resto de las variables aleatorizadas, es decir, se guiaba el comportamiento con una variación reducida. Cada grupo de variables fijas que definían un comportamiento suponía crear una secuencia concreta que aplicar en un test. Con la implementación final se encapsuló los diferentes comportamientos planteados en una sola secuencia llamada desde un único test valiéndose de la base de datos de UVM para configurar cada prueba.

1.6.1 Planteamiento del test

El test fue planteado como una prueba que facilite ser guiada a diferentes comportamientos, debido a que uno completo largo consume mucho tiempo y recursos computacionales, si lo que se quiere comprobar en el momento está más o menos localizado, se puede orientar un test corto en el que esté prácticamente asegurado que dentro de los posibles estímulos generados se encuentren ejemplos de lo que se busca comprobar, para ello se plantearon 4 modos de test y diferentes seleccionadores de aleatorización.

Primero, hay un seleccionador independiente al modo de test por cada variable que se consideró aleatorizable, con la que se escoge si se usan o no valores aleatorios para esa variable en el test. Si no se usan valores aleatorios se tiene que definir el valor fijo de dicha variable.

El resto de los seleccionadores están para escoger en qué rango se aleatoriza una variable. Tomando por ejemplo el registro `f_samp`, la frecuencia de muestreo, están considerados 4 rangos de aleatorización según el valor del selector.

```
0: [100MHz:80MHz] 1: [80MHz:50MHz] 2: [25MHz:80MHz] 3: [1MHz:25MHz]
```

Con esto se puede orientar el análisis sabiendo que el resto de las variables está en el mismo orden de frecuencia o uno diferente y como afecta a los valores generados.

Dicho esto, los modos de test son:

Cada selector por separado

Se considera cada uno de los selectores por separado y se mantiene el comportamiento de los selectores escogidos durante todo el test.

Selectores compartidos

Se toma en cuenta un único rango para que todas las variables estén más o menos en el mismo orden de magnitud de frecuencia durante todo el test.

Barrido de comportamientos

Durante la duración del test cambia los selectores de manera conjunta para simular comportamientos desde la máxima frecuencia a la mínima, a modo de una revisión general.

Hasta 1/16 de la duración del test va a máximas frecuencias, disminuye al siguiente rango hasta 1/4, luego disminuye al siguiente rango a 1/2 y la última mitad del test lo hace en los rangos de frecuencia mínima.

Totalmente Aleatorio

Aleatoriza los selectores con lo que los valores aleatorios quedan dentro de los parámetros de diseño y se generan interacciones dentro de dichos parámetros que no podrían haber sido generados empleando los modos anteriores.

1.6.2 Configuración de test.

La clase `wave_gen_config` usada para la configuración del test se extiende de la clase `uvm_object`, para que forme parte del sistema UVM y pueda ser llamado desde su base de datos. La clase está compuesta de estructuras con los registros necesarios para cada uno de los selectores antes mencionados.

Exceptuando el selector de modo y el selector compartido usado en su respectivo modo, las variables consideradas para aleatorizar son: `f_sample`, `len_PRI`, `duration_wave`, `type_wave`, `bandwidth`, `f_signal`, `t_ramp` y `gain_percent`. Cabe destacar que dichos registros son directamente parte del puerto de entrada del DUT y, además, se introducen 4 variables aleatorizables más que se encargan de retrasos entre señales de control en número de muestras. Estas son: `vita_delay`, el número de muestras que tarda en alcanzarse el `vita_time_trigger` desde el accionamiento de `sRun`, `sRun_delay`, el número de muestras entre desactivaciones `sRun` que marca el momento de reconfiguración de las ondas generadas, `sRun_drop`, el número de muestras que la señal `sRun` permanece desactivada y `reset_delay`, el número de muestras que tarda en accionarse una señal de `reset` desde la anterior.

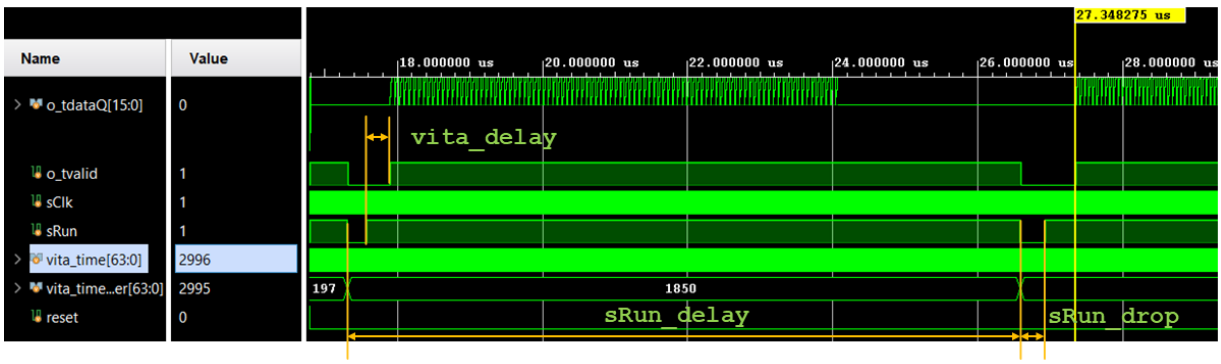


Figura 25 - Diferentes retrasos de señal aleatorizados

1.6.3 Secuencia de agente de puerto de entrada

La secuencia se planteó parametrizada para facilitar su reutilización. Su estructura está definida en tres partes:

1. El ítem secuencia que es la unidad considerada para la transacción entre componentes de UVM.
2. Una secuencia genérica base que configura lo necesario para su correcta definición en UVM.
3. La secuencia configurada.

ítem secuencia

El ítem de secuencia es la clase `wave_gen_in_transaction` que se extiende de la clase `uvm_sequence_item`, para hacerla de uso genérico únicamente contiene los registros del puerto de entrada del DUT. En la definición de los elementos de la transacción puede estar definido también si serán aleatorios. Definirlos de tipo aleatorios (`rand`) hace que se aleatorice desde SystemVerilog, que es la manera apropiada de tratar con aleatorizaciones usando UVM. Cabe destacar también que lo que se aleatorice de otra forma se tiene que declarar únicamente con su respectivo tipo, como es el caso, por ejemplo, de la frecuencia de muestreo que se aleatoriza desde la secuencia del agente reloj o la señal `sRun` donde lo que se aleatoriza son retrasos en su comportamiento, no la señal en sí.

En esta clase también están definidas las restricciones de aleatorización. Las restricciones pueden ser condicionadas a otro valor aleatorio, mientras no se defina nada que sea explícitamente contradictorio con otra restricción, la gestión de valores aleatorios de SystemVerilog se encarga de que se cumplan las condiciones.

Un ejemplo de una restricción condicionada está en el siguiente fragmento, en el que una sola restricción se aplica a dos variables.

```
constraint duration_wave_shorterThan_lenPRI {
   duration_wave <= (len_PRI*9)/10; //90%
   duration_wave >= (len_PRI)/50; // 20%
   solve len_PRI before duration_wave;
}
```

Figura 26 - Restricción de aleatorización condicionada

Dicho lo anterior, para describir los diferentes rangos posibles de una sola variable, fue necesario describir restricciones contradictorias, un ejemplo de ello en el siguiente fragmento.

```
constraint f_signal_fast {
   f_signal inside {[32'd25_000_000:32'd50_000_000]};
}
constraint f_signal_med {
   f_signal inside {[32'd10_000_000:32'd25_000_000]};
}
constraint f_signal_slow{
   f_signal inside {[32'd1_000_000:32'd10_000_000]};
}
```

Figura 27 - Varios rangos de aleatorización para un mismo registro

Por lo que se hace necesaria una forma de tratar con restricciones contradictorias.

Secuencia genérica base

La clase `wave_gen_in_basic_seq#(string cfg_name)` se extiende de la clase `uvm_sequence#(wave_gen_in_transaction)`, y se encarga de todo lo previo a la fase de ejecución en la secuencia, fue creada para extender la secuencia configurada a partir de ésta y ser parametrizada con una configuración.

En ella se instancian el objeto de ítem de secuencia, el objeto de configuración y la interfaz con el agente reloj, además de variables auxiliares.

Al inicio de la fase de ejecución, usa la base de datos para conectar la interfaz y leer la configuración del test y prepararlo si el comportamiento del test no cambia a lo largo del test. Después muestra por terminal la configuración escogida para el test a modo de confirmación para comprobar que se configuró correctamente.

```
[PRE-SEQUENCE] Every value is randomized
[PRE-SEQUENCE] Test mode:3
[PRE-SEQUENCE] Every range is chosen randomly at each sRun negative edge
```

Figura 28 - Mensaje de resumen de la configuración seleccionada previa al test

```
[PRE-SEQUENCE] Used at least one fixed value
 [PRE-SEQUENCE] f_sample fixed to: 100000000 Hz
 [PRE-SEQUENCE] gain_percent fixed to:100
 [PRE-SEQUENCE] Test mode:2
 [PRE-SEQUENCE] Behaviour sweep, defined in sequence
```

Figura 29 - Mensaje de resumen de configuración distinta

Por último, la clase contiene una serie de tareas con las que se gestionan las restricciones contradictorias. Cada una de ellas funciona de manera similar, por lo que un ejemplo es suficiente para aclarar su trato.

```
task activate_f_signal_constraint(shortint sel);
// Desactiva todas
   req.f_signal_fast.constraint_mode(0);
   req.f_signal_med.constraint_mode(0);
   req.f_signal_slow.constraint_mode(0);
   if(sel>2) begin randomize(auxsel) with {auxsel inside {[0:2]};};
   end else auxsel=sel;
   // Activa sólo la seleccionada
   case(auxsel)
      0: req.f_signal_fast.constraint_mode(1);
      1: req.f_signal_med.constraint_mode(1);
      2: req.f_signal_slow.constraint_mode(1);
      default:;
   endcase
endtask
```

Figura 30 - Gestión de restricciones contradictorias

Primero se desactivan cada una de las restricciones asociadas a la variable, si el selector está dentro de uno de sus valores válidos se activa únicamente dicha restricción. Si el número es mayor, se aleatoriza el selector, con lo que se logra el comportamiento del modo totalmente aleatorizado y si el selector es negativo deja desactivadas todas las restricciones, con lo que se puede definir un valor fijo para la variable.

Como se aplica a la variable dentro del objeto `req`, que es creado en cada iteración del ciclo de secuencia, es necesario hacer la llamada a la tarea de manejo de restricciones en cada ciclo.

Secuencia configurada

La clase `wave_gen_in_seq_customRandTest#(string cfg_name)` se extiende de la secuencia base antes explicada, es esta en la que se define la generación de la secuencia de estímulos y al ser la última en su jerarquía de clase es la que se tiene que registrar en la base de datos de UVM. En esta clase se declaran múltiples variables auxiliares para controlar la generación de secuencias de una manera que sea fácilmente legible para su análisis. Está implementado de manera que únicamente se aleatoricen los valores cuando vayan a ser utilizados para evitar ruido en las señales de entrada. En la siguiente imagen se muestra un fragmento de la simulación pudiéndose ver las variables fácilmente para facilitar el análisis posterior en caso de dar con un error.

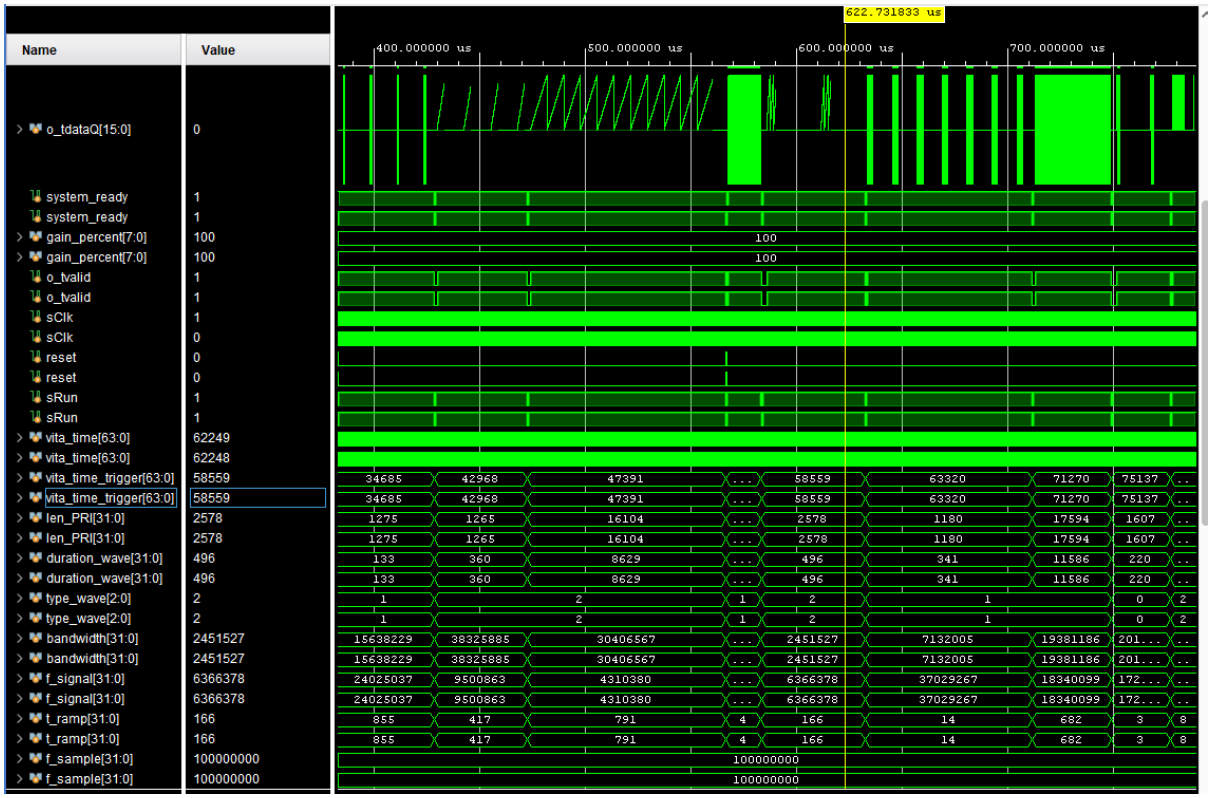


Figura 31 - Visión general de simulación de formas de onda en Vivado

El bucle de generación de la secuencia en sí puede separarse en 5 fases.

Ejecución en todas las transacciones

El número de transacciones está definido en un archivo por separado de `defines`, el bucle de creación de transacciones necesita tener un final definido para que llegue la condición de finalización de fase de ejecución. Esta es que no haya una transacción siguiente.

Dentro del bucle se genera una señal de reloj del DUT a la mitad de la frecuencia de lo que está generando el agente reloj, ya que la secuencia solo se ejecuta a flanco positivo de dicho reloj. Se utiliza el reloj del DUT para controlar el resto de la secuencia.

También se crea el objeto `req` que es la transacción que será enviada al controlador.

Manejo de modos que varían el comportamiento en fase de ejecución

Se asigna los valores fijos a las variables que fueron definidas como tal, además de hacer el cambio de selectores si el modo de barrido de comportamientos fue seleccionado.

Ejecución en flanco positivo de reloj

En el flanco positivo de reloj del DUT aumenta el contador de `vita_time` y después se gestionan las variables de control, es decir `sRun` y `reset`, que, si bien no están aleatorizadas como tal, sí se aleatoriza el retardo del accionamiento de cada una como se explicó en la configuración del test. Para ello se utiliza el propio `vita_time`, guardando su valor en el momento de la última conmutación de la señal, comparado con el `vita_time` actual aleatorizando la diferencia que debe tener para volver a accionar la señar y volver a aleatorizarla.

Como ejemplo del manejo tomo la señal de `reset`, que es ligeramente más simple que la de `sRun`. Cabe destacar que si está escogido el uso de valores fijos (`cfg.use_random.variable=0`) el valor ya estaría definido. También es destacable que como no se habían definido los rangos de aleatorización de las variables de retardo de los registros de control se definen aquí, con una estructura similar a las tareas encargadas de desactivar las restricciones de aleatorización.

```
if((vita_counter - prev_reset_counter) >= reset_delay) begin
   req.reset = 1;
   prev_reset_counter = vita_counter;
   if(cfg.use_random.reset_delay) begin
      if(cfg.sel.reset_delay>3) begin
         randomize(aux_sel) with {aux_sel inside {[0:3]};};
      end else begin
         aux_sel= cfg.sel.reset_delay;
      end
      case(aux_sel)
      0: randomize(reset_delay) with {reset_delay inside {[32768:65536]};};
      1: randomize(reset_delay) with {reset_delay inside {[65536:262144]};};
      2: randomize(reset_delay) with {reset_delay inside {[262144:1048576]};};
      3: randomize(reset_delay) with {reset_delay inside {[1048576:4194304]};};
      endcase
   end
end else begin
   req.reset=0;
end
   prev_reset= req.reset;
```

Figura 32 - Aleatorización de intervalo entre resets

La diferencia con la señal `sRun` está en que dicha señal necesita también mantenerse un número aleatorio de ciclos desactivada, este retardo se gestiona de manera similar.

Reconfiguración de la onda

La reconfiguración de la onda, es decir, la aleatorización de todos los registros de configuración solo ocurre cuando la señal `sRun` cae o la señal `reset` se acciona. En ese

ciclo de reloj se define cuantos ciclos `sRun` permanecerá a 0, cuantos ciclos de reloj del DUT pasará entre que se accione la señal `sRun` y `vita_time` llege al `vita_time_trigger` para que de inicio la generación de la onda.

Se aplican las selecciones de restricciones para el objeto `req` actual y se aleatoriza lo que aún no tiene un valor definido con una aserción para que resulte en un error que detenga el sistema si no llega a poder ser posible aleatorizar correctamente dentro de los rangos indicados.

Por último, se guardan los valores de la transacción actual para ser usados en el flanco de reloj negativo que viene o hasta que los valores sean aleatorizados una vez más.

Ejecución en flanco negativo de reloj

Se mantienen los valores usados en el flanco positivo anterior, la única diferencia es que la señal de reloj del DUT cambia a 0, pero como se utiliza una transacción, se tiene que definir el objeto completo para enviarlo al controlador.

Por último, también ejecutado en todas las transacciones ya que es necesario para la transmisión de la secuencia del secuenciador al controlador, se realiza el handshake mencionado en el controlador.

Una vez terminado el ciclo de generación de secuencias, se acciona una bandera en el objeto de configuración para sincronizar el fin de la secuencia con la generada en el agente reloj. Ejecutando una única secuencia esto resultaría innecesario, pero de cara a un test más largo es necesaria para sincronizar la reconfiguración de las secuencias.

1.6.4 Secuencia de agente reloj

La secuencia del agente reloj consta de dos partes ítem de secuencia y secuencia configurada, pero en este caso, el controlador tiene una participación mayor a solo volcar el objeto de transacción en su interfaz.

Ítem secuencia

El ítem secuencia es una clase llamada `wave_gen_clk_transaction` se extiende de la clase `uvm_sequence_item`, el objetivo de la secuencia es aleatorizar frecuencias de un reloj

que también tiene que ser generado por su agente, para ello se optaron por dos acercamientos, aleatorizando frecuencia y aleatorizando el periodo.

Finalmente se optó por la aleatorización del periodo porque permite evitar los errores de decimales en el periodo a una determinada frecuencia.

Tomemos, por ejemplo, una frecuencia que dé lugar a un periodo con muchos decimales.

$$T = \frac{1}{f} = \frac{1}{56MHz} = 17.85714286 \ldots ns$$

El simulador no puede generar correctamente esa frecuencia, haciendo un redondeo en el semiperiodo que genera de acuerdo con la precisión del tiempo definido en top de 1 picosegundo, por lo que el reloj va a una frecuencia ligeramente diferente en el orden de los kHz, pero todos los cálculos en el DUT se realizan con los 56MHz exactos, por lo que ya no solo hay un error en la generación del reloj si no en los valores generados respecto a la frecuencia en la que son generados.

Este error se puede disminuir significativamente cuando se consideran rangos de periodos ya que es la unidad con la que trata directamente las sentencias de tiempo de SystemVerilog. Para asegurar que se respete ese periodo en la simulación, se aleatoriza el periodo con un paso de 10 picosegundos, el fragmento de las restricciones está en la Figura 33, lo que asegura que al calcular el semiperiodo en el controlador como se muestra en la Figura 35, este sea divisible entero en la precisión de tiempo asignada.

```
constraint high_freq     { t_step inside {[32'd500:32'd625]}; }
constraint mid_high_freq { t_step inside {[32'd625 :32'd1000]}; }
constraint accuracy      { t_ps == t_step*32'd10;};
```

Figura 33 - Restricciones de aleatorización de periodo del reloj generado

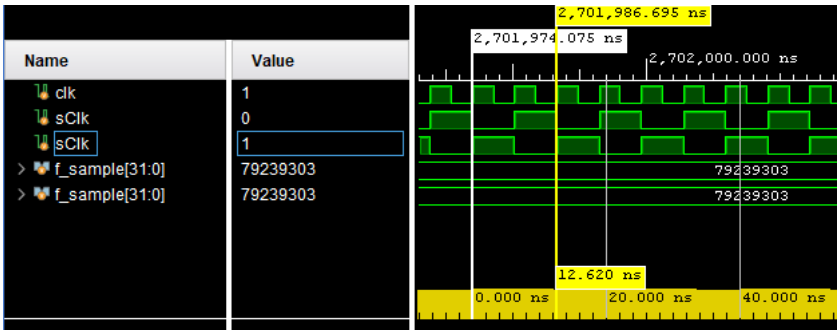


Figura 34 - Discrepancia entre frecuencia generada e indicada

Como se muestra en la Figura 34, la generación de reloj aún conserva un error entre la frecuencia deseada, en este caso de 79239303Hz y la que se genera realmente calculada del periodo del reloj, que en este caso son 79239302.69Hz, pero es únicamente en su parte decimal, por lo que es un error esperado al tratar con números enteros.

Secuencia configurada

La clase `wave_gen_clk_seq#(string cfg_name)` se extiende de la clase `uvm_sequence#(sequence_item)`, en ella se instancia el objeto ítem de secuencia, una interfaz virtual del puerto de entrada del DUT y un objeto de configuración.

Al inicio de la fase de ejecución utiliza la base de datos de UVM para conectar la interfaz y leer la configuración con la que fue parametrizada. También asigna los seleccionadores de la configuración relevantes para el modo de selectores compartidos y totalmente aleatorizado.

Al igual que la secuencia del agente de puerto de entrada tiene un bloque en el que asigna el valor fijo del periodo según la frecuencia escogida, un bloque en el que cambia el selector usado en el modo de prueba de barrido de comportamientos y la aleatorización final del valor.

Este código solo se puede ejecutar si la señal `sRun` está desactivada para que no cambie la frecuencia de muestreo durante la generación de una onda.

Una vez la transacción es enviada al controlador, se ejecuta el cálculo del semiperiodo del reloj y se actualiza el valor en el hilo de generación de reloj.

```
fork

////// Thread que genera el clock
   forever begin
      #(half_period) vif.clk = ~vif.clk;
// cabe destacar que este es un reloj que va al doble de la frecuencia
de lo que va el sClk del resto del sistema
   end

///// Thread que actualiza frecuencia cuando llega un nuevo seq_item
   forever begin
      seq_item_port.get_next_item(req);
      if (req == null) begin
      `uvm_error("CLK_DRV", "Received null req from sequencer");
      continue;
      end
      half_period = (req.t_ps / 1000.0) / 2.0;
      // 1/1000 para ns y /2 para semiciclo
      if(half_period<=1.25) begin
// Condicional para impedir que el semiperiodo sea excesivamente corto
// Con valores aleatorizados no importa porque está de por sí limitado
en los rangos, pero para valores fijos establezco el límite en 200MHz
```

```
          `uvm_info("CLK_DRV", $sformatf(
           "Too high frequency (%d) , maximum allowed is 200MHz,
           autoset to 50MHz",
           1e9/(half_period*4) ), UVM_MEDIUM)
           half_period = 5;
      end

      calc_freq = 1e9/(half_period*4);
// *2 para ciclo completo *2 para reloj del sistema
      vif.freq_hz = calc_freq;
      if(calc_freq != prev_freq) begin
          `uvm_info("CLK_DRV", $sformatf("Set freq=%0d Hz,
           rand  t_ps= %d, half_period = %0.4f ",
           calc_freq, req.t_ps, half_period  ), UVM_MEDIUM)
// No enteramente para debugging pero casi -- el paso mínimo posible
para frec aleatorizada es 10ps
      end
      seq_item_port.item_done();
      prev_freq=req.freq_hz;
   end
join_none
```

Figura 35 - Generación del reloj y su recalculación

Para la generación del reloj se utiliza la expresión de tiempo de SystemVerilog `#(time)` que se interpreta por defecto como nanosegundos, por lo que el valor aleatorizado tiene que convertirse a esas unidades. Se hace también manejo de dos posibles casos de error, como el sistema no puede reaccionar si no recibe una señal de reloj, se asegura de dar una advertencia del error, pero generando un reloj fijado a 50Mhz, también toma esta frecuencia si el periodo es demasiado corto, es decir, la frecuencia del reloj demasiado alta.

Se utiliza también el sistema de mensajería de UVM para informar por terminal de la frecuencia y los valores calculados, aviso que fue necesario para detectar el origen del error de incongruencia entre la frecuencia indicada y la generada realmente.

1.6.5 Llamada a test configurado

El test también está definido en dos clases, una base que hace el manejo genérico de las fases de construcción y ejecución, y la configurada, que se extiende de la base para configurar el test como tal.

Test base

La clase base, `wave_gen_basic_test` se extiende de la clase `uvm_test`, en ella se instancia el entorno y las secuencias a utilizar, que en este caso son las del puerto de entrada y el reloj.

En fase de construcción crea en factory de UVM el entorno y ambas secuencias. Cabe destacar que ambas secuencias se crean parametrizadas al objeto configuración que no está en esta clase si no en la heredada.

```
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    env = wave_gen_environment::type_id::create("env", this);

    clk_seq = wave_gen_clk_seq#("cfg_t")::type_id::create("clk_seq");

    if (!uvm_config_db#(uvm_sequence_base)::get(this, "", "seq_in",
    seq_in)) begin
// Se configura como uvm_sequence_base para flexibilixar
el uso del basic test
      `uvm_fatal("NO_SEQ_IN", "No sequence configured via
       uvm_config_db")
    end

  endfunction : build_phase
```

Figura 36 - Fase de construcción del test

Otra cosa para remarcar es que para la secuencia reloj se usa el método `create` que la registra en factory, porque se aprovecha de que el objeto `cfg_t` ya está creado en factory.

En un inicio, antes de incluir la secuencia del reloj, la configuración se limitaba a la secuencia del puerto de entrada, por lo que no había razón de registrarla en factory, es por esto por lo que la secuencia de entrada `seq_in` utiliza el método `get`. Esta forma de configurar la secuencia permite tratar como secuencia genérica al objeto `seq_in`, mientras su configuración y registro en la base de datos se hace en otra clase. Ambos acercamientos son válidos para la metodología y depende de la implementación que se quiera realizar, aunque es recomendado dar uso a la factory ya que facilita la parametrización de las pruebas.

En la fase de ejecución se inician ambas secuencias en sus respectivos secuenciadores, que tienen que ser iniciados de manera concurrente utilizando una sentencia `fork-join`.

```
task run_phase(uvm_phase phase);
   phase.raise_objection(this);
   fork
      clk_seq.start(env.wave_gen_clk_agnt.sequencer);
      seq_in.start(env.wave_gen_in_agnt.sequencer);
   join
   phase.drop_objection(this);
endtask : run_phase
```

Figura 37 - Fase de ejecución del test

Como se explicó en la sección de fases de UVM, para acabar una fase, se tiene que bajar todas las objeciones levantadas, siendo esta la única que tiene que ser explícitamente

definida por el usuario, ya que para todo el resto de las fases y subfases se encarga la propia librería. En este caso, la secuencia `seq_in` sí tiene un fin definido con un número de transacciones máximo, pero la secuencia `clk_seq` tiene un bucle infinito necesario para la generación del reloj del que tiene que ser forzado a salir. La solución por la que se optó fue utilizar la base de datos de UVM para levantar una bandera al momento de acabar la secuencia `seq_in`, que ejecutase el paro de la secuencia `clk_seq`, lo que, como se explicará más adelante, permitió extender el enfoque del test. Una solución más simple válida para este caso concreto podría haber sido sustituir la sentencia `fork-join` por un `fork-join_any`, haciendo que en lugar de requerir que todas las secuencias acabasen para salir del `fork`, bastase con que acabase una.

Test configurado

La clase `wave_gen_test_customRandTest` se extiende de `wave_gen_basic_test`, esta se encarga de toda la configuración del test, planteado de esta manera para aislar todo lo que se puede modificar de cara a una prueba concreta en una sola clase.

En ella se instancian el objeto de configuración y la secuencia parametrizada de entrada, para ello se le tienen que dar valores a cada uno de los registros de configuración, luego registrar en la base de datos de UVM la configuración, instanciar y crear la secuencia. Todo ello en la fase de construcción y es necesario que se ejecute antes que lo que se ejecuta en la clase base, por lo que la función `super.build_phase(phase)` tiene que ir al final de la definición de la función de fase de construcción de esta clase.

En esta clase también se asigna el nivel de verbosidad a nivel de test, por lo que afecta al entorno completo. Para mostrar información útil en depuración el nivel tiene que ser `UVM_HIGH` y para mostrar únicamente la información previa y posterior al test, el nivel tiene que ser `UVM_LOW`.

1.6.6 Parámetros de cobertura

La cobertura se realiza para comprobar en qué medida se han probado los posibles valores de determinados registros o interacciones entre ellos, para ello es necesario conocer los parámetros de funcionamiento del sistema para excluir de la cobertura valores en los registros que ocasionen errores. De forma resumida, los rangos considerados para cada variable aleatorizada son:

```
cfg.sel.f_sample = 7;
//0: [100MHz:80MHz] 1: [80MHz:50MHz] 2: [25MHz:80MHz] 3: [1MHz:25MHz]

cfg.sel.len_PRI = 7;
//0:[1000:2000] 1:[2000:12000] 2:[12000:32000]

cfg.sel.bandwidth = 7;
//0: [25MHz:50MHz] 1: [10MHz:25MHz] 2: [1MHz:10MHz]
cfg.sel.f_signal = 7;
//0: [25MHz:50MHz] 1: [10MHz:25MHz] 2: [1MHz:10MHz]
cfg.sel.t_ramp = 7;
//0: [3:16] 1: [16:500] 2: [500:1000]

cfg.sel.gain_percent = 7;
//0: [10:50] 1: [50:80] 2: [80:120]

cfg.sel.reset_delay = 7;
//0: [32768:65536] 1: [65536:262144] 2: [262144:1048576]
3: [1048576:4194304]
cfg.sel.sRun_delay = 7;
//0: [1024:2047] 1: [2048:8191] 2: [8192:32767] 3: [32768:262114]
cfg.sel.vita_delay = 7;
//0: [30:38] 1: [34:63] 2: [64:127] 3: [128:255]
cfg.sel.sRun_drop = 7;
//0: [4:12] 1: [8:40] 2: [40:60] 3: [60:80]
```

Figura 38 - Rangos de aleatorización junto a los selectores en el objeto configuración

De los cuales se mantuvieron los 0 de `sRun_drop` y `vita_delay` para forzar uno de los errores encontrados, pero no están orientados a usarse en una prueba larga dentro de los parámetros de diseño del DUT.

Grupos de cobertura

Una vez explicada la secuencia, se hace clara la separación de los dos grupos de cobertura antes mencionados.

El grupo de cobertura `sync_sRun_cg` se muestrea cuando se acciona la señal `sRun`, en este grupo se almacenan los valores de configuración de la onda generada en el momento, junto a los retardos de la señal de control, que, si bien no llegan en la transacción, son contados en la función que controla el muestreo de ambos grupos de cobertura como se muestra en la Figura 24.

El grupo de cobertura `sync_sClk_cg` se muestrea cada flanco positivo de la señal de reloj del DUT. Si bien en la Figura 24 no se muestra ningún control de reloj, en la Figura 18, correspondiente a la fase de ejecución del marcador, solo se hace la llamada a la función de muestreo en flancos positivos del reloj. En este grupo de cobertura se almacenan cuantos resets y reconfiguraciones de la señal se realizaron en la simulación completa, en qué parte

del ciclo del pulso de generación de onda se produjeron las interrupciones, así como también cuantos ciclos de los pulsos de generación de onda se terminaron.

Los rangos de los bins se asignaron para crear suficientes como para verificar rangos relativamente pequeños sin aumentar el número de bins excesivamente. Por ejemplo:

```
wave_gen_f_sample: coverpoint cov_trans.f_sample{
   bins low_freq [15]= {[32'd10_000_000:32'd25_000_000]};
   bins mid_range_freq [55]= {[32'd25_000_000:32'd80_000_000]};
   bins high_freq[20] = {[32'd80_000_000:32'd100_000_000]};
}
```

Figura 39 - Ejemplo de estructura de los puntos de cobertura.

Se consideró que el rango del selector 3 de la frecuencia de `muestreo f_sample (10MHz – 25MHz)` era demasiado bajo para el que será su uso real, por lo que tendría que ser excluido de la cobertura y de la ejecución del test. Además, como se puede ver, los bins están definidos para subdividir cada rango en rangos de 1MHz.

1.6.7 Modulabilidad de los componentes.

Es necesario recordar que todos los anteriores componentes descritos son clases, se tienen que incluir explícitamente en el código para que puedan ser instanciados. Esto se hace a través de los paquetes de SystemVerilog, cada módulo del sistema UVM que se espera que exista al mismo nivel tiene que ser incluido explícitamente en su paquete correspondiente.

El nivel más exterior de estos paquetes es la lista de tests:

```
package wave_gen_test_list;
 import uvm_pkg::*;
 `include "uvm_macros.svh"
 import wave_gen_env_pkg::*;
 import wave_gen_in_seq_list::*;
 //import wave_gen_out_seq_list::*;
 import wave_gen_clk_seq_list::*;
 `include "wave_gen_config_test.svh"
 `include "wave_gen_basic_test.svh"
 //`include "wave_gen_test4testing.svh"
 //`include "wave_gen_test_lenTest.svh"
 //`include "wave_gen_test_freqSweepTest.svh"
 //`include "wave_gen_test_guidedRandomizedTest.svh"
 `include "wave_gen_test_customRandTest.svh"
 `include "wave_gen_test_compoundTest.svh"
Endpackage
```

Figura 40 - Paquete de lista de tests

Comentados se encuentran las versiones de test que quedaron obsoletas y una secuencia aplicada en el puerto de salida que se optó por no incluir al test final.

Dentro del paquete de lista de test de encuentra el paquete de entorno `wave_gen_env_pkg`:

```
package wave_gen_env_pkg;

    import uvm_pkg::*;
    `include "uvm_macros.svh"

    import wave_gen_in_agent_pkg::*;
    import wave_gen_out_agent_pkg::*;
    import wave_gen_clk_agent_pkg::*;

   `include "wave_gen_coverage.svh"
   `include "wave_gen_ref_model_transaction.svh"
   `include "wave_gen_scoreboard.svh"
   `include "wave_gen_env.svh"
   `include "wave_gen_config_test.svh"
Endpackage
```

Figura 41 - Paquete del entorno

Y dentro de él están importados los paquetes de los agentes, como también incluidos los componentes que existen a nivel de entorno.

Con la estructura de paquetes del entorno de simulación se establece la estructura UVM con la que se presentó el test en la Figura 6.

1.6.8 Simulación Vivado

Antes de arrancar la simulación es necesario configurar la simulación en vivado. La librería UVM está incluida por defecto en su versión 1.2, en caso de querer usar una diferente tendría que especificarse. Lo que sí es necesario configurar es el manejo de cobertura, los campos modificados se muestran en la Figura 42. Es necesario asignar un nombre para el archivo de cobertura y el directorio en el que almacena la ejecución, además se puede especificar qué tipo de cobertura de código se desea realizar. El tipo sbct seleccionado realiza cobertura completa que ofrece vivado, en concreto es de línea, rama, condición y conmutación. Que si bien no son parte del foco del proyecto fueron de ayuda al depurar el sistema de verificación.

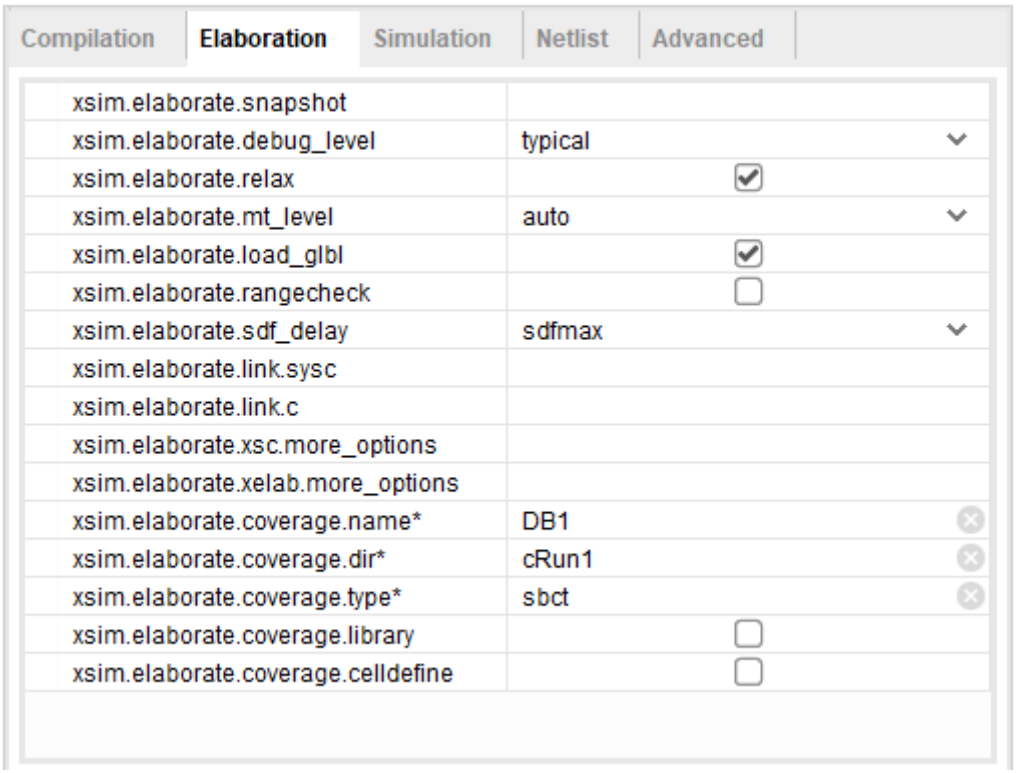


Figura 42 - Configuración de elaboración de la simulación en Vivado

La simulación en vivado está estructurada desde el archivo top con el que está conectado al módulo real y al modelo de referencia a través de sus respectivas interfaces que son las que se tiene acceso de manera legible una vez iniciada la simulación.

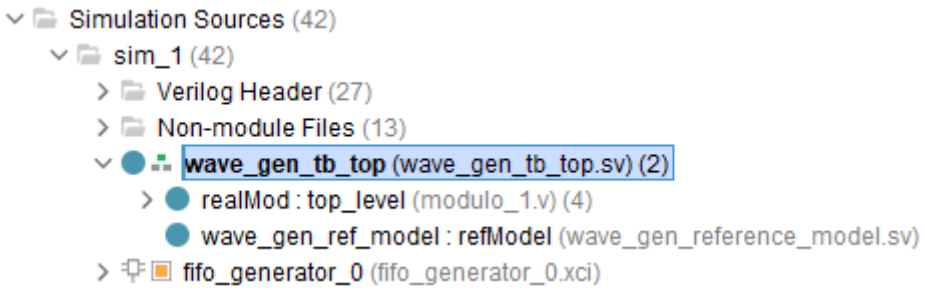


Figura 43 - Jerarquía de fuentes de simulación en Vivado

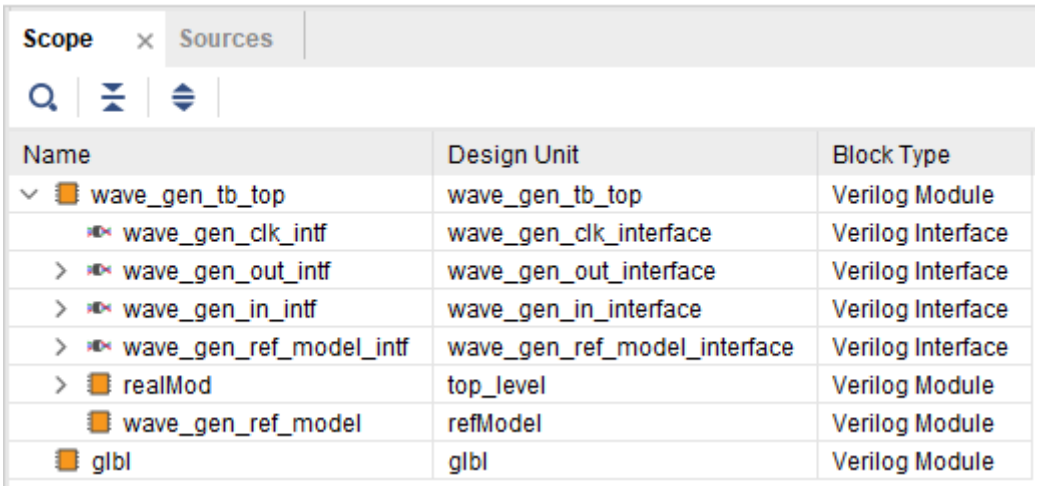


Figura 44 - Interfaces a las que se tiene acceso en la simulación de Vivado

La simulación de vivado permite hacer un seguimiento visual los puertos del DUT y el modelo de referencia. La manera en la que se organizó fue ordenar las señales de forma que queden la señal del DUT seguida por la misma del modelo de referencia.

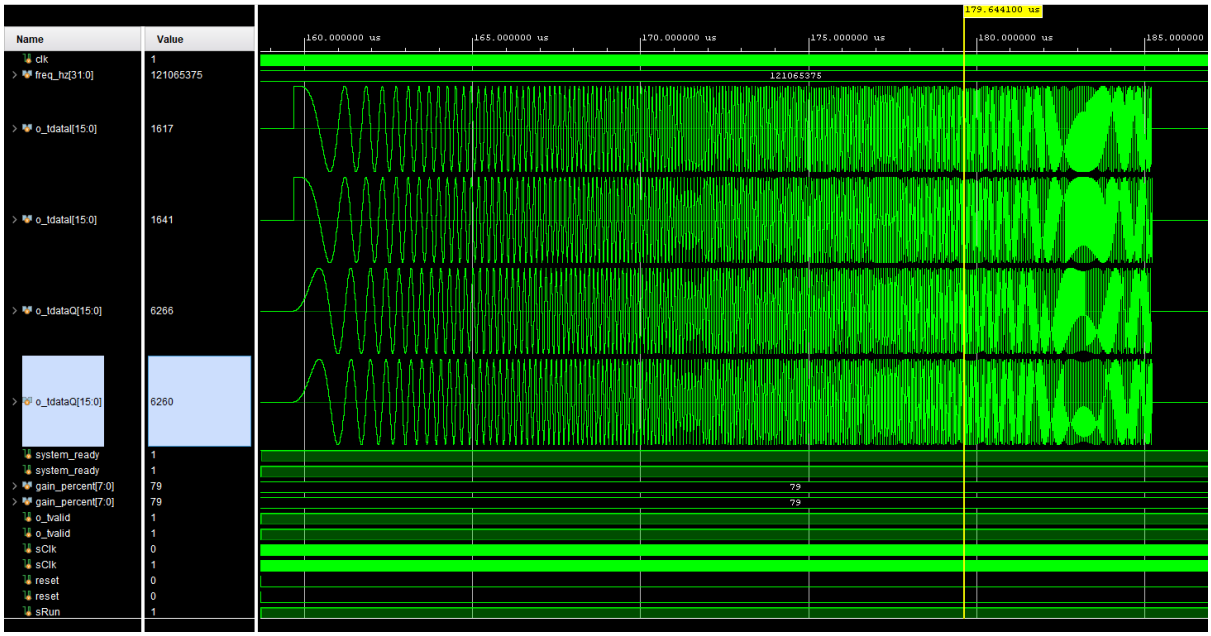


Figura 45 - Registros del DUT y modelo real

1.6.9 Tras la simulación

Una vez finalizada una simulación, se muestran por terminal todas las métricas consideradas en el marcador y se crea un csv con los mismos datos. El sistema UVM hace un resumen de todo lo que fue impreso usando su sistema de mensajería al final para facilitar la búsqueda de mensajes. En concreto, usando las etiquetas creadas por el usuario. El siguiente es un

ejemplo de ejecución con verbosidad baja en el que los únicos mensajes no definidos por el sistema corresponden a las etiquetas [PRE-SEQUENECE] y [SCOREBOARD].

```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  100
UVM_WARNING :    8
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[PRE-SEQUENCE]     3
[RNTST]     1
[SCOREBOARD]    93
[TEST_DONE]     1
[TPRGED]     8
[UVM/COMP/NAMECHECK]     1
[UVM/RELNOTES]     1
```

Figura 46 - Resumen de mensajes generados por entorno UVM tras la simulación

Para que se genere un informe de cobertura es necesario ejecutar por terminal los siguientes comandos.

```
write_xsim_coverage -cov_db_dir cRun1 -cov_db_name DB1
```

```
export_xsim_coverage -cov_db_name DB1 -cov_db_dir cRun1 -output_dir cReport1 -open_html true
```

Con los que escribe el archivo de base de datos de cobertura con los datos de la simulación actual y genera un informe en html de ambas coberturas. Los datos de cobertura se corresponden a una prueba corta, por lo que tanto la cobertura de código como la funcional resulta muy baja.

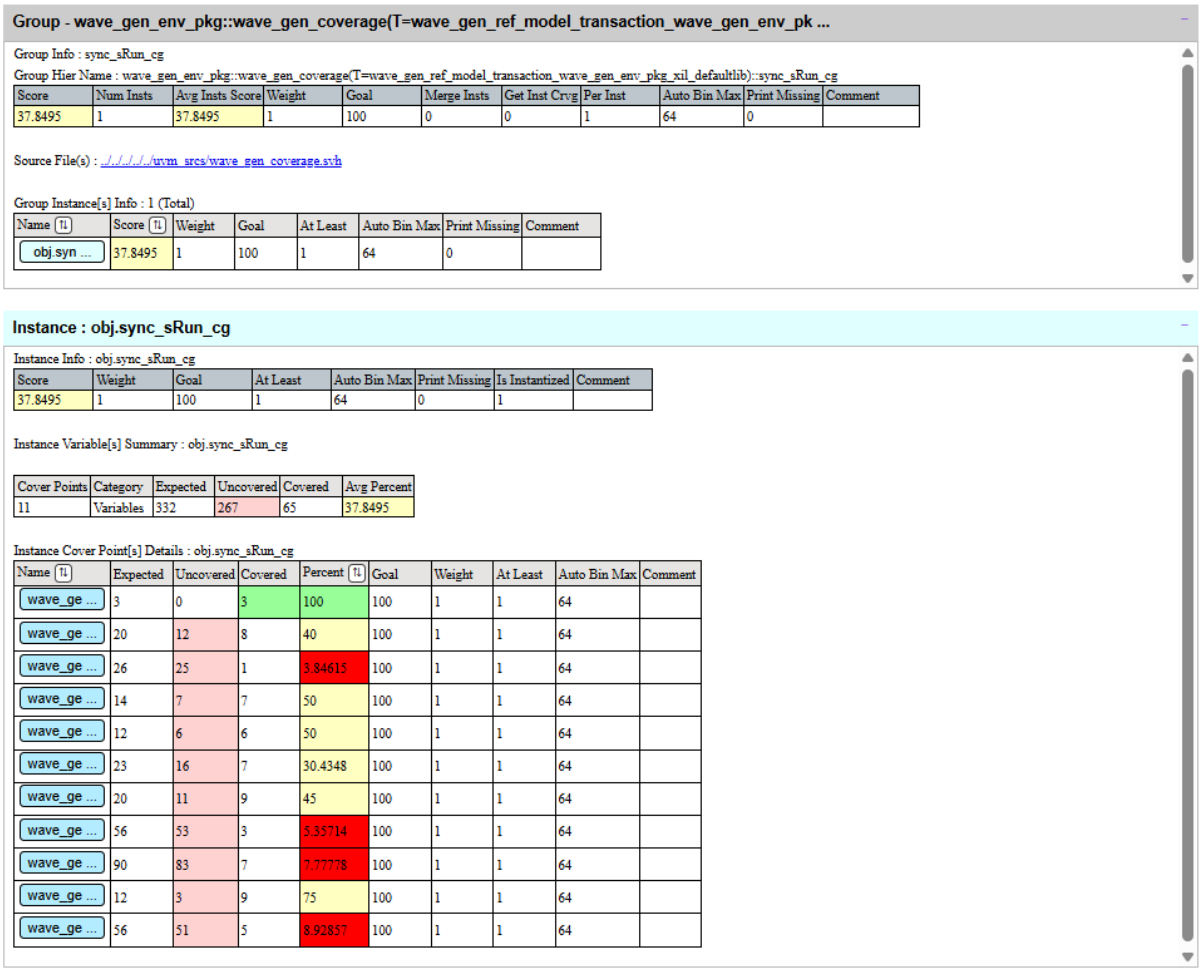
```
write_xsim_coverage -cov_db_dir cRun1 -cov_db_name DB1
INFO:  (TCL) Writing Functional Coverage Database at path : cRun1\xsim.covdb\DB1\xsim.covinfo
INFO-SAVE : (TCL) Code Coverage DB saved successfully at : cRun1\xsim.codeCov\DB1\xsim.CCInfo
export_xsim_coverage -cov_db_name DB1 -cov_db_dir cRun1 -output_dir cReport1 -open_html true

 Options Set are :

   Report Format              : html
   Log File                   : xcrg.log
   Functional Coverage Dir[s] : cRun1
   Functional Coverage DB[s]  : DB1
   Code Coverage Dir[s]       : cRun1
   Code Coverage DB[s]        : DB1


*************************************************************
                  Code Coverage Report
*************************************************************

 Restoring Code Coverage DB[s] from :
  1 cRun1/xsim.codeCov/DB1/xsim.CCInfo

 Created Coverage Report Directory     : cReport1/codeCoverageReport

 Code Coverage HTML Report directory              : cReport1/codeCoverageReport

 Line Coverage Score 69.4142
 Branch Coverage Score 22.1754
 Condition Coverage Score 27.9807
 Toggle Coverage Score 3.44
```

Figura 47 - Generación y resumen de cobertura de código

```
 ***********************************************************
                   Functional Coverage Report
 ***********************************************************

  Restoring Functional Coverage DB[s] from :
   1 cRun1/xsim.covdb/DB1/xsim.covinfo

  Line Coverage Score 69.4142
  Branch Coverage Score 22.1754
  Condition Coverage Score 27.9807
  Toggle Coverage Score 3.44

  Created Coverage Report Directory    : cReport1/functionalCoverageReport

  Functional Coverage HTML Report directory        : cReport1/functionalCoverageReport

  Creating dashboard.html

  Creating groups.html

  Creating grp0.html

  Creating grp1.html
```

Figura 48 - Generación de cobertura funcional

El informe completo se puede visualizar con un navegador al estar generado como html.



Group - wave_gen_env_pkg::wave_gen_coverage(T=wave_gen_ref_model_transaction_wave_gen_env_pk ...

Group Info : sync_sRun_cg

Group Hier Name : wave_gen_env_pkg::wave_gen_coverage(T=wave_gen_ref_model_transaction_wave_gen_env_pkg_xil_defaultlib)::sync_sRun_cg

Score	Num Insts	Avg Insts Score	Weight	Goal	Merge Insts	Get Inst Cvg	Per Inst	Auto Bin Max	Print Missing	Comment
37.8495	1	37.8495	1	100	0	0	1	64	0	

Source File(s) : ../../../../../uvm_srcs/wave_gen_coverage.svh

Group Instance[s] Info : 1 (Total)

Name	Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Comment
obj.syn ...	37.8495	1	100	1	64	0	

Instance : obj.sync_sRun_cg

Instance Info : obj.sync_sRun_cg

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
37.8495	1	100	1	64	0	1	

Instance Variable[s] Summary : obj.sync_sRun_cg

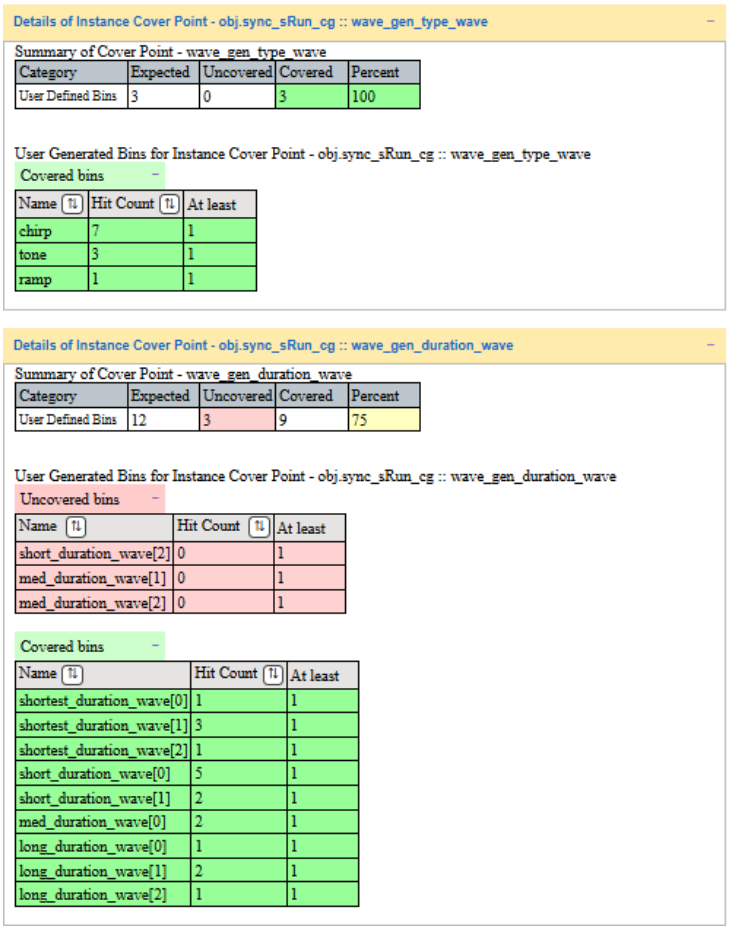
Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
11	Variables	332	267	65	37.8495

Instance Cover Point[s] Details : obj.sync_sRun_cg

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
wave_ge ...	3	0	3	100	100	1	1	64	
wave_ge ...	20	12	8	40	100	1	1	64	
wave_ge ...	26	25	1	3.84615	100	1	1	64	
wave_ge ...	14	7	7	50	100	1	1	64	
wave_ge ...	12	6	6	50	100	1	1	64	
wave_ge ...	23	16	7	30.4348	100	1	1	64	
wave_ge ...	20	11	9	45	100	1	1	64	
wave_ge ...	56	53	3	5.35714	100	1	1	64	
wave_ge ...	90	83	7	7.77778	100	1	1	64	
wave_ge ...	12	3	9	75	100	1	1	64	
wave_ge ...	56	51	5	8.92857	100	1	1	64	

Figura 49 - Informe de un grupo de cobertura

También se puede ver en mayor detalle la cobertura tanto a nivel de grupo de cobertura como se ve en la Figura 49 o a nivel de punto de cobertura con sus respectivos bins como se ve en la Figura 50.



Details of Instance Cover Point - obj.sync_sRun_cg :: wave_gen_type_wave

Summary of Cover Point - wave_gen_type_wave

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	3	0	3	100

User Generated Bins for Instance Cover Point - obj.sync_sRun_cg :: wave_gen_type_wave

Covered bins

Name	Hit Count	At least
chirp	7	1
tone	3	1
ramp	1	1

Details of Instance Cover Point - obj.sync_sRun_cg :: wave_gen_duration_wave

Summary of Cover Point - wave_gen_duration_wave

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	12	3	9	75

User Generated Bins for Instance Cover Point - obj.sync_sRun_cg :: wave_gen_duration_wave

Uncovered bins

Name	Hit Count	At least
short_duration_wave[2]	0	1
med_duration_wave[1]	0	1
med_duration_wave[2]	0	1

Covered bins

Name	Hit Count	At least
shortest_duration_wave[0]	1	1
shortest_duration_wave[1]	3	1
shortest_duration_wave[2]	1	1
short_duration_wave[0]	5	1
short_duration_wave[1]	2	1
med_duration_wave[0]	2	1
long_duration_wave[0]	1	1
long_duration_wave[1]	2	1
long_duration_wave[2]	1	1

Figura 50 - Informe de puntos de cobertura con bins individuales

Como se podría esperar de un test corto ejecutado para la generación del informe de cobertura mostrado en las figuras previas, no se cumple con un porcentaje de cobertura lo suficientemente alto como para considerar que el diseño ha sido verificado completamente.

1.6.10 Test compuesto

Para implementar un test lo suficientemente largo como para lograr una cobertura suficiente, se planteó múltiples secuencias configuradas para asegurar que se comprueban casos en todo el rango de posibilidades. La manera en la que se planeó esto fue un barrido de comportamientos en todo el rango de frecuencia por cada una de las ondas y finalmente una secuencia totalmente aleatoria para dar con casos en los que las frecuencias y periodos pudiesen dar conflicto.

La manera con la que se implementó fue una clase llamada `wave_gen_test_compoundTest` extendida de `uvm_test` en la que se configurasen 4 comportamientos diferentes.

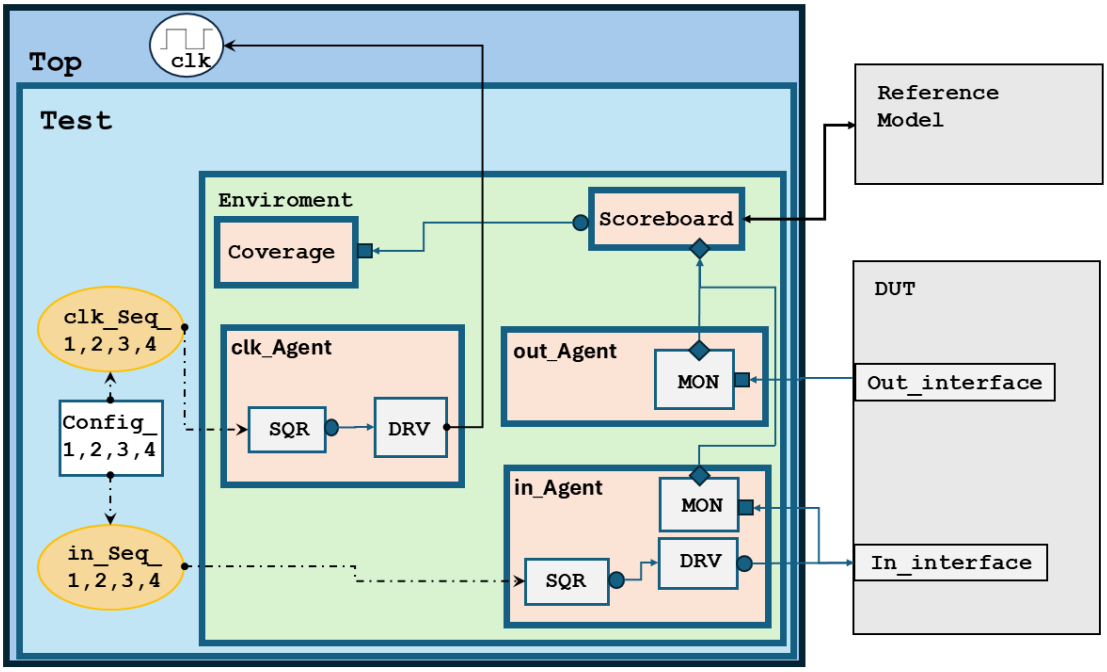


Figura 51 - Estructura del test largo

Para ello basta con definir cada uno de estos comportamientos, junto con las secuencias parametrizadas del puerto de entrada y del reloj. Es para este caso que se hace necesario incluir una bandera que indique cuando se ha acabado la secuencia de puerto de entrada, ya que, si no se detiene explícitamente la secuencia de reloj, este no puede ser reconfigurado y el sistema entraría en conflicto.

```
task run_phase(uvm_phase phase);
   phase.raise_objection(this);
   `uvm_info("CLK_DRV", "Inicia primera secuencia" , UVM_MEDIUM)
   fork
      clk_seq1.start(env.wave_gen_clk_agnt.sequencer);
      seq_in1.start(env.wave_gen_in_agnt.sequencer);
   join
   `uvm_info("CLK_DRV", "Inicia segunda secuencia" , UVM_MEDIUM)
   fork
      clk_seq2.start(env.wave_gen_clk_agnt.sequencer);
      seq_in2.start(env.wave_gen_in_agnt.sequencer);
   join
    `uvm_info("CLK_DRV", "Inicia tercera secuencia" , UVM_MEDIUM)
   fork
      clk_seq3.start(env.wave_gen_clk_agnt.sequencer);
      seq_in3.start(env.wave_gen_in_agnt.sequencer);
   join
   `uvm_info("CLK_DRV", "Inicia cuarta secuencia" , UVM_MEDIUM)
   fork
      clk_seq4.start(env.wave_gen_clk_agnt.sequencer);
      seq_in4.start(env.wave_gen_in_agnt.sequencer);
   join
   phase.drop_objection(this);
endtask : run_phase
```

Figura 52 - Fase de ejecución del test largo

Con esto se logra, en una sola simulación, abarcar más escenarios para la verificación.

1.7 RESULTADOS Y ANALISIS

1.7.1 Errores detectados durante el proceso de implementación

Durante el proceso de implementación se fueron realizando pruebas para comprobar que el sistema de verificación funcionase correctamente. Una vez estuvo definido el modelo de referencia se comenzaron a realizar diferentes pruebas antes de llegar a lo que fue la prueba configurable final. Estas pruebas si bien eran más orientadas dentro de diferentes parámetros de aleatoriedad, se lograron detectar diferentes errores que pudieron ser corregidos en el diseño del módulo a verificar, de entre los cuales, los más destacables son los siguientes:

Velocidad de respuesta del diseño

Si bien es evidente que un diseño hardware no puede responder a la misma velocidad que un código enteramente simulado, es necesario establecer en qué medida. Para ello se introdujeron los retrasos aleatorizados en la señal de control `sRun`. Utilizando rangos bajos en la aleatorización se puede forzar el error, que era más pronunciado en las ondas sinusoidales.

Fue para resincronizar el modelo de referencia con el módulo real que se introdujo la detección de desincronización. Cómo se explicó en el marcador, en caso de que el módulo real no pudiera generar onda cuando el modelo de referencia está por empezar, se deja de transmitir el reloj al modelo de referencia hasta que el módulo real pueda generar valores válidos.

En simulación, el error se presenta y detecta de la siguiente manera:

Configurando el test para alta verbosidad, se muestra por terminal los retardos que tiene cada onda y su tipo.

```
@ 134232000: uvm_test_top.env.wave_gen_in_agent.sequencer@@seq_in [IN_SEQ] sRun_drop=  7 vita_delay=   34 vita_counter=              13127
@ 134232000: uvm_test_top.env.wave_gen_in_agent.sequencer@@seq_in [IN_SEQ] type_wave=1
@ 150364000: uvm_test_top.env.wave_gen_in_agent.sequencer@@seq_in [IN_SEQ] sRun_drop=  7 vita_delay=   30 vita_counter=              15080
@ 150364000: uvm_test_top.env.wave_gen_in_agent.sequencer@@seq_in [IN_SEQ] type_wave=1
000: uvm_test_top.env.sb [SCOREBOARD] o_tvalid FAILED at vita=              15118
000: uvm_test_top.env.sb [SCOREBOARD] o_tvalid FAILED at vita=              15119
000: uvm test top.env.sb [SCOREBOARD] o tvalid FAILED at vita=              15120
```

Figura 53 - Desincronización entre DUT y modelo de referencia

En la Figura 53 se muestra como hay una onda tono que no hace saltar error con un `vita_delay` de 34 ciclos seguido de una onda tono que sí hace saltar el error con un `vita_delay` de 30 ciclos.

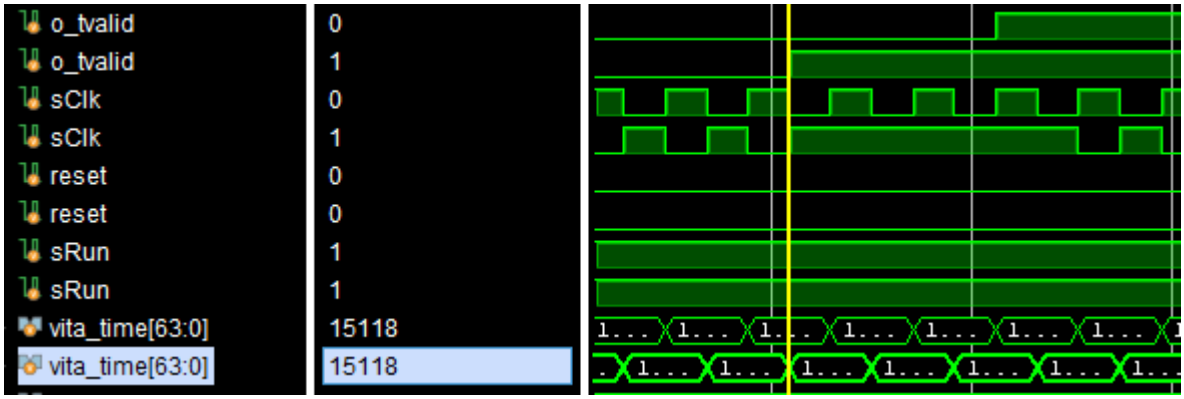


Figura 54 - No transmisión del reloj al modelo de referencia para resincronizar el comportamiento

En la Figura 54 se puede ver como el sistema deja de transmitir el reloj al modelo de referencia durante 3 ciclos para resincronizarlo con el DUT.

Se realizaron numerosas pruebas de las que se extrajeron 2 conclusiones.

Aunque se había asumido que el tiempo en el que la señal `sRun` se mantiene a 0, `sRun_drop`, tendría un efecto, resultó no ser el caso, el retardo importante resultó ser el tiempo que pasa entre que la señal `sRun` se acciona y el contador `vita_time` llega al valor `vita_trigger`, es decir, `vita_delay`.

El `vita_delay` mínimo que es necesario dejar para que el DUT pueda generar valores válidos es de 34 ciclos para tanto el chirp como el tono.

Error en rampa

El primer tipo de onda en el que se comprobaron errores en sus valores fue la rampa ya que sería la más sencilla de detectar tanto desfases como errores de cálculo. En lo que se esperaba que fuese una prueba con errores bajos aparecieron errores altos que parecían ser cíclicos, utilizando los mensajes de error al acabar la prueba se encontraron varias instancias de rampas en las que no se llegaba a hacer el número de pasos indicado antes de regresar

a 0. Tras un breve análisis se concluyó que el error sucedía si la rampa se generaba con un número de muestras que fuese potencia de 2 +1. Utilizando esta información, el diseñador del módulo pudo depurar el error, lo que dejó la generación de rampas con el error mínimo esperado al tratar con números enteros, con un RMSE de 0.59.

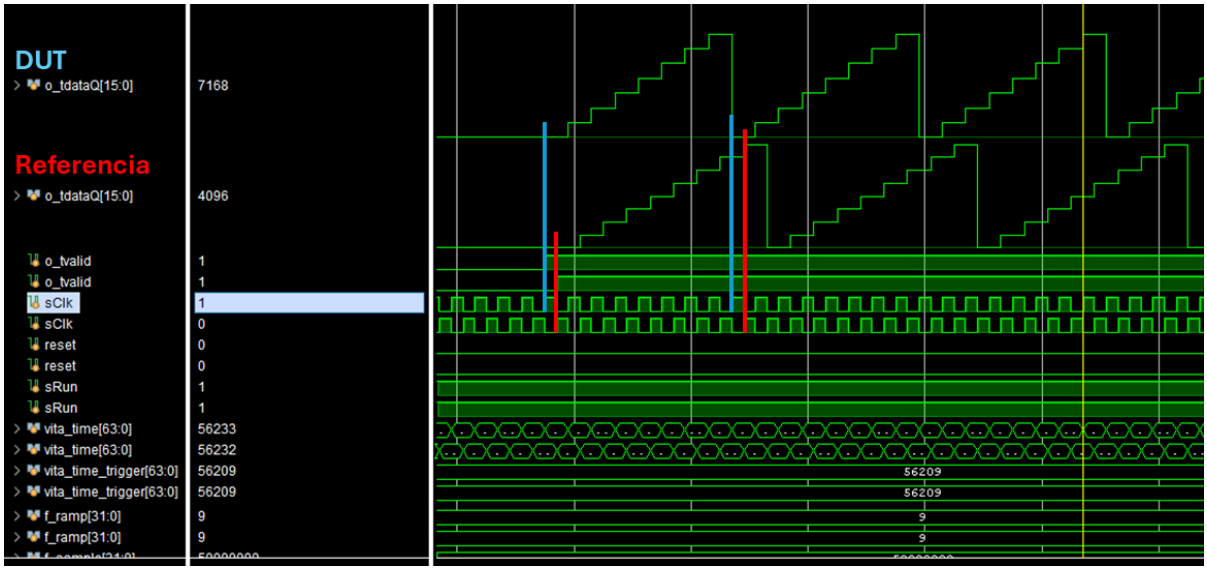


Figura 55 - Error de número de muestras en la generación de rampas

Error acumulativo en funciones trigonométricas

Desde un principio se asumió que, de haber errores significativos en la generación de onda, serían en las que utilizan funciones trigonométricas por la manera en la que se operan en hardware.

Para contextualizar, antes de realizar una corrección, el histograma de errores generados en un test en modo de barrido de comportamiento con los 3 tipos de onda tenía de forma consistente el siguiente aspecto.

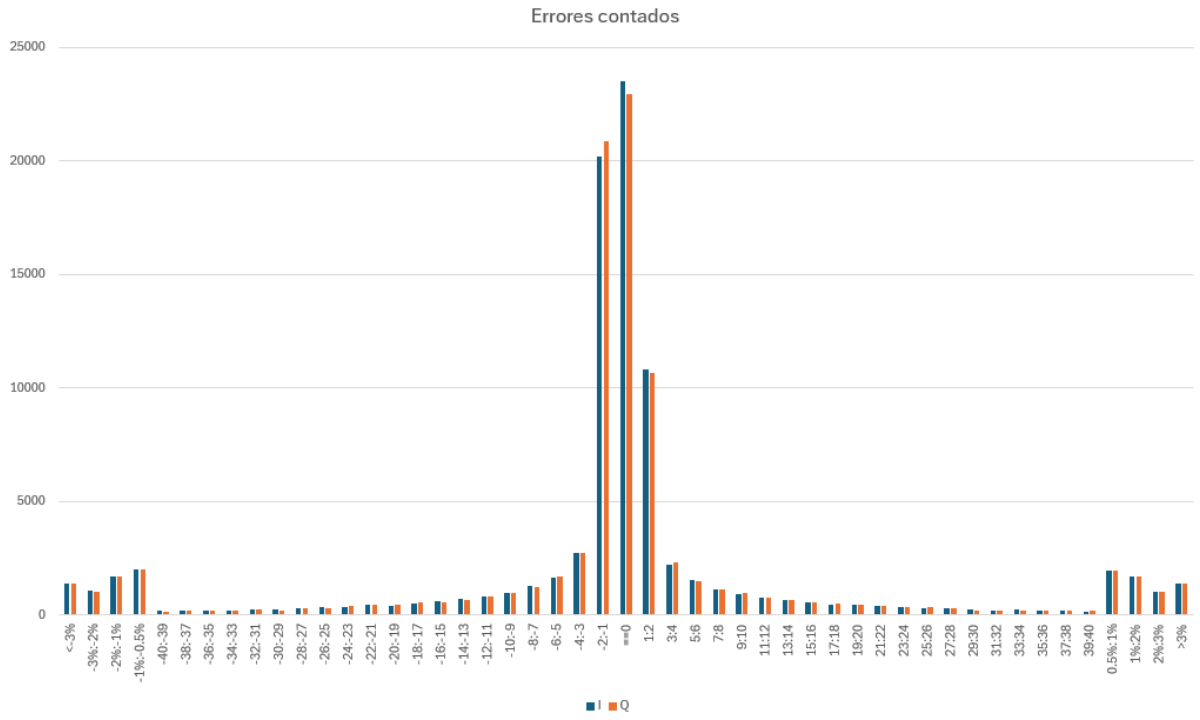


Figura 56 - Histograma de errores en una prueba con los 3 tipos de onda

En la gráfica se puede ver como para valores con bajo error había una inclinación a errores negativos, pero una vez el error aumentaba se acercaba a ser simétrico. Los valores de error más altos del 0.5% de 8192, se acumulaban en varios grupos, teniendo una cantidad relativamente alta en los grupos más exteriores en los que el error era mayor al 3%. Al agruparlos de esta manera se camuflan errores altos, es por esto por lo que también se incluye el pico de error en el informe en la fase de extracción. Utilizando ese dato se localizan errores en el orden de miles de unidades, disparando el valor del RMSE a varios cientos. Evidentemente había un error importante, por lo que se empezó a hacer pruebas más dirigidas.

```
UVM_INFO ../../../../../uvm_srcs/wave_gen_scoreboard.svh(445) @ 11452961000: uvm_test_top.env.sb [SCOREBOARD]
RMSE I: 422.255, RMSE Q: 422.203, valid samples: 266221
UVM_INFO ../../../../../uvm_srcs/wave_gen_scoreboard.svh(448) @ 11452961000: uvm_test_top.env.sb [SCOREBOARD]
Error [max I: 2928 phase: 1.644 rad time:              223426 ] | [min I: -2938 phase: 4.938 rad time:              223404 ]
Error [max Q: 2941 phase: 3.290 rad, time:              223415]  | [min Q: -2927 phase: 0.195 rad time:              223333]
```

Figura 57 - Errores altos en una prueba con los 3 tipos de onda

El análisis comenzó con la generación del tono, múltiples pruebas de barrido en la generación del tono dieron como resultado una concentración de errores de en los grupos de -2:-1 de sobre el 30% de las muestras, error nulo sobre el 50% y error de 1:2 sobre el 10%, repartiendo el 10% restante en todo el resto de los grupos con una inclinación a errores negativos. Pero no llegaba realmente a generar errores altos, por lo que el enfoque se cambió a la generación de chirps.

Tras un par de pruebas se hizo evidente la presencia de un error acumulativo ya que los picos de error se encontraban al final del pulso de los chirps más largos. Por lo que se procedió a hacer un análisis del error para aislar su origen utilizando Matlab.

Para ello, se utilizó la función opcional del marcador de escribir en un csv todos los valores de onda generados y se leyeron desde Matlab. Se calcularon los valores exactos de fase conocidos los parámetros de la onda y se compararon tanto con la fase calculada en el modelo de referencia como las fases correspondientes a los valores de ambas señales. Es conveniente recordar que por utilizar modulación I/Q, el cálculo de la fase en cada muestra es trivial:

$$\phi_k = atan\left(\frac{Q_k}{I_k}\right)$$

De esta forma se comprobó que el cálculo de fase en modelo de referencia tiene un RMSE de 2.91e-7 respecto al ideal calculado en Matlab y que la fase que se puede extraer de los valores de IQ del modelo de referencia tiene un RMSE de 2.92e-7 respecto de la fase con la que se calculan, con lo que se válida el cálculo del modelo de referencia. Es apropiado recordar que los valores RMSE comparten unidades con la variable analizada, que en este caso son radianes.

Después, se comprobó el cálculo de fase del DUT extraído de los valores IQ respecto al modelo de referencia, lo que dio un RMSE de 1.015e-3, comprobando que es varios ordenes de magnitud superior al anterior calculado. Graficando el error se puede ver que el error tiene una clara tendencia exponencial respecto a la duración del pulso:

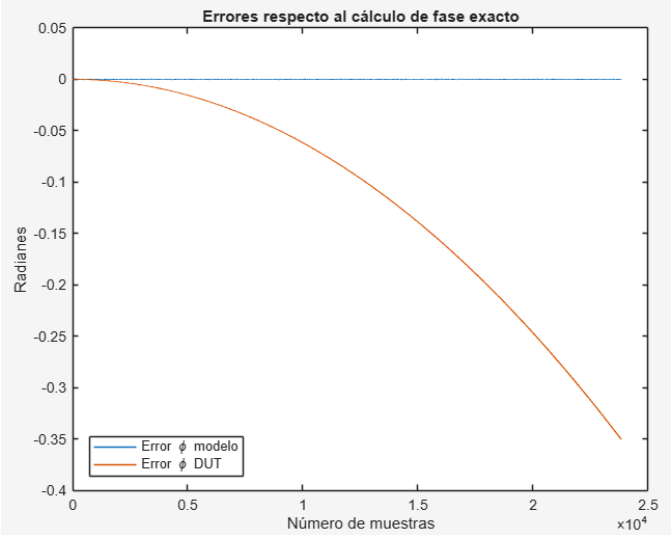


Figura 58 - Error acumulativo en el cálculo de la fase del chirp

Confirmando así que el error está causado en el cálculo de la fase.

Una vez fue depurado el error en el DUT, se realizaron las pruebas pertinentes para comprobar el funcionamiento de lo que pudo haber sido afectado por los cambios. Primero habiendo implementado el cambio tanto para el tono como para el chirp. Una prueba de solo chirps dio como resultado:

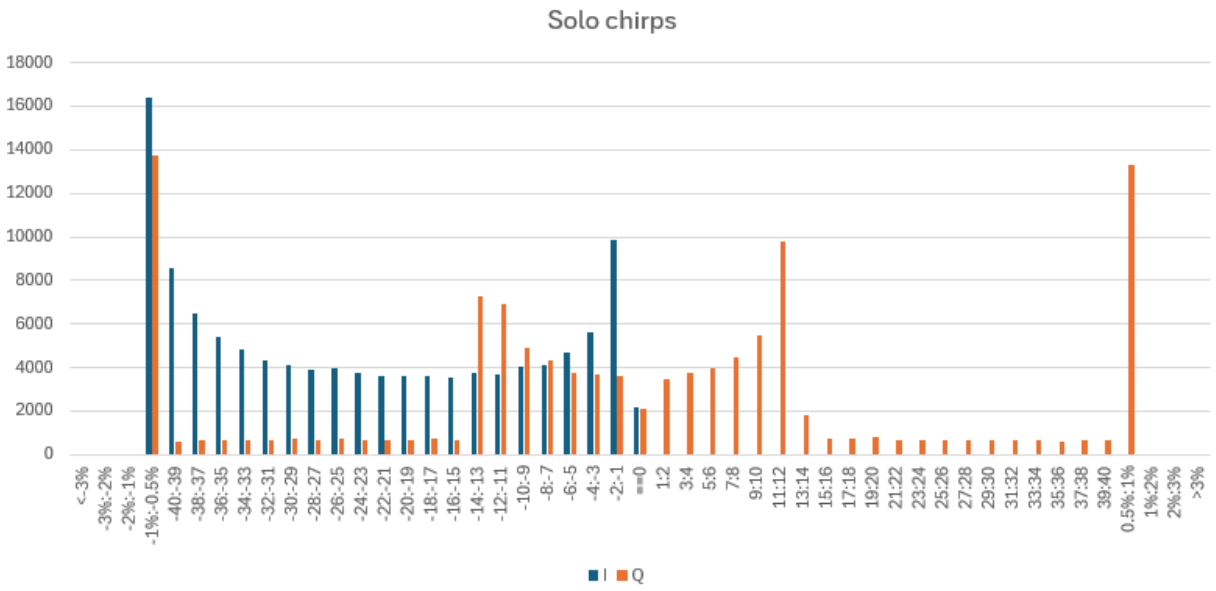


Figura 59 - Histograma de errores de solo chirps tras la corrección del error de fase

Si bien deja de estar el error centrado en el 0, se limitan los errores a inferiores al 1% por lo que se opta por esta solución de compromiso. Se puede comprobar que el RMSE disminuye del orden de cientos a un valor de 27.41 para la `I` y 31.61 para la `Q` en esta prueba especifica.

Se comprueba también el tono y se observa que, tras la corrección, su comportamiento se asemeja al mostrado por el chirp, por lo que se revierte parte del último cambio en el DUT, logrando así el siguiente comportamiento.

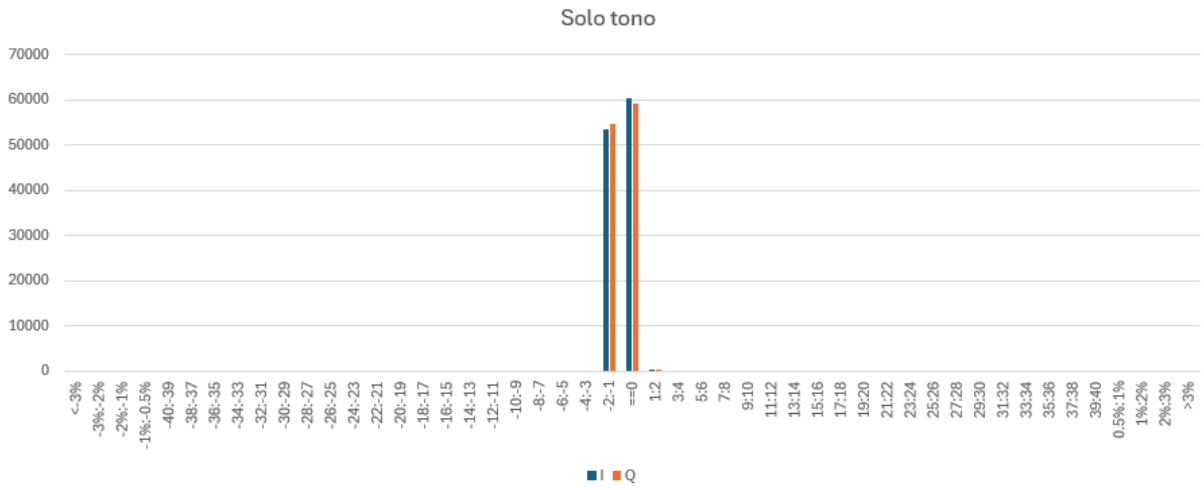


Figura 60 - Histograma de solo tonos final

Llegando en este caso a un RSME para `I` de 0.68 y para `Q` de 0.69.

1.7.2 Resultados de la prueba final

Marcador

La prueba final se planteó como una prueba lo suficientemente larga como para lograr un porcentaje de cobertura suficiente, para ello se hizo uso del test compuesto con las secuencias explicadas en su sección.

Para llevar a cabo la simulación fue necesario cambiar el equipo hardware ya que la simulación en vivado requiere muchos recursos computacionales que el portátil usado para toda la implementación no posee. Una vez exportado el proyecto al segundo equipo se comprobó un aumento de rendimiento en las simulaciones sustancial, una simulación de sobre 30 min en el portátil se redujo a poco más de 3 minutos.

Se configuró una prueba de 5 millones de transacciones y se dejó simular entera.

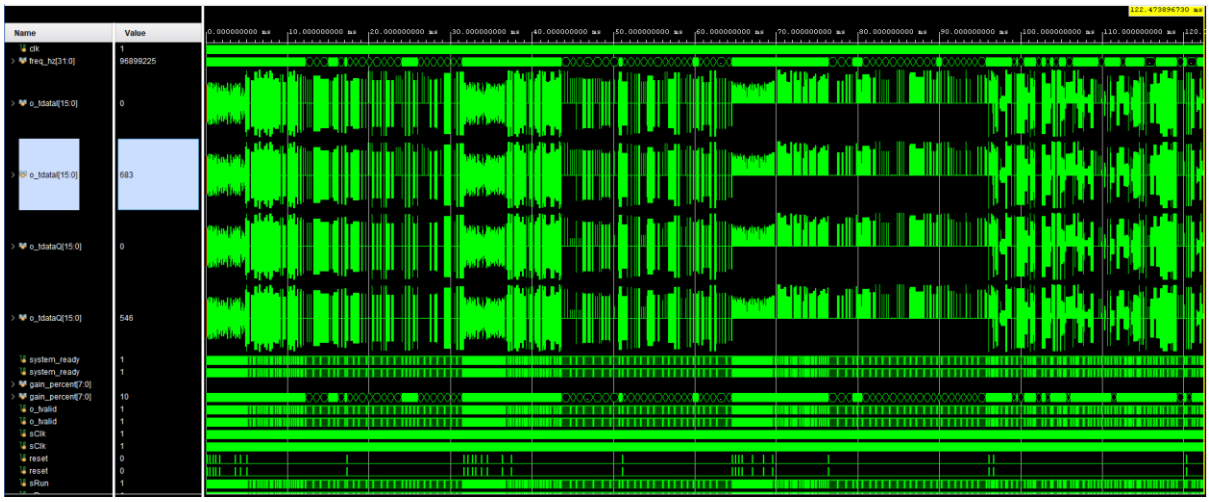


Figura 61 - Simulación de verificación completa

Del informe generado en el marcador podemos extraer que de las 5 millones de transacciones, 4778110 dieron lugar a muestras válidas, no se produjo ninguna desincronización entre el DUT y el modelo de referencia, los errores pico en `o_tdataI` son de -35 y +1, mientras que los de `o_tdataQ` son de ±59. Los RMSE son, para `I` de 9.01 y para `Q` de 10.57. Por último, se grafica el histograma de los errores generado en escala logarítmica para ver mejor la acumulación de valores. Se puede apreciar un comportamiento de error simétrico para la señal `Q` pero asimétrico para la señal `I`, cuya distribución está mucho más concentrada que en el punto de partida.

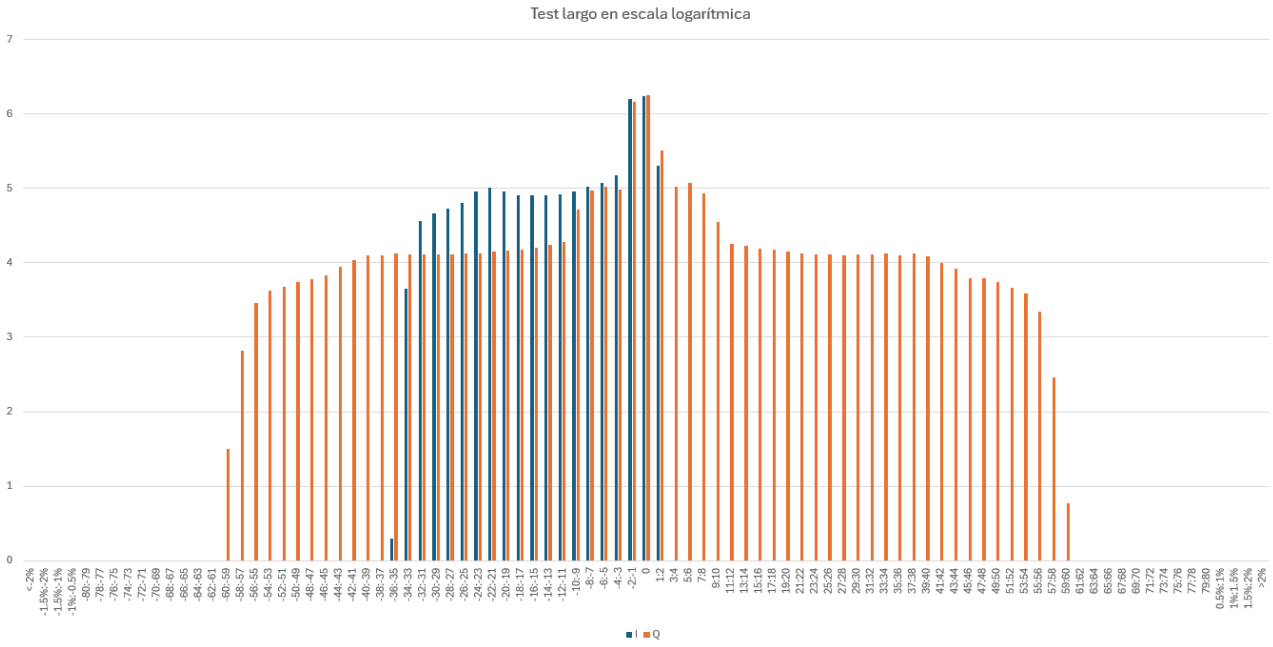


Figura 62 - Histograma de la prueba larga en escala logarítmica

Es destacable que, para este test, se aumentó el número de rangos en el acumulador para el histograma en el marcador y los grupos porcentuales más externos se consideraron respecto a todo el rango de 2^14 para que no se camuflasen acumulaciones en el exterior del histograma.

Cobertura

Tras generar el informe de cobertura nos encontramos con un porcentaje de cobertura relativamente alto para cada uno de los grupos.

Groups Coverage Summary

Score	Inst Score
91.1246	91.1246

Total groups in report: 2

Name	Score	Num Instances	Avg Instances Score
wave_gen_env_pkg::wave_gen_coverage(T=wave_gen_ref_model_transaction_wave_gen_env_pkg_xil_defaultlib)::sync_sClk_cg	88.3315	1	88.3315
wave_gen_env_pkg::wave_gen_coverage(T=wave_gen_ref_model_transaction_wave_gen_env_pkg_xil_defaultlib)::sync_sRun_cg	93.9177	1	93.9177

Figura 63 - Resumen del informe de cobertura funcional

sync_sClk_cg

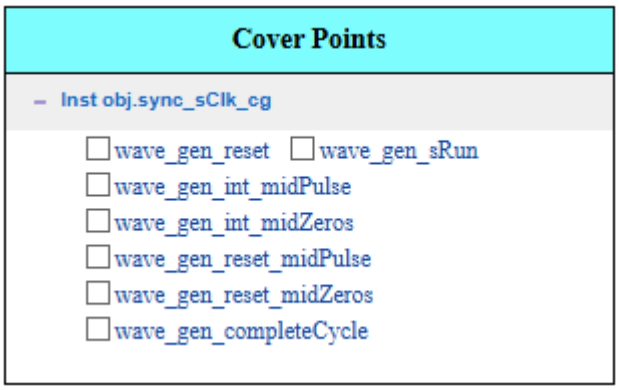
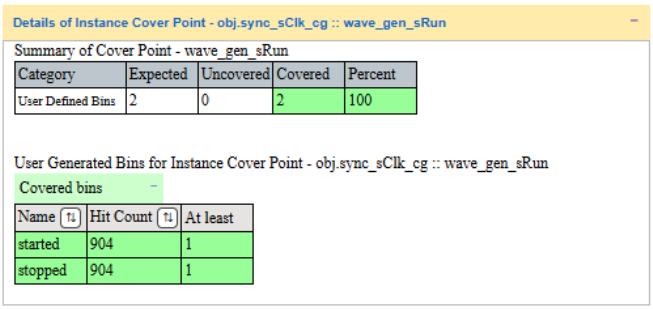


Figura 64 - Puntos de cobertura muestreados cada ciclo de reloj

En el grupo de cobertura muestreado cada flanco de reloj positivo encontramos que la generación de onda ha sido interrumpida múltiples veces por la señal de `sRun` y la señal de `reset` mientras se genera el pulso y mientras se generan ceros. Podemos también ver que se configuraron 904 ondas durante todo el test



Details of Instance Cover Point - obj.sync_sClk_cg :: wave_gen_sRun

Summary of Cover Point - wave_gen_sRun

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	2	0	2	100

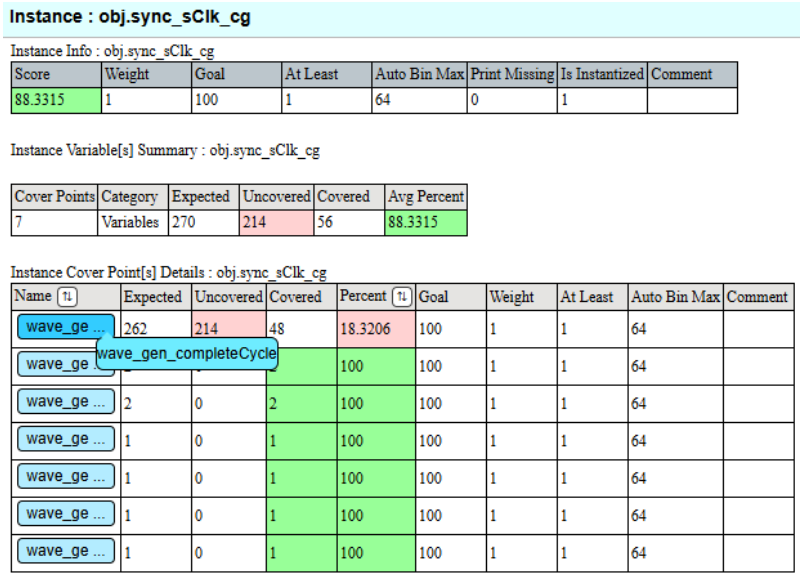
User Generated Bins for Instance Cover Point - obj.sync_sClk_cg :: wave_gen_sRun

Covered bins

Name	Hit Count	At least
started	904	1
stopped	904	1

Figura 65 - Punto de cobertura de la señal sRun

La razón por la que este grupo de cobertura no tiene 100% de cobertura se encuentra únicamente en el punto de cobertura que indica el número de ciclos enteros completados.



Instance : obj.sync_sClk_cg

Instance Info : obj.sync_sClk_cg

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
88.3315	1	100	1	64	0	1	

Instance Variable[s] Summary : obj.sync_sClk_cg

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
7	Variables	270	214	56	88.3315

Instance Cover Point[s] Details : obj.sync_sClk_cg

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
wave_ge ...	262	214	48	18.3206	100	1	1	64	
wave_ge ...				100	100	1	1	64	
wave_ge ...	2	0	2	100	100	1	1	64	
wave_ge ...	1	0	1	100	100	1	1	64	
wave_ge ...	1	0	1	100	100	1	1	64	
wave_ge ...	1	0	1	100	100	1	1	64	
wave_ge ...	1	0	1	100	100	1	1	64	

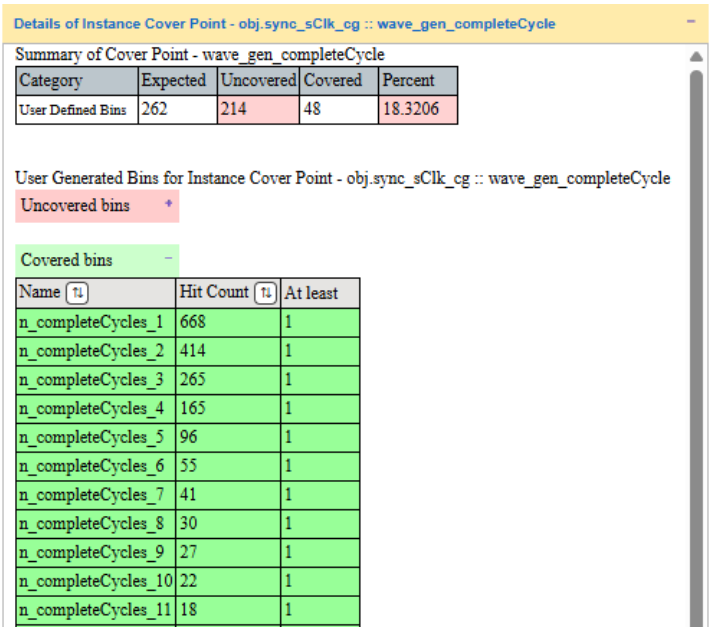
wave_gen_completeCycle

Figura 66 - Informe del grupo de cobertura sync_sClk_cg completo

En el que están cubiertos solo 48 de los 262 considerados como máximo posible dentro de los parámetros de aleatorización, el caso en el que hubiese un `len_PRI` mínimo de 1000 y un `delay_sRun` de 262000 a 262114, que es el valor máximo de su rango más alto.

El que solo estén cubiertos 48 significa que la onda que se mantuvo generándose sin interrupciones por más tiempo fue 48 ciclos.

Viendo en más detalle este punto de cobertura, podemos ver que el número de ciclos completos disminuye drásticamente, por lo que podría ser buena idea ajustar los rangos de aleatorización para orientar el caso o aumentar la duración de la prueba para que se permita la simulación de más casos de manera aleatoria. Además, se puede ver que el número de ondas que terminan un ciclo entero es de 668, por lo que hubieron 236 que se interrumpieron antes de terminar un solo ciclo.



Details of Instance Cover Point - obj.sync_sClk_cg :: wave_gen_completeCycle

Summary of Cover Point - wave_gen_completeCycle

Category	Expected	Uncovered	Covered	Percent
User Defined Bins	262	214	48	18.3206

User Generated Bins for Instance Cover Point - obj.sync_sClk_cg :: wave_gen_completeCycle

Uncovered bins +

Covered bins −

Name	Hit Count	At least
n_completeCycles_1	668	1
n_completeCycles_2	414	1
n_completeCycles_3	265	1
n_completeCycles_4	165	1
n_completeCycles_5	96	1
n_completeCycles_6	55	1
n_completeCycles_7	41	1
n_completeCycles_8	30	1
n_completeCycles_9	27	1
n_completeCycles_10	22	1
n_completeCycles_11	18	1

Figura 67 - Punto de cobertura de ciclos de generación de señal completados

sync_sRun_cg

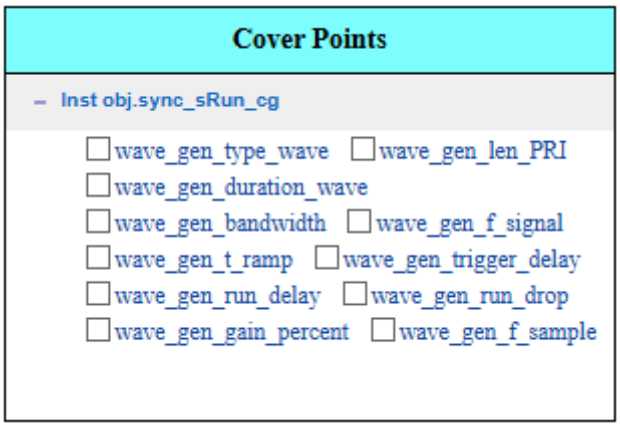


Figura 68 - puntos de cobertura muestreados cada reconfiguración de señal

En el grupo de cobertura muestreado cada accionamiento de la señal `sRun` se puede ver que fueron varios los puntos que no fueron cubiertos

Instance : obj.sync_sRun_cg

Instance Info : obj.sync_sRun_cg

Score	Weight	Goal	At Least	Auto Bin Max	Print Missing	Is Instantized	Comment
93.9177	1	100	1	64	0	1	

Instance Variable[s] Summary : obj.sync_sRun_cg

Cover Points	Category	Expected	Uncovered	Covered	Avg Percent
11	Variables	332	46	286	93.9177

Instance Cover Point[s] Details : obj.sync_sRun_cg

Name	Expected	Uncovered	Covered	Percent	Goal	Weight	At Least	Auto Bin Max	Comment
wave_ge ...	3	0	3	100	100	1	1	64	
wave_ge ...	23	0	23	100	100	1	1	64	
wave_ge ...	12	0	12	100	100	1	1	64	
wave_ge ...	56	0	56	100	100	1	1	64	
wave_ge ...	56	2	54	96.4286	100	1	1	64	
wave_ge ...	26	0	26	100	100	1	1	64	
wave_ge ...	20	0	20	100	100	1	1	64	
wave_ge ...	12	2	10	83.3333	100	1	1	64	
wave_ge ...	14	0	14	100	100	1	1	64	
wave_ge ...	20	0	20	100	100	1	1	64	
wave_ge ...	90	42	48	53.3333	100	1	1	64	

Figura 69 - Resumen de puntos de cobertura de sync_sRun_cg

Estos son, por orden en el que están en la Figura 69, el punto correspondiente a `f_signal`, es decir, la frecuencia del tono, `run_delay`, es decir, el tiempo entre reconfiguraciones de onda y `f_sample`, la frecuencia de muestreo.

Para `f_sample`, los bins que no llegaron a ocurrir fueron los de la Figura 70, que corresponden a los rangos de 1MHz a 1.5Mhz y 3MHz a 3.5Mhz. Por lo que, dado suficiente tiempo de simulación, se podrían haber alcanzado.

Uncovered bins

Name	Hit Count	At least
slow_f_signal[0]	0	1
slow_f_signal[4]	0	1

Figura 70 - Bins no cubiertos del punto de cobertura asociados a f_signal

Para `run_delay`, los bins faltantes son los de la Figura 71,que corresponden a los rangos de 109216 muestras a 185665 muestras y 185666 muestras a 262114 muestras, por lo que al igual que el caso anterior, con más tiempo de simulación se hubiesen alcanzado.

Uncovered bins

Name	Hit Count	At least
longest_run_delay[1]	0	1
longest_run_delay[2]	0	1

Figura 71 - Bins no cubiertos del punto de cobertura asociados a sRun_delay

Para `f_sample` faltan 42 bins de 90 posibles, por lo que en un principio parece un fallo de definición de cobertura más que un fallo del sistema de verificación. Tras mirar en detalle los valores considerados para la frecuencia de muestreo, tal como indica la Figura 39 es de 10Mhz a 100Mhz mientras que la generación aleatoria de reloj hace que se genere un `sClk` de frecuencias de 50MHz a 100MHz, la mitad de los calculados por los periodos aleatorizados mostrados en Figura 33. Por lo que los valores de frecuencia inferiores a 50MHz no deberían haber estado incluidos en la cobertura. Ignorando los bins inferiores a 50Mhz, es decir, todos los inferiores a `mid_range_freq[24]`, quedan 50 bins válidos, de los cuales no se llegan a cubrir 2.

mid_range_freq[22]	0	1
mid_range_freq[23]	0	1
mid_range_freq[24]	0	1
mid_range_freq[39]	0	1
mid_range_freq[48]	0	1

Figura 72 - Bins no cubiertos del punto de cobertura asociado a f_sample

`mid_range_freq[39]`, que corresponde con el rango de 64MHz a 65Mhz y `mid_range_freq[48]`, que corresponde con el rango 73MHz a 74MHz. Por lo que nuevamente está en el caso de que hubiese podido alcanzarlos con tiempo suficiente.

Considerando esa corrección, el porcentaje de cobertura del punto sube de 53.33% a 96% y el porcentaje del grupo subiría de 93.9177% a 97.7965%.

1.8 CONCLUSIONES

Una vez terminado el proyecto de verificación y habiendo visto la progresiva mejora del diseño verificado se pueden extraer las siguientes conclusiones:

Una vez comprendida correctamente la estructura de UVM y visto lo dinámico que puede llegar a ser el proceso de desarrollo de un diseño hardware, se puede afirmar que la propiedad modular que ofrece UVM es fundamental para la adaptación del entorno de verificación al diseño en sí. Esto se ve acentuado al hacer la verificación de manera concurrente al diseño hardware, ya que fue necesaria la inclusión de más señales una vez el entorno ya se había completado, lo cual resulto en una cantidad pequeña de cambios estando ya familiarizado con el sistema UVM al completo.

La complejidad de un diseño hardware hace que se requiera una verificación cada vez más robusta. Un banco de pruebas sencillo puede comprobar que el comportamiento totalmente dentro de los parámetros de funcionamiento sea correcto, pero tanto para establecer dichos parámetros de funcionamiento, encontrar errores cerca de dichos parámetros, detectar comportamientos anómalos o tendencias de error es necesaria una verificación aleatorizada con un informe suficientemente completo como para poder replicar las condiciones que dieron con el error.

Si bien en este caso el diseño era una caja blanca, lo común es que el diseño a verificar sea una caja negra, ya sea por complejidad del código o la ofuscación de este. La metodología UVM permite abstraer el entorno de verificación para únicamente tratar con las señales de entrada y salida, por lo que es útil valorar qué son realmente las señales a las que sí se tiene acceso. Es trabajo de quien verifica establecer cuáles son las métricas consideradas en la verificación y el tratamiento posterior a la prueba de las señales, solo con un entendimiento adecuado del funcionamiento que se busca verificar se puede realizar una verificación eficiente.

Una verificación del 100% real de un diseño mínimamente complejo es inviable, por lo que se tiene que establecer de manera realista una serie comportamientos que tienen que cumplirse durante una prueba, estos se pueden pormenorizar en mayor o menor medida, pero se debe tener presente qué rangos o secuencias son realmente importantes para poder responder si se ha completado la verificación o no.

Por último, resulta interesante la posibilidad de explorar otro simulador como lo es el Questa de Siemens para comparar de manera directa las diferencias que hay entre usar el simulador que los diferentes recursos online sobre UVM recomiendan y su implementación en Vivado.

1.9 BIBLIOGRAFÍA

[1] MCLELLAN, P., 2014. A Brief History of Functional Verification. *Semiwiki* [en línea]. [consulta: 16 agosto 2025]. Disponible en: https://semiwiki.com/eda/3348-a-brief-history-of-functional-verification/.

[2] SUBBU, [sin fecha]. How Intel makes sure the FDIV bug never happens again. [en línea]. [consulta: 24 agosto 2025]. Disponible en: https://www.chiplog.io/p/how-intel-makes-sure-the-fdiv-bug.

[3] SystemVerilog Tutorial. [en línea], [sin fecha]. [consulta: 24 agosto 2025]. Disponible en: https://www.asic-world.com/systemverilog/tutorial.html.

[4] UVM - Universal Verification Methodology | Siemens Verification Academy. *Verification Academy* [en línea], [sin fecha]. [consulta: 24 agosto 2025]. Disponible en: https://verificationacademy.com/topics/uvm-universal-verification-methodology/verificationacademy.com/topics/uvm-universal-verification-methodology/.

[5] Download UVM (Universal Verification Methodology) - Accellera Systems Initiative. [en línea], [sin fecha]. [consulta: 24 agosto 2025]. Disponible en: https://www.accellera.org/downloads/standards/uvm.

[6] Verification Methodology Cookbooks | Siemens Verification Academy. *Verification Academy* [en línea], 2015. [consulta: 24 agosto 2025]. Disponible en: https://verificationacademy.com/cookbook/verificationacademy.com/cookbook/.

[7] Vivado Design Suite User Guide: Logic Simulation (UG900) • Reader • AMD Technical Information Portal. [en línea], [sin fecha]. [consulta: 24 agosto 2025]. Disponible en: https://docs.amd.com/r/en-US/ug900-vivado-logic-simulation.

[8] verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/index.html. [en línea], [sin fecha]. [consulta: 24 agosto 2025]. Disponible en: https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/index.html.

[9] RICH, D., [sin fecha]. Siemens Digital Industries Software. The missing link: the testbech to DUT connection. *2018*.

[10] HORN, M., VAN DER SCHOOT, H., PERYER, M., FIRZOATRICK, T. y STICKLEY, J., 2018. *Universal Verification Methodology UVM Cookbook*. 2018.

[11] SMITH, Doug y AYNSLEY, John. A practical look @ SystemVerilog coverage – tips, tricks, and gotchas. Morgan Hill (CA): Doulos, 2010.

[12] INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800-2023. New York: IEEE, 2024.

[13] In-phase and quadrature components. En: Page Version ID: 1301795961, Wikipedia [en línea], 2025. [consulta: 25 agosto 2025]. Disponible en: https://en.wikipedia.org/w/index.php?title=In-phase_and_quadrature_components&oldid=1301795961.

[14] BERGERON, J.; CERNY, E., HUNTER, A. y NIGHTINGALE, A. Verification methodology manual for SystemVerilog. New York: Springer Science+Business Media, 2006.

2 PRESUPUESTO

Índice de presupuesto.

2 PRESUPUESTO 81

2.1 INTRODUCCIÓN 83
2.2 DESGLOSE DE COSTES 83
2.2.1 Software 83
2.2.2 Equipo Hardware 83
2.2.3 Mano de obra 84
2.3 RESUMEN DE COSTES 85

2.1 Introducción

En este capítulo se establecerán los costes asociados con el proceso de implementación de un sistema de verificación utilizando la metodología UVM desde el aprendizaje hasta la verificación completa del módulo a verificar, asumiendo que es un gasto interno a una empresa orientado a realizar la verificación a más módulos. Cabe destacar que todos los recursos de aprendizaje utilizados son de libre acceso, por lo que ninguno supuso un coste adicional.

2.2 Desglose de costes

2.2.1 Software

Para la implementación y verificación del diseño simulado se utilizó íntegramente el entorno Vivado en su versión base que es gratuita. Dado que los recursos online relacionados a UVM recomiendan el uso del simulador QuestaSim, se podría considerar su uso para futuras implementaciones de entornos de verificación, lo que implicaría el coste de una licencia cuyo precio ronda los 15.000€ en su versión base y este se elevaría conforme se necesiten diferentes características adicionales, pero permitiría el uso de más características para la verificación como el pleno uso de SVA como también diferentes herramientas para flexibilizar el análisis de cobertura funcional.

Para el análisis del error se utilizó Matlab sin ningún add-on, lo que supone el coste de 938€ de una licencia standard. Pero no es realmente necesario, siendo fácilmente sustituido por alternativas gratuitas como lo puede ser Octave, considerado el equivalente libre de Matlab o Python.

2.2.2 Equipo Hardware

Ya que la verificación del diseño se realiza únicamente a nivel de simulación, no se necesita un equipo especifico en el que sintetizar el diseño. Pero sí es necesario un equipo para implementar el sistema de verificación, se utilizó un portátil Vivobook con un procesador 12th Gen Intel(R) Core(TM) i7-1255U (1.70 GHz) y una RAM de 16GB de 3200MT/s de 930€, apto para pruebas cortas, pero para una prueba más larga se hace necesario utilizar un equipo con mejores prestaciones. El equipo utilizado posee un procesador 12th Gen Intel(R) Core(TM) i7-12700F (2.1 GHz) y 64Gb de RAM de 3200MT/s de 2000€, lo que supuso un incremento del rendimiento sustancial respecto a las simulaciones en el portátil.

2.2.3 Mano de obra

El tiempo invertido en la implementación del sistema de verificación puede subdividirse en 3 periodos.

El primer periodo de aprendizaje y familiarización con tanto el lenguaje SystemVerilog como la metodología UVM, En este caso tomó sobre 3 semanas a tiempo completo, es decir, sobre 120h. Este periodo de tiempo solo fue necesario para una primera implementación.

El segundo periodo es la implementación del sistema de verificación en sí, que en este caso fueron sobre 4 semanas a tiempo completo, es decir, sobre 160h. De cara a una futura implementación este periodo de tiempo se podría ver reducido por la modularidad característica de UVM y la experiencia adquirida en la primera implementación.

El tercer y último periodo fue el de pruebas y análisis, que en este caso tomó sobre 2 semanas, es decir sobre 80h. La duración de este periodo es la que podría resultar más variable en futuras implementaciones debido a los posibles imprevistos o errores que se puedan encontrar al ejecutar pruebas.

Se considera de referencia el salario bruto medio de un ingeniero junior en España por hora de 13.85€/h.

Concepto	**Horas**	**Coste (€)**
Periodo de aprendizaje	120	1662
Periodo de implementación	160	2216
Periodo de pruebas	80	1108
Total	360	4986

2.3 Resumen de costes

Concepto	**Coste (€)**
Software	938
Hardware	2930
Mano de obra	4986
Total	8854

Considerando los costes antes expuestos, todo el proceso de entrenamiento e implementación de un entorno de verificación le costaría un total de 8854€ a una empresa. Cabe destacar nuevamente que una parte significativa de los costes son gastos únicos para la empresa, como lo pueden ser el equipo de simulación o licencias software y que futuras implementaciones podrían tardar menos gracias a la experiencia que se pueda ir adquiriendo. Dicho esto, se podría considerar un coste reducido, considerando únicamente el tiempo de implementación y pruebas de 3324€ por diseño, que poder comparar con la alternativa común en el campo de la verificación de diseños hardware, que es la externalización del proceso.

3 CODIGO

Tanto el código completo como el archivo de Vivado, compatible con su versión v2024.2, están en el siguiente repositorio Github.

jcmu_tfg_Implementation_of_UVM_on_Vivado