ULPGC Universidad de Las Palmas de Gran Canaria | eite ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

“Integración de propiedades de verificación en un testbench UVM. Caso práctico para la verificación de un IP”

Titulación:	**Grado en Ingeniería en Tecnologías de la Telecomunicación**
Mención:	**Sistemas Electrónicos**
Autor:	**D. Isaac Rodríguez Díaz**
Tutores:	**Dr. D. Valentín de Armas Sosa** **Dr. D. Félix Bernardo Tobajas Guerrero**
Fecha:	**Junio de 2025**

AGRADECIMIENTOS

Quiero expresar mi más profundo agradecimiento a quienes han sido mi apoyo incondicional y mis referentes a lo largo de este apasionante recorrido:

A mis padres y a mi hermana, mis héroes, por su amor infinito, su confianza y sus enseñanzas diarias. Sin su ejemplo y su aliento constante, este logro no habría sido posible.

A mis abuelos, que han participado activamente en mi educación, transmitiéndome valores, paciencia y sabiduría. En particular a mi abuela Gunda, la cual fue mi primera profesora. Su presencia y sus consejos han sido una guía imprescindible en cada paso.

A mi prima Zaida, culpable —en el mejor de los sentidos— de que yo esté hoy aquí: fuiste tú quien me descubrió la carrera de Telecomunicaciones y encendió en mí la pasión por este mundo.

A mis compañeros de grado, por formar un equipo sólido desde el primer día, por las horas de estudio compartidas, las risas en los descansos y el apoyo mutuo que hizo más llevadero cada reto.

A todos los profesores que he tenido a lo largo del grado, por su profesionalidad, su dedicación y su vocación docente, que han impregnado de conocimiento y entusiasmo cada clase.

Una mención muy especial a mis tutores, el Dr. Valentín de Armas Sosa y el Dr. Félix Bernardo Tobajas Guerrero, por su paciencia infinita, su disponibilidad 24/7, su excelencia profesional y, sobre todo, por su cercanía y humanidad. Ha sido un verdadero privilegio contar con su guía y mentoría en cada fase de este Trabajo Fin de Grado.

"No es la montaña lo que conquistamos, sino a nosotros mismos."
— Sir Edmund Hillary

RESUMEN

En la actualidad, la verificación funcional juega un papel crucial en el desarrollo de sistemas hardware, especialmente en aquellos con alta complejidad, como los *System on Chip* (SoC), que integran múltiples bloques IP en un solo chip. Este contexto ha incrementado la necesidad de metodologías avanzadas que mejoren el flujo de trabajo y optimicen el proceso de verificación.

El presente Trabajo Fin de Grado (TFG) tiene como objetivo la integración de propiedades de verificación *SystemVerilog Assertions* (SVA) en un *testbench* basado en la metodología Universal *Verification Methodology* (UVM). Esta integración pretende mejorar la eficiencia y efectividad en la verificación de un IP, aportando una solución más elegante y reutilizable en comparación con enfoques tradicionales.

Inicialmente, se realizará un análisis exhaustivo del lenguaje *SystemVerilog* y la metodología UVM. Posteriormente, se desarrollará un *testbench* que permita la integración de propiedades de verificación definidas en SVA. Este enfoque, además de ofrecer ventajas en términos de cobertura y detección temprana de errores, contribuirá a un proceso de verificación más robusto, adecuado para los retos que presenta el desarrollo de sistemas electrónicos modernos.

ABSTRACT

Currently, functional verification plays a crucial role in the development of hardware systems, especially those with high complexity, such as System on Chip (SoC), which integrate multiple IP blocks into a single chip. This scenario has increased the need for advanced methodologies that improve workflow and optimize the verification process.

The objective of this Bachelor's Thesis (TFG) is to integrate SystemVerilog Assertions (SVA) properties into a testbench based on the Universal Verification Methodology (UVM). This integration aims to enhance the efficiency and effectiveness of verifying an IP by providing a more elegant and reusable solution compared to traditional approaches.

Initially, an in-depth analysis of the SystemVerilog language and the UVM methodology will be carried out. Subsequently, a testbench will be developed to integrate verification properties defined in SVA. This approach, in addition to providing advantages in terms of coverage and early error detection, will contribute to a more robust verification process, suitable for the challenges presented by modern electronic systems development.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ACRÓNIMOS

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
COITT	Colegio Oficial de Ingenieros Técnicos de Telecomunicación
DSI	División de Diseño de Sistemas Integrados
DUT	Device Under Test
DUV	Device Under Verification
EDA	Electronic Design Automation
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
FIFO	First In, First Out
GITT	Grado en Ingeniería en Tecnologías de la Telecomunicación
HDL	Hardware Description Language
IEEE	Institute of Electrical and Electronics Engineers
IGIC	Impuesto General Indirecto Canario
IP	Intellectual Property
IUMA	Instituto Universitario de Microelectrónica Aplicada
LRU	Least Recently Used
OOP	Object-Oriented Programming
PIFO	Priority In, First Out
RTL	Register Transfer Level
SVA	SystemVerilog Assertions
SV	SystemVerilog
SoC	System on Chip
TFG	Trabajo Fin de Grado
TLM	Transaction-Level Methodology
UVM	Universal Verification Methodology
ULPGC	Universidad de Las Palmas de Gran Canaria
VHDL	VHSIC Hardware Description Language

MEMORIA

1º Capítulo: Introducción

En este primer capítulo se detallarán los antecedentes y las necesidades que han dado lugar a la elaboración de este Trabajo Fin de Grado (TFG). Además, se expondrán los objetivos planteados en la realización de este TFG junto a la estructura del documento correspondiente a la memoria, con el fin de proporcionar una visión general del trabajo y diferenciar los apartados tratados a lo largo de la memoria.

1.1. Antecedentes

El diseño hardware está estrechamente ligado a la verificación funcional, siendo este uno de los procedimientos más importantes en el desarrollo de estos sistemas. Tanta es su relevancia que el porcentaje de tiempo empleado en la verificación funcional en proyectos de diseño IP (*Intelectual Property*) y de diseño de sistemas suele ser mayor del 60% [1]. Esta distribución temporal ha tenido como consecuencia que, desde el año 2007 hasta la actualidad, la relación entre ingenieros de diseño e ingenieros de verificación se invierta, siendo el número de ingenieros de verificación superior en proyectos relacionados con circuitos integrados, como se muestra en la Figura 1.

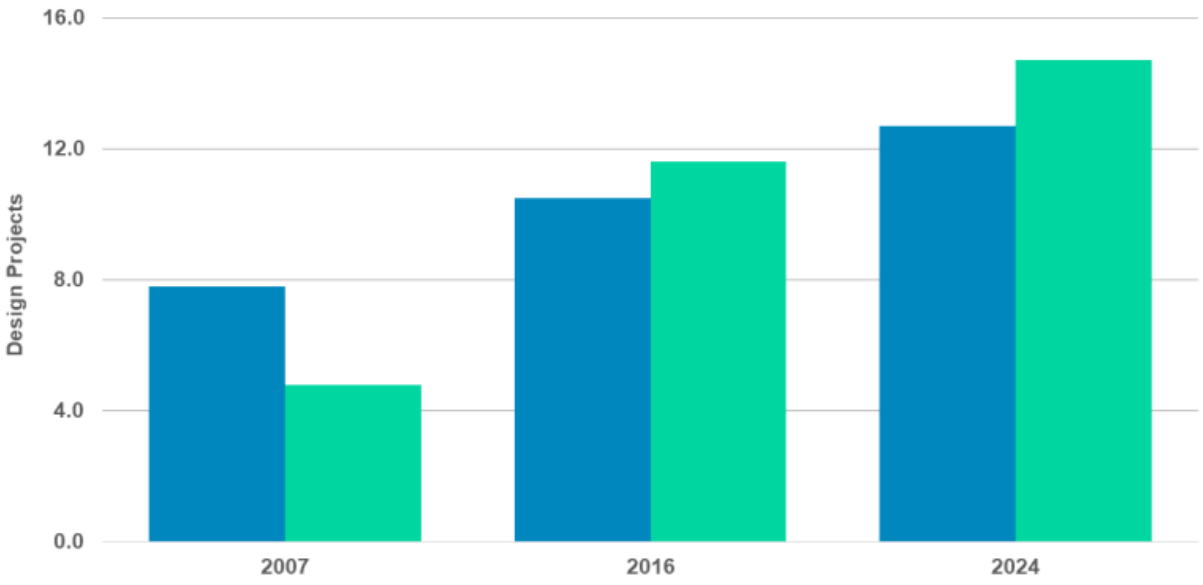

Figura 1 Media de ingenieros en proyectos de circuitos integrados

Actualmente, el desarrollo de sistemas electrónicos integrados se caracteriza por su gran complejidad, como pueden ser los *System on Chip* (SoC), en los que encontramos distintos bloques IP dentro de un mismo chip. Esta situación plantea la necesidad de encontrar un método de verificación distinto a los convencionales, basados en el análisis de formas de onda, para optimizar el flujo de trabajo.

La metodología *Universal Verification Methodology* (UVM) [2] combina el lenguaje de descripción y verificación *hardware SystemVerilog* (SV), que combina las prestaciones de los lenguajes de descripción *hardware* (*Hardware Description Languages,* HDL), como *Verilog,* con otras propiedades y características específicas para la verificación. Entre estas características destacan que añade la funcionalidad de lenguajes como C y C++ [3]; hace uso de TLM (*Transaction-Level Methodology*), que permite la comunicación entre componentes en un nivel muy alto de abstracción; e integra las características de la programación orientada a objetos (*Object-Orieted Programming,* OOP), lo que permite la reutilización de los componentes diseñados. Así, se define un entorno de verificación funcional formado por bloques independientes generados a partir de descripciones SV que interactúan mediante transacciones TLM.

Entre las principales ventajas de adoptar UVM como metodología de verificación funcional, se puede destacar la generación de componentes y objetos reutilizables, requiriendo un mayor esfuerzo en las primeras etapas de la verificación, pero con la capacidad de ser capaz de adaptarse a nuevos sistemas. Esto conlleva a que UVM como metodología y SV como lenguaje sean los más utilizados actualmente, acaparando más del 80% de los proyectos, como se muestra en la Figura 2.

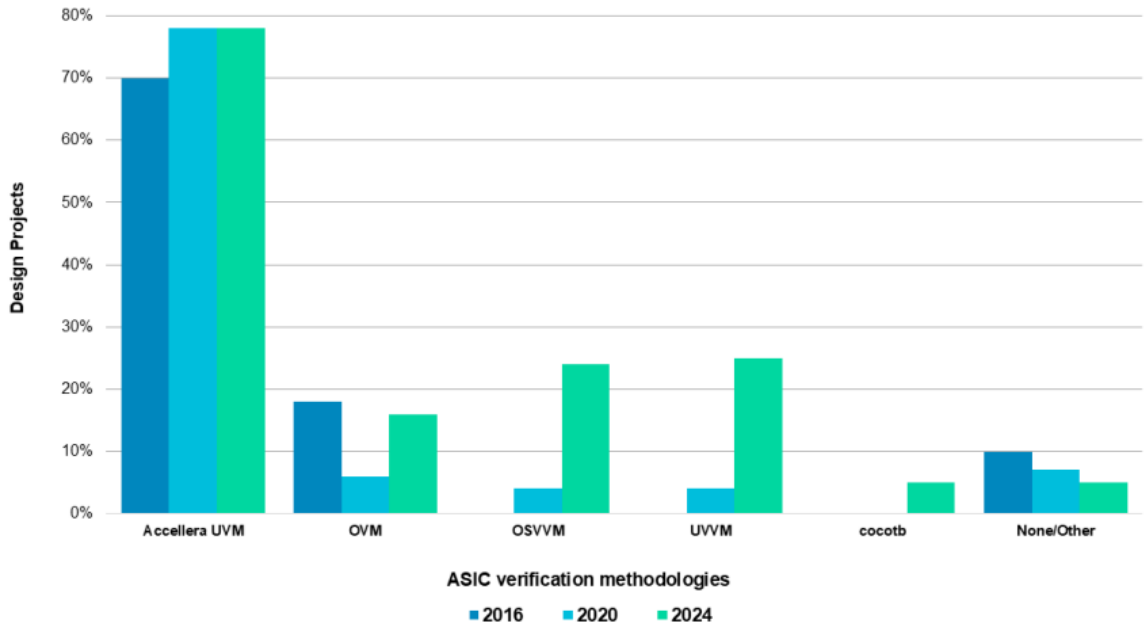

Figura 2 Uso de metodologías para la creación de testbench

En el mercado laboral actual, la importancia de UVM para el perfil de ingeniero de verificación es muy elevada, siendo su conocimiento y la experiencia en su uso requisitos obligatorios para la mayoría de los puestos de trabajo relacionados con este sector, en los que no hay propuesta en las que no se soliciten estas características [4]. Esto sucede en empresas como *Meta*, antigua *Facebook*, que pide como requisito para el puesto de ingeniero de verificación tener una experiencia mínima de más de cinco años en SV y UVM [5].

SystemVerliog además de ser un lenguaje de descripción hardware proporciona un lenguaje para la descripción de propiedades de verificación, *SystemVerilog Assertions* (SVA). Las *assertions* son expresiones que especifican ciertos comportamientos o condiciones que deben cumplirse en un diseño. Se utilizan para describir propiedades formales y restricciones funcionales en el diseño del hardware.

Las SVAs permiten a los diseñadores y verificadores especificar de manera concisa y precisa las expectativas del comportamiento del hardware a lo largo del tiempo. Estas afirmaciones se pueden utilizar durante la simulación para verificar automáticamente si el diseño cumple con las condiciones especificadas.

Además, las SVAs son una herramienta valiosa en la verificación formal, donde se utilizan para demostrar matemáticamente la corrección del diseño en lugar de depender únicamente de pruebas de simulación [6], las cuales no cubren todo el espacio de funcionamiento de un diseño.

El objetivo de este Trabajo Fin de Grado (TFG) radica en la integración de propiedades de verificación, previamente definidas en SystemVerilog, de manera efectiva en un *testbench* basado en la metodología de verificación universal (UVM). Tanto la metodología UVM como el lenguaje SystemVerilog presentan una complejidad suficiente como para ser abordada en su totalidad dentro de los tiempos establecidos para un TFG. Por este motivo se partirá de los trabajos previos realizados en el seno de la división de Diseño de Sistemas Integrados (DSI) del Instituto Universitario de Microelectrónica Aplicada (IUMA) la cual tiene experiencia con la metodología UVM pero carece de experiencia en el uso e implementación de estas propiedades en UVM ya que no es algo directo.

1.2. Objetivos

El objetivo final del presente TFG, consiste en la integración de propiedades de verificación, definidas en SystemVerilog (SV), a un *testbench* UVM. La integración de propiedades SVA en un *testbench* UVM no es directa. Esto es debido a que las propiedades en SVA son propiedades concurrentes (creadas en tiempo de compilación) y este tipo de propiedades no se pueden definir dentro de las clases de SystemVerilog (creadas en tiempo de ejecución). La mayor parte de la literatura existente resuelve este inconveniente ubicando las propiedades en el único objeto no dinámico definido en SystemVerilog, como es el "interfaz" (*interface*). En este Trabajo Fin de Grado se tratará de dar una solución más elegante (entendiendo la elegancia en términos de reusabilidad) tal y como se presenta en el artículo [7]. Una vez finalizada la integración, se ejecutará el *testbench* y se comprobará el cumplimiento de las propiedades. La combinación de propiedades de SystemVerilog con el marco de trabajo UVM ofrece ventajas en términos de expresividad, verificación formal, integración con UVM, mejora de la cobertura y detección temprana de errores. Esta combinación fortalece la metodología de verificación y contribuye a un proceso más efectivo y eficiente en el desarrollo de sistemas de hardware complejos.

Para cumplir este objetivo, se propone concretar en los siguientes objetivos:

O1. Analizar las tendencias metodológicas en el campo de la verificación funcional dinámica, identificando sus dependencias a nivel de lenguajes de diseño y verificación.

O2. Ser capaz de describir un *testbench* funcional de un módulo IP, así como de interpretar los resultados obtenidos e identificar casos no verificados.

O3. Estudiar y entender las ventajas de los lenguajes de verificación formal basados en propiedades, sus ventajas y sus limitaciones en la verificación funcional de un sistema electrónico.

O4. Ser capaz de describir propiedades de verificación, así como de integrarlas en un *testbench* tradicional, sabiendo justificar el uso de las *mismas* en la verificación de un diseño.

O5. Ser capaz de describir un *testbench* UVM de un módulo IP, así como de interpretar los resultados obtenidos e identificar casos no verificados.

O6. Adquirir la habilidad para definir propiedades de verificación y su integración efectiva en un *testbench* UVM, respaldando la elección de estas propiedades en el proceso de verificación de un diseño.

1.3. Peticionario

El peticionario de este Trabajo Fin de Grado es la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), en calidad de institución pública que solicita la realización de dicho trabajo con el fin de superar los requisitos impuestos en la asignatura Trabajo Fin de Grado en el plan de estudios de la titulación Grado en Ingeniería en Tecnologías de la Telecomunicación (GITT).

1.4. Estructura del documento

El presente documento se divide en cuatro secciones: Memoria, Pliego de condiciones, Presupuesto y Anexo. La Memoria se estructura en siete capítulos y la bibliografía empleada. A continuación, se resume el contenido de estos capítulos:

- Capítulo 1º. Introducción. En este primer capítulo se presentan los antecedentes y la importancia de la verificación funcional en el diseño de sistemas electrónicos, lo que da lugar a la motivación para realizar este TFG. Además, se plantean los objetivos del trabajo, así como el alcance y la metodología que se seguirá.
- Capítulo 2º. Metodología y herramientas. Este capítulo describe las principales herramientas y metodologías empleadas para el desarrollo del proyecto, como el lenguaje SystemVerilog, la metodolgía de verificación UVM y las propiedades SystemVerilog *Assertions* (SVA). Se analiza en detalle la relevancia de cada una de estas tecnologías y su papel en la integración de un entorno de verificación funcional robusto.
- Capítulo 3º. Desarrollo de un testbench funcional. En este capítulo se estudia el dispositivo que será objeto de verificación (*Device Under Verification*, DUV). Se profundiza en sus características principales y se detalla el comportamiento que se espera del IP, con el fin de establecer un punto de partida para la verificación, elaborando un *testbench* funcional del mismo.
- Capítulo 4º. Lenguajes de verificación formal. Este capítulo analiza los lenguajes de verificación formal basados en propiedades, destacando sus ventajas y limitaciones en la verificación de sistemas electrónicos.
- Capítulo 5º. Propiedades de verificación en SystemVerilog. Aquí se aborda el análisis de las propiedades en SystemVerilog Assertions (SVA). Se describen las propiedades diseñadas para verificar el comportamiento del IP estudiado y se explica cómo se integran dentro del *testbench* funcional.
- Capítulo 6º. Desarrollo del entorno UVM. En este capítulo se desarrolla el entorno de verificación basado en UVM para el módulo IP seleccionado. Se detalla el proceso de implementación y los componentes de verificación diseñados, así como la ejecución del *testbench*.
- Capítulo 7º. Definición e integración de propiedades en UVM. Este capítulo se centra en la habilidad para definir propiedades de verificación en SystemVerilog y su integración efectiva en un *testbench* UVM. Se detallan los criterios para la elección de

las propiedades más adecuadas, así como el proceso de implementación y validación dentro del entorno de verificación.

- Capítulo 8º. Resultados de verificación. En este capítulo se presentan los resultados obtenidos tras la ejecución del *testbench* UVM. Se analiza el cumplimiento de las propiedades de verificación y se evalúa el comportamiento del IP, identificando posibles mejoras.
- Capítulo 9º. Conclusiones y líneas futuras. En este último capítulo se resumen las principales conclusiones obtenidas a lo largo del desarrollo del proyecto. Además, se sugieren posibles líneas de desarrollo y futuras ampliaciones que puedan derivarse del presente TFG.

Capítulo 2º: Metodología y herramientas

Este capítulo describe las principales herramientas y metodologías empleadas para el desarrollo del proyecto, destacando el papel del lenguaje SystemVerilog, la metodología de verificación Universal Verification Methodology (UVM) y las propiedades SystemVerilog Assertions (SVA). A continuación, se analiza en detalle la relevancia de cada una de estas tecnologías y cómo contribuyen a la integración de un entorno de verificación funcional robusto.

2.1. Lenguaje SystemVerilog

SystemVerilog es un lenguaje de descripción de hardware (HDL) y de verificación que se utiliza ampliamente en el diseño y modelado de circuitos digitales complejos dentro del campo de la electrónica VLSI. Surge como una extensión de Verilog, que fue estándar en la industria desde 1995 bajo el nombre IEEE 1364, y posteriormente fue mejorado hasta convertirse en SystemVerilog, un *superset* designado como IEEE 1800 en 2005, que se actualizó en 2012 [8]. SystemVerilog se creó para facilitar tanto la descripción estructural como la verificación de sistemas complejos, aportando mayores niveles de abstracción y capacidades de verificación.

Las principales diferencias entre Verilog y SystemVerilog se encuentran en sus niveles de abstracción y en sus capacidades de verificación. Verilog es un lenguaje de bajo nivel que se enfoca en describir el comportamiento del hardware en detalle, mientras que SystemVerilog [9] permite la descripción y modelado de sistemas complejos de forma más concisa y eficiente, gracias a su mayor nivel de abstracción. Además, SystemVerilog introduce potentes capacidades de verificación integradas que lo hacen una herramienta preferida para ingenieros de verificación, como la posibilidad de usar programación orientada a objetos con conceptos como clases y objetos, que facilitan la reutilización del código y mejoran la organización.

2.1.1. Características de SystemVerilog

- Extensión de Verilog: SystemVerilog es una extensión de Verilog que agrega numerosas características nuevas para facilitar el diseño y verificación de sistemas complejos.
- Mayor nivel de abstracción: Permite trabajar con niveles de abstracción más altos, lo cual facilita la descripción de sistemas complejos de forma más sencilla y eficiente en comparación con Verilog.
- Capacidades de verificación avanzadas: Incluye una serie de características integradas para la verificación, como la capacidad de describir propiedades, dominadas aserciones. cobertura funcional y proporciona un entorno de pruebas basado en clases, que hacen que la verificación sea más robusta y completa.
- Programación Orientada a Objetos (OOP): SystemVerilog introduce conceptos de programación orientada a objetos, como clases, herencia y polimorfismo. Esto permite una mejor organización del código, facilita la reutilización y mejora la escalabilidad de los proyectos de verificación.
- Aserciones *Assertions*: Las aserciones permiten la descripción de propiedades, lo que facilita verificar que ciertas condiciones se cumplan durante la simulación, lo cual es crucial para detectar errores de diseño de forma temprana.
- Cobertura Funcional: SystemVerilog permite medir la cobertura funcional para asegurar que todas las partes del diseño han sido verificadas adecuadamente, garantizando una mayor calidad del producto final.
- Interfaz de Múltiples Niveles: Ofrece la capacidad de describir interfaces complejas y manejar la comunicación entre módulos de manera más eficiente mediante el uso de interfaces y *modports* [10].
- Capacidad de Testbench Dinámico: SystemVerilog permite la creación de *testbenches* dinámicos y reutilizables que simplifican la generación de estímulos y la comprobación de resultados, mejorando la eficiencia del proceso de verificación.

2.1.2.Bancos de pruebas en SystemVerilog

SystemVerilog se destaca particularmente en la creación de bancos de prueba (*testbench*) eficientes. Un banco de prueba en SystemVerilog es un entorno de verificación

compuesto por una serie de clases que permiten validar la correcta funcionalidad del Diseño Bajo Verificación (DUV). Los componentes de un banco de prueba incluyen:

- *Transaction:* Clase que contiene la estructura usada para comunicarse con el DUV, proporcionando información crítica sobre el estímulo creado.
- *Generator:* Responsable de generar el estímulo de prueba y asignarlo aleatoriamente antes de enviarlo al componente *driver*.
- *Driver:* Componente que recibe las transacciones del generador y las envía al DUV.
- *Monitor:* Componente que observa la actividad de las señales del DUV y ,dependiendo de la lógica implementada, procede enviándolas a otros componentes como el *scoreboard*.
- *Scoreboard:* Componente que compara los datos observados con los valores esperados, que pueden ser valores de referencia o generados por un modelo de referencia descrito en algún lenguaje de alto nivel.
- *Agent:* Contiene clases como el generador, driver y monitor, y agrupa componentes específicos para una interfaz o protocolo, normalmente estándares.
- *Environment:* Componente que actúa como un contenedor de componentes de alto nivel como los componentes agentes y scoreboards.
- *Test:* Es el componente responsable de configurar el banco de prueba, iniciar la construcción de componentes y conducir la estimulación.
- *Testbench_top:* Conecta el DUV con el banco de pruebas e incluye las referencias al DUV, la prueba y las interfaces. Este componente es el encargado de activar la generación dinámica del banco de pruebas.

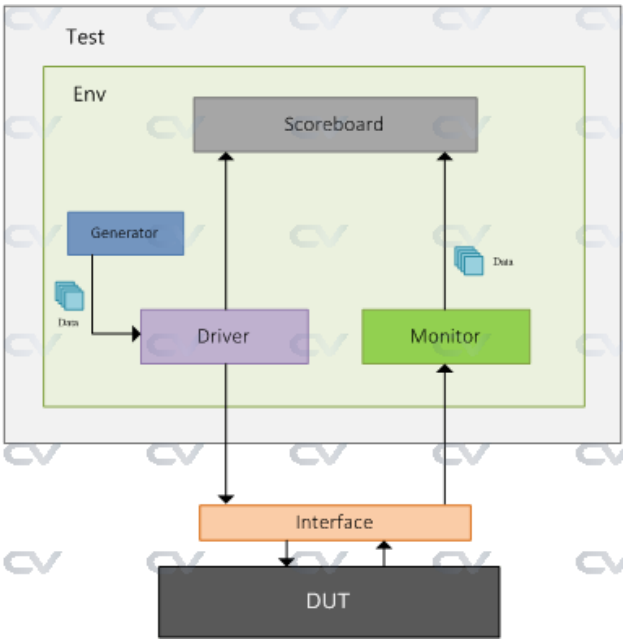

Figura 3: Esquema básico testbench

El objetivo del banco de pruebas de SystemVerilog es generar diferentes tipos de estímulos, aplicar estos estímulos al diseño y comparar los resultados obtenidos con el comportamiento esperado. De esta forma, se pueden identificar fallos funcionales y poder realizar correcciones en el diseño, asegurando que el circuito funcione de manera confiable [11].

2.2. Metodología de Verificación Universal (UVM)

La metodología *Universal Verification Methodology* (UVM) es un estándar de la industria para la creación de entornos de verificación. La metodología UVM proporciona una estructura modular y reutilizable que facilita la verificación de diseños complejos, reduciendo el esfuerzo de desarrollo y mejorando la calidad del proceso. En este proyecto, la metodología UVM se utiliza para implementar un *testbench* estructurado que incluye componentes como drivers, monitores, secuencias y agentes, cada uno de los cuales tiene un rol específico en la generación y monitoreo de estímulos para el dispositivo bajo verificación (DUV). La modularidad de UVM permite crear un entorno de verificación robusto y fácilmente adaptable a nuevos requisitos [12].

2.2.1.Características de UVM

Las principales características de la metodología UVM son:

- Estandarización: UVM proporciona un entorno de verificación estándar que facilita la colaboración entre equipos y empresas, asegurando que todos trabajen bajo un mismo marco metodológico, lo cual mejora la interoperabilidad de componentes de verificación.
- Reutilización de Componentes: UVM promueve la reutilización de componentes de verificación mediante el uso de clases base y patrones de diseño comunes, lo cual permite crear *testbenches* que se puedan reutilizar para múltiples proyectos o diferentes versiones de un mismo diseño.
- Jerarquía Modular: UVM organiza los componentes de verificación de manera modular, dividiéndolos en agentes, secuencias, monitores y otros elementos. Esto facilita la construcción de entornos de verificación flexibles y la escalabilidad de los mismos.
- Capacidades de Generación de Estímulos: Con UVM se pueden generar estímulos de prueba complejos y variados, utilizando generadores aleatorios y controlados que permiten probar diferentes escenarios y condiciones límites para asegurar la robustez del diseño.
- Automatización de Verificación: UVM proporciona una infraestructura que automatiza la creación y ejecución de pruebas, incluyendo la generación de estímulos, la recolección de cobertura y la comparación de resultados. Esto reduce el esfuerzo manual y minimiza los errores humanos.
- Informes y Cobertura: UVM tiene capacidades integradas para generar informes detallados de las pruebas y medir la cobertura funcional del diseño, permitiendo identificar áreas no verificadas y garantizar que se cumplan los objetivos de calidad.

2.2.2. Bancos de prueba en UVM

Un banco de pruebas, o *testbench,* en UVM es un entorno de prueba diseñado para interactuar con el *Design Under Verification* (DUV) y verificar su comportamiento bajo diversas condiciones. La arquitectura típica de un *testbench* UVM, como muestra la Figura 4, incluye los siguientes componentes clave:

- ***Test:*** Es el componente de nivel superior que configura y controla la ejecución del *testbench*. Define las condiciones iniciales, configura los parámetros del entorno y lanza las secuencias de estímulos que se aplicarán al DUT.
- ***Environment:*** Agrupa y organiza los diferentes agentes y otros componentes necesarios para la verificación. Proporciona una estructura jerárquica que facilita la gestión y configuración del *testbench*.
- ***Agent:*** Encapsula los elementos necesarios para la generación y monitorización de estímulos en una interfaz específica del DUV. Un componente *agent* típico incluye un componente *driver*, un *monitor* y un componente *sequencer*.
- ***Driver:*** Convierte las transacciones generadas por las secuencias en señales que se aplican directamente a las entradas del DUV. Actúa como un puente entre el nivel abstracto de las transacciones y el nivel físico de las señales.
- ***Monitor****:* Observa las señales del DUV y las convierte en transacciones que pueden ser analizadas. Es esencial para la verificación pasiva y la recopilación de datos para el análisis de cobertura y la comparación con los resultados esperados.
- ***Sequencer:*** Controla el flujo y la generación de transacciones que se enviarán al componente *driver*. Permite la creación de secuencias de estímulos complejas y controladas.
- ***Scoreboard****:* Compara las salidas reales del DUV con las salidas esperadas para determinar la corrección funcional. Es fundamental para la validar de que el DUV se comporta según lo especificado.

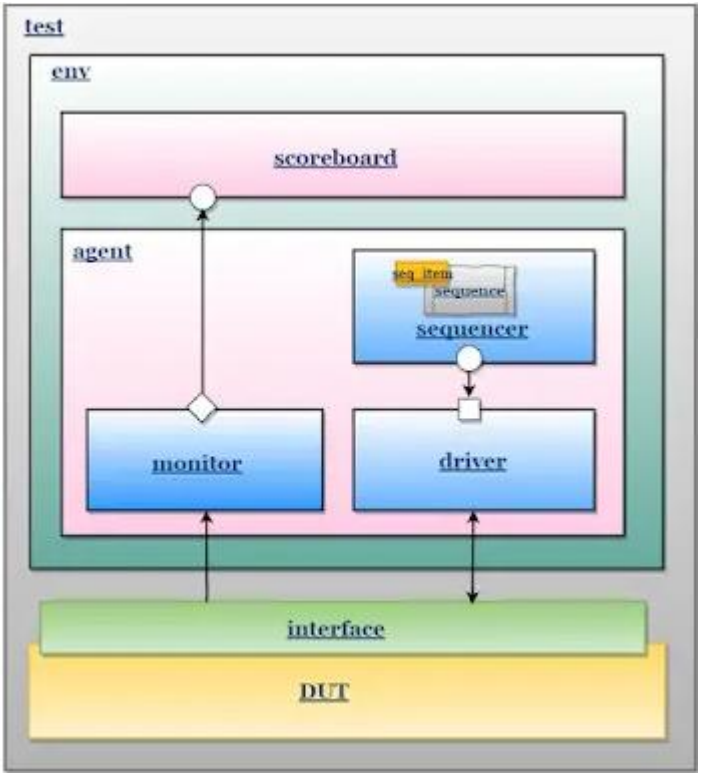

Figura 4: Arquitectura testbench UVM

2.2.3. Librerías de clases UVM

Los componentes en UVM proporciona una librería completa de clases, que son las unidades básicas que conforman el *testbench*. Cada componente, que deriva de una de estas clases tiene un rol específico y se integra en la jerarquía del *testbench* para facilitar una verificación estructurada y eficiente [13]. Los principales componentes de UVM incluyen:

- ***uvm_component***: Es la clase base para todos los componentes en UVM. Proporciona la infraestructura necesaria para la construcción jerárquica, la configuración y la sincronización de fases en el *testbench*. Todos los componentes, como *driver*, *monitor* y *scoreboard*, heredan de *uvm_component*, lo que les permite participar en el flujo de fases y la gestión jerárquica del *testbench*.
- ***uvm_object:*** Es la clase base para los objetos que no forman parte de la jerarquía de componentes, como las transacciones y las secuencias. Proporciona métodos para la

impresión, comparación y copia de objetos, facilitando la manipulación y análisis de datos en el *testbench*.

- ***uvm_sequence_item****:* Representa las transacciones o estímulos que se envían al DUV. Es una extensión de *uvm_object* y se utiliza para encapsular los datos y comportamientos necesarios para las pruebas. Las secuencias de transacciones se construyen utilizando la clase *uvm_sequence_item*, permitiendo la generación de estímulos complejos y variados.

- ***uvm_sequence*:** Define y asigna una serie transacciones de *uvm_sequence_item* que se ejecutan en un componente *sequencer*. A esta agrupación se la denomina secuencia. Permite la creación de patrones de estímulos específicos para verificar diferentes aspectos del DUV. Las secuencias pueden ser reutilizadas y combinadas para crear escenarios de prueba complejos, mejorando la eficiencia de la verificación.

- ***uvm_driver:*** Es un componente que recibe un objeto del tipo *uvm_sequence_item* a través del *sequencer* y lo traduce en señales que se aplican al DUV. Actúa como un controlador activo que estimula el DUV según las transacciones definidas. El componente *driver* es esencial para la aplicación precisa y controlada de estímulos al DUV.

- ***uvm_monitor:*** Este componente realiza la operación inversa, observa las señales del DUV y crea objetos del tipo *uvm_sequence_item* basados en la actividad observada. Es un componente pasivo que no influye en el funcionamiento DUV, pero proporciona información crucial para el análisis y la verificación. El componente *monitor* es fundamental para la recopilación de datos y la evaluación del comportamiento del DUV.

- ***uvm_agent:*** Agrupa los componentes *driver*, el *sequencer* y el *monitor* para una interfaz específica del DUT. Facilita la gestión y configuración de estos componentes como una unidad coherente, simplificando la estructura del *testbench*. El componente *agent* permite una integración modular y reutilizable de los componentes de verificación.

- ***uvm_env:*** Es un contenedor que agrupa múltiples componentes *agents* y otros componentes necesarios para la verificación. Proporciona una estructura organizada y jerárquica para el *testbench*, facilitando la gestión y escalabilidad del entorno de verificación como muestra la Figura 5.

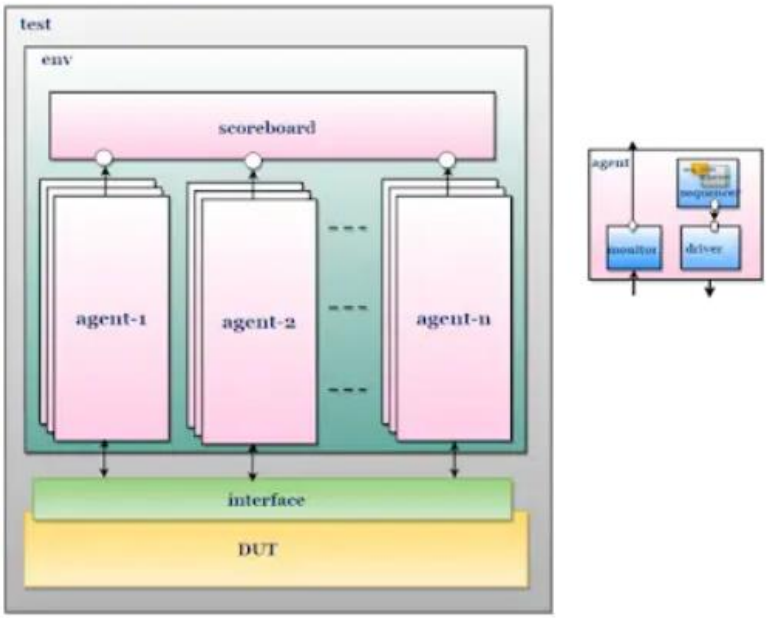

Figura 5: Arquitectura UVM 2

2.3. SystemVerilog Assertions (SVA)

Las SystemVerilog *Assertions* (SVA), o aserciones, son una herramienta fundamental para la verificación formal y funcional del diseño. Permiten especificar de manera concisa las condiciones y restricciones que deben cumplirse durante la ejecución del diseño, proporcionando una manera efectiva de detectar errores en etapas tempranas del desarrollo. En este proyecto, las aserciones SVA se integran en el *testbench* UVM para asegurar que el dispositivo bajo verificación cumpla con las especificaciones establecidas. El uso de las aserciones SVA mejora la cobertura de verificación y facilita la identificación de problemas complejos que podrían pasar desapercibidos mediante métodos de simulación convencionales [14].

2.3.1.Tipos de *Assertions* en SystemVerilog:

SystemVerilog proporciona los tipos de *assertions*, como son:

- *Immediate Assertions*

Estas *assertions* se evalúan inmediatamente en el contexto procedural en el que se colocan. Son ideales para verificar condiciones que deben cumplirse en un momento específico de la simulación.

- *Concurrent Assertions*

Estas *assertions* evalúan relaciones entre señales a lo largo del tiempo y permiten describir secuencias temporales. Se implementan utilizando propiedades descritas mediante el token (*property*) y secuencias descritas mediante el token *(sequence*).

2.3.2. Ventajas de SystemVerilog Assertions:

De forma muy resumida, las principales ventajas de las assertions en SystemVerilog son:

- Detección Temprana de Errores: Las SVA, permiten y facilitan la detección de problemas en las primeras etapas del ciclo de diseño, lo que reduce los costos y el tiempo de depuración.
- Automatización: Las *assertions* permiten automatizar la verificación, reduciendo la necesidad de revisar manualmente las señales utilizando las formas de onda.
- Cobertura Funcional: Al capturar explícitamente los requisitos del diseño, las SVA aseguran que todas las condiciones críticas se evalúan durante la simulación.
- Compatibilidad con Herramientas de Verificación Formal: Las *assertions* son interpretadas por todas las principales herramientas de verificación formal para analizar el diseño y demostrar matemáticamente su corrección.

2.3.3. Comparativa: Immediate vs Concurrent *SystemVerilog Assertions*:

Característica	Immediate Assertions	Concurrent Assertions
Evaluación	Inmediata	A lo largo de la ejecución
Uso Principal	Condiciones puntuales	Secuencias temporales
Complejidad	Baja	Alta
Aplicación Típica	Verificar I/O	Verificar protocolos

Tabla 1: Comparativa Inmediate vs Concurrent Assertions

La Tabla 1 muestra una comparativa entre los dos tipos principales de *SystemVerilog Assertions*: las *Immediate Assertions* y las *Concurrent Assertions*. Cada tipo está orientado a un propósito distinto dentro del flujo de verificación y presenta características específicas en cuanto a su evaluación, uso, complejidad y ámbito de aplicación.

Las *Immediate Assertions* son más simples y se evalúan de forma inmediata en el mismo ciclo de simulación en el que se ejecutan, lo que las hace ideales para comprobar condiciones puntuales en señales o interfaces. Por otro lado, las *Concurrent Assertions* permiten describir secuencias temporales complejas, capturando comportamientos esperados del diseño a lo largo del tiempo, lo cual resulta especialmente útil en la verificación de protocolos o lógica secuencial compleja.

Comprender las diferencias entre ambas resulta clave a la hora de decidir qué tipo de *assertion* emplear en función del objetivo de verificación, el contexto temporal requerido.

3º Capítulo: Desarrollo testbench funcional

En este capítulo se analiza en profundidad el módulo IP que se usará para el proceso de verificación, el cual constituye el dispositivo bajo verificación (DUV). Su entendimiento detallado es esencial para poder desarrollar un *testbench* funcional coherente, capaz de evaluar correctamente su comportamiento ante distintos estímulos. Este módulo puede entenderse como una memoria caché, una estructura fundamental en sistemas donde no solo se valora la secuencia de llegada de los datos, sino también la importancia relativa de cada uno.

3.1.Analogía con sistemas caché

En arquitectura de ordenadores, una memoria caché es un tipo de almacenamiento intermedio que se sitúa entre el procesador y la memoria principal. Su propósito es reducir el tiempo de acceso a los datos más utilizados, almacenándolos temporalmente en una estructura de menor latencia. Las cachés suelen estar organizadas en bloques o líneas, y cada entrada almacena tanto los datos como información de control, como etiquetas y bits de validez.

Una de las claves en el diseño de una caché es la política de reemplazo, es decir, el criterio por el cual se decide qué bloque de datos se debe eliminar cuando la caché está llena y se necesita hacer espacio para un nuevo dato. Las políticas más comunes incluyen FIFO (el primero en entrar, primero en salir), LRU (el menos recientemente usado) o por prioridad, donde se evalúa la importancia del dato [14].

El módulo IP presenta un comportamiento análogo a una caché con política de reemplazo por prioridad: cuando el registro está lleno y se intenta insertar un nuevo dato, este solo será almacenado si su prioridad (*rank*) es mayor, es decir, menor en valor numérico, que la del dato actualmente menos prioritario. En ese caso, el dato de menor prioridad es reemplazado. Esta lógica convierte al IP en una solución eficaz para sistemas donde se requiere mantener solo los elementos más relevantes, descartando los de menor interés conforme llegan nuevas entradas.

3.2.Análisis funcional del DUV

Cada elemento insertado en el módulo está caracterizado por una prioridad, representada por la señal *rank_in*, y un conjunto de metadatos (*meta_in*). Internamente, el módulo mantiene un conjunto de registros con estructuras auxiliares para almacenar estos datos junto a su validez, lo que le permite mantener la coherencia en su comportamiento.

El proceso de inserción se lleva a cabo únicamente si el número de entradas no ha alcanzado el máximo definido por el parámetro *L2_REG_WIDTH*, lo que establece la profundidad del registro mediante una potencia de dos. En este caso, el nuevo valor se añade al final del conjunto y se marca como válido. Si el módulo está lleno (*full* = 1), el sistema evalúa si el nuevo elemento tiene una prioridad suficientemente alta como para reemplazar el que en ese momento ostenta la menor prioridad. Este comportamiento emula una política de reemplazo basada en prioridad, similar a las empleadas en memorias caché selectivas, donde se protege la permanencia de los datos más relevantes.

En cuanto a las operaciones de eliminación, el módulo permite extraer el dato de mayor prioridad, es decir el de menor valor en *rank*, accediendo directamente al índice calculado mediante un sistema jerárquico de comparadores. Este mecanismo evita tener que recorrer toda la estructura, aumentando así la eficiencia. Tras la eliminación, el resto de los elementos son desplazados para cerrar el hueco, y se actualizan las señales de estado como *empty y num_entries,* indicando esta última el número de entradas válidas para la PIFO.

Además, el módulo proporciona información sobre el dato menos prioritario a través de las señales *max_rank_out* y *max_meta_out*, lo cual resulta clave para la toma de decisiones externas cuando el registro está lleno. Las señales *valid_out* y *max_valid_out* actúan como indicadores para determinar si los datos extraídos o visualizados en la salida son efectivamente válidos, lo que refuerza la robustez del diseño.

3.3.Propósito del testbench funcional

El *testbench* funcional se ha desarrollado con el propósito de validar los distintos modos de operación del módulo, prestando especial atención a los límites del sistema, las

transiciones de estado y la correcta actualización de las señales de salida. Se busca, por tanto, comprobar que el IP inserta correctamente los datos cuando el registro no está lleno, que responde adecuadamente a la lógica de reemplazo cuando está al límite de su capacidad, y que los datos extraídos tras una operación de *remove* corresponden efectivamente al de menor prioridad.

También se evalúa el correcto funcionamiento de las señales auxiliares como *full*, *empty*, *valid_out* y *max_valid_out*, así como el adecuado conteo interno de elementos. Estos indicadores no solo reflejan el estado del sistema, sino que también forman parte de las condiciones necesarias para que el módulo se integre de forma segura en un sistema de mayor complejidad.

El entorno de verificación diseñado para este propósito se estructura en torno a agentes generadores y monitores, y un *scoreboard* que actúa como referencia para validar las salidas del DUV.

3.4.Desarrollo del testbench funcional

Se ha desarrollado un *testbench* funcional escrito íntegramente en *SystemVerilog* y ejecutado sobre la plataforma EDAPlayground. Este entorno no sigue la metodología UVM, pero adopta una estructura modular y jerárquica que facilita tanto la estimulación del dispositivo como la observación y verificación de sus respuestas.

El motivo de no usar la metodología UVM en este punto es debido a que lo que se persigue en este apartado es el estudio del IP y no el de la metodología en sí. Por otro lado, la implementación de un testbench en SystemVerilog permitirá, además, comparar los resultados obtenidos con los futuros resultados usando la metodología UVM y, de esta manera, poder comparar ambos.

La implementación se basa en un enfoque clásico de *testbench*, en el que cada parte del sistema tiene una responsabilidad bien definida. El diseño se inspira en un ejemplo publicado en *Verification Guide* [15], en el que se describe un banco de pruebas funcional para un sumador. A partir de ese modelo, se ha construido un entorno adaptado específicamente al comportamiento y señales del IP.

3.4.1. Arquitectura del entorno funcional

El *testbench* se estructura en torno a dos bloques principales tal y como se muestra en la Figura 4. El camino de entrada, que se encarga de generar y aplicar datos al módulo bajo prueba, y el camino de salida, que observa las respuestas del sistema y verifica su validez. Ambos caminos están conectados mediante interfaces *SystemVerilog* que encapsulan las señales de control y datos, mejorando la claridad y la reutilización del entorno.

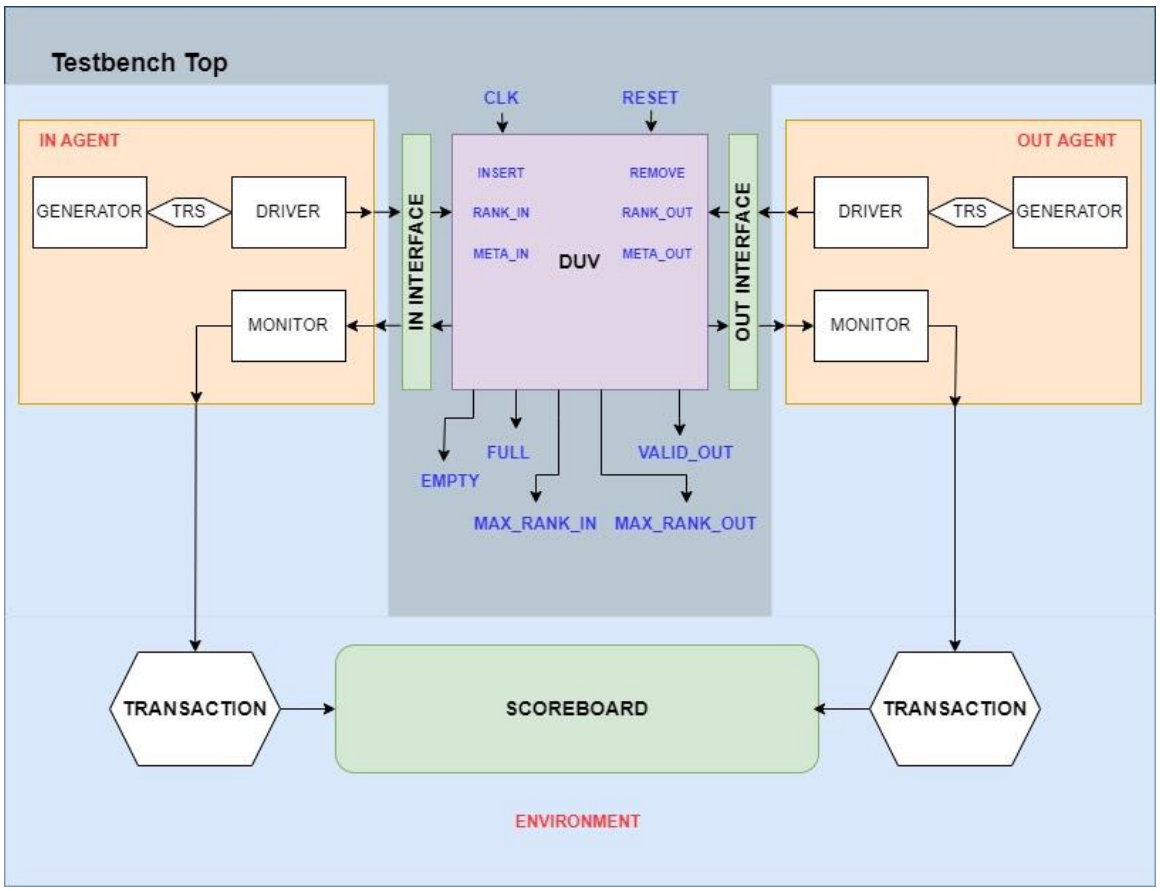

Figura 6: Arquitectura del entorno de verificación propuesto

En el camino de entrada (*In Agent*), el componente *generator_in* es el encargado de crear estímulos aleatorios, empaquetados en estructuras de datos denominadas transacciones (*transaction_in*). Cada transacción contiene la información necesaria para simular una operación de inserción: una prioridad (*rank_in*), metadatos asociados (*meta_in*) y la señal de control *insert*. Estas transacciones son enviadas al, componente *in_driver*, que se encarga de traducirlas en señales concretas que se aplican al DUV a través de la interfaz de entrada. De forma simultánea, el componente *in_monitor* observa continuamente la actividad en dicha

interfaz, detectando cuándo se ha producido una inserción válida y registrando el contenido de la transacción en él, componente *scoreboard*, permitiendo así llevar un seguimiento de los datos introducidos en el sistema.

En paralelo, el *generator_out* controla el momento en el que se producen extracciones del módulo IP mediante la activación de la señal *remove*. Las salidas del DUV (*rank_out*, *meta_out*, *valid_out*, etc.) son observadas por el componente *out_monitor*, que reconstruye las respuestas en forma de transacciones (*transaction_out*) y las envía al componente *scoreboard*.

El componente *scoreboard* compara cada salida observada con el resultado esperado, en función de las operaciones de entrada previamente registradas. Esto permite verificar no solo la validez de los datos entregados, sino también que el módulo cumpla con las condiciones de prioridad esperadas.

Todo el sistema se encuentra coordinado desde el código descrito en el archivo *testbench.sv*, que actúa como componente *testbench* principal. Es aquí se instancian todos los componentes y se conectan mediante el módulo *environment*, que agrupa los caminos de entrada y salida y permite su ejecución conjunta en un entorno coherente.

3.4.2. Implementación de componentes clave

A continuación, se analizan algunos fragmentos de código representativos del *testbench* funcional desarrollado, con el objetivo de ilustrar cómo se han implementado los distintos bloques que forman parte del entorno de verificación. Se ha priorizado mostrar aquellos elementos que presentan mayor interés técnico, ya sea por su lógica, su sincronización con el reloj, o por su papel en la validación de los datos generados o recibidos. El código completo del entorno de test desarrollados se adjunta en el anexo que acompaña a la memoria de este TFG.

- Driver de entrada:

El módulo *in_driver* tiene como función principal recibir transacciones desde el generador de entrada (*generator_in*) y traducirlas en señales físicas que se aplican al DUV a través de la interfaz virtual *in_interface*. Además de conducir señales, este componente implementa un sistema de control del *reset*, así como un mecanismo opcional para aplicar

retardos aleatorios entre transacciones, lo que permite simular condiciones temporales más realistas.

La primera función destacada del *driver* es la tarea *reset*, que prepara el sistema antes del inicio de la simulación funcional, tal y como se muestra en la Figura 7.

```
1  task reset;
2    wait(in_vif.rst);
3    $display("[IN DRIVER] ----- Reset Started -----");
4    // Reinicia las señales de la interfaz
5    `DRIV_IF.insert <= 0;
6    `DRIV_IF.rank_in <= '0;
7    `DRIV_IF.meta_in <= '0;
8    wait(!in_vif.rst);
9    $display("[IN DRIVER] ----- Reset Ended   -----");
10 endtask
```

Figura 7: Tarea reset perteneciente al driver de entrada

En esta tarea se espera a que se active la señal de *reset (in_vif.rst)* para iniciar la inicialización de todas las señales de entrada. Esta lógica sincroniza el arranque del *testbench* con la señal de *reset*, poniendo a cero todas las señales de entrada. El uso del macro *DRIV_IF* permite acceder a los *clocking blocks* definidos en la interfaz, garantizando que las asignaciones se realicen de forma sincronizada y evitando condiciones de carrera. Los *clocking blocks* son un mecanismo que permite definir el sincronismo y el comportamiento de las señales asociadas a los puertos de E/S con respecto a la señal de reloj de la que dependen. Esto es muy útil en la definición de un *testbench*.

La parte más importante del módulo reside en la tarea *main*, que se ejecuta de forma indefinida durante toda la simulación. El código de dicha tarea se muestra en la Figura 8.

```
task main;
  forever begin
    // Crea una instancia de la transacción
    transaction_in trans;
    // Obtiene la transacción del buzón
    gen2driv_in.get(trans);
    // Comprobar si la transacción viene con delay
    if(trans.delay) begin
      int unsigned delay;
      $display("[IN-DRV-DELAY] At time %d get trans from mailbox", $time);
      //Generar delay
      delay = $urandom_range(5); //Genera numeros aleatorios entre 0-5
      $display("[DELAY]=> %d", delay);
      // Espera al flanco de subida del reloj
      repeat(delay) @(posedge in_vif.in_DRIVER.clk);
      $display("[TIEMPO] %d", $time);
      // Establece la señal de validación en 1
      `DRIV_IF.insert <= 1;
      // Asigna los valores de la transacción a las señales de la interfaz
      `DRIV_IF.rank_in <= trans.rank_in;
      `DRIV_IF.meta_in <= trans.meta_in;
      // Espera al siguiente flanco de subida del reloj
      @(posedge in_vif.in_DRIVER.clk);
      // Reinicia la señal de inserción
      `DRIV_IF.insert <= 0;
      // Incrementa el contador de transacciones
      no_transactions++;
      $display("[IN-DRV-DELAY] At time %d no_transactions %d", $time, no_transactions);
    end
    else begin
      $display("[IN-DRV] At time %d get trans from mailbox", $time);
      // Espera al flanco de subida del reloj
      @(posedge in_vif.in_DRIVER.clk);
      // Establece la señal de validación en 1
      `DRIV_IF.insert <= 1;
      // Asigna los valores de la transacción a las señales de la interfaz
      `DRIV_IF.rank_in <= trans.rank_in;
      `DRIV_IF.meta_in <= trans.meta_in;
      // Espera al siguiente flanco de subida del reloj
      @(posedge in_vif.in_DRIVER.clk);
      // Reinicia la señal de inserción
      `DRIV_IF.insert <= 0;
      // Incrementa el contador de transacciones
      no_transactions++;
      $display("[IN-DRV] At time %d no_transactions %d", $time, no_transactions);
    end
  end
endtask
```

Figura 8: Tarea main perteneciente al driver de entrada

Dentro de un bucle infinito, se declara una referencia de tipo transacción, denominada *trans*, y espera la llegada de una transacción desde el generador mediante un objeto *mailbox*. En la línea 6 se muestra como haciendo uso del método *get* del mailbox *gen2driv_in*, este componente queda a ala espera de recibir una transacción. Posteriormente evalúa si la transacción requiere aplicar un retardo antes de ser conducida. En caso afirmativo, se introduce un *delay* aleatorio de entre 0 y 5 ciclos de reloj mediante una espera activa controlada por *repeat*, con el objetivo de retardar intencionadamente las inserciones, simulando así un entorno más realista y asíncrono. Una vez vencido el posible retardo, o si no existe, el componente *driver* se sincroniza, esperando un flanco de subida del reloj, y aplica la transacción activando la señal *insert* junto con los datos *rank_in y meta_in*. Esta activación se mantiene durante un único ciclo de reloj, tras el cual la señal de *insert* se desactiva para evitar duplicaciones o lecturas no deseadas por parte del DUV. Finalmente, se incrementa un contador de transacciones y se imprime un mensaje por consola con el *timestamp* actual y el número de operaciones realizadas, lo que permite un seguimiento preciso del comportamiento del *driver* durante la simulación. Esta implementación, además de estar completamente sincronizada con el reloj del sistema, incluye mecanismos de trazabilidad y control de flujo, y proporciona al IP un patrón de estímulo representativo de condiciones reales de operación.

- Monitor de entrada:

El componente *in_monitor* de la Figura 6 desempeña una función clave en el proceso de verificación funcional: observar las señales de entrada que se están aplicando al DUV y capturarlas de forma no intrusiva para su análisis posterior. En este contexto, su misión es detectar cuándo se produce una operación de inserción válida (*insert* = 1) y registrar los datos asociados (*rank_in y meta_in*) que fueron aplicados, enviándolos al *scoreboard* para ser comparados con las salidas correspondientes.

La lógica principal del monitor se implementa dentro de la tarea *main*, cuyo código completo se muestra en el Figura 9.

```
task main;
 forever begin
   transaction_in trans;
   trans = new();

   @(posedge in_vif.in_MONITOR.clk);
   wait(`MON_IF.insert);
     trans.rank_in = `MON_IF.rank_in;
     trans.meta_in = `MON_IF.meta_in;
     $display("[IN MONITOR] At time %d Put trans", $time);
     trans.display();
     in_mon2scb.put(trans);
     end
   endtask
```

Figura 9: Tarea main perteneciente al módulo de entrda

Cada vez que se detecta una señal de *insert* sincronizado tras un flanco positivo del reloj, el monitor crea una nueva transacción, copia los datos *rank_in y meta_in* observados en la interfaz, y los envía a través del buzón *in_mon2scb* al *scoreboard*. Esta comunicación se realiza a través del método *put* del mailbox utilizado. También imprime en consola el contenido de la transacción, lo cual facilita el seguimiento del proceso durante la simulación.

Una observación importante es que el monitor no comprueba si el módulo está lleno (full) antes de registrar la transacción. Esta decisión, expresamente planteada en un comentario del propio código, responde a una filosofía correcta en verificación funcional: el monitor debe limitarse a observar, no a decidir. Su rol no es validar si la operación fue aceptada por el DUV, sino registrar la intención de inserción. Será el *scoreboard* quien, tras recibir las transacciones desde el monitor y comparar con las salidas reales del DUV, determine si el módulo actuó correctamente. De este modo se conserva la separación de responsabilidades entre observación pasiva (*monitor*) y validación activa (*scoreboard*), lo que favorece la claridad, escalabilidad y mantenibilidad del entorno de verificación.

- Interfaz de entrada:

La interfaz *in_interface* representa uno de los elementos estructurales más importantes del entorno de verificación, ya que actúa como el punto de conexión entre el *testbench*

funcional y el módulo bajo prueba *pifo_reg*. Su función principal es encapsular las señales de entrada asociadas a las operaciones de inserción, y proporcionar un mecanismo de acceso sincronizado tanto para los drivers como para los monitores mediante bloques de temporización *clocking blocks* y *modports*. Los modports es un mecanismo de SystemVerilog que permite especificar el acceso a las señales definidas en una interfaz desde diferentes puntos de vista, como puede ser, desde el punto de vista del DUV o del testbench.

Este diseño modular y parametrizable permite desacoplar la lógica del *testbench* del DUV, mejorando la claridad del código y facilitando su reutilización. A continuación, se resumen las principales características y decisiones de diseño implementadas en la interfaz. La Figura 10 muestra un pequeño fragmento de la definición de la interfaz *in_interface*.

```
1  interface in_interface
2      (
3        input bit                clk,
4        input bit         rst
5      );
6
7      // Insertion interface
8      logic                      full;
9      logic                      insert;
10     logic [RANK_WIDTH-1:0]     rank_in;
11     logic [META_WIDTH-1:0]     meta_in;
```

Figura 10: Señales pertenecientes a la interfaz de entrada

En esta interfaz se definen las señales para gestionar el proceso de inserción de datos en el DUV. La señal *insert* actúa como un pulso de control que habilita la operación de escritura, indicando al DUV que debe capturar los datos presentes en sus entradas. Por su parte, *rank_in* y *meta_in* transportan la información asociada a cada elemento: una prioridad y sus metadatos respectivos. Finalmente, la señal full, generada por el propio sistema, informa al entorno de verificación de que el registro ha alcanzado su capacidad máxima, impidiendo nuevas inserciones hasta que se libere espacio.

El uso de definir las señales usando como tipo *logic* en lugar de *wire* o *reg* responde a las convenciones modernas de *SystemVerilog*, y permite mayor flexibilidad de uso dentro de

módulos de prueba y simulación. Una de las partes más sofisticadas de esta interfaz es el uso de *clocking blocks*, que tal y como se comentó, permiten sincronizar correctamente las operaciones de entrada y observación con el reloj del sistema. Esta técnica es fundamental para evitar condiciones de carrera entre el *testbench* y el DUV. La Figura 11 muestra un ejemplo de *clocking block* para el interfaz de entrada.

```
1  //Driver clocking block
2    clocking in_driver_cb @(posedge clk);
3      default input #1 output #1;
4      input full;
5      output insert;
6      output rank_in;
7      output meta_in;
8    endclocking
```

Figura 11: Driver Clocking Block en el Interfaz de entrada

Este bloque está asociado al módulo *driver*, y define que todas las señales se actualizarán o leerán en el flanco positivo del reloj, con un retardo de una unidad de tiempo (#1) por defecto. Esto garantiza que las señales se apliquen en el momento apropiado y sean consistentes con el comportamiento del DUV. Por su parte, el *clocking block* del *monitor* se define tal y como muestra la Figura 12.

```
1  //Monitor clocking block
2  clocking in_monitor_cb @(posedge clk);
3    default input #1 output #1;
4    input insert;
5    input full;
6    input rank_in;
7    input meta_in;
8  endclocking
```

Figura 12: Monitor Clocking Block en el Monitor de entrada

Este bloque asegura que el monitor lea las señales también en el mismo instante del ciclo de reloj, permitiendo capturar exactamente los valores que se aplicaron, sin interferencias

ni lecturas fuera de fase. Finalmente, se declaran los *modports*, que definen, tal y como se comentó, los roles de acceso de cada componente del *testbench*. La Figura 13 muestra el código de un *modport* para el interfaz de entrada.

```
1  //driver modport
2  modport in_DRIVER  (clocking in_driver_cb,input clk, rst);
3
4  //monitor modport
5  modport in_MONITOR (clocking in_monitor_cb,input clk, rst);
```

Figura 13: Modports pertenecientes a la interfaz de entrada

El uso de interfaces con *clocking blocks* y *modports* en este entorno funcional permite una sincronización precisa entre el DUV y el *testbench*.

- Scoreboard:

El *scoreboard* constituye el corazón de la verificación funcional en este entorno, ya que su función es comprobar si el comportamiento observado del DUT coincide con lo que se espera según el modelo funcional. En este caso, no se limita a comparar valores de forma directa, sino que reconstruye el funcionamiento interno de una cola PIFO, como se puede observar en la Figura X, manteniendo una representación paralela de los datos insertados y su orden de prioridad. De este modo, puede anticipar con precisión cuál debería ser el valor de salida en cada operación de eliminación, validando si el módulo opera de forma correcta.

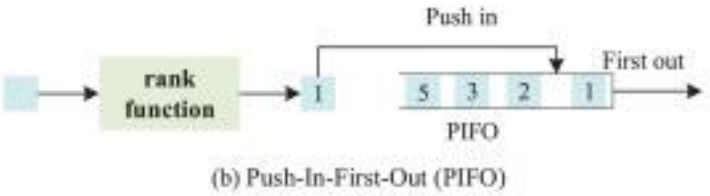

Figura 14: Esquema de funcionamiento sistema PIFO[16]

El componente se comunica con el entorno a través de dos buzones, *mailbox*. El primero que recibe transacciones desde el monitor de entrada (*in_mon2scb*) y un segundo *mailbox,* que recibe las salidas capturadas por el monitor de salida (*out_mon2scb*). Además, mantiene dos colas (*queues*) para modelar el comportamiento de la PIFO: *pifo_rank* y

pifo_meta, que almacenan respectivamente los valores de prioridad y metadatos de cada elemento.

```
1  logic [META_WIDTH-1:0] pifo_meta [$];
2  logic [RANK_WIDTH-1:0] pifo_rank [$];
3
4  //used to count the number of transactions
5  logic [META_WIDTH-1:0] pifo_meta_pop;
6  logic [RANK_WIDTH-1:0] pifo_rank_pop;
```

Figura 15: Variables auxiliares del Scoreboard

Estas colas permiten simular el contenido interno del IP en cada momento de la simulación. Mediante el uso del mecanismo *fork_join* de SystemVerilog, se puede simultanear la atención de ambas colas al mismo tiempo. Para ello se declaran dos ramas (*threads*) dentro de la tarea *main* de este módulo. Cada vez que se recibe una transacción desde el monitor de entrada, el *scoreboard* la inserta en el lugar correcto dentro de la estructura, respetando el orden por prioridad (*rank*). Este comportamiento se reproduce en la primera rama del *fork* dentro de la tarea *main*. Este proceso de inserción se muestra el código de la Figura 16.

```
1    if((pifo_rank.size() == 0) || (pifo_rank[0] > trans_in.rank_in)) begin
2      pifo_rank.push_front(trans_in.rank_in);
3      pifo_meta.push_front(trans_in.meta_in);
4      $display("\tPush rank[%d] = %p", pifo_rank.size(), pifo_rank);
5      $display("\tPush meta[%d] = %p", pifo_meta.size(), pifo_meta);
6    end
7    else begin
8      foreach(pifo_rank[i]) begin
9        if(trans_in.rank_in < pifo_rank[i]) begin
10         pifo_rank.insert(i, trans_in.rank_in);
11         pifo_meta.insert(i, trans_in.meta_in);
12         break;
13       end
14     end
15     $display("\tInsert rank[%d] = %p", pifo_rank.size(), pifo_rank);
16     $display("\tInsert meta[%d] = %p", pifo_meta.size(), pifo_meta);
17   end
18 end
```

Figura 16: Lógica de inserción de datos perteneciente al Scoreboard

Aquí se implementa una lógica que mantiene las colas ordenadas según la prioridad del elemento (*rank_in*). Si el valor recibido es más prioritario que todos los existentes, se coloca al frente, líneas 1,2 y 3. De lo contrario, se busca la posición correspondiente y se inserta allí, bucle *foreach* del código anterior. Esta estructura asegura que el primer elemento de las colas siempre represente al dato con mayor prioridad, es decir, el que debería salir primero del *pifo_reg*. Esta lógica convierte al *scoreboard* en un modelo funcional del DUV, que opera en paralelo, pero sin contacto directo, sirviendo como referencia para la validación de su correcto funcionamiento.

La verificación de las salidas se realiza en paralelo a la recepción de inserciones, mediante una segunda rama del *fork* dentro de la tarea *main*. En esta sección, el *scoreboard* espera recibir una transacción del *out_monitor*, que representa el valor realmente entregado por el DUV durante una operación de extracción *remove*. Una vez recibida, el *scoreboard* compara los valores de salida, *rank_out* y *meta_out*, con el primer elemento de sus colas internas, que representa el valor esperado según la política de prioridad mínima. Para evitar errores o comparaciones inconsistentes, el elemento esperado se extrae previamente de la cola correspondiente mediante *pop_front()* y se guarda en variables temporales denominadas, en este caso, *exp_rank* y *exp_meta*. A continuación, se realiza la comparación con los valores observados. Este procedimiento se describe en el código de la Figura 17.

```
1    // Output transaction
2    transaction_out trans_out;
3    trans_out = new();
4    out_mon2scb.get(trans_out);
5    //Comparacion SCB
6    exp_rank = pifo_rank.pop_front();
7    exp_meta = pifo_meta.pop_front();
8
9    if (exp_meta == trans_out.meta_out && exp_rank == trans_out.rank_out) begin
10     $display("[SCB-PASS] meta_out expected: %0h, actual: %0h", exp_meta, trans_out.meta_out);
11     $display("[SCB-PASS] rank_out expected: %0h, actual: %0h", exp_rank, trans_out.rank_out);
12   end else begin
13     $display("[SCB-FAIL] meta_out expected: %0h, actual: %0h", exp_meta, trans_out.meta_out);
14     $display("[SCB-FAIL] rank_out expected: %0h, actual: %0h", exp_rank, trans_out.rank_out);
15   end
16   no_transactions++;
17 end
```

Figura 17: Lógica de extracción de datos perteneciente al Scoreboard

La extracción de los valores se realiza en las líneas 6 y 7 y la comparación en la línea 9. Si los valores coinciden, se imprime un mensaje [SCB-PASS], confirmando que el DUV ha entregado el elemento correcto en ese ciclo. En caso contrario, se muestra un mensaje [SCB-FAIL] detallando la discrepancia. Esta estructura asegura que la comparación sea precisa, sincronizada y trazable, ya que el uso de variables temporales evita el consumo prematuro de elementos en las colas y garantiza que los datos mostrados en consola correspondan con exactitud a lo que se ha evaluado. Además, se incrementa un contador de transacciones verificadas, lo que permite llevar un seguimiento del progreso del *test*.

3.4.3. Comportamiento del *test*

El comportamiento del *test* se basa en una arquitectura jerárquica formada por tres niveles funcionales: el módulo *testbench*, que actúa como contenedor físico del DUV y de las interfaces; el módulo *environment*, que organiza y coordina los agentes de verificación y el *scoreboard*; y el bloque *random_test*, que inicializa el entorno y lanza la simulación. Esta estructura modular permite una separación clara de responsabilidades, facilita el mantenimiento del entorno y proporciona flexibilidad para definir distintos escenarios de prueba. La Figura 18 muestra parte del código del módulo *testbench*.

```
module tbench_top;
  // Declaración de señales de reloj y reset
  bit clk = 0;
  bit reset;

  // Generación del reloj
  always #5 clk = ~clk;

  // Generación del reset
  initial begin
    reset = 1;
    #35 reset = 0;
  end

  // Creación de instancia de la interfaz para conectar DUT y testbench
  in_interface in_intf(clk, reset);
  out_interface out_intf(clk, reset);

  // Instancia de las interfaces, y se pasa el manejador
  //de cada interfaz al test como argumento
  test t1(in_intf, out_intf);

  // Instancia del DUT, se conectan las señales de la
  //interfaz a los puertos del DUT

  pifo_reg DUT (
    .clk(clk),
    .rst(reset),
    .full(in_intf.full),
    .insert(in_intf.insert),
    .rank_in(in_intf.rank_in),
    .meta_in(in_intf.meta_in),
    .valid_out(out_intf.valid_out),
    .remove(out_intf.remove),
    .rank_out(out_intf.rank_out),
    .meta_out(out_intf.meta_out),
    .max_valid_out(out_intf.max_valid_out),
    .max_rank_out(out_intf.max_rank_out),
    .max_meta_out(out_intf.max_meta_out),
    .num_entries(out_intf.num_entries),
    .empty(out_intf.empty)
   );

```

Figura 18: Testbench

- Inicio del test:

En el módulo *testbench* se define la infraestructura básica para la ejecución de la simulación. Además, se generan las señales de reloj (*clk*) y de *reset*. El *reset* se mantiene activo durante los primeros 35 ns, asegurando así que tanto el DUV como los módulos del *testbench* arranquen desde un estado conocido.

Posteriormente, se referencian las dos interfaces: in_*interface* y out_*interface*, conectadas tanto al DUV como al bloque test, que contiene el programa *random*_test. Esta conexión garantiza que el test tenga acceso directo a las señales del DUV a través de los *modports* definidos en las interfaces. El DUV, se conecta explícitamente a través de todas sus señales clave, incluyendo *insert, remove, rank_in, meta_in, valid_out, rank_out, meta_out, max_valid_out, empty, full*, entre otras.

- Ejecución del programa random_test:

Constituye el punto de entrada del *test* funcional. Dentro del bloque *initial*, se instancia el objeto *environment* pasando como argumentos las interfaces de entrada y salida. Posteriormente, se ejecuta el método *env.run()*, que es el responsable de lanzar todas las tareas paralelas del entorno. La lógica interna del *environment* define patrones fijos a través de listas *in_test* y *out_test* que determinan cuántas inserciones y extracciones se realizarán. La Figura 19 muestra el código del módulo test.

```
1  `include "environment.sv"
2  program test(in_interface in_intf, out_interface out_intf);
3
4    //declaring environment instance
5    environment env;
6
7    initial begin
8      //creating environment
9      env = new(in_intf, out_intf);
10     env.run();
11   end
12 endprogram
```

Figura 19: Random test

- Coordinación en el entorno:

El módulo *environment* es el núcleo operativo del *test*. Se encarga de instanciar los agentes de entrada *in_agent* y salida *out_agent*, así como el componente *scoreboard*. Internamente, define dos colas (*queues*) llamadas *in_test* y *out_test*, que controlan el número de transacciones que se generan y conducen durante la simulación. En el código actual, estas listas están inicializadas con dos valores cada una, por lo que se espera que se realicen dos operaciones de inserción y dos de eliminación, ejecutadas en el orden establecido por los generadores. Como se puede observar en la Figura 20.

La tarea *run* del entorno ejecuta en paralelo los métodos ***run()*** de ambos agentes. Esta ejecución concurrente permite que se apliquen operaciones de inserción y extracción en el mismo periodo de simulación, lo cual es fundamental para verificar la robustez del DUV frente a estímulos simultáneos o solapados. Cada agente, a su vez, lanza su generador, *driver* y monitor asociados.

```
1  class environment;
2    //Test
3    logic [2:0] delay;
4    int in_test[$] = {3,1};
5    int out_test[$] = {2,1};
6
7    //generator and driver instance
8    in_agent      in_age;
9    out_agent     out_age;
10   scoreboard scb;
11
12   //virtual interface
13   virtual in_interface in_vif;
14   virtual out_interface out_vif;
```

Figura 20: Clase Environment-Test e Instanciaciones

3.4.4. Validación del resultado de la simulación

Tras ejecutar el entorno funcional con la configuración definida por las listas `in_test = {3,1}` y `out_test = {2,1}`, se ha generado una simulación compuesta por un total de cuatro operaciones de inserción y tres operaciones de extracción. La figura adjunta recoge la traza temporal del comportamiento del DUV durante la ejecución del *test*, permitiendo verificar con claridad que el módulo se comporta conforme al modelo funcional definido en el *scoreboard*.

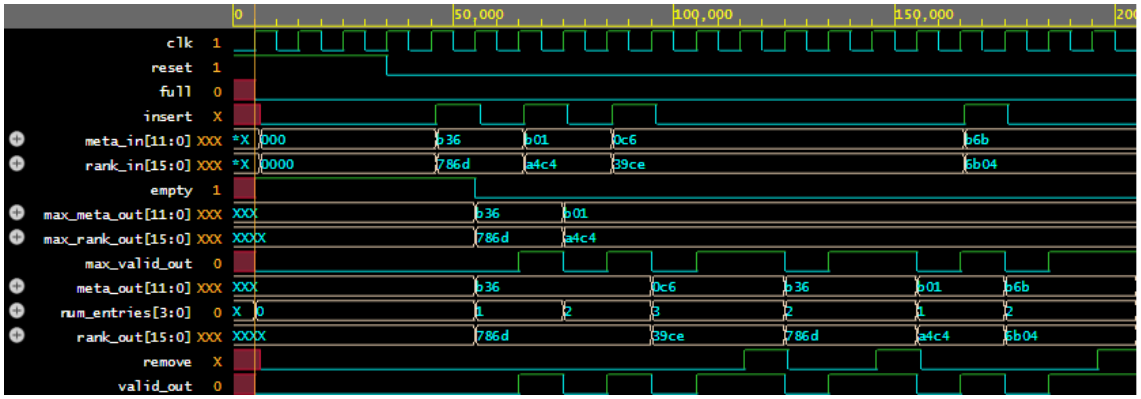

Figura 21: Formas de onda resultado de la ejecución de los test

En un primer bloque de inserciones, se observa cómo se introducen tres datos consecutivos en el DUV, activando la señal *insert* y aplicando los valores *rank_in* y *meta_in* correspondientes. Con cada inserción, la señal *num_entries* se incrementa correctamente de 0 a 3 y la señal *empty* pasa de 1 a 0 tras la primera operación, indicando que el registro deja de estar vacío. A medida que se acumulan elementos, las salidas *rank_out* y *meta_out* reflejan siempre el elemento de mayor prioridad almacenado, es decir, el de menor valor de *rank_in*. Esta información se actualiza automáticamente por el DUV tras cada inserción, sin necesidad de realizar extracciones.

A continuación, se produce una secuencia de dos operaciones de *remove*, reflejadas en la activación de la señal *remove* y la generación de un pulso en *valid_out*. En ambos casos, las señales *rank_out* y *meta_out* reflejan correctamente el contenido del elemento más prioritario que estaba en la cola, y *num_entries* se decrementa en consecuencia. Esta secuencia

confirma que la lógica de extracción funciona correctamente y que el DUV mantiene internamente el orden de los datos según prioridad. Posteriormente, se realiza una cuarta inserción aislada, que incrementa de nuevo el número de entradas. Esta acción también actualiza las señales de salida de forma coherente, incluyendo *max_rank_out*, *meta_out*, y *valid_out*, reforzando así la validez de los cálculos internos del DUV.

Finalmente, tras la última operación de eliminación, el DUV reduce su número de entradas, reflejado en la señal *num_entries* decrece, pero no alcanza el estado de cola vacía, ya que se observa que la señal *empty* permanece en 0. Esto indica que aún queda al menos un dato almacenado internamente. Además, la señal *valid_out* continúa activa, reflejando que sigue existiendo un valor de prioridad mínimo disponible para extracción, el cual se mantiene actualizado en las salidas *rank_out* y *meta_out*. Este comportamiento confirma que el DUV gestiona correctamente la coexistencia de inserciones y extracciones intercaladas, manteniendo la integridad del contenido de la cola y el orden de prioridad, incluso bajo condiciones dinámicas. La Figura 22 muestra la traza de las operaciones realizadas por el sistema.

```
[IN MONITOR] At time 95 Put trans
       Transaction: insert=0, rank_in=39ce, meta_in=0c6
[IN-DRV] At time                 95 no_transactions          3
       Push rank[          2] = '{'h39ce, 'h786d}
       Push meta[          2] = '{'hc6, 'hb36}
[IN-AGENT] At time                  95 gen.repeat_count          3 driv.no_transactions          3
[IN-AGENT] At time                  95 post_test finished
[OUT-AGENT] At time                 95 test finished
[OUT-DRV] At time                 115 valid_out detected
[OUT-DRV] At time                 115 remove asserted
Transaction: valid_out=0, remove=1, rank_out=0000, meta_out=000, max_valid_out=0, max_rank_out=0000, max_meta_out=000, num_entries=0, empty=0
[SCB-PASS] meta_out expected: c6, actual: c6
[SCB-PASS] rank_out expected: 39ce, actual: 39ce
[OUT-DRV] At time                 145 valid_out detected
[OUT-DRV] At time                 145 remove asserted
Transaction: valid_out=0, remove=0, rank_out=0000, meta_out=000, max_valid_out=0, max_rank_out=0000, max_meta_out=000, num_entries=0, empty=0
[OUT-AGENT] At time                 155 gen.repeat_count          2 driv.no_transactions          2
[OUT-AGENT] At time                 155 post_test finished
```

Figura 22: Log generado por el Scoreboard

3.4.5. Observaciones finales

El desarrollo del *testbench* funcional ha permitido verificar el comportamiento del módulo, confirmando que el entorno implementado ofrece una solución completa, trazable y robusta para evaluar operaciones de inserción, extracción y gestión de prioridad. La arquitectura modular, basada en interfaces sincronizadas, agentes y un *scoreboard* funcional,

ha demostrado ser eficaz para detectar discrepancias entre el comportamiento esperado y el real del DUV.

Uno de los principales logros ha sido la correcta implementación del *scoreboard*, cuya lógica interna simula una estructura PIFO y verifica que los elementos extraídos por el DUV respeten el criterio de menor *rank*. Este componente es para detectar errores en el orden de extracción durante las fases de depuración, y su comparación estructurada ha garantizado la validez de los resultados.

Otro aspecto destacable ha sido la trazabilidad obtenida gracias a los *displays* en consola, los cuales han proporcionado información detallada sobre cada transacción, facilitando el análisis de fallos y la validación del sistema. Además, la separación entre estimulación, observación y comparación ha reforzado la mantenibilidad del entorno, permitiendo una detección precisa de errores sin interferir en el flujo de datos del DUV.

No obstante, durante el desarrollo se identificó un error en la comparación original del *scoreboard*, en la que se realizaban múltiples operaciones `pop_front()` consecutivos sobre las colas internas, lo que provocaba desincronizaciones y resultados incorrectos. Esta situación fue corregida mediante el uso de variables intermedias, lo que restableció la coherencia entre la evaluación y la trazabilidad de los datos esperados y observados. Por último, se destaca que el entorno funcional, aún sin basarse en UVM, ha logrado replicar muchas de sus buenas prácticas: modularidad, reutilización, control sincronizado y separación de responsabilidades. Esto no solo valida su diseño, sino que lo convierte en una base sólida para su posible evolución hacia metodologías de verificación más avanzadas.

4º Capítulo: Lenguajes de verificación formal

En el proceso de diseño y validación de sistemas electrónicos digitales, la verificación juega un papel crítico. Tradicionalmente, esta verificación se ha abordado mediante *testbench* funcionales, donde se estimula el diseño bajo verificación DUV con distintos patrones de entrada y se observan sus salidas. Si bien esta metodología ha demostrado ser eficaz en múltiples contextos, presenta limitaciones: la cobertura funcional es parcial, la generación de casos de prueba puede no ser exhaustiva, y ciertos errores pueden permanecer ocultos si no se dan las condiciones exactas que los activan.

Frente a este enfoque, surge la verificación formal, una técnica que permite comprobar automáticamente si un sistema cumple ciertas propiedades, sin necesidad de aplicar estímulos específicos ni recorrer manualmente el espacio de estados. En lugar de verificar comportamientos individuales, la verificación formal evalúa reglas lógicas que el sistema debe cumplir en todos los casos posibles, lo que la convierte en una herramienta para detectar errores lógicos y asegurar el cumplimiento de requisitos críticos.

Para aplicar verificación formal, se utilizan lenguajes específicos basados en propiedades, que permiten expresar de forma precisa y estructurada las condiciones que deben cumplirse en el funcionamiento del sistema. Estos lenguajes, como *SystemVerilog Assertions* (SVA), permiten definir temporalmente relaciones entre señales, eventos esperados, condiciones de activación y consecuencias, entre otros aspectos.

A lo largo de este capítulo se presentarán los fundamentos de la verificación formal, se analizarán los lenguajes de especificación de propiedades más utilizados, y se evaluarán tanto las ventajas que ofrecen como los desafíos que presentan al integrarse en entornos reales de verificación.

4.1. Fundamentos de verificación formal

La verificación formal es una técnica que emplea métodos matemáticos para demostrar que un diseño hardware cumple una especificación definida de forma precisa.

A diferencia de la verificación funcional tradicional basada en simulaciones, donde se aplican estímulos concretos para observar respuestas, la verificación formal analiza exhaustivamente todos los comportamientos posibles del sistema, garantizando que ciertas propiedades lógicas se cumplan bajo cualquier condición de entrada o estado interno.

En general, se distinguen dos enfoques principales en la verificación formal: *model checking* y demostración de teoremas. Ambos comparten el objetivo de validar la corrección de un sistema frente a una especificación, pero difieren significativamente en su metodología.

- Model checking: es una técnica automática que comprueba si un modelo finito de un sistema satisface una propiedad expresada en lógica temporal, como LTL (*Linear Temporal Logic*) o CTL (*Computation Tree Logic*). El verificador recorre sistemáticamente el espacio de estados del modelo, evaluando si la propiedad se cumple en todos los posibles escenarios. En caso de encontrar una violación, se genera un contraejemplo útil para el ingeniero. Esta técnica es especialmente potente en diseños hardware debido a su capacidad de detectar errores complejos sin necesidad de generar estímulos externos.
- La demostración de teoremas: Se basa en métodos deductivos. Requiere que tanto el diseño como las propiedades se expresen en un lenguaje lógico, y emplea reglas de inferencia y axiomas para probar que una propiedad se cumple. Este enfoque es más general y puede aplicarse a sistemas infinitos o altamente abstractos, pero suele requerir intervención manual, experiencia en lógica matemática y un mayor esfuerzo de formalización.

Ambos métodos tienen sus ventajas. El *model checking* ofrece automatización y retroalimentación rápida con contraejemplos precisos, mientras que la demostración de teoremas aporta flexibilidad y mayor capacidad de abstracción. En la práctica, muchas metodologías de verificación industrial combinan ambos enfoques para obtener un análisis más completo del sistema [17].

4.2. Ventajas del uso de lenguajes de verificación formal

El uso de lenguajes de verificación formal basados en propiedades, como *SystemVerilog Assertions* (SVA), aporta múltiples ventajas que complementan los enfoques de verificación funcional tradicionales. Estas ventajas se manifiestan tanto en fases tempranas del diseño como en etapas avanzadas de validación, contribuyendo a mejorar la calidad, eficiencia y trazabilidad del proceso de verificación.

- Detección temprana de errores: Una de las principales fortalezas de las aserciones es su capacidad para detectar errores en etapas iniciales del desarrollo. Al definir condiciones que deben cumplirse de forma obligatoria durante la ejecución del sistema, cualquier violación se reporta de manera inmediata y precisa. Esto permite localizar errores lógicos que podrían permanecer ocultos en una simulación tradicional si no se generaran estímulos específicos que los activaran. Además, las herramientas de verificación formal pueden utilizar estas aserciones como objetivos a comprobar exhaustivamente, lo que acelera la depuración y reduce el coste de las correcciones.

- Automatización y simplificación del análisis: Las aserciones introducen un mecanismo automático para comprobar el cumplimiento de requisitos funcionales sin necesidad de analizar manualmente las formas de onda o recorrer todos los casos de prueba. Una vez activadas, actúan como observadores permanentes del diseño, generando mensajes detallados en caso de incumplimiento. Esto reduce la dependencia del ingeniero en tareas repetitivas de análisis y permite dedicar más tiempo a tareas de valor añadido, como la optimización del diseño o la mejora de cobertura.

- Mejora de la cobertura funcional y formal: El uso de aserciones permite definir explícitamente condiciones que podrían no quedar cubiertas por los *tests* funcionales, especialmente aquellas asociadas a casos límite, errores asincrónicos o interacciones temporales complejas. Además, es posible hacer uso de las aserciones para medir qué propiedades han sido activadas durante la simulación, lo que proporciona una métrica objetiva de cobertura a nivel de especificación. En entornos formales, estas mismas propiedades pueden utilizarse para comprobar exhaustivamente el cumplimiento de

reglas en todos los caminos posibles del sistema, algo prácticamente inalcanzable con verificación funcional convencional.

- Integración con herramientas de verificación formal: *SystemVerilog Assertions* está ampliamente soportado por las principales herramientas EDA, tanto en simulación como en verificación formal (*model checking*). Esto permite reutilizar el mismo conjunto de propiedades en ambos entornos, reduciendo la duplicidad de trabajo y asegurando coherencia entre la verificación funcional y la formal. Las herramientas formales utilizan estas propiedades como objetivos de prueba, y pueden demostrar su validez matemática o generar contraejemplos en caso de violación.

- Claridad en la especificación: Otra ventaja fundamental de los lenguajes basados en propiedades es que sirven como documentación ejecutable del comportamiento esperado del sistema. Las propiedades definen de forma clara y estructurada lo que se espera que ocurra en determinadas condiciones, lo cual mejora la comprensión del diseño por parte de todo el equipo y reduce la ambigüedad en la interpretación de los requisitos. Esta formalización favorece la mantenibilidad del código y facilita la incorporación de nuevos miembros al equipo de verificación.

4.3. Fundamentos técnicos de SystemVerilog *Assertions*

Tras haber analizado el rol general de las *SystemVerilog Assertions* (SVA) en la verificación funcional y formal, en este apartado se presenta un estudio más profundo de su estructura y funcionamiento interno. El objetivo es entender cómo se construyen, cómo operan a nivel de simulación y verificación formal, y qué capacidades ofrece el lenguaje para controlar comportamientos temporales complejos.

Esto se logra gracias a una sintaxis estructurada basada en bloques tipo *sequence* y *property*, operadores temporales avanzados, mecanismos de control de evaluación y funciones del sistema.

A lo largo de las siguientes secciones se describen los bloques fundamentales de una aserción, sus operadores temporales, patrones de uso habituales y técnicas para estructurar propiedades reutilizables y robustas.

4.3.1. Estructura interna de una SVA

Las *SystemVerilog Assertions* (SVA) proporcionan una forma estructurada de capturar y verificar el comportamiento esperado del hardware mediante construcciones formales basadas en lógica temporal. Internamente, una SVA se compone principalmente de dos bloques fundamentales: *sequence* y *property*, que se combinan para construir expresiones verificables a lo largo del tiempo.

- Bloque sequence: El bloque *sequence* se utiliza para describir una serie ordenada de eventos o condiciones que deben cumplirse con una relación temporal concreta. Estas secuencias son esenciales para expresar protocolos, *handshakes*, y restricciones que no pueden capturarse fácilmente con condiciones puntuales. Un ejemplo básico de secuencia es:

```
1  sequence s1;
2      a ##1 b;
3  endsequence
```

Figura 23: Estructura básica de una secuencia SVA

En esta secuencia, denominada S1, se espera que la señal b ocurra exactamente un ciclo después de que a sea verdadera. El operador ## representa un retraso entre eventos, y puede usarse para modelar ventanas temporales y relaciones causales.

Las secuencias también pueden incorporar, repeticiones usando el formato [N], [*N] o [*N:M] para exigir que una condición ocurra N veces consecutivas o dentro de un rango temporal. Operadores de concatenación (##), intercalación (*and, intersect, within*), y elección (*or*). Tambien puede hacer uso de eventos de activación mediante el uso del simbolo @, para indicar el reloj bajo el que se evaluará la secuencia.

- Bloque property: Una *property* encapsula una secuencia para convertirla en una regla lógica que puede afirmarse (*assert*), suponerse (*assume*) o cubrirse (*cover*). Las propiedades permiten controlar en qué condiciones se activa la evaluación de la secuencia,

incluyendo mecanismos como `disable iff`, que desactiva la evaluación bajo ciertas condiciones, típicamente un *reset*, tal y como se muestra en la Figura 24.

```
1  property p1;
2      disable iff (reset)
3      s1;
4  endproperty
```

Figura 24: Estructura básica de una property de SVA

En la evaluación y control de tiempo, *SystemVerilog* proporciona múltiples mecanismos para manejar el tiempo y la sincronización dentro de las SVA:

- `@(<event>):` especifica el reloj o evento de evaluación.
- `$rose(signal), $fell(signal), $stable(signal):` funciones del sistema para detectar transiciones.
- `$past(expression, N)`: accede al valor pasado de una señal `N` ciclos atrás.
- `disable iff (cond)`: desactiva temporalmente la evaluación de la propiedad si se cumple la condición.

La correcta sincronización de eventos y condiciones es fundamental para evitar errores de verificación. Por ejemplo, para un entorno sensible a flancos de reloj, una propiedad básica para descubrir la relación entre una solicitud (`req`) y un reconocimiento (`ack`) seria la que se muestra en la Figura 25.

```
1  assert property (@(posedge clk) disable iff (rst)
2                   req |-> ##1 ack);
3  property handshake_prop;
```

Figura 25: Ejemplo de entorno sensible a flancos en SVA

Este tipo de afirmaciones pueden modelar protocolos de *handshake*, relaciones maestro-esclavo y muchas otras situaciones críticas.

Una de las principales fortalezas de las SVA es su expresividad sintáctica, que permite capturar de forma compacta comportamientos complejos. Esta expresividad, sin embargo, conlleva un coste: cuanto más complejas y ricas sean las condiciones temporales (por ejemplo, secuencias anidadas o con múltiples alternativas), mayor es la carga computacional en herramientas de verificación formal como los *model checkers*.

4.3.2. Operadores temporales y semántica

Las *SystemVerilog Assertions* permiten expresar de manera formal comportamientos temporales complejos gracias a una serie de operadores específicos. Estos operadores temporales dotan al lenguaje de una gran expresividad, permitiendo modelar restricciones de tiempo, repeticiones y duraciones.

Uno de los operadores más utilizados es el operador de concatenación temporal, representado mediante el símbolo **##**. Este operador indica que un determinado evento debe ocurrir un número específico de ciclos de reloj después de otro evento. Por ejemplo, si una propiedad establece que una señal `b` debe activarse exactamente un ciclo después de que se active la señal `a`, se emplearía este operador con un valor de retardo igual a uno. Además, este operador puede aceptar rangos, permitiendo definir ventanas en las que una condición debe cumplirse, como por ejemplo entre uno y tres ciclos tras un evento inicial.

La semántica de repetición se expresa a través del operador de corchetes con asteriscos, **`[*]`**. Este permite establecer cuántas veces consecutivas debe mantenerse una condición. Así, se puede exigir que una determinada señal esté activa durante tres ciclos continuos, o bien entre un mínimo y un máximo de ciclos. Esta capacidad resulta especialmente útil en contextos donde se requiere estabilidad de una señal o cuando se implementan protocolos que demandan periodos de espera controlados.

Otro operador fundamental es `throughout`, que obliga a que una condición se mantenga verdadera durante toda la duración de una secuencia determinada. Es decir, si se define una secuencia compuesta por varios eventos, el operador `throughout` garantizará que otra condición externa se mantenga constante mientras esa secuencia se

desarrolla. Por su parte, el operador **within** especifica que una determinada secuencia debe tener lugar dentro del intervalo de tiempo delimitado por otra. Esto es útil, por ejemplo, para expresar que una operación debe completarse dentro de una ventana temporal permitida.

Los operadores ***until*** y ***until_with*** permiten controlar la duración de una condición hasta que se produce otra. La diferencia entre ambos radica en que **until** simplemente mantiene la condición inicial hasta que la segunda se cumple, mientras que **until_with** exige que ambas condiciones sean verdaderas simultáneamente en el ciclo final.

Uno de los aspectos más relevantes en la semántica de las SVA es la diferencia entre los operadores de implicación **|->** y **|=>**. Aunque ambos indican que una condición inicial debe ir seguida de una consecuencia, difieren en su comportamiento temporal. El operador **|->**, conocido como implicación superpuesta, permite que la consecuencia comience en el mismo ciclo en el que se activa la condición. En cambio, **|=>**, o implicación secuencial, obliga a que la consecuencia ocurra a partir del siguiente ciclo, es decir, después de que la condición haya finalizado. Esta diferencia es clave en el modelado de protocolos donde el orden exacto de los eventos es fundamental.

Todos estos operadores se combinan con el mecanismo de censado de reloj, que se expresa mediante el uso del símbolo **@**, especificando el dominio temporal en el que se evaluarán las condiciones. Esto permite al diseñador definir con precisión cuándo se deben comprobar las relaciones entre señales, típicamente en flancos de reloj.

A modo de resumen, se adjunta la Tabla 2, la cual agrupa ciertos operadores temporales y semántica.

Expresión	Significado
a ##1 b	'b' ocurre un ciclo después de 'a'
a[*3]	'a' se mantiene durante 3 ciclos seguidos
a[*1:4]	'a' se mantiene entre 1 y 4 ciclos consecutivos
a throughout s	'a' permanece activa durante toda la secuencia s
s1 within s2	la secuencia s1 ocurre dentro del marco temporal de s2

a until b	'a' es válida hasta que 'b' se cumpla

Tabla 2: Resumen operadores principales SVA

4.3.3. Ejemplos técnicos aplicados

En este apartado se presentan algunos ejemplos prácticos de propiedades aplicadas, que permiten verificar patrones comunes en el diseño digital, como protocolos de handshake, ventanas de tiempo y secuencias de inicialización.

- Verificación de handshake req → ack: Un ejemplo clásico en la verificación de protocolos es el patrón de *handshake*, en el que una señal de petición (*req*) debe ir seguida de una señal de reconocimiento (*ack*) dentro de un intervalo de tiempo determinado. Esta propiedad garantiza que cada vez que se emite una petición, el sistema responde de forma correcta y oportuna. La propiedad puede modelarse mediante la secuencia mostrada en la Figura 26:

```
1  sequence handshake;
2    req ##[1:3] ack;
3  endsequence
```

Figura 26: Ejemplo Sequence SVA

Aquí se especifica que tras activarse la señal **req**, se espera que la señal **ack** se active entre uno y tres ciclos después. La propiedad asociada a esta secuencia se puede evaluar tal y como se muestra en la Figura 27:

```
1  assert property (@(posedge clk) disable iff (reset) handshake);
```

Figura 27: Assert Property, ejemplo sequence SVA

Este tipo de afirmación es fundamental para verificar buses, interfaces de comunicación y controladores periféricos, donde una respuesta tardía o ausente puede indicar un fallo funcional crítico.

- Ventana de tiempo para respuesta: Más allá del simple *handshake*, en muchas arquitecturas es necesario que ciertas operaciones ocurran dentro de una ventana

temporal estricta. Por ejemplo, una instrucción enviada debe procesarse antes de que se agote un temporizador, o una señal de validación debe coincidir con un rango específico de ciclos. Este tipo de verificación se puede expresar con operadores de rango:

```
property respuesta_oportuna;
  @(posedge clk) trigger |-> ##[2:5] done;
endproperty
```

Figura 28: Ejemplo venetana temporal SVA

En este caso, tras una señal de **trigger**, se espera que la señal **done** se active entre los ciclos 2 y 5 posteriores, lo que define explícitamente la ventana temporal válida para la operación. Este patrón se utiliza ampliamente en diseños con limitaciones temporales estrictas, como pueden ser los protocolos de red.

- Secuencia de inicialización y reset: Otro ejemplo común en la verificación de sistemas digitales es la validación de secuencias de inicialización. En muchas arquitecturas, tras la desactivación del *reset*, el sistema debe pasar por una serie de pasos definidos antes de comenzar su operación normal. La secuencia puede modelarse como se muestra en la Figura 29.

```
sequence init_seq;
  !reset ##1 init == 1'b1 ##2 ready == 1'b1;
endsequence

assert property (@(posedge clk) init_seq);
```

Figura 29: Ejemplo Inicialización y Reset SVA

Esto asegura que, una vez desactivado el *reset*, el sistema habilita la señal **init** en el siguiente ciclo y, posteriormente, tras dos ciclos adicionales, la señal **ready** se activa. Este patrón es clave para detectar errores en la lógica de funcionamiento.

4.3.4. Operadores temporales y semántica

Además de las propiedades que se afirman con el token *assert*, *SystemVerilog Assertions* (SVA) proporciona otros dos mecanismos fundamentales para enriquecer el proceso de verificación formal y funcional: la cobertura formal, a través del token cover property, y las suposiciones del entorno, mediante el token *assume* property. Ambos conceptos cumplen funciones distintas pero complementarias en un flujo de verificación moderno.

- Cobertura formal (cover property): Las propiedades de cobertura permiten verificar que ciertos escenarios o secuencias efectivamente ocurren durante la simulación o el análisis formal. A diferencia de las *assert*, que detectan fallos, las *cover* permiten medir la visibilidad de los casos de uso del diseño. Son, por tanto, una herramienta clave para evaluar la completitud de la verificación. Por ejemplo, una propiedad de cobertura que comprueba si se produce una transacción válida tras una petición podría escribirse como se muestra en la Figura 30.

```
1   cover property (@(posedge clk) req ##1 ack);
```

Figura 30: Ejemplo cover property SVA

Esta instrucción no genera fallos si no se cumple, pero sí reporta si la situación se ha producido al menos una vez durante la ejecución. Su utilidad es doble: sirve como métrica de cobertura funcional y como guía para mejorar los patrones de estímulo del *testbench*.

En contextos de verificación formal, las herramientas de *model checking* también usan el mecanismo de cobertura, especificado mediante el token *cover* como objetivos para buscar ejemplos de ejecución que cumplan ciertas secuencias. Esto permite comprobar el alcance de situaciones deseadas, como rutas críticas de control, modos de operación o cambios de estado en máquinas de estados.

Suposiciones del entorno (*assume property*): Por otro lado, las suposiciones de entorno, denominadas *assume* permiten restringir el espacio de búsqueda del verificador formal o definir condiciones que el entorno debe cumplir obligatoriamente. Mientras que las

propiedades activadas con *assert* se aplican sobre el diseño bajo verificación DUV, las *assume* describen comportamientos esperados del entorno del DUV. Un ejemplo típico es suponer que, tras un *reset*, no se activarán señales de control durante algunos ciclos. Esto se especifica tal y como se muestra en la Figura 31.

```
1   assume property (@(posedge clk) disable iff (reset) !start[*5]);
```

Figura 31: Ejemplo assume property SVA

Esta suposición indica que no se debe iniciar ninguna operación durante los cinco primeros ciclos tras el *reset*. Si el entorno no cumple esta condición, la herramienta formal puede considerarlo un caso inválido y descartarlo, enfocando el análisis en escenarios realistas.

Las *assume* son especialmente útiles para evitar falsos negativos en verificación formal, reducir el espacio de estados explorado, y modelar el comportamiento esperado de otros módulos que no están directamente en el análisis.

La Tabla 3 muestra una comparativa y resumen las diferencias clave entre los distintos tipos de propiedades utilizadas en *SystemVerilog Assertions*.

Tipo de propiedad	Propósito	Resultado
assert	Verificar que se cumpla una regla	Falla si la regla no se cumple
cover	Confirmar que algo ocurre	Informa si se alcanza el objetivo
assume	Restringir el entorno	El análisis solo considera escenarios compatibles

Tabla 3: Tipos de propiedad SVA

En flujos de verificación formal, es habitual utilizar las tres en conjunto: las *assert* definen lo que se quiere comprobar, las *assume* lo que se da por hecho y las *cover* lo que se desea observar.

4.3.5. Rol de las SVA en la abstracción formal

En el contexto de la verificación formal, las *SystemVerilog Assertions* (SVA) juegan un papel fundamental al constituir una interfaz precisa y ejecutable del comportamiento esperado del sistema. Más allá de su uso como meras condiciones de comprobación, las SVA actúan como un modelo abstracto de especificación, capaz de describir requisitos funcionales con

una granularidad temporal detallada, pero sin necesidad de codificar el comportamiento completo del sistema.

Esta capacidad de abstracción permite a las herramientas de verificación formal, como los *model checkers*, utilizar las SVA como objetivos de análisis exhaustivo. Al aplicar técnicas como la exploración simbólica de estados, estas herramientas intentan demostrar que todas las ejecuciones posibles del diseño cumplen las propiedades expresadas en las aserciones. Si se detecta una violación, se genera un contraejemplo: una traza temporal que muestra paso a paso cómo se llega a la situación errónea. Este contraejemplo es extremadamente útil para la depuración, ya que localiza con precisión la fuente del fallo.

Las propiedades también pueden utilizarse como criterios de prueba para herramientas automáticas. Por ejemplo, en verificación dirigida por propiedades (*property-driven verification*), el objetivo no es solo comprobar que una propiedad se cumple, sino también generar estímulos que la activen (*cover*) o explorar qué condiciones del entorno (*assume*) permiten su satisfacción. Esto convierte las SVA en elementos activos del flujo de verificación, no solo en comprobaciones pasivas.

No obstante, su uso como abstracción formal presenta también limitaciones inherentes.

A medida que las propiedades se vuelven más expresivas, por ejemplo, incorporando secuencias anidadas, ventanas temporales amplias o condiciones múltiples, el coste computacional para verificarlas aumenta considerablemente. Las herramientas deben analizar un mayor número de caminos posibles, lo que puede provocar explosión del espacio de estados, tiempos de ejecución elevados o incluso la imposibilidad de concluir el análisis.

Por este motivo, un diseño eficaz de propiedades debe buscar el equilibrio entre expresividad y verificabilidad. Se recomienda priorizar propiedades modulares, bien documentadas y orientadas a casos funcionales clave. Además, resulta útil refinar propiedades complejas en subpropiedades más simples que puedan verificarse por separado, incrementando así la escalabilidad del proceso.

En conclusión, las SVA permiten elevar el nivel de abstracción en la verificación, al capturar de forma formal los requisitos esenciales del sistema sin necesidad de modelar cada detalle interno. Esta capacidad las convierte en una herramienta estratégica para asegurar la corrección del diseño y facilitar su mantenimiento.

5º Capítulo: Propiedades de verificación en SystemVerilog

El presente capítulo se centra en el análisis y la implementación de propiedades de verificación utilizando *SystemVerilog Assertions* (SVA) como herramienta para validar el comportamiento funcional del módulo, que actúa como Dispositivo Bajo Verificación (DUV) en este proyecto. Tras haber abordado en el capítulo anterior los fundamentos teóricos de las SVA, aquí se da el paso hacia su aplicación práctica, integrándolas dentro del entorno funcional previamente desarrollado.

El objetivo principal es mostrar cómo estas propiedades permiten capturar de forma concisa las condiciones que el diseño debe cumplir y cómo se integran sin modificar el código fuente del DUV. Esta aproximación no solo mejora la eficiencia de la verificación, sino que también permite detectar errores lógicos difíciles de identificar mediante observación manual de formas de onda.

Durante esta fase del proyecto se ha optado por una aproximación progresiva, enfocada en una toma de contacto práctica con el uso de SVA. Para ello, se han implementado una serie de propiedades sencillas pero representativas que validan aspectos fundamentales del funcionamiento del módulo, como la correcta gestión de inserciones, la validez de salidas y la respuesta del sistema ante condiciones de reinicio.

Además, se experimentó con dos metodologías distintas para la integración de las SVA en el entorno de verificación: la inclusión directa mediante la directiva *include* y la vinculación no intrusiva mediante la directiva *bind* que proporciona SystemVerilog. Esta comparación permitió comprender los beneficios y limitaciones de cada enfoque en términos de modularidad, escalabilidad y reutilización.

A lo largo de este capítulo se describirán las propiedades diseñadas, se detallará cómo fueron integradas en el *testbench* funcional y se analizarán los primeros resultados obtenidos. Con ello, se sientan las bases para su posterior evolución en un entorno UVM más estructurado, que será abordado en los siguientes capítulos del presente trabajo.

5.1. Comportamiento a verificar

Antes de abordar el diseño de propiedades en *SystemVerilog Assertions* (SVA), es esencial identificar los elementos funcionales del módulo que resultan más relevantes desde el punto de vista de la verificación. En este proyecto, el componente bajo análisis implementa una lógica de almacenamiento y gestión de datos basada en prioridades, donde cada entrada se acompaña de un valor de orden (*rank_in*) que determina su relevancia relativa frente al resto.

El módulo permite realizar operaciones de inserción y extracción, y mantiene señales de estado que informan sobre su ocupación y validez de los datos en salida. Dado que estas señales y operaciones forman parte de la interfaz observable desde el *testbench*, y son además determinantes para el correcto funcionamiento del sistema, se decidió centrar en ellas la primera etapa de validación mediante SVA.

Durante esta fase inicial, el objetivo fue desarrollar un conjunto reducido pero representativo de propiedades que validaran los siguientes comportamientos esenciales:

- Inserción de datos (**`insert`**): Verificar que una operación de inserción solo se realiza cuando el módulo no está lleno (`full` = 0) y que, en caso contrario, se respeta la lógica interna de prioridad.
- Extracción de datos (***`remove`***): Comprobar que la extracción solo se permite cuando hay datos válidos (***`empty`*** = 0) y que la salida generada (*rank_out, meta_out*) corresponde, efectivamente, a un valor válido, lo cual debe reflejarse en la señal *`valid_out`*.
- Señales de estado (***`full, empty, valid_out`***): Validar que estas señales evolucionan de forma coherente con las operaciones realizadas. Por ejemplo, la señal `full` debe activarse tras insertar el último elemento permitido, y la señal ***`empty`*** debe hacerlo al vaciar completamente el módulo.
- Sincronización y condiciones de reset: Asegurar que todas las propiedades se evalúan en sincronismo con el reloj y se desactivan correctamente cuando la señal de *reset* está activa, evitando falsas detecciones durante fases de inicialización.

5.2. Diseño e implementación de las propiedades SVA

Para esta primera aproximación a la verificación mediante *SystemVerilog Assertions*, se diseñaron un conjunto de propiedades enfocadas a comprobar el comportamiento esencial del módulo. Estas propiedades fueron implementadas en el archivo ***`assertions.svh`*** y vinculadas al diseño utilizando las técnicas explicadas en el apartado anterior.

El diseño de cada propiedad siguió un enfoque sistemático: se partió del análisis funcional del módulo, se identificaron las señales implicadas en cada operación crítica y se definieron condiciones que debían cumplirse durante su ejecución. Las propiedades se construyeron en base a bloques definidas mediante el token ***`property`***, activadas con sensibilidad a la señal reloj (**`@posedge clk`**) y desactivadas bajo condición de *reset* (**`disable iff (rst)`**), asegurando así su correcta evaluación temporal. A continuación, se describen algunas de las propiedades implementadas, seleccionadas por su valor didáctico y por cubrir situaciones típicas en módulos de control.

- Propiedad 1: Inserción solo permitida si el módulo no está lleno

Esta propiedad comprueba que no se realicen operaciones de inserción (*insert* = 1) mientras la señal full está activa. De este modo, se evita que se sobrescriban datos en un registro saturado. La Figura 32 muestra el código correspondiente a dicha propiedad.

```
1  property p_no_insert_when_full;
2    disable iff (rst)
3    (full && insert) |-> ##1 $fatal("ERROR: Inserción en estado FULL");
4  endproperty
```

Figura 32: Propiedad insert_when_full

Esta expresión define una condición de error: si las señales ***`full`*** e ***`insert`*** están activas simultáneamente, se genera un error en la simulación. En este caso, si bien se usa una acción de tipo **`fatal`** para mostrar la condición de error, también podría emplearse una notificación menos intrusiva, como seria el uso de una declaración de tipo **`assert`**.

- Propiedad 2: Activación de **valid_out** implica datos disponibles

Aquí se comprueba que, cuando la salida ***valid_out*** está activa, el módulo no se encuentre en estado *empty*. Es decir, no se deben declarar datos válidos si no hay elementos almacenados. La Figura 33 muestra la implementación de la propiedad en cuestión.

```
property p_valid_out_implies_not_empty;
  disable iff (rst)
  valid_out |-> !empty;
endproperty
```

Figura 33: Propiedad valid_out_implies_not_empty

Esta propiedad asegura la coherencia interna entre las señales de salida del sistema.

- Propiedad 3: Desactivación de **valid_out** al vaciar el módulo

Esta propiedad complementa la anterior, exigiendo que si la señal de ***empty*** se activa, el módulo se vacía, lo que implica que la señal ***valid_out*** debe desactivarse en ese momento o inmediatamente después. La Figura 34 muestra el código correspondiente a dicha propiedad.

```
property p_empty_disables_valid_out;
  disable iff (rst)
  empty |=> !valid_out;
endproperty
```

Figura 34: Propiedad, empty_disables_valid_out

Con esto se evita que se mantenga activo el indicador de ***valid_out*** cuando no existe información real que extraer.

- Propiedad 4: Activación de ***reset*** cancela cualquier salida válida

Como práctica habitual en diseños síncronos, se espera que al activarse la señal de ***reset***, las salidas del sistema vuelvan a un estado seguro. Esta propiedad verifica que la señal **valid_out** se desactiva inmediatamente tras el ***reset***. La Figura 35 muestra la propiedad que verifica este comportamiento.

```
1  property p_reset_clears_valid_out;
2    rst |=> !valid_out;
3  endproperty
```

Figura 35: Propiedad reset_clears_valid_out

En conjunto, estas propiedades no sólo comprueban condiciones funcionales relevantes, sino que sirven como base para futuras extensiones. Cada una de ellas está escrita de forma modular y explícita, con una estructura clara que facilita su mantenimiento y depuración. Durante su implementación, se prestó especial atención a aspectos como la sincronización con el reloj, la activación condicional bajo **`disable iff (rst)`**, y la elección entre implicación superpuesta (**`|->`**) y secuencial (**`|=>`**) según la lógica del comportamiento verificado.

5.3. Integración en el testbench funcional

Para evaluar el comportamiento del módulo, se exploraron dos métodos distintos de integración de propiedades en el entorno funcional. Esta doble aproximación tuvo como objetivo comparar su viabilidad práctica y familiarizarse con las opciones que ofrece el lenguaje para incorporar SVA sin necesidad de alterar el diseño original del DUV.

- Método 1: Inclusión directa mediante la directiva ***`include`***

La primera técnica consistió en incluir directamente el archivo de aserciones (***`assertions.svh`***) al final del archivo principal del *testbench*. Esta opción es la más sencilla desde el punto de vista sintáctico y permite una visualización inmediata de las propiedades junto con el banco de pruebas. Para su implementación, en el archivo testbench.sv, se habilitó el código que muestra la Figura 36:

```
1  //Assertions
2  `include "assertions.svh"
```

Figura 36: Método inclusión directa SVA

Este enfoque presenta la ventaja de la inmediatez y la simplicidad, especialmente en fases tempranas de desarrollo o en entornos no modulares. Sin embargo, su principal inconveniente es que acopla las propiedades al archivo del *testbench*, lo cual puede limitar su reutilización en otros entornos o dificultar su mantenimiento.

- Método 2: Vinculación externa mediante la directiva *bind*

El segundo método utilizado fue el empleo del mecanismo ***bind***, una característica de *SystemVerilog* que permite conectar propiedades externas a instancias internas del diseño sin modificar su código fuente.

En este caso, se utilizó el archivo **bind_assertions.sv** para aplicar las propiedades al módulo dentro del entorno de prueba. La asociación entre el DUV, el archivo con las aserciones es el que se muestra en la Figura 37.

```
1  bind tbench_top.DUT bind_assertions bind_assertions_sva_inst(
2    clk, rst, in_intf.insert, out_intf.remove, out_intf.valid_out);
```

Figura 37: Método vinculación externa mediante bind

Aquí, el token ***bind*** se aplica sobre la instancia DUT contenida dentro del módulo *tbench_top*. Esto permite referenciar un módulo auxiliar denominado *bind_assertions*, definido como contenedor de SVA y conectarlo a las señales necesarias (***clk, rst, insert, remove, valid_out***) para evaluar las propiedades de manera no intrusiva.

Este segundo método es más potente y escalable, ya que permite mantener las propiedades en módulos separados, facilitando su reutilización y organización.

Ambos métodos fueron funcionales en el entorno de pruebas utilizado, y permitieron validar las mismas propiedades. No obstante, se observó que el uso del método ***bind*** representa una práctica más profesional y alineada con entornos de verificación avanzados, mientras que la inclusión directa mediante la directiva ***include*** es más adecuada para entornos pequeños o pruebas exploratorias.

5.4. Resultados y observaciones

Tras la implementación y conexión de las propiedades en el entorno de verificación funcional, se procedió a la ejecución de varias simulaciones orientadas a validar tanto la sintaxis como la lógica de las *SystemVerilog Assertions* diseñadas. El resultado fue satisfactorio, todas las propiedades se activaron correctamente en el momento previsto y respondieron de manera coherente ante los estímulos generados por el *testbench*.

En ninguna de las simulaciones se registraron violaciones de propiedades, lo cual es coherente con el hecho de que el diseño funcional del módulo ya había sido depurado en fases anteriores. No obstante, las SVA ofrecieron una garantía adicional, al verificar de forma automática condiciones críticas que de otro modo requerirían inspección manual mediante formas de onda.

La experiencia con esta primera integración de SVA dejó varias observaciones de valor:

- Utilidad inmediata: incluso con un conjunto reducido de propiedades, fue posible detectar o confirmar comportamientos que en otros entornos podrían pasar desapercibidos. La propiedad `p_no_insert_when_full`, por ejemplo, asegura una condición de seguridad relevante que podría producir errores silenciosos si no se controla adecuadamente.

- Simplicidad y expresividad: las propiedades diseñadas resultaron concisas y legibles, lo que facilitó tanto su escritura como su mantenimiento. La separación entre condiciones de activación (`@posedge clk`) y desactivación (`disable iff (rst)`) resultó especialmente útil para evitar evaluaciones no deseadas durante la fase de *reset*.

- Flexibilidad en la integración: la prueba de ambos métodos de conexión, `include` y `bind`, ofreció una perspectiva clara sobre las ventajas de desacoplar las

  propiedades del banco de pruebas. La opción mediante `bind` es relativamente óptima por su escalabilidad y por mantener la estructura del *testbench* limpia y modular.

- Limitaciones actuales: aunque el conjunto de propiedades abordó aspectos esenciales del funcionamiento del módulo, se trató de una primera toma de contacto. No se contemplaron aún propiedades más complejas como relaciones de prioridad entre entradas, condiciones de reemplazo o validación cruzada de múltiples ciclos. Estas se reservan para una fase posterior en el entorno UVM.
- Facilidad de ampliación: uno de los puntos positivos fue comprobar que, una vez establecida la infraestructura de aserciones y su integración, la adición de nuevas propiedades puede hacerse de forma incremental, sin necesidad de modificar el DUV ni alterar el flujo de simulación.

Aunque en esta fase no se exploraron explícitamente escenarios forzados que violasen las propiedades definidas, el objetivo principal era confirmar su correcta activación e integración dentro del entorno funcional bajo condiciones normales de operación. No obstante, en fases posteriores del proyecto, especialmente en el capítulo de resultados de verificación, se propone realizar variaciones controladas en el número de operaciones de inserción y extracción, así como en el estado interno del módulo, con el fin de provocar violaciones intencionadas y observar la respuesta efectiva de las SVA. Esta estrategia permitirá validar no solo que las propiedades están correctamente formuladas, sino que también son capaces de detectar comportamientos indebidos en situaciones límite, reforzando así su valor como herramienta de verificación.

6º Capítulo: Desarrollo del entorno UVM

Tras una primera aproximación a la verificación funcional mediante un *testbench* clásico en *SystemVerilog*, el siguiente paso en el desarrollo del proyecto ha sido la migración hacia un entorno estructurado basado en la *Universal Verification Methodology* (UVM). Este cambio no solo responde a criterios técnicos, sino que está alineado con el objetivo principal del trabajo: implementar un sistema de encapsulación de *SystemVerilog Assertions* (SVA) dentro de un entorno UVM, tal y como propone el artículo de *Verilab* "*SVA Encapsulation in UVM: Enabling Phase and Configuration Aware Assertions*".

La motivación para adoptar UVM radica en sus ventajas frente al enfoque funcional tradicional: permite una mayor modularidad, reutilización de componentes, generación aleatoria de estímulos, cobertura funcional estructurada y, sobre todo, una integración más robusta y escalable de propiedades formales dentro de entornos complejos. UVM se convierte así en el marco ideal para evolucionar desde la verificación estática basada en estímulos manuales, hacia una verificación automatizada, parametrizable y dirigida por configuración.

Un cambio importante en esta fase fue el abandono de plataformas online como *EDAPlayground*, utilizadas durante las etapas iniciales, para dar paso a un entorno de desarrollo profesional ejecutado sobre un servidor local del Instituto Universitario de Microelectrónica Aplicada (IUMA). Dicho servidor opera sobre un sistema Linux y cuenta con el simulador *Questasim* de Siemens EDA, una herramienta industrial ampliamente reconocida en verificación de sistemas hardware.

El acceso a este entorno se realiza a través de una máquina virtual, que actúa como cliente y permite una experiencia de trabajo remota. Además, se incorporó el uso de la plataforma GitHub como sistema de control de versiones y gestión colaborativa del código, permitiendo una trazabilidad clara del desarrollo, identificación rápida de errores y mejora progresiva del entorno de verificación. Esta herramienta también ha sido clave para experimentar con ramas paralelas, depuración controlada y pruebas reproducibles.

El objetivo específico de este capítulo es, por tanto, documentar la construcción del entorno UVM, describiendo los componentes principales que lo conforman.

Este desarrollo representa un avance significativo tanto en términos de metodología como de complejidad técnica, y sienta las bases para una verificación más automatizada, escalable y alineada con los estándares profesionales de la industria.

6.1. Arquitectura general del testbench UVM

El entorno de verificación UVM desarrollado para el módulo sigue las directrices clásicas de la metodología UVM, dividiendo las responsabilidades de estimulación, observación, comparación y control.

El diseño del *testbench* se organiza en torno a dos flujos diferenciados: entrada (*insert*) y salida (*remove*). Cada uno de estos flujos cuenta con sus propios agentes (*agent*_i y *agent*_r), lo que permite tratar de forma independiente las transacciones que entran y salen del DUV. Esta separación mejora la trazabilidad y facilita la instrumentación de propiedades específicas en fases posteriores. La Figura 38 presenta un esquema general del *testbench* desarrollado.

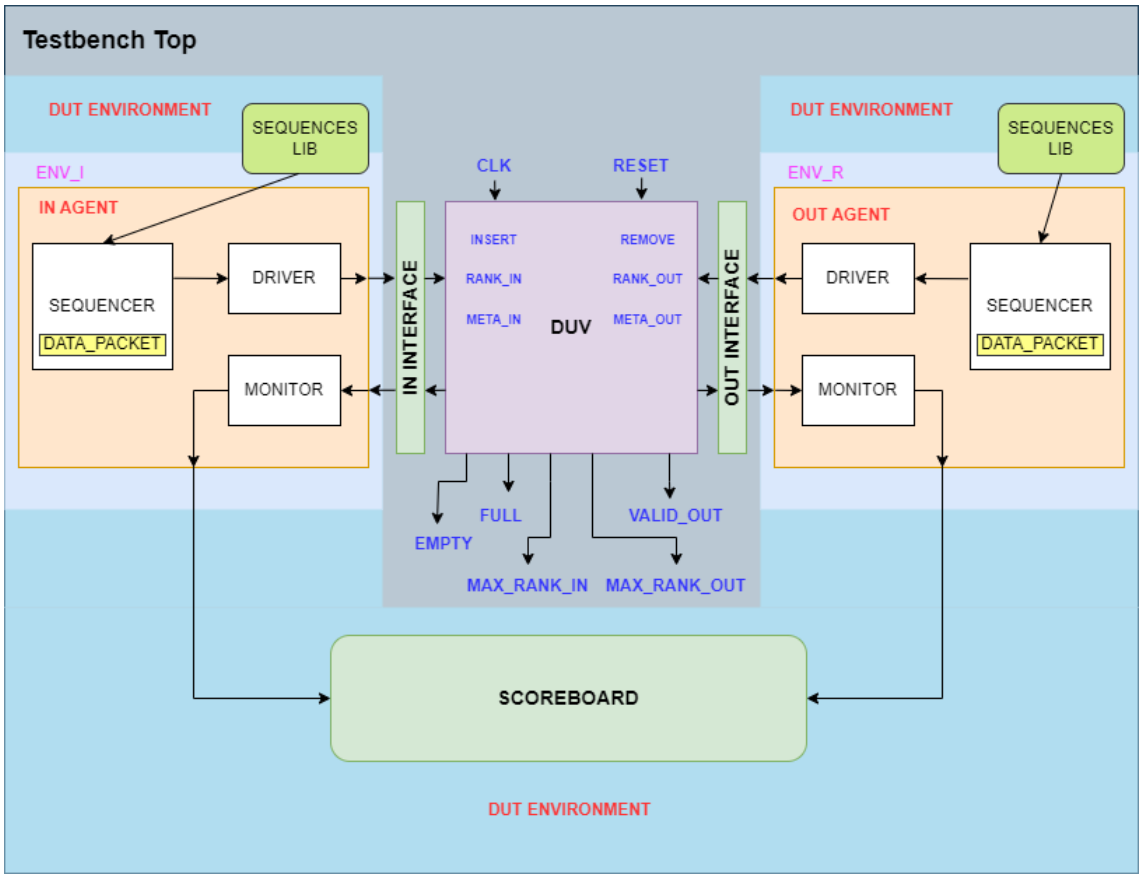

Figura 38: Arquitectura de verificación UVM

El entorno UVM desarrollado incluye los siguientes niveles jerárquicos:

- Interfaces virtuales: encapsulan las señales necesarias para conectar el entorno UVM con el diseño RTL. Son esenciales para permitir la comunicación entre clases UVM y el hardware descrito en *Verilog* RTL.
- *Data packets*: definen las transacciones que viajarán a través del sistema. Contienen los campos necesarios para representar tanto la información útil como los metadatos de control. Estas transacciones se derivan de la clase base *uvm_sequence_item*.
- *Drivers*: responsables de convertir las transacciones generadas en señales concretas para el diseño. Se activan mediante estímulos generados por el componente *sequencer*.
- *Sequencers*: controlan la generación de secuencias de transacciones, permitiendo tanto pruebas aleatorias como dirigidas.
- Monitores: observan pasivamente el comportamiento del DUV en ambos flujos, capturan la actividad relevante y la reportan al componente *scoreboard*.
- Agentes: encapsulan cada conjunto formado por *driver*, *monitor* y *sequencer* para las operaciones de entrada y salida respectivamente. Aportan modularidad y facilitan la reutilización.
- Scoreboard: componente que implementa el modelo funcional de referencia. Compara los datos capturados por los monitores con los resultados esperados, validando el correcto funcionamiento del módulo.
- Entorno (env): se encarga de referenciar y conectar los componentes agentes y el componente *scoreboard*. Es el núcleo jerárquico del *testbench*.
- *Test*: referencia el entorno (*env*), define los parámetros del test y configura el tipo de estímulo que se va a aplicar al DUV.
- *Top-level testbench*: referencia el diseño, las interfaces físicas y la clase test. También proporciona el reloj, la señal de *reset* y realiza el control general de la simulación.

6.2. Descripción de componentes implementados

Durante el desarrollo del entorno de verificación UVM, se implementaron todos los componentes fundamentales definidos en la metodología UVM, organizados en torno al patrón clásico: *sequence item* → *sequencer* → *driver* → *monitor* → *scoreboard*, agrupados dentro de agentes y entornos jerárquicos. Esta estructura sigue fielmente las recomendaciones de los libros guía en los que se ha basado el proyecto, favoreciendo la claridad, modularidad y escalabilidad del entorno.

A continuación, se describen con detalle cada uno de los bloques UVM desarrollados.

- Transacciones: (*data_packet_i y data_packet_r*)

En la metodología UVM, las transacciones representan las unidades básicas de comunicación entre componentes como secuenciadores, drivers y monitores. Estas transacciones se implementan como clases que derivan de la clase base uvm_*sequence*_item. En este proyecto se han definido dos clases de transacción: una para la entrada (data_*packet*_i) y otra para la salida (data_*packet*_r), adaptadas a las señales de control del módulo. La Figura 39 muestra el código en SystemVerilog en la descripción de la transacción data_*packet*_i.

```
1  import uvm_pkg::*;
2  import pifo_pkg::*;
3  `include "uvm_macros.svh"
4
5  class data_packet_i extends uvm_sequence_item;
6      bit insert;
7      bit full;
8      rand bit [RANK_WIDTH-1:0]  rank_in;
9      rand bit [META_WIDTH-1:0]  meta_in;
10     bit max_valid_out;
11     bit [RANK_WIDTH-1:0]  max_rank_out;
12     bit [META_WIDTH-1:0]  max_meta_out;
13     bit [L2_REG_WIDTH:0] num_entries;
14     //delay utilizado entre transacciones
15     rand int delay;
16
17   constraint timing {delay inside {[1:5]};}
18
19   `uvm_object_utils_begin(data_packet_i)
20       `uvm_field_int(insert, UVM_DEFAULT)
21       `uvm_field_int(full, UVM_DEFAULT)
22       `uvm_field_int(rank_in, UVM_DEFAULT)
23       `uvm_field_int(meta_in, UVM_DEFAULT)
24       `uvm_field_int(max_valid_out, UVM_DEFAULT)
25     `uvm_field_int(max_rank_out, UVM_DEFAULT)
26       `uvm_field_int(max_meta_out, UVM_DEFAULT)
27     `uvm_field_int(num_entries, UVM_DEFAULT)
28       `uvm_field_int(delay, UVM_DEFAULT)
29    `uvm_object_utils_end
30
31   function new(string name = "data_packet_i");
32       super.new(name);
33    endfunction: new
```

Figura 39: Clase data_packet_ i UVM

Las clases *data_packet_*i y *data_packet_r* se derivan de *uvm_sequence_item*, la clase base estándar en UVM para representar transacciones o estímulos intercambiables entre componentes del entorno de verificación. Esta clase base proporciona mecanismos internos para aleatorizar los campos definidos en cada transacción. En el caso de ***data_packet_i***, los atributos *rank* y *meta* están declarados como tipo *rand*, lo que permite su generación aleatoria. Estos campos modelan, respectivamente, la prioridad del dato y los metadatos asociados a una inserción en el módulo. A ellos se suma el campo ***insert***, una señal de control binaria que determina si se debe realizar la operación de inserción en ese ciclo. La clase ***data_packet_r***, por el contrario, representa una transacción observada tras una operación de extracción, por lo que sus campos (***rank, meta, valid, remove***) no son aleatorizables,

sino reconstruidos por los monitores a partir de las señales del DUV. En ambos casos, se emplean las macros *uvm_field_int* dentro del bloque *uvm_object_utils_begin* y *uvm_object_utils_end*, lo que activa automáticamente los métodos de trazabilidad, comparación y generación de logs. El uso del *flag UVM_ALL_ON* garantiza que todos los campos se incluyan en las operaciones internas de depuración, como recomiendan las guías *Getting Started with UVM* [18] y *A Practical Guide to Adopting the UVM* [19]. Además, el constructor explícito new, que invoca **`super.new(name)`**, permite asignar nombres identificadores a cada instancia de transacción generada, lo cual facilita el seguimiento en los reportes UVM y mejora la trazabilidad durante la simulación.

- *Sequencers*: (*pifo_sequencer_i y pifo_sequencer_r*)

En la metodología UVM, los secuenciadores actúan como intermediarios entre las secuencias de prueba (*uvm_sequence*) y los *drivers*. Su función principal es arbitrar el flujo de transacciones hacia el driver de manera controlada, permitiendo aplicar estímulos aleatorios, dirigidos o incluso programados de forma reactiva. En este proyecto se han definido dos secuenciadores independientes: uno para la entrada (***`pifo_sequencer_i`***) y otro para la salida (***`pifo_sequencer_r`***), reflejando la arquitectura simétrica del módulo. La Figura 39 muestra el código implementado en la clase ***`pifo_sequencer_i.`***

```
1  import pifo_pkg::*;
2
3  class pifo_sequencer_i extends uvm_sequencer #(data_packet_i);
4
5    `uvm_component_utils(pifo_sequencer_i)
6
7     function new(string name, uvm_component parent);
8        super.new(name, parent);
9     endfunction: new
10 endclass: pifo_sequencer_i
```

Figura 40: Clase pifo_sequencer_i UVM

Ambos *sequencers* derivan de la clase base *uvm_sequencer*, una clase base parametrizada que debe referenciarse con el tipo de transacción correspondiente (***`data_packet_i`*** *o* ***`data_packet_r`***). Esta estructura permite al secuenciador conocer exactamente qué tipo de objetos manejará durante su comunicación con el *driver*.

Se ha definido el constructor *new* con los argumentos *name y parent*, invocando internamente al constructor de la clase padre `super.new(...)` como requiere la jerarquía UVM. Esta llamada es esencial para que el secuenciador se registre correctamente dentro del árbol de componentes de UVM, habilitando su trazabilidad, configuración y la ejecución de fases como *build_phase, connect_phase*, etc.

La macro *uvm_component_utils* es obligatoria en cada componente UVM y se utiliza para registrar la clase en el sistema de fábrica de UVM. Esto permite su creación dinámica mediante el método `type_id::create` en el entorno o en el agente correspondiente, además de habilitar la introspección automática por ejemplo, para la impresión de nombres jerárquicos.

En definitiva, los secuenciadores *pifo_sequencer_i y pifo_sequencer_r* constituyen un componente estructural imprescindible dentro del flujo de verificación UVM. A pesar de su simplicidad funcional, su inclusión garantiza el desacoplamiento entre la generación de estímulos y su aplicación al DUV, permitiendo una arquitectura modular, escalable y alineada con los principios de diseño recomendados por la metodología UVM.

- *Drivers*: (`pifo_driver_i` y `pifo_driver_r`)

Los drivers son los componentes del entorno UVM encargados de traducir las transacciones abstractas generadas por el *sequencer* en señales concretas que se aplican al diseño bajo verificación DUV. Constituyen el único elemento del agente que interactúa de forma activa con las interfaces físicas del DUV. Su diseño requiere una sincronización precisa con el reloj del sistema y un manejo correcto del canal de comunicación estándar con el *sequencer*. La Figura 41 muestra la implementación de la clase `pifo_driver_i.`

```
1  import pifo_pkg::*;
2
3  class pifo_driver_i extends uvm_driver #(data_packet_i);
4     virtual pifo_if_i vif;
5
6  //registrar el driver en la factory
7    `uvm_component_utils(pifo_driver_i)
8
9  //Constructor UVM
10    function new(string name, uvm_component parent);
11       super.new(name, parent);
12    endfunction: new
13
14 //build_phase, se usa el uvm_config_db para obtener la int virtual
15    function void build_phase(uvm_phase phase);
16       super.build_phase(phase);
17      if(!uvm_config_db#(virtual pifo_if_i)::get(this, "", "in_intf", vif))
18        `uvm_fatal("NOVIF", {"virtual interface must be set for: ", get_full_name( ), ".vif"})
19        `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
20    endfunction
21
22 //run_phase
23    virtual task run_phase(uvm_phase phase);
24       fork
25          reset( );
26          get_and_drive( );
27       join
28    endtask: run_phase
```

Figura 41: Clase pifo_driver_i Parte 1 UVM

La clase *pifo_driver_i* se deriva de la clase base *uvm_driver* y está especializada para manejar transacciones del tipo *data_packet_i*, que representan operaciones de inserción en el módulo. Declara una interfaz virtual *pifo_if_i*, la cual se obtiene durante la fase *build_phase* a través de la clase *uvm_config_db*, permitiendo que el *driver* interactúe con las señales del diseño sin estar acoplado físicamente al código RTL. Si la interfaz no se encuentra registrada, se lanza un error fatal para evitar una simulación inválida. El constructor *new* y la macro *uvm_component_utils* permiten que el componente se registre en la jerarquía UVM y sea referenciado dinámicamente. En la fase *run_phase*, el *driver* lanza dos tareas en paralelo: *reset()* y *get_and_drive()*. Esta estructura paralela, implementada con el mecanismo *fork/join* ya explicado, permite separar la lógica de reinicio de señales del proceso principal de conducción de transacciones. La tarea *reset* se mantiene activa durante toda la simulación, y cada vez que detecta un flanco de subida en la señal de reset, inicia todas las señales de entrada (*insert, rank_in y meta_in*, entre otras) para asegurar que el diseño arranca en un estado definido y libre de valores residuales. La Figura 42 muestra la implementación de las tareas `get_and_drive` y `drive_packet.`

```
1     virtual task get_and_drive( );
2        forever begin
3          @(negedge vif.rst);
4          while(vif.rst != 1'b1) begin
5             //coge el item del sequencer (bloqueante), Request transaction type
6             seq_item_port.get_next_item(req);
7             //envia a la interfaz el contenido de la transaccion
8             drive_packet(req);
9            `uvm_info(get_type_name,"just drove a packet", UVM_HIGH);
10            //se finaliza
11            seq_item_port.item_done( );
12           `uvm_info(get_type_name,"just returned item_done", UVM_HIGH);
13         end
14       end
15    endtask: get_and_drive
16
17      virtual task drive_packet(data_packet_i pkt);
18        repeat(pkt.delay) @(posedge vif.clk);
19        `uvm_info(get_type_name,"in the drive_packet_i", UVM_HIGH);
20         vif.insert <= 1'b1;
21         vif.meta_in <= pkt.meta_in;
22         vif.rank_in <= pkt.rank_in;
23       @(posedge vif.clk);
24         vif.insert <= 1'b0;
25    endtask
26
27 endclass:pifo_driver_i
```

Figura 42: Clase pifo_driver_i Parte 2 UVM

La tarea *get_and_drive* es el núcleo funcional del driver, encargada de recibir transacciones desde el componente *sequencer* y aplicarlas al diseño cuando el sistema no está en estado de *reset*. Primero, espera un flanco de bajada en la señal *rst* (lo que indica que el *reset* ha finalizado) y, en segundo lugar, entra en un bucle donde, mientras la señal *rst* siga desactivada, va solicitando transacciones una a una mediante el método *seq_item_port.get_next_item(req)*. En la llamada al método, *req* es un puntero que hereda de la clase padre *uvm_sequence_item* y que, en este caso, contiene una referencia a ***data_packet_i***, con todos los campos necesarios para configurar una inserción (como *rank*, *meta*, *insert* y *delay*). Una vez recibida la transacción, se delega su aplicación a la tarea *drive_packet*, que se encarga de generar los estímulos correspondientes. Esta función puede incluir un retardo configurable (***pkt.delay***) antes de activar las señales, lo cual permite simular entornos realistas con temporización variable. A continuación, la tarea *drive_packet*

activa la señal *insert* junto con los valores *rank_in* y *meta_in*, manteniéndolos activos durante un único ciclo de reloj, y luego los desactiva, simulando un pulso de inserción. Finalizada la estimulación, la tarea ***get_and_drive*** llama al método *seq_item_port.item_done()* para notificar al secuenciador que la transacción ha sido completada, cerrando así el ciclo clásico de *handshake* entre los componentes *sequencer* y *driver* en UVM.

```
1    virtual task drive_packet(data_packet_r pkt);
2      while (vif.valid_out == 0) begin //Correccion para evitar bucle infinito
3        repeat(pkt.delay) @(posedge vif.clk);
4       end
5       repeat(pkt.delay) @(posedge vif.clk);
6      `uvm_info(get_type_name, $sformatf("\t VALID_OUT = %d", vif.valid_out), UVM_LOW);
7      `uvm_info(get_type_name, $sformatf("\t insert = %d", vif.insert), UVM_LOW);
8      `uvm_info(get_type_name,"in the drive_packet_r", UVM_HIGH);
9       vif.remove <= 1'b1;
10      @(posedge vif.clk);
11      vif.remove <= 1'b0;
12 endtask
13
```

Figura 43: Tarea drive_packet de la clase data_packet_r UVM

La función *drive_packet* del *driver* de salida, mostrada en la Figura 43, tiene como objetivo aplicar una operación de extracción al diseño, es decir, activar la señal ***remove*** para que el módulo entregue el siguiente dato en cola. Para evitar aplicar esta operación de forma prematura, la tarea primero entra en un bucle ***while*** que espera a que la señal ***valid_out*** se active, lo que indica que hay datos disponibles para ser extraídos. Durante esta espera, se introduce un retardo programado mediante **`repeat(pkt.delay)`**, lo que simula el tiempo de espera entre comprobaciones y evita que el simulador quede bloqueado en un bucle sin avance. Una vez que *valid_out* se active, se aplica un segundo retardo configurable, también con **`repeat(pkt.delay)`**, que representa una latencia realista entre la disponibilidad del dato y su consumo. A continuación, se imprimen mensajes de depuración con el estado de señales clave como *valid_out* e *insert* y, finalmente, se activa la señal *remove* durante exactamente un ciclo de reloj las líneas 9, 10 y 11 de la Figura 42. Esto provoca la generación de un pulso limpio y controlado. Este comportamiento garantiza que la extracción solo se produce en condiciones válidas, evitando errores de protocolo.

- Monitors: (pifo_monitor_i y pifo_monitor_r)

El monitor de entrada (*pifo_monitor_i*) es un componente pasivo, cuya función es observar las señales del diseño sin alterarlas para reconstruir las transacciones que realmente han ocurrido y enviarlas al componente *scoreboard* para su verificación. A diferencia del componente *driver*, el monitor no genera estímulos, sino que actúa como un "vigilante" del comportamiento del DUV. La Figura 44 muestra el código correspondiente de la tarea `collect_data` de la clase `pifo_monitor_i.`

```
1  virtual task collect_data( );
2     forever begin
3       @(posedge vif.clk);
4       wait(vif.insert)
5        data_collected.insert <= vif.insert;
6        data_collected.meta_in <= vif.meta_in;
7        data_collected.rank_in <= vif.rank_in;
8     @(posedge vif.clk);
9        data_collected.max_valid_out <= vif.max_valid_out;
10       data_collected.max_meta_out <= vif.max_meta_out;
11       data_collected.max_rank_out <= vif.max_rank_out;
12       data_collected.num_entries <= vif.num_entries;
13       data_collected.full <= vif.full;
14       $cast(data_clone, data_collected.clone( ));
15       item_collected_port.write(data_clone);
16       num_pkts++;
17    end
18 endtask: collect_data
19
20 virtual function void report_phase(uvm_phase phase);
21    `uvm_info(get_type_name( ), $sformatf("REPORT: COLLECTED PACKETS = %d", num_pkts)
22    , UVM_HIGH)
23 endfunction: report_phase
```

Figura 44: Tarea collect_data de la clase pifo_monitor_i UVM

La clase *pifo_monitor_i* declara una interfaz virtual *pifo_if_i*, que proporciona acceso a las señales del diseño (*insert, rank_in, meta_in*, etc.) y se vincula en la fase *build* utilizando el mecanismo estándar definido en la clase base *uvm_config_db;* Si dicha interfaz no se encuentra disponible, se lanza un error fatal, ya que el monitor depende completamente de esa conexión. Además, el componente define un puerto TLM denominado *uvm_analysis_port,* parametrizado con *data_packet_i*, el tipo de transacción que representa una inserción, y cuya instancia se inicializa en el constructor. Este puerto permite que las transacciones observadas se transmitan a otros bloques del entorno, como el *scoreboard*. La lógica principal reside en la, fase de ejecución *run_phase*, donde el componente monitor se sincroniza con el flanco de subida del reloj `(@(posedge vif.clk))` y comprueba en cada ciclo si se está señalizando una inserción

(`vif.insert == 1`) y si el diseño no está lleno (`vif.full` == 0). Si ambas condiciones se cumplen, se crea dinámicamente un objeto de clase *data_packet_i*, se copian en él los valores de las señales observadas (*rank_in*, *meta_in*, *insert*), y se envía a través del puerto *analysis_port*. Esta transacción reconstruida representa lo que ha ocurrido en el hardware. Finalmente, se imprime un mensaje con los datos capturados, facilitando el seguimiento del comportamiento del sistema. Este enfoque permite que el monitor funcione de forma precisa, pasiva y alineada con el protocolo implementado, capturando únicamente eventos válidos sin interferir en el flujo operativo del diseño.

```
    virtual task collect_data();
     forever begin
         fork
             // Proceso principal que espera la señal vif.remove
             begin
                 @(posedge vif.clk);  // Espera un ciclo de reloj
                 wait(vif.remove) begin
                     // Captura los datos solo cuando `vif.remove` está activo
                     data_collected.remove <= vif.remove;
                     data_collected.meta_out <= vif.meta_out;
                     data_collected.rank_out <= vif.rank_out;
                     @(posedge vif.clk);  // Espera al siguiente ciclo de reloj para capturar más datos
                     data_collected.num_entries <= vif.num_entries;
                     data_collected.empty <= vif.empty;
                     data_collected.valid_out <= vif.valid_out;
                     data_collected.max_valid_out <= vif.max_valid_out;
                     data_collected.max_meta_out <= vif.max_meta_out;
                     data_collected.max_rank_out <= vif.max_rank_out;
                     data_collected.num_entries <= vif.num_entries;

                     // Clonar y escribir los datos recolectados
                     $cast(data_clone, data_collected.clone());
                     item_collected_port.write(data_clone);
                     num_pkts++;
                 end
             end

             // Proceso de timeout
             begin
                 #100000;  // Esperar 1000 ciclos de reloj
                 `uvm_info(get_type_name(), $sformatf("[MON-R] Error: No se detecto REMOVE dentro del Timeout."), UVM_HIGH);
                 $finish;  // Terminar la simulación si no se detectó `vif.remove`
             end
         join_any

         // Finalizar cuando cualquiera de los procesos en el `fork/join_any` termine
         disable fork;  // Detener el otro proceso cuando uno de los dos termine
     end
endtask: collect_data
    virtual function void report_phase(uvm_phase phase);
       `uvm_info(get_type_name( ), $sformatf("REPORT: COLLECTED PACKETS = %d", num_pkts), UVM_HIGH)
    endfunction: report_phase
```

Figura 45: Tarea collect_data de la clase pifo_monitor_r UVM

El componente *pifo_monitor_r* hereda de la clase base *uvm_monitor* y, al igual que su contraparte de entrada, tiene como objetivo observar pasivamente el comportamiento del DUV, pero en este caso enfocado en el canal de salida. La Figura 44 muestra a diferencia del *pifo_monitor_i*, que detecta intentos de inserción, este monitor está diseñado para capturar los datos generados como resultado de una extracción (*remove*). Para ello, accede a la interfaz virtual pifo_if_r, cuyo nombre se obtiene primero como un *string (monitor_intf_r)* desde la clase *uvm_config_db* y luego se usa como clave para recuperar el *handle* o manejador de la interfaz correspondiente.

En la fase *build*, también se referencian dos objetos *data_packet_r*: El primero (*data_collected*), para almacenar los valores leídos de la interfaz y el segundo, (*data_clone*) como copia que será enviada por el puerto TLM *uvm_analysis_port (item_collected_port)* al *scoreboard* u otros receptores. La lógica principal se implementa en la tarea *collect_data*, ejecutada durante la fase *run_phase*, donde el monitor espera en cada ciclo de reloj a que la señal *remove* se active. Cuando esto ocurre, se capturan múltiples señales del DUV: *meta_out, rank_out, valid_out, empty, num_entries* y, al mismo tiempo, los valores máximos del sistema, lo que permite una reconstrucción completa del estado del módulo en el momento de la extracción. Esta transacción es posteriormente clonada y enviada para análisis. Como mecanismo de robustez, se incluye un proceso paralelo de *timeout* que detiene la simulación si no se detecta una operación de *remove* en un periodo prolongado, ayudando a identificar bloqueos o fallos lógicos. Finalmente, en la fase *report_phase*, se imprime la cantidad total de paquetes observados. Esta estructura, más rica en contenido que su equivalente de entrada, permite no solo reconstruir la transacción de salida, sino también capturar el contexto completo del módulo en ese instante, lo que aporta información crítica para las etapas de validación y de depuración.

- Scoreboard:

La Figura 45 muestra el código correspondiente a la clase *scoreboard*, donde en la primera parte del código se define la estructura inicial del componente *pifo_scoreboard*, que hereda de *uvm_scoreboard*, la clase base que agrupa lógica de comparación o verificación de

resultados en entornos UVM. En esta fase se establecen tres bloques fundamentales: las colas internas, las transacciones temporales y los puertos de análisis.

```
1  import pifo_pkg::*;
2  //definir analysis_imp ports
3  `uvm_analysis_imp_decl(_port_insert)
4  `uvm_analysis_imp_decl(_port_remove)
5  class pifo_scoreboard extends uvm_scoreboard;
6    //Declaracion de colas:
7    bit [META_WIDTH-1:0] pifo_meta [$];
8    bit [RANK_WIDTH-1:0] pifo_rank [$];
9    //Variables Auxiliares
10   bit [META_WIDTH-1:0] meta_expected;
11   bit [RANK_WIDTH-1:0] rank_expected;
12   //Transacciones
13   data_packet_i insert_packet;
14   data_packet_r remove_packet;
15   //Declaracion analysis_imp ports
16   uvm_analysis_imp_port_insert #(data_packet_i, pifo_scoreboard) analysis_imp_insert;
17   uvm_analysis_imp_port_remove #(data_packet_r, pifo_scoreboard) analysis_imp_remove;
18
19   `uvm_component_utils(pifo_scoreboard)
20
21    function new(string name, uvm_component parent);
22       super.new(name, parent);
23    endfunction: new
24
25    function void build_phase(uvm_phase phase);
26       super.build_phase(phase);
27       //crear analysis_imp ports
28       analysis_imp_insert = new("analysis_imp_insert", this);
29       analysis_imp_remove = new("analysis_imp_remove", this);
30       insert_packet = data_packet_i::type_id::create("insert_packet");
31       remove_packet = data_packet_r::type_id::create("remove_packet");
32       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
33    endfunction: build_phase
```

Figura 46: Clase Scoreboard UVM

Primero, se declaran las colas dinámicas *pifo_meta* y *pifo_rank*, que almacenarán temporalmente los metadatos y prioridades de cada transacción insertada. Estas colas representan una representación interna del estado del módulo y son clave para validar posteriormente la secuencia de extracción. En paralelo, se definen dos variables auxiliares (*meta_expected y rank_expected*) que permiten guardar los valores esperados durante las comparaciones con las salidas reales del diseño.

A continuación, se declaran dos transacciones de tipo *data_packet_i* y *data_packet_r*, llamadas *insert_packet* y *remove_packet*, que se utilizan como receptores temporales para las

operaciones que llegan a través de los monitores. Son instanciadas dinámicamente más adelante, dentro de la fase *build_phase*, mediante el método `type_id::create`.

Un aspecto clave del diseño es la utilización de puertos TLM del tipo *uvm_analysis_imp*, concretamente con sufijos personalizados (*_port_insert, _port_remove*) definidos mediante macros. Esto permite conectar directamente los monitores con el *scoreboard*, asignando una función específica (`write_port_insert` *o* `write_port_remove`) que será ejecutada automáticamente cada vez que llegue una nueva transacción.

En la fase *build_phase*, se referencian ambos puertos (*analysis_imp_insert y analysis_imp_remove*), y se inicializan las dos transacciones mencionadas. Este paso garantiza que el *scoreboard* está listo para recibir datos y realizar comparaciones durante la simulación. Finalmente, se imprime un mensaje UVM con prioridad UVM_HIGH para confirmar que la fase de construcción ha finalizado correctamente.

Esta primera parte configura la base lógica y estructural del componente *pifo_scoreboard*, preparando todos los elementos necesarios para su funcionamiento en fases posteriores, tanto en operaciones de inserción como en validación de extracción.

La función *write_port_insert* define cómo el componente *scoreboard* simula internamente el comportamiento del módulo cada vez que se recibe la inserción de una transacción. El objetivo es mantener una representación precisa de cómo deberían organizarse los datos en el diseño, de acuerdo con su política de prioridad (*rank_in*).

Al recibir una transacción desde el *monitor_i* mediante el puerto *analysis_imp_insert*, se evalúan cuatro casos posibles que determinan cómo y dónde insertar el nuevo elemento en las colas internas *pifo_rank* y *pifo_meta*:

- Caso 1: Inserción al inicio

  La Figura 47 muestra la lógica asociada al primer tipo de inserción.

```
// Caso 1: Inserción al inicio del buffer
if((pifo_rank.size() == 0) || (pifo_rank[0] > insert_packet.rank_in)) begin
   pifo_rank.push_front(insert_packet.rank_in);
   pifo_meta.push_front(insert_packet.meta_in);
   `uvm_info(get_type_name, $sformatf("\tPush Front rank[%d] = %p at %0t", pifo_rank.size(), pifo_rank, $time), UVM_MEDIUM)
   `uvm_info(get_type_name, $sformatf("\tPush Front meta[%d] = %p at %0t", pifo_meta.size(), pifo_meta, $time), UVM_MEDIUM)
end
```

Figura 47: Lógica de inserción Caso 1, UVM

Si la cola está vacía, o el primer elemento actual tiene menor prioridad, es decir, su *rank* es mayor, el nuevo dato se inserta al frente. En este caso se ha de realizar una operación de (*push_front*). Este comportamiento emula un sistema que coloca las prioridades más altas con los valores más bajos de *rank* en las primeras posiciones.

- Caso 2: Inserción al final

  La Figura 48 muestra la lógica asociada al segundo tipo de inserción.

```
// Caso 2: Rank mayor prioridad y espacio disponible en la cola
else if(pifo_rank.size() != QUEUE_WIDTH && insert_packet.rank_in > pifo_rank[pifo_rank.size()-1]) begin
   pifo_rank.push_back(insert_packet.rank_in);
   pifo_meta.push_back(insert_packet.meta_in);
   `uvm_info(get_type_name, $sformatf("\tPush Back rank[%d] = %p at %0t", pifo_rank.size(), pifo_rank, $time), UVM_MEDIUM)
   `uvm_info(get_type_name, $sformatf("\tPush Back meta[%d] = %p at %0t", pifo_meta.size(), pifo_meta, $time), UVM_MEDIUM)
end
```

Figura 48: Lógica de inserción Caso 2, UVM

Si la cola aún tiene espacio disponible (`pifo_rank.size() != QUEUE_WIDTH`) y el nuevo dato tiene la prioridad más baja, es decir, *rank_in* mayor que el último de la cola, entonces se coloca al final (*push_back*). Este caso refleja un sistema que conserva el orden si el nuevo dato no afecta al orden actual y no supera en prioridad a ningún elemento.

- Caso 3: Descarte por baja prioridad

  La Figura 49 muestra la lógica asociada al tercer tipo de inserción.

```
1  //Caso 3: Datos no insertados debido a su prioridad dentro de la cola llena
2   else if(pifo_rank.size() == QUEUE_WIDTH && insert_packet.rank_in >= pifo_rank[pifo_rank.size()-1]) begin
3         `uvm_info(get_type_name, $sformatf("\033[31m\tData NOT inserted:\033[0m"), UVM_MEDIUM)
4         `uvm_info(get_type_name, $sformatf("\033[31m\tRank = %p and Meta = %p\033[0m", insert_packet.rank_in, insert_packet.meta_in),
5         `uvm_info(get_type_name, $sformatf("\033[31m\tQueue Maxs:\033[0m"), UVM_MEDIUM)
6         `uvm_info(get_type_name, $sformatf("\033[31m\t--> [%d] rank = %p and meta = %p\033[0m", pifo_rank.size(), pifo_rank[pifo_rank.
7        pifo_meta[pifo_meta.size()-1]), UVM_MEDIUM)
8   end
```

Figura 49: Lógica de inserción Caso 3, UVM

Si la cola ya está llena y el nuevo dato tiene menor prioridad que todos los almacenados, se descarta. Esta condición modela con precisión la política de saturación del módulo, en la que no se permite la inserción de datos que no mejoren la calidad de los que ya están presentes. En este caso, se imprime un mensaje que advierte explícitamente que el dato no fue insertado, junto con la comparación del *rank* y meta con los elementos actuales de menor prioridad.

- Caso 4: Inserción ordenada por prioridad

  La Figura 50 muestra la lógica asociada al la inserción ordenada por *rank*.

```
1  // Caso 4: Inserción Ordenada por Rank
2  else begin
3     foreach(pifo_rank[i]) begin
4        if(insert_packet.rank_in < pifo_rank[i]) begin
5           pifo_rank.insert(i, insert_packet.rank_in);
6           pifo_meta.insert(i, insert_packet.meta_in);
7           break;
8        end
9     end
10    `uvm_info(get_type_name, $sformatf("\tInsert rank[%d] = %p at %0t", pifo_rank.size(), pifo_rank, $time), UVM_MEDIUM)
11    `uvm_info(get_type_name, $sformatf("\tInsert meta[%d] = %p at %0t", pifo_meta.size(), pifo_meta, $time), UVM_MEDIUM)
12 end
```

Figura 50: Lógica de inserción Caso 4, UVM

Si el nuevo *rank_in* debe ir en una posición intermedia según la prioridad, se recorre la cola usando un bucle **`foreach`** hasta encontrar la posición correcta, y se inserta allí con el código descrito en la líneas 5 y 6 Esto mantiene el orden dentro de la cola, garantizando que los elementos con mayor prioridad queden al frente. Es un paso esencial para que el componente *scoreboard* replique con fidelidad el algoritmo de ordenamiento del módulo.

La función *write_port_insert* no solo añade elementos a una cola, sino que implementa la lógica de ordenamiento y aceptación por prioridad que define el comportamiento del

módulo. Gracias a sus cuatro caminos diferenciados, el componente *scoreboard* puede distinguir operando entre inserciones aceptadas y descartadas, controlar el orden de los elementos y simular saturaciones.

La función *write_port_remove*, se activa automáticamente cada vez que el monitor de salida (*pifo_monitor_r*) envía una transacción de tipo *data_packet_r* al *scoreboard* a través del puerto *analysis_imp_remove*. Su propósito es verificar si los datos extraídos por el diseño (*rank_out, meta_out*) coinciden con los que el *scoreboard* había almacenado previamente mediante la función *write_port_insert*. La Figura 51 muestra la lógica asociada a la extracción de elementos de la cola.

```
1  //Remove Transaction
2  virtual function void write_port_remove(data_packet_r remove_packet);
3   // Comprobacion de contenido de las colas
4   if(pifo_rank.size() > 0) begin
5      // Collect Extract Data
6      rank_expected = pifo_rank.pop_front();
7      meta_expected = pifo_meta.pop_front();
8      // Comparacion SCB
9      if(meta_expected == remove_packet.meta_out && rank_expected == remove_packet.rank_out) begin
10        `uvm_info(get_type_name, $sformatf("\t[SCB-PASS] Meta_out expected: %0h,  actual: %0h at %0t",
11        meta_expected, remove_packet.meta_out, $time), UVM_MEDIUM)
12        `uvm_info(get_type_name, $sformatf("\t[SCB-PASS] Rank_out expected: %0h,  actual: %0h at %0t",
13        rank_expected, remove_packet.rank_out, $time), UVM_MEDIUM)
14     end
15     else begin
16        `uvm_info(get_type_name, $sformatf("\t[SCB-FAIL] Meta_out expected: %0h,  actual: %0h at %0t",
17        meta_expected, remove_packet.meta_out, $time), UVM_MEDIUM)
18        `uvm_info(get_type_name, $sformatf("\t[SCB-FAIL] Rank_out expected: %0h,  actual: %0h at %0t",
19        rank_expected, remove_packet.rank_out, $time), UVM_MEDIUM)
20     end
21  end
22  else begin
23     `uvm_info(get_type_name, $sformatf("\t[SCB-FAIL] Empty Queue rank and meta size = %d at: %0t",
24     pifo_rank.size(), $time), UVM_MEDIUM)
25  end
```

Figura 51: Lógica de extracción UVM

La verificación realizada en la función *write_port_remove* comienza asegurándose de que existen elementos en las colas internas del *scoreboard* mediante la condición `if(pifo_rank.size() > 0)` descrita en la línea 4 del código anterior, lo que establece un criterio lógico esencial: si el diseño ha producido una salida, se espera que previamente haya habido al menos una inserción válida registrada. Si esta condición se cumple, el *scoreboard* simula la extracción del primer elemento de ambas colas internas (*pifo_rank y pifo_meta*) usando el método `pop_front()`. De esta manera se obtienen así los valores *rank_expected* y

meta_expected, que representan el contenido que el diseño debería haber entregado según el orden de prioridad establecido. A continuación, se realiza una comparación directa con los valores realmente observados en la transacción de salida (*remove_packet.meta_out* y *remove_packet.rank_out*). Si ambos coinciden, el resultado se considera correcto y se emiten mensajes tipo [SCB-PASS] mostrando los valores esperados y los recibidos, junto con la marca de tiempo de simulación. En caso de discrepancia en cualquiera de los campos, se genera un fallo de validación con mensajes tipo [SCB-FAIL], detallando también la diferencia encontrada. Este mecanismo de comparación cruzada asegura que el módulo no solo esté generando salidas, sino que lo hace respetando el orden lógico de prioridad y el contenido asociado a cada inserción. Por último, si el diseño emite una operación de extracción, pero las colas del *scoreboard* están vacías, se lanza un mensaje [SCB-FAIL] adicional que reporta que la cola interna está vacía, evidenciando una condición funcional errónea. Este tipo de fallo suele indicar un problema en el control interno del DUV, como por ejemplo una señal *empty* mal implementada o una lógica de vaciado incorrecta, y permite detectar errores sutiles en el sistema.

- Agents:

Los agentes *pifo_agent_i* y *pifo_agent_r* son los componentes encargados de encapsular toda la funcionalidad de verificación asociada a los canales de entrada y salida del módulo, respectivamente. Ambos agentes heredan de la clase base *uvm_agent* y siguen una arquitectura idéntica, compuesta por tres subcomponentes: un *sequencer*, un *driver* y un *monitor*. En el caso del agente *pifo_agent_i*, el componente *sequencer* genera transacciones tipo *data_packet_i,* que son aplicadas al diseño por el componente *pifo_driver_i*, mientras que el componente *pifo_monitor_i* observa las inserciones aceptadas y reconstruye las transacciones válidas para enviarlas al componente *scoreboard*. De forma simétrica, el agente *pifo_agent_r* maneja el canal de salida, generando operaciones de extracción a través de *pifo_driver_r* y capturando las respuestas reales con *pifo_monitor_r*. En ambos agentes, los componentes se referencian en la fase *build_phase* mediante el método `type_id::create`, lo que permite una construcción jerárquica limpia, configurable y trazable. Esta estructura modular permite que cada agente actúe como una unidad autónoma dentro del entorno de verificación, lo que facilita su integración, reutilización y mantenimiento.

Además de su estructura jerárquica, el agente UVM implementa la fase *connect_phase*, una etapa clave dentro del ciclo de simulación UVM. En esta fase, se establece la conexión entre el *driver* y el *sequencer*, concretamente mediante la asignación del puerto *seq_item_port* del *driver* al *seq_item_export* del *sequencer* como se puede observar en la línea 31 de la Figura 52. Esta conexión es necesaria para que las secuencias generadas desde el *test* puedan ser transmitidas correctamente hasta el *driver*, que es el encargado de aplicarlas físicamente a la interfaz del diseño.

```
1  import pifo_pkg::*;
2  class pifo_agent_i extends uvm_agent;
3     protected uvm_active_passive_enum is_active = UVM_ACTIVE;
4
5     pifo_sequencer_i sequencer;
6     pifo_driver_i    driver;
7     pifo_monitor_i   monitor;
8
9    `uvm_component_utils_begin(pifo_agent_i)
10       `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
11    `uvm_component_utils_end
12
13    function new(string name, uvm_component parent);
14       super.new(name, parent);
15    endfunction
16
17    function void build_phase(uvm_phase phase);
18       super.build_phase(phase);
19       if(is_active == UVM_ACTIVE) begin
20         sequencer = pifo_sequencer_i::type_id::create("sequencer_i", this);
21         driver = pifo_driver_i::type_id::create("driver_i", this);
22       end
23
24      monitor = pifo_monitor_i::type_id::create("monitor_i", this);
25
26       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
27    endfunction: build_phase
28
29    function void connect_phase(uvm_phase phase);
30       if(is_active == UVM_ACTIVE)
31          driver.seq_item_port.connect(sequencer.seq_item_export);
32          `uvm_info(get_full_name( ), "Connect stage complete.", UVM_HIGH)
33    endfunction: connect_phase
34 endclass: pifo_agent_i
```

Figura 52: Clase pifo_agent_i UVM

- Environment:

En el desarrollo del entorno de verificación UVM para el módulo, se optó por una arquitectura jerárquica dividida en dos entornos parciales: uno para el canal de entrada *pifo_env_i* y otro para el canal de salida *pifo_env_r*. Esta separación responde a un criterio de

modularidad que permite verificar, depurar y escalar cada flujo de datos de forma independiente, respetando el principio de responsabilidad única y facilitando la reutilización de bloques. Ambos entornos encapsulan sus propios agentes y se encargan de conectar los monitores respectivos con el componente *scoreboard*. Posteriormente, se define un entorno superior denominado, *dut_env*, que integra ambos caminos y centraliza las conexiones con el comparador funcional, creando así una estructura robusta, organizada y coherente con los estándares de verificación profesional basados en UVM.

El componente *pifo_env_i* representa un entorno parcial dentro de la arquitectura UVM desarrollada, diseñado para agrupar y coordinar los elementos relacionados exclusivamente con el canal de entrada del módulo. Este entorno hereda de la clase base *uvm_env* y encapsula al agente de entrada *pifo_agent_i*, así como al componente *scoreboard*, estableciendo las conexiones necesarias entre el *monitor_i* del agente y el puerto de análisis de inserciones del *scoreboard*. En su fase de construcción *build_phase*, el entorno referencia ambos componentes mediante el metodo ***type_id::create***, siguiendo las directrices clásicas de modularidad UVM. En la fase de conexión *connect_phase*, establece el vínculo funcional entre el puerto de salida del monitor *mon_analysis_port* y el puerto *analysis_imp_insert* del componente *scoreboard*, garantizando así que todas las transacciones observadas en el canal de entrada se envíen directamente al comparador funcional. Este entorno tiene una estructura simétrica con el componente *pifo_env_r*, que agrupa de manera idéntica el agente *pifo_agent_r* y conecta su *monitor_r* con el puerto *analysis_imp_remove* del *scoreboard*. Esta separación de responsabilidades en entornos parciales permite mantener el diseño jerárquico limpio y modular, facilitando la trazabilidad, reutilización y verificación independiente de cada camino (entrada y salida), y se convierte en una base sólida sobre la cual se construye el entorno completo integrado denominado, *dut_env*.

El componente *dut_env* constituye el entorno jerárquico principal del sistema de verificación UVM y es el encargado de integrar y coordinar todos los bloques funcionales desarrollados: Engloba tanto el canal de entrada *pifo_env_i* como el canal de salida *pifo_env_r*, junto con el componente *scoreboard* que valida la coherencia del comportamiento observado. Esta clase hereda de la clase base *uvm_env* y representa el nivel superior desde el cual se gestionan los estímulos, la observación y la comparación de resultados. Durante la fase de

construcción *build_phase*, el entorno referencia de forma explícita ambos entornos parciales (`pifo_env_i` y `pifo_env_r`) mediante el método `type_id::create`, manteniendo una separación clara de responsabilidades. En la fase de conexión *connect_phase* el entorno, *dut_env* se encarga de enlazar los monitores de entrada y salida con el componente *scoreboard*, estableciendo las rutas de análisis necesarias: el puerto *mon_analysis_port* del monitor de entrada se conecta al puerto *analysis_imp_insert* del *scoreboard*, mientras que el puerto del monitor de salida se enlaza al puerto *analysis_imp_remove*. Esta doble conexión asegura que las transacciones observadas tanto en inserciones como en extracciones son reenviadas al *scoreboard*, donde se contrastan con las expectativas generadas internamente. El diseño de *dut_env* permite, además, una fácil expansión del entorno para futuras pruebas, como escenarios multicanal, validación cruzada o cobertura funcional. En conjunto el componente, *dut_env* actúa como núcleo operativo del entorno UVM, integrando todos los elementos individuales en una estructura coherente, trazable y lista para ejecutar simulaciones completas del módulo. La Figura 53 muestra el código correspondiente al componente *dut_env*.

```
1  import pifo_pkg::*;
2  class dut_env extends uvm_env;
3
4     pifo_env_i       pifo_insert;
5     pifo_env_r       pifo_remove;
6     pifo_scoreboard  scb;
7
8     `uvm_component_utils(dut_env)
9
10    function new(string name, uvm_component parent);
11       super.new(name, parent);
12    endfunction
13
14    function void build_phase(uvm_phase phase);
15       super.build_phase(phase);
16
17       uvm_config_db#(int)::set(this, "pifo_insert.agent_i", "is_active",
18       UVM_ACTIVE);
19       uvm_config_db#(int)::set(this, "pifo_remove.agent_r", "is_active",
20       UVM_ACTIVE);
21
22       uvm_config_db#(string)::set(this, "pifo_insert.agent_i.monitor_i",
23       "monitor_intf_i", "in_intf");
24       uvm_config_db#(string)::set(this, "pifo_remove.agent_r.monitor_r",
25       "monitor_intf_r", "re_intf");
26
27       pifo_insert = pifo_env_i::type_id::create("pifo_insert", this);
28       pifo_remove = pifo_env_r::type_id::create("pifo_remove", this);
29
30       scb = pifo_scoreboard::type_id::create("scb", this);
31
32       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
33    endfunction: build_phase
34
35    function void connect_phase(uvm_phase phase);
36      pifo_insert.agent.monitor.item_collected_port.connect(scb.analysis_imp_insert);
37      pifo_remove.agent.monitor.item_collected_port.connect(scb.analysis_imp_remove);
38      `uvm_info(get_full_name( ), "Connect phase complete.", UVM_HIGH)
39    endfunction: connect_phase
40
41 endclass: dut_env
42
```

Figura 53: Clase dut_env UVM

6.3.Generación y control del estímulo de prueba

Una vez definido el entorno jerárquico de verificación, el siguiente paso es la generación y aplicación del estímulo de prueba al módulo. Este proceso se organiza en dos niveles principales: la definición de *tests* UVM y la inclusión física del diseño e interfaces en el *testbench*. Ambas piezas trabajan en conjunto para lanzar simulaciones, generar transacciones e iniciar la ejecución coordinada del entorno UVM.

6.3.1.Librería de *tests*

En la metodología UVM, es una práctica recomendada la organización de los *tests* en forma de librería de *tests*. Esta estructura se fundamenta en definir una clase base propia llamada *base_test*, que hereda, a su vez de la clase base *uvm_test*, y cuya finalidad es encapsular toda la lógica compartida entre distintos escenarios de verificación. En concreto, esta clase *base_test* se encarga de referenciar el entorno jerárquico completo *dut_env*, registrar los componentes necesarios mediante el uso del mecanismo *factory,* fábrica, UVM (`type_id::create(...)`), y establecer las conexiones con las interfaces virtuales del diseño a través del sistema de configuración *uvm_config_db*. Esta clase actúa como punto central de inicialización, asegurando que todos los *tests* derivados partan de una configuración coherente y lista para ejecutar.

Una vez definida esta clase base, se construye la librería de *tests* mediante la creación de múltiples clases especializadas que heredan de la clase *base_test*. Cada una de estas clases puede redefinir fases específicas del ciclo UVM, como *run_phase*, para implementar un comportamiento concreto, por ejemplo, ejecutar una secuencia dirigida. Esta organización permite mantener una separación clara entre la configuración común gestionada por la clase *base_test* y la lógica particular de cada prueba. Además, facilita la reutilización de código.

Este patrón de diseño, ampliamente recomendado en guías prácticas de UVM, como *Getting Started with UVM* [18], ha sido aplicado rigurosamente en este TFG. Gracias a esta organización, cualquier nuevo *test* puede añadirse fácilmente a partir de la clase *base_test* y escribiendo únicamente el comportamiento específico que se desea verificar.

A continuación, se muestra la implementación concreta de la clase *base_test* desarrollada para este proyecto, que sigue exactamente la estructura anteriormente descrita. En ella, se crea la clase a partir de la clase base de *uvm_test* y se registra el componente en UVM usando el mecanismo *factory* mediante la macro `uvm_component_utils(base_test)`. Dentro de la fase `build_phase`, se crea dinámicamente el entorno completo *dut_env* mediante la función `type_id::create(...)`, lo que permite mantener la jerarquía del sistema. Además, se crea un objeto de tipo `uvm_table_printer`, configurado con una profundidad de impresión `(printer.knobs.depth = 5)` que se utilizará para visualizar la estructura jerárquica del entorno.

Durante la fase `end_of_elaboration_phase` se imprime la topología completa del test usando `this.sprint(printer)`, lo cual permite verificar gráficamente que todos los componentes han sido correctamente creados y conectados antes de comenzar la simulación. Por último, en la fase `run_phase`, se establece un tiempo de drenado (*drain_time*) de 500 unidades de simulación mediante `phase.phase_done.set_drain_time(this, 500);` Este tiempo de drenaje garantiza que cualquier actividad pendiente (como envíos tardíos del monitor o comparaciones en el *scoreboard*) pueda completarse antes de que la simulación finalice. Esta implementación de *base_test* proporciona una base robusta y reutilizable a partir de la cual se pueden construir múltiples clases de *test*, cada una heredando esta estructura y definiendo únicamente su comportamiento específico, como la ejecución de secuencias concretas o la modificación de parámetros del entorno.

A partir de la clase *base_test* se han definido diversos escenarios de prueba que representan casos funcionales concretos sobre el módulo. Estas clases de test heredan directamente de la clase *base_test*, lo que les permite reutilizar toda la configuración del entorno y centrarse exclusivamente en la definición de los estímulos a aplicar. Cada clase redefine la fase *run_phase* para crear, referenciar y lanzar una o varias secuencias (*sequences*) que generan el comportamiento deseado sobre los agentes de entrada y salida del sistema.

Por ejemplo, la Figura 54 muestra el código correspondiente al *test t_i00*, que simula una situación controlada en la que primero se inserta un elemento con la prioridad mínima (`insert_data_min_prio`) seguido por otro con la máxima prioridad

(`insert_data_max_prio`). Ambas secuencias se ejecutan de forma secuencial sobre el *sequencer* del canal de entrada, y su objetivo es verificar si el sistema respeta correctamente el criterio de orden en función del campo *rank*.

```
 //SE INSERTA ELEMENTO MIN PRIO Y SEGUIDO MAX PRIO
class t_i00 extends base_test;
  `uvm_component_utils(t_i00)

  function new(string name, uvm_component parent);
      super.new(name, parent);
   endfunction: new

  function void build_phase(uvm_phase phase);
      super.build_phase(phase);
   endfunction: build_phase

  virtual task run_phase(uvm_phase phase);
    insert_data_min_prio seq_1;
    insert_data_max_prio seq_2;
    super.run_phase(phase);
    phase.raise_objection(this);
    seq_1 = insert_data_min_prio::type_id::create("seq_1");
    seq_2 = insert_data_max_prio::type_id::create("seq_2");
    seq_1.start(env.pifo_insert.agent.sequencer);
    seq_2.start(env.pifo_insert.agent.sequencer);
    phase.drop_objection(this);
  endtask: run_phase
endclass: t_i00
```

Figura 54: Clase test t_i00 UVM

En cambio, el *test t_r00,* cuyo código se proporciona junto a la presente memoria, implementa una prueba combinada, en la que se inserta un único elemento con valores aleatorios (`random_sequence_i`) y, a continuación, se ejecuta una extracción (`remove_data`). Este tipo de prueba permite validar el correcto traspaso de datos a través del diseño y comprobar que la transacción observada en salida coincide con la esperada, como se evalúa en el componente *scoreboard*.

Finalmente, el *test t_ir01* representa un escenario más completo y paralelo, en el que se ejecutan simultáneamente múltiples inserciones aleatorias (`many_random_sequence_i`) y extracciones continuas (`remove_into_empty`), tal y como se muestra en la Figura 55. La ejecución concurrente se logra mediante un bloque *fork-join*, que permite lanzar ambas

secuencias al mismo tiempo sobre sus respectivos sequencers (`env.pifo_insert.agent.sequencer`) y (`env.pifo_remove .agent.sequencer`). Este tipo de *test* es especialmente útil para comprobar el comportamiento del sistema bajo condiciones más realistas y de carga, incluyendo el vaciado completo del buffer.

```
1  class t_ir01 extends base_test;
2    `uvm_component_utils(t_ir01)
3
4    function new(string name, uvm_component parent);
5        super.new(name, parent);
6     endfunction: new
7
8    function void build_phase(uvm_phase phase);
9        super.build_phase(phase);
10    endfunction: build_phase
11
12   virtual task run_phase(uvm_phase phase);
13     many_random_sequence_i seq_i;
14     remove_into_empty seq_r;
15     super.run_phase(phase);
16     phase.raise_objection(this);
17     seq_i = many_random_sequence_i::type_id::create("seq_i");
18     seq_r = remove_into_empty::type_id::create("seq_r");
19     fork
20     seq_i.start(env.pifo_insert.agent.sequencer);
21     seq_r.start(env.pifo_remove.agent.sequencer);
22     join
23     phase.drop_objection(this);
24   endtask: run_phase
25 endclass: t_ir01
```

Figura 55: Clase test_ir01 UVM

6.3.2. Secuencias y generación de transacciones

En UVM, el estímulo hacia el diseño bajo verificación se genera mediante secuencias denominadas *sequences*, que son clases derivadas de la clase base y son las *uvm_sequence* encargadas de construir y enviar transacciones a través del *sequencer*. En este proyecto las secuencias se han organizado en dos librerías diferenciadas: ***`pifo_sequence_i_lib.sv`***, para el canal de entrada, y ***`pifo_sequence_r_lib.sv`***, para el canal de salida. Esta separación permite modularizar los distintos tipos de estímulos y facilita la reutilización y expansión del banco de pruebas.

Cada secuencia define, en su método *body()*, el conjunto de transacciones que se generarán y enviarán al componente *driver*. Estas transacciones son de tipo *data_packet_i* o *data_packet_r* según el canal, y contienen campos como *rank, meta, insert, remove y delay*, todos ellos configurables o aleatorizables. Las secuencias se activan en la fase *run_phase* de los *tests*, a través de llamadas al método **`start(sequencer)`**, definido en la clase base y permiten probar distintos comportamientos del sistema.

Por ejemplo, en el archivo **`pifo_sequence_i_lib.sv`** se encuentran secuencias como **`insert_data_min_prio`**, cuyo código se muestra en la Figura 56, que crea una única transacción con la prioridad mínima (*rank* bajo) e **`insert_data_max_prio`**, que genera un paquete con prioridad máxima. Ambas se utilizan en el test t_i00 para comprobar si el módulo pifo_reg ordena correctamente las inserciones. También se incluyen secuencias aleatorias como **`random_sequence_i`** o **`many_random_sequence_i`**, que generan una o múltiples transacciones con valores completamente aleatorios en cada campo, permitiendo validar el comportamiento general del diseño bajo estímulos no deterministas.

```
1  class insert_data_min_prio extends uvm_sequence #(data_packet_i);
2    `uvm_object_utils(insert_data_min_prio)
3
4    function new(string name = "insert_data_min_prio");
5      super.new(name);
6      req = new("data_packet_i");
7     endfunction:new
8
9    virtual task body();
10     `uvm_do_with(req, {req.rank_in == 16'h0000; req.delay == 2;});
11   endtask: body
12 endclass: insert_data_min_prio
13
14
15 class insert_data_max_prio extends uvm_sequence #(data_packet_i);
16   `uvm_object_utils(insert_data_max_prio)
17
18   function new(string name = "insert_data_max_prio");
19     super.new(name);
20     req = new("data_packet_i");
21    endfunction:new
22
23   virtual task body();
24     `uvm_do_with(req, {req.rank_in == 16'hFFFF; req.delay == 2;});
25   endtask: body
26 endclass: insert_data_max_prio
```

Figura 56: Clase insert_data_min_prio UVM

Por su parte, la librería `pifo_sequence_r_lib.sv` contiene secuencias orientadas a la extracción. La clase `remove_data` genera una transacción de extracción básica, activando la señal *remove* en un instante concreto, mientras que `remove_into_empty` permite aplicar extracciones de forma reiterada, incluso cuando el módulo pueda encontrarse vacío, lo cual es útil para comprobar la robustez del diseño frente a situaciones límite. El código de todas estas clases se distribuye junto con la presente memoria.

Todas las secuencias implementadas utilizan el mecanismo `start_item()` y `finish_item()` para controlar el envío de las transacciones, siguiendo el protocolo estándar, de handshake entre *sequence* y *driver*. En algunos casos se emplea el campo *delay* dentro de la transacción para introducir retardos entre la generación de estímulos, simulando condiciones de latencia o carga variable en el entorno. Esta capacidad de parametrización es clave para lograr una cobertura temporal amplia del diseño y observar su comportamiento bajo diferentes ritmos de operación.

6.4. Ejecución del entorno

La simulación del sistema comienza en el archivo `testbench.sv`, donde se instancian el diseño bajo verificación, las señales de control reloj, *reset* y las interfaces necesarias para la comunicación con el entorno UVM. Estas interfaces se conectan a sus equivalentes virtuales mediante el mecanismo de configuración *uvm_config_db*, lo que permite a los componentes del entorno (*drivers*, monitores y *scoreboard*) interactuar con las señales del diseño sin acoplamiento directo.

La activación del mecanismo de fases de UVM se activa mediante la función `run_test("nombre_del_test")`, una llamada estándar proporcionado por la propia librería UVM. Esta función es la responsable de inicializar todo el sistema de verificación, activar el *test* especificado por su nombre (como *t_i00*, *t_r00 o t_ir01*) y gestionar automáticamente todas las fases de simulación definidas por el *framework*, desde la construcción de los componentes hasta su ejecución y finalización. Al ejecutarse el *test*, se pone en marcha todo el flujo de estimulación, observación y comprobación del diseño: las secuencias generan transacciones que, a través de los *sequencers,* los drivers aplican al diseño bajo verificación (DUV), mientras que los monitores capturan su comportamiento y lo transmiten al componente *scoreboard*,

donde se realiza la validación funcional. Esta interacción se produce de forma sincronizada y controlada, siguiendo la secuencia de fases estándar definida en la arquitectura UVM.

6.5. Simulación con QuestaSim

Para facilitar el proceso de compilación y simulación del entorno de verificación, se ha desarrollado un fichero *Makefile,* como se muestra en la Figura 57, específico para el simulador *QuestaSim*, permitiendo una ejecución controlada, parametrizable y compatible con los flujos profesionales de desarrollo en entornos Linux. Este *Makefile* define una serie de macros y reglas que encapsulan las opciones de configuración de la metodología UVM, las trazas de depuración y los argumentos específicos del simulador, simplificando el lanzamiento de pruebas desde consola.

```
#
# Makefile template for Questa 10.1
#
DEFAULT_TEST    = t_i00
UVM_VERBOSITY   = "UVM_HIGH"
UVM_OPT     = +UVM_VERBOSITY=${UVM_VERBOSITY}
UVM_AWARE_DEBUG = -classdebug \
          -msgmode both             \
          -uvmcontrol=all           \
          -debugDB=questa.dbg           \
          -onfinish stop            \
          +uvm_set_config_int=*,recording_detail,400 \
          +UVM_CONFIG_DB_TRACE          \
          +UVM_OBJECTION_TRACE

# vsim options
VSIM_OPT    = -do "log -r /*;radix hex -showbase"   \
          -voptargs="+acc"          \
          -sv_seed 1

vlib: clean
    vlib work

comp:   vlib
    vlog                        \
        -timescale 1ns/1ns          \
        +incdir+.               \
        design.sv               \
        testbench.sv

run: comp
    vsim                        \
        top                 \
        -c -l top.log               \
        ${UVM_AWARE_DEBUG}          \
        ${UVM_OPT}              \
        ${VSIM_OPT}             \
        -do "run -all ; q"          \
        +UVM_TESTNAME=${DEFAULT_TEST}

rung:   comp
    vsim                                            \
                top                                     \
                -l top.log                              \
                ${UVM_AWARE_DEBUG}                      \
                ${UVM_OPT}                              \
                ${VSIM_OPT}                             \
                -do "run 0; do wave.do; run -all; q"    \

```

Figura 57: Archivo de configuración Makefile

En la primera parte del archivo, se definen varias macros clave. La variable `DEFAULT_TEST` especifica el nombre del *test* por defecto que se ejecutará si no se indica otro, mientras que `UVM_VERBOSITY` controla el nivel de detalle de los mensajes (`UVM_HIGH`, `UVM_MEDIUM`, etc.). A partir de ella se construye la variable `UVM_OPT`, que se pasa como argumento al simulador en formato `+UVM_VERBOSITY=...`, activando así el sistema de *log* del entorno. Por otro lado, la macro `UVM_AWARE_DEBUG` agrupa múltiples opciones avanzadas para depuración y trazabilidad: `-classdebug` permite inspeccionar estructuras de clases en tiempo de simulación, `-msgmode both` activa la impresión simultánea por consola y fichero, `-debugDB=questa.dbg` habilita la base de datos de depuración, y `+UVM_CONFIG_DB_TRACE` junto con `+UVM_OBJECTION_TRACE` activan la traza de configuraciones y objeciones, facilitando el seguimiento del flujo UVM.

Además, se define la macro `VSIM_OPT`, que incluye argumentos específicos para la ejecución del simulador *vsim* QuestaSim. Entre ellos destacan el argumento `-do` con comandos para abrir la interfaz gráfica, cargar configuraciones de visualización (`wave.do`), ejecutar la simulación `run -all` y salir automáticamente (`q`). También se activan argumentos como `-sv_seed 1` (semilla de aleatorización reproducible) y `+acc,` que permite aumentar la visibilidad de las variables de las señales del diseño.

La segunda parte del código implementa la regla "*rung*", que es la encargada de ejecutar la simulación en modo gráfico. Esta regla depende de una compilación previa definida en la regla `comp` y lanza la herramienta *vsim* con todos los argumentos definidos por las macros anteriores. Se simula el diseño definido como modulo principal *top-level (top)*, se registra la salida en el fichero `top.log`, y se pasa la configuración UVM extendida junto a los comandos interactivos que controlan la ejecución gráfica (`do wave.do; run -all; q`). De esta forma, el entorno se ejecuta con trazabilidad completa y permite al usuario visualizar la evolución de señales, aplicar *breakpoints* o analizar el comportamiento temporal del diseño. Este enfoque permite ejecutar una simulación completa con un único comando: `make rung`

A lo largo de este capítulo se ha detallado la construcción del entorno de verificación basado en UVM, su estructura jerárquica y el flujo completo de ejecución, desde la generación del estímulo hasta la comprobación automática de resultados. La adopción de una librería de

test organizada, el uso de secuencias predefinidas, y la incorporación de un *Makefile* específico para el simulador *QuestaSim* han permitido establecer un sistema robusto, reproducible y escalable. Esta infraestructura no solo cumple con las buenas prácticas de verificación modernas, sino que sienta las bases para futuras extensiones del proyecto.

7º Capítulo: Definición e integración de propiedades en UVM

La integración de *SystemVerilog Assertions* (SVA) dentro de entornos UVM representa el eje central de este proyecto. Aunque las SVA son una herramienta poderosa para expresar y comprobar propiedades temporales directamente sobre señales del diseño, su uso dentro de UVM plantea retos técnicos importantes: no pueden vivir dentro del entorno de clases, requieren visibilidad estructural directa y, por defecto, no están sincronizadas con las fases del *testbench*. Resolver estas limitaciones de forma limpia, modular y mantenible ha sido el objetivo principal del trabajo.

Este capítulo se centra en aplicar, analizar y contrastar las dos metodologías propuestas en el artículo de Verilab *"SVA Encapsulation in UVM: Enabling Phase and Configuration Aware Assertions"*.[7] Estas metodologías permiten encapsular *assertions* de manera que sean configurables y compatibles con la arquitectura por fases de UVM. La implementación y evaluación de estas soluciones dentro del entorno desarrollado constituye la aportación principal del proyecto, no solo a nivel técnico, sino también metodológico, ya que propone una forma escalable y profesional de integrar propiedades formales en flujos de verificación avanzados.

7.1. Análisis de la solución propuesta por Verilab

El artículo *"SVA Encapsulation in UVM: Enabling Phase and Configuration Aware Assertions"*, desarrollado por *Verilab*, plantea una solución formal y estructurada a uno de los problemas más relevantes en la verificación moderna basada en UVM: cómo integrar *SystemVerilog Assertions* de forma limpia, escalable y sincronizada dentro de un entorno de verificación orientado a clases. En particular, este artículo pone el foco en tres desafíos prácticos que, hasta su publicación, se resolvían habitualmente mediante soluciones ad-hoc, poco mantenibles o con elevado acoplamiento entre el diseño, la interfaz y el entorno de verificación.

- Problema: incompatibilidad entre el mundo estructural y el mundo de clases

El núcleo del problema es sencillo, pero estructural: En *SystemVerilog*, las SVA concurrentes deben declararse dentro de bloques *module* o *interface*, ya que son elementos de simulación orientados al tiempo y al contexto físico. En contraste, UVM está construido sobre clases *uvm_component*, las cuales no pueden contener directamente SVA concurrentes. Esto obliga a separar las propiedades del entorno de verificación, situándolas en módulos o interfaces accesibles desde el *testbench*. El resultado es una fragmentación de la estructura que conlleva varios efectos secundarios:

- Duplicación de información: las señales, variables o *flags* necesarios por las *assertions* deben definirse o copiarse en la interfaz.
- Falta de encapsulación: las *assertions* quedan disociadas del componente o funcionalidad a la que pertenecen.
- Pérdida de trazabilidad: cuando una *assertion* falla, resulta difícil relacionarla con el estado de configuración del *test* o con la fase de verificación activa.
- Sincronización manual: los datos necesarios para activar o parametrizar la *assertion* deben pasarse a mano desde el monitor o el entorno, con riesgo de desincronización o error humano.

Estos problemas, frecuentes en la práctica, se agravan en entornos grandes, con múltiples configuraciones, *assertions* reusables o varias instancias del diseño bajo prueba.

- Filosofía del enfoque Verilab: recuperar el control desde UVM

El enfoque de *Verilab* parte de un objetivo claro: devolver a UVM el control sobre las SVA. La idea no es mover las *assertions* dentro de las clases (lo cual no es posible), sino estructurar su uso de manera que puedan ser configuradas, activadas y monitorizadas desde el entorno UVM, sin perder sus propiedades formales ni su potencia como mecanismo de verificación.

Para ello, el artículo de referencia establece tres principios fundamentales que deben cumplir las SVA integradas en un entorno moderno de verificación:

1. Modularidad: las *assertions* deben estar encapsuladas en un bloque lógico separado y reutilizable, evitando contaminar las interfaces funcionales con lógica de verificación.
2. Configurabilidad: debe ser posible activar, desactivar o parametrizar las *assertions* desde el entorno de clases, preferiblemente utilizando la clase base *uvm_config_db* o algún otro mecanismo de paso de información jerárquico.
3. Conciencia de fase (*Phase-awareness*): las *assertions* deben activarse o adaptarse en la fase adecuada del ciclo de simulación UVM, garantizando que el entorno esté listo y que los valores de configuración sean válidos y estables.

A partir de estos objetivos, el artículo propone dos estrategias complementarias para implementar *assertions* que cumplen con estos requisitos: La primera, basada en un mecanismo de configuración manual (API) y, la segunda, basada en un componente automático, embebido y *fase-aware*.

- Caso base: *assertions* embebidas en interfaces funcionales

Antes de presentar sus soluciones, el artículo describe el enfoque que muchas veces se toma por defecto: definir las *assertions* directamente dentro de la interfaz funcional del diseño, por ejemplo *my_if*, mezclando señales de control, lógica RTL y propiedades formales. Aunque funcional, este enfoque tiene varios inconvenientes:

1. Violación del principio de separación de responsabilidades: la interfaz deja de ser una simple abstracción de conexión física y pasa a contener lógica de verificación.
2. Dificultad de mantenimiento: si se quiere reutilizar la interfaz en otros contextos o eliminar las *assertions* temporalmente, se requiere modificar código fuente sensible.
3. Configuración externa forzada: la única forma de controlar el comportamiento de las assertions es declarando variables adicionales dentro de la interfaz, por ejemplo, bit `checks_enable` y asignándolas externamente.

A partir de esta situación inicial, el trabajo justifica la necesidad de externalizar y encapsular las *assertions*.

- Primer objetivo: encapsular las *assertions* fuera de la interfaz funcional

El primer paso clave que propone *Verilab* es mover las *assertions* a una interfaz de verificación separada, por ejemplo `my_checker_if`, que se pueda referenciar dentro de la interfaz funcional, pero cuyo contenido esté dedicado exclusivamente a lógica de verificación. Esta separación permite; reutilizar las *assertions* en distintos entornos, activarlas o desactivarlas mediante variables internas (`bit checks_enable`) y conservar la visibilidad estructural necesaria para que las SVA funcionen.

No obstante, esta estructura sigue siendo estática y para permitir su control dinámico desde UVM es necesario introducir mecanismos que hagan de puente entre el entorno de clases y la interfaz física.

Segundo objetivo: comunicar el entorno de clases con las *assertions*

Aquí es donde el trabajo introduce sus dos métodos de encapsulación:

1. El enfoque API: define métodos, por ejemplo (`set_config(...)`) que se pueden invocar desde componentes UVM para copiar valores del entorno a variables internas de la interfaz que usa las SVA. El punto clave es que estas llamadas se hacen en fases bien definidas tal es el caso de fases como (`end_of_elaboration_phase` o `start_of_simulation`), y que el monitor o test debe tener acceso a la interfaz a través de la clase base *`uvm_config_db`*.

2. El enfoque automático y fase-aware: incorpora un componente `uvm_component` directamente en la interfaz de verificación (`my_checker`). Este componente participa en el ciclo de fases como cualquier otro, accede a la clase base `uvm_config_db` desde su fase `build_phase o end_of_elaboration_phase`, y copia las configuraciones internas que luego serán utilizadas por las *assertions*. Así, no requiere intervención manual externa ni riesgo de que se olviden pasos críticos.

- Implicaciones prácticas del enfoque

Ambos enfoques tienen ventajas y limitaciones. El enfoque API es más simple, más explícito y fácil de depurar, pero depende del usuario para que las llamadas se hagan correctamente. El enfoque automático es más robusto, más limpio desde el punto de vista

arquitectónico y escalable a múltiples instancias, pero requiere una integración más profunda con el entorno UVM y un uso más cuidadoso de las jerarquías y configuraciones.

El valor principal del artículo de referencia no está solo en proponer una solución, sino en formalizar un problema que hasta entonces se resolvía de forma poco profesional, y en proporcionar una guía metodológica concreta para adoptar buenas prácticas en entornos de verificación complejos.

7.2. Metodología 1: Encapsulación mediante API

En una primera fase del proyecto se optó por implementar la integración de *SystemVerilog Assertions* (SVA) dentro del entorno UVM mediante la metodología API descrita en el *paper* de Verilab. Esta aproximación propone una conexión explícita entre el entorno de clases UVM y la interfaz que contiene las *assertions*, utilizando funciones públicas que actúan como un punto de entrada para pasar información desde el *testbench* al dominio estructural.

El enfoque se basa en encapsular las SVA dentro de una interfaz auxiliar llamada *pifo_sva_checker_i*, dedicada exclusivamente a la lógica de verificación. Esta interfaz se referencia dentro de la interfaz funcional *pifo_if_i*, de forma que dispone de visibilidad directa sobre todas las señales necesarias del diseño (*insert, rank_in, meta_in, full, num_entries*, etc.). Para mantener un control global sobre la activación de las *assertions*, se declara una variable interna bit *checks_enable_i* y una función pública `set_check_enable_i(bit en)`, que permite habilitarlas o deshabilitarlas desde el exterior. La Figura 58 muestra el código correspondiente a la definición de dicha variable, así como de la función.

```
1  bit checks_enable_i;
2
3  function void set_check_enable_i(bit en);
4    checks_enable_i = en;
5  endfunction
```

Figura 58: Integración variable checks_enable_i método API

Esta señal *checks_enable_i* se utiliza dentro de las propiedades mediante el bloque `disable iff (!checks_enable_i)`, lo que permite activar o desactivar toda la lógica formal en función de esta variable sin necesidad de modificar cada *assertion* individualmente.

Activación desde el entorno UVM, para comunicar esta interfaz con el entorno de clases, se sigue el patrón API: desde el monitor, en su fase *end_of_elaboration_phase*, se accede a la interfaz virtual del *checker (vif)* y se llama directamente a la función de activación, `vif.set_check_enable_i(checks_enable_i);`

El valor de la varible *checks_enable_i* se obtiene previamente desde la clase base *`uvm_config_db`* durante la fase *build_phase*, permitiendo que sea configurado de forma centralizada desde el test o desde la clase *base_test*. En efecto, en la clase *base_test* se realiza la asignación global del *flag*, mediante el código `uvm_config_db#(bit)::set(this, "*", "checks_enable_i", 1);`

Esto permite que todas las referencias del monitor que accedan a la clase *uvm_config_db* con esa etiqueta reciban el mismo valor, manteniendo la coherencia de activación entre distintos *tests*.

Este enfoque ha demostrado ser simple, efectivo y perfectamente funcional en una primera fase de desarrollo. Ha permitido incorporar *assertions* en el entorno sin modificar la arquitectura general del *testbench*, y mantener su activación completamente bajo control desde UVM. Las principales ventajas observadas han sido:

- Separación limpia entre lógica de verificación y entorno funcional.
- Activación centralizada desde el test o entorno base (*base_test*).
- Modularidad y encapsulamiento, permitiendo escalar el checker fácilmente.

Sin embargo, esta metodología también presenta limitaciones estructurales importantes:

- Dependencia del usuario: si el programador olvida invocar la función *set_check_enable_i()*, las *assertions* no se activan y la simulación puede arrojar resultados incorrectos o engañosamente correctos.
- Falta de automatismo: cada componente debe gestionar manualmente cuándo y cómo realizar la llamada.

- Rigidez jerárquica: la solución requiere acceso explícito a la interfaz y conocimiento de su estructura, lo que dificulta su uso con múltiples instancias.

7.3. Metodología 2: Encapsulación mediante *Automatic* y *Phase-Aware*

El segundo enfoque propuesto en el *paper* de *Verilab*, plantea una solución avanzada para integrar *SystemVerilog Assertions* (SVA) dentro de un entorno UVM, resolviendo las limitaciones prácticas del enfoque API. Esta metodología se basa en dos pilares fundamentales: la configuración automática de las *assertions* desde dentro del propio entorno UVM y la sincronización consciente de las fases del ciclo de simulación *phase-awareness*. El resultado es una solución que no solo elimina la intervención manual, sino que garantiza que las *assertions* se activen únicamente cuando el entorno está listo y la configuración es válida.

El método basado en API, aunque funcional, introduce una dependencia directa del usuario: si este olvida llamar a la función que activa las SVA (`set_check_enable()`), la verificación puede quedar incompleta sin indicios evidentes. Además, obliga a que otros componentes UVM, como los monitores, accedan explícitamente a la interfaz *checker*, lo cual rompe el principio de encapsulación.

La metodología *phase-aware* elimina por completo esta fragilidad al convertir el checker en un elemento activo del entorno UVM, capaz de gestionarse y configurarse a sí mismo de forma autónoma. Esto se logra insertando dentro de la interfaz que contiene las *assertions* un componente `uvm_component` especializado, que participa de manera nativa en el ciclo de fases del entorno. Esta estrategia se basa en tres elementos principales:

- Encapsulación estructural: Las *assertions* se agrupan en una interfaz independiente, por ejemplo, *pifo_sva_checker_if*, separada del diseño funcional, donde también residen variables internas de control como *checks_enable*, que habilita/deshabilita la comprobación.

- Componente embebido: Dentro de esa misma interfaz se referencia un componente que hereda de, la clase base *uvm_component*, registrado con la macro *uvm_component_utils*, que forma parte del árbol jerárquico de UVM. Este componente

tiene acceso completo a las variables internas de la interfaz, y se convierte en el puente entre la clase *uvm_config_db* y las SVA.

- Acceso phase-aware a la configuración: El componente embebido accede al *uvm_config_db* en una fase concreta, normalmente *build_phase* o *end_of_elaboration_phase*, obteniendo los valores definidos por el test o la clase base *base_test*. Copia esos valores en las variables internas de la interfaz, asegurando así que las *assertions* recibirán los parámetros correctos antes de entrar en la fase *run_phase*, que es cuando se activan y comienzan a evaluarse.

Una de las claves de esta metodología es que el componente embebido se comporta como cualquier otro bloque UVM:

- Durante *build_phase*: accede a la clase `uvm_config_db` y registra los valores de configuración necesarios.
- Durante la fase end_of_elaboration_phase: puede realizar trazas, comprobaciones o ajustar señales de control.
- Durante la fase run_phase: no es necesario que actúe, ya que las SVA ya disponen de los valores correctos.

Este comportamiento *phase-aware* asegura que el *checker* está preparado justo a tiempo, sin activar *assertions* antes de estar debidamente configuradas, y sin requerir interacción adicional desde el entorno. Las ventajas metodológicas de este enfoque ofrecen una serie de beneficios clave en comparación con el método API:

- Autonomía total: el *checker* se configura solo, sin que ningún otro componente tenga que preocuparse de activarlo.
- Sincronización precisa: los valores se leen en la fase adecuada del ciclo UVM, eliminando el riesgo de condiciones de carrera o evaluaciones incorrectas.
- Modularidad absoluta: se puede replicar el *checker* para múltiples instancias del diseño sin modificar su lógica ni su activación.

- Escalabilidad: es ideal para entornos complejos con múltiples interfaces, configuraciones por test o jerarquías profundas.

Además, este enfoque restaura el principio de encapsulación, ya que el entorno de verificación no necesita saber cómo está construido el *checker* ni cómo se activan sus *assertions*. Solo necesita colocar en la clase base *uvm_config_db* los valores adecuados y dejar que el *checker* se encargue del resto.

7.4. Implementación de las metodologías en el entorno

En este apartado se exponen las implementaciones realizadas en el presente Trabajo Fin de Grado de ambas metodologías.

7.4.1.Implementación de la metodología 1: API manual

En una primera fase de la implementación se adoptó el enfoque más directo para encapsular *assertions* dentro de UVM: la metodología basada en una API manual. Esta estrategia requiere que un componente UVM, en este caso el componente monitor, acceda directamente a la interfaz que contiene las *assertions* (*pifo_sva_checker_i*) y realice la configuración necesaria mediante llamadas explícitas.

En la estructura de la interfaz se definió una interfaz auxiliar denominada *pifo_sva_checker_i*, que contiene todas las propiedades SVA asociadas al canal de entrada. Esta interfaz se conecta estructuralmente con la interfaz funcional *pifo_if_i*, referenciándola de forma directa mediante el método `pifo_sva_checker_i checker_i();`

Dentro de la clase *pifo_sva_checker_i*, se declaró una variable interna tipo bit, denominado `checks_enable_i` y una función pública, denominada `set_check_enable_i(bit en),` que permite habilitar o deshabilitar globalmente las *assertions*.

En el componente *pifo_monitor_i* se incluyó una lógica en la fase *end_of_elaboration_phase* para acceder a la interfaz y activar las *assertions*. Esto se logró recuperando el valor *checks_enable_i* desde la base *uvm_config_db*, y llamando después a la función expuesta por la interfaz, tal y como se muestra en la Figura 59.

```
1  //!API
2    if(!uvm_config_db#(bit)::get(this,"","checks_enable_i",checks_enable_i))
3      `uvm_fatal("NO_BIT_EN_i", {"Need bit enable for: ", get_full_name( ), ".checks_enable_i"})
4
5     `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
6  endfunction: build_phase
7
8  //!API
9  function void end_of_elaboration_phase(uvm_phase phase);
10    vif.set_check_enable_i(checks_enable_i);
11 endfunction: end_of_elaboration_phase
```

Figura 59: Implementación método API

Esta llamada conecta el entorno de clases UVM con la lógica estructural de las SVA, trasladando la configuración deseada al *checker* de manera explícita.

En la clase *base_test* se realiza la asignación del valor de control global mediante la llamada al método set de la clase *uvm_config_db*, `uvm_config_db#(bit)::set(this, "*", "checks_enable_i", 1).`

Esto permite que cualquier componente que acceda a la base *uvm_config_db* utilizando la etiqueta "`checks_enable_i`" reciba el valor adecuado. Gracias a esto, todos los *tests* que hereden del test base *base*_test activan automáticamente las *assertions* sin tener que redefinir la configuración.

Esta implementación cumple perfectamente con la metodología API descrita por *Verilab*. La lógica de activación queda centralizada, el *checker* mantiene su encapsulación, y la activación se realiza en el momento correcto del ciclo UVM. No obstante, como se señaló en el análisis metodológico, esta estrategia requiere que el monitor recuerde siempre realizar la llamada manual, lo que introduce una dependencia externa que puede generar errores si se omite.

7.4.2. Implementación de la metodología 2: *Automatic* y *Phase-Aware*

En la segunda fase del proyecto se aplicó la metodología avanzada descrita en el *paper* de *Verilab*, basada en la configuración automática y sincronizada de las *assertions*, eliminando toda dependencia manual desde el entorno. Esta implementación representa un salto de calidad respecto al enfoque API, ya que logra una integración limpia, autónoma y completamente alineada con el ciclo de fases del entorno UVM.

Al igual que en la metodología anterior, se encapsularon las *assertions* del canal de entrada en la interfaz *pifo_sva_checker_i*. Esta se referenció dentro de la interfaz funcional **pifo_if_i**, permitiendo acceder a todas las señales relevantes del diseño usando el método **pifo_sva_checker_i checker_i().**

La interfaz contiene la variable de control **checks_enable_i**, que actúa como bandera global para habilitar o deshabilitar las *assertions*.

Siguiendo la metodología *Verilab*, se creó una clase *checker_phaser* que hereda de la clase base *uvm_component* y se referenció directamente dentro de la propia interfaz *pifo_sva_checker_i*. Esta clase fue registrada con la macro *uvm_component_utils*, lo que le permite participar como cualquier otro componente del entorno UVM. La referencia del componente se realizó mediante la sentencia **checker_phaser m_phase = new($psprintf("%m.m_phase_i"))**, que invoca al constructor de la misma.

El uso de "%m" genera un identificador único y jerárquico para cada referencia del módulo *checker*, garantizando la trazabilidad en entornos con múltiples *checkers* o interfaces. En la fase *end_of_elaboration_phase*, el componente *checker_phaser* accede a la base *uvm_config_db* para recuperar el valor de activación de las *assertions*. Esto se muestra en la Figura 60.

```
1  if (!uvm_config_db#(bit)::get(this, "", "checks_enable_i", checks_enable_i))
2    `uvm_fatal("NO_BIT_EN", {"Need bit enable for: ", get_full_name(), ".checks_enable_i"});
```

Figura 60: Implementación método Automatic y Phase-Aware

El valor obtenido a través del método **get**, se guarda directamente en la variable interna de la interfaz (*checks_enable_i*), la cual es utilizada por las *assertions* a través de bloques **disable iff (!checks_enable_i)** para controlar su evaluación.

Desde el entorno UVM, la configuración se realiza una única vez en la clase *base_test*, utilizando una ruta jerárquica parcial que coincida con el nombre generado dinámicamente. En este caso, se invoca al método **set** de la clase *uvm_config_db*, utilizando el siguiente formato: **uvm_config_db#(bit)::set(null, "*m_phase_i", "checks_enable_i", 1).**

Debido al uso del comodín "`*m_phase_i`", no es necesario conocer la ruta completa de cada *checker*. Esto permite que cualquier referencia correctamente registrada en el entorno UVM reciba su configuración automáticamente, sin interacción manual por parte del *test* o el monitor. Por tanto la implementación realizada refleja con fidelidad los principios del enfoque *phase-aware*:

- El checker es autónomo: no necesita que otros componentes lo configuren.
- La configuración se realiza en la fase correcta: antes de que las *assertions* entren en evaluación.
- No hay dependencias externas ni llamadas API: todo el proceso ocurre dentro del *checker*.
- Se mantiene el encapsulado: el entorno UVM no necesita conocer la estructura interna del *checker*.

Esta solución establece una base sólida para el desarrollo de entornos de verificación altamente escalables y profesionalmente estructurados. Al permitir que cada instancia del *checker* se configure de forma autónoma y sincronizada con el ciclo de simulación, se elimina por completo la necesidad de intervención manual o conocimiento de la jerarquía interna del entorno, lo que reduce los posibles errores.

En entornos reales de verificación donde pueden coexistir múltiples interfaces, configuraciones específicas por test o distintos modos de operación, este enfoque garantiza una gestión consistente y automática de las *assertions*, sin duplicación de lógica ni dependencias rígidas entre componentes.

7.5.Nuevas propiedades SVA integradas en UVM

Como parte del desarrollo de este entorno de verificación, se han implementado múltiples propiedades SVA destinadas a validar de forma precisa el comportamiento del módulo. Estas *properties* están distribuidas de manera modular entre dos interfaces: una orientada a la verificación de operaciones de entrada *pifo_sva_checker_i* y otra dedicada a las señales de salida *pifo_sva_checker_r*.

7.5.1.Propiedades en la interfaz de entrada

Esta interfaz observa las señales implicadas en las operaciones de inserción, y las propiedades aquí implementadas tienen como objetivo garantizar que el módulo opera correctamente cuando recibe solicitudes de entrada. A continuación se detallarán las nuevas propiedades que han sido implementadas:

- Propiedad 1: Desactivación de *full* tras un *reset*

Aquí se comprueba que, cuando se activa la señal de reinicio, el sistema debe limpiar la señal full. Es decir, tras un *reset*, no debe considerarse que el registro de prioridad está lleno. Esta propiedad garantiza que el módulo comience siempre en estado no lleno después de una inicialización. La Figura 61 muestra el código implementado de la propiedad.

```
// When reset, full must be clear
  property reset_full;
    disable iff (!checks_enable_i)
    @(posedge clk) rst |=> !full; //si rst activo en el siguiente flanco de reloj !full
  endproperty: reset_full

  ap_reset_full: assert property (reset_full)
    $display("At %d  PROP. OK. - reset_full", $time);
    else $error("ERROR ap_reset_full");
```

Figura 61: Propiedad 2 UVM, reset_full

- Propiedad 2: Reinicio desactiva *max_valid_out*

Esta propiedad comprueba que la señal *max_valid_out* se desactiva inmediatamente después del *reset*. En otras palabras, no debe marcarse como válido ningún dato máximo tras un reinicio. Con ello se asegura que no se expongan valores inválidos como salidas útiles al sistema exterior tras un *reset*. La Figura 62 presenta el código correspondiente a la implementación de la propiedad.

```
// When reset, max_valid_out must be clear
property reset_max_valid_out;
  disable iff (!checks_enable_i)
  @(posedge clk) rst |=> !max_valid_out; // si rst activo en el siguiente flanco de reloj max_valid_out debe estar desactivado
endproperty: reset_max_valid_out

ap_reset_max_valid_out: assert property (reset_max_valid_out)
  $display("#### SI #### At time %d, reset_max_valid_out OK", $time);
  else $error("ERROR ap_reset_max_valid_out");
```

Figura 62: Propiedad 3 UVM, reset_max_valid_out

- Propiedad 3: Activación de *max_valid_out* al insertar un nuevo máximo

Esta propiedad asegura que, si se inserta un dato cuyo rango y metadatos coinciden con el nuevo máximo calculado, entonces *max_valid_out* debe activarse. Así se garantiza que el sistema comunica correctamente cuándo ha cambiado el elemento de menor prioridad. En la Figura 62 se observa el código utilizado para implementar la propiedad.

```
1  // max_valid_out is set two cycles after insert
2  property set_max_valid_out;
3    @(posedge clk) disable iff (rst || !checks_enable_i) //Desactiva la propiedad si rst esta activo
4    $rose(insert) |-> ##2 max_valid_out;
5  endproperty: set_max_valid_out
6
7  ap_set_max_valid_out: assert property (set_max_valid_out)
8    $display("#### SI #### At time %d, set_max_valid_out OK", $time);
9    else $error("At time %d ERROR set_max_valid_out", $time);
```

Figura 63: Propiedad 3 UVM, set_max_valid_out

- Propiedad 4: Actualización del contador *num_entries* tras inserción

Esta propiedad se comprueba que el contador de entradas aumente en uno tras una inserción válida. No se deben producir errores de conteo si *insert* está activo y el sistema no está lleno. Esta propiedad asegura la integridad del seguimiento del número de elementos almacenados. La Figura 64 muestra la implementación de la propiedad mediante código.

```
1  //num_entries ++ when insert
2  property num_entries_update;
3  @(posedge clk) disable iff ( rst || !checks_enable_i)
4    $rose(insert) |=> num_entries == $past(num_entries, 1) + 1;
5  endproperty: num_entries_update
6
7  ap_num_entries_update: assert property (num_entries_update)
8      $display("#### SI #### At time %d, num_entries_update OK", $time);
9      else $error("At time %d ERROR num_entries_update", $time);
```

Figura 64: Propiedad 4 UVM, num_entries_update

- Propiedad 5: Prohibición de inserción si *full* está activo

Esta propiedad evita que se realicen inserciones cuando el sistema ha alcanzado su capacidad máxima (*full* = 1). Con ello se impide cualquier condición de desbordamiento que

pueda comprometer el estado interno del registro. En la Figura 65 se observa el código utilizado para implementar la propiedad.

```
1  //Detecta fallo si se cumple que cuando full hay un insert
2  property insert_when_full;
3      @(posedge clk) disable iff (rst || !checks_enable_i)
4        full |-> (insert);
5  endproperty: insert_when_full
6
7  ap_insert_when_full: assert property (insert_when_full)
8      $display("#### SI #### At time %d, Insert when full detected", $time);
```

Figura 65: Propiedad 5 UVM, insert_when_full

- Propiedad 6: Condición de activación del *flag full*

Se verifica que la señal *full* únicamente se active cuando el contador de entradas ha alcanzado su valor máximo **(*2**L2_REG_WIDTH*).** Esta propiedad aporta coherencia interna entre la señal de llenado y el estado real del sistema. La Figura 66 muestra el código implementado de la propiedad.

```
1  property full_requires_max_entries;
2      @(posedge clk) disable iff (rst || !checks_enable_i)
3      full |-> (num_entries == 16);
4  endproperty: full_requires_max_entries
5
6  ap_full_requires_max_entries: assert property (full_requires_max_entries)
7      $display("#### SI #### At time %d, full_requires_max_entries OK", $time);
8      else $error("At time %0t: ERROR - full is active, but num_entries is not 16.
9      Current num_entries =
   ", $time, num_entries);
```

Figura 66: Propiedad 6 UVM, full_requires_max_entries

- Propiedad 7: Duración de pulso de *insert*

Se comprueba que, después de una inserción, la señal *insert* debe desactivarse en el siguiente ciclo. Esto evita que un solo pulso se interprete erróneamente como múltiples inserciones. Esta propiedad previene errores debidos a señales sostenidas más allá de un ciclo de reloj, lo cual podría generar múltiples actualizaciones involuntarias. La Figura 67 muestra la implementación de la propiedad mediante código.

```
property insert_period_check;
  @(posedge clk) disable iff (rst || !checks_enable_i)
    insert |-> (($past(!insert,1)) || !insert);
endproperty: insert_period_check

ap_insert_period_check: assert property(insert_period_check)
    else $error("At time %0t: ERROR - Insert timing violated at:",$time);
```

Figura 67: Propiedad 6 UVM, insert_period_check

7.5.2. Propiedades en la interfaz de salida

Esta interfaz observa las señales implicadas en las operaciones de eliminación y las *properties* aquí implementadas tienen como objetivo garantizar que el módulo opera correctamente cuando se solicitan extracciones de datos. A continuación se detallan las propiedades que han sido implementadas:

- Propiedad 1: Activación de *empty* tras un *reset*

Esta propiedad verifica que, al activarse el *reset*, la señal *empty* se activa en el siguiente ciclo de reloj. Es decir, el sistema debe comenzar en estado vacío, garantizando una inicialización coherente del sistema, asegurando que no se interprete la presencia de datos inexistentes tras el reinicio. La Figura 68 detalla el fragmento de código empleado para la implementación de la propiedad.

```
// When reset, empty must be set
property reset_empty;
  @(posedge clk) disable iff (!checks_enable_r)
    rst |=> empty; //si rst activo en el siguiente flanco de reloj empty debe estar activo
endproperty: reset_empty

ap_reset_empty: assert property (reset_empty)
  $display("At %d  PROP. OK. - reset_empty", $time);
  else $error("ERROR ap_reset_empty");
```

Figura 68: Propiedad 6 UVM, reset_empty

- Propiedad 2: empty permanece activo hasta que se inserte un elemento

Esta propiedad se comprueba que, una vez el sistema está vacío, este permanecerá en

dicho estado hasta que se detecte una inserción *insert* o la combinación de *insert* y *remove*. Esta propiedad impide que el sistema entre en un estado "no vacío" sin que se hayan insertado datos válidos, lo cual preserva la integridad funcional del módulo. La Figura 69 contiene el código fuente correspondiente a la implementación de la propiedad.

```
1  // Empty is set until insert
2  property empty_until_insert;
3    bit insert_remove;
4    @(posedge clk) disable iff (rst || !checks_enable_r)
5    empty |-> empty until (insert || ($past(insert,1) && $past(remove,1)));
6    //empty se debe manter true hasta (until) que se cumpla alguna de las dos condiciones
7  endproperty: empty_until_insert
8
9  ap_empty_until_insert: assert property (empty_until_insert)
10   else $error("ERROR empty_until_insert");
```

Figura 69: Propiedad 2 UVM, empty_until_insert

- Propiedad 3: Actualización correcta del contador tras una inserción

Esta propiedad verifica que, al producirse una inserción, el contador *num_entries* se incremente en uno respecto al valor anterior. Esta comprobación asegura que el mecanismo de conteo interno del sistema se actualiza correctamente durante operaciones de inserción. En la Figura 70 se presenta el código que da soporte a la implementación de la propiedad.

```
1  //Remove decrements number of entries
2  property i02_insert2numentries;
3    logic [L2_REG_WIDTH:0] pnum_entries;
4    @(posedge clk) disable iff (rst || !checks_enable_r)
5    (!insert && remove, pnum_entries=num_entries) |=> (num_entries == pnum_entries-1);
6  endproperty: i02_insert2numentries
7
8  ap_i02_insert2numentries: assert property (i02_insert2numentries)
9    else $error("ERROR ap_i02_insert2numentries");
```

Figura 70: Propiedad 3 UVM, i02_insert2numentries

- Propiedad 4: Inserción o eliminación deben durar un solo ciclo

Se comprueba que las señales *insert* y *remove*, cuando se activan, deben desactivarse al menos en el ciclo siguiente. Esto evita repeticiones indeseadas.

Esta propiedad protege al sistema de efectos indeseados derivados de señales que persisten más de un ciclo, evitando errores por interpretaciones múltiples de una misma

operación. La Figura 71 representa el código desarrollado para la implementación de la propiedad.

```
property insert_remove_period_check;
  @(posedge clk) disable iff (rst || !checks_enable_r)
    (insert |-> ##1 !remove) or (remove |-> ##1 insert);
endproperty: insert_remove_period_check

ap_insert_remove_period_check: assert property(insert_remove_period_check)
    else $error("At time %0t: ERROR - Insert-Remove timing violated at:",$time);
```

Figura 71: Propiedad 4 UVM, insert_remove_period_check

7.6. Conclusión del capítulo

A lo largo de este capítulo se ha abordado la integración de *SystemVerilog Assertions* (SVA) dentro del entorno UVM desde una perspectiva metodológica y aplicada. Partiendo del enfoque propuesto por *Verilab*, se han explorado en profundidad dos estrategias complementarias: el uso de una API manual para controlar la activación de las *assertions*, y la incorporación de un componente automático y *phase-aware* que permite una configuración autónoma, sincronizada y desacoplada del *checker*.

Ambas metodologías han sido implementadas y evaluadas dentro del entorno UVM desarrollado para el módulo, permitiendo validar formalmente su comportamiento sin comprometer la modularidad ni la escalabilidad del sistema. Las *assertions* diseñadas y encapsuladas en las interfaces de entrada y salida han reforzado la cobertura funcional, facilitado la detección de errores y mejorado la trazabilidad de la verificación.

En conjunto, esta integración representa una contribución técnica relevante del proyecto, al demostrar que es posible llevar las ventajas del *model checking* y la verificación temporal directamente al flujo UVM de manera controlada y conforme a buenas prácticas. Esta base permite además ampliar fácilmente el entorno hacia objetivos más complejos, como propiedades dependientes de configuración dinámica, cobertura formal dirigida o entornos *multipuerto* con múltiples instancias del *checker*

8º Capítulo: Resultados de verificación

Este capítulo presenta los resultados obtenidos tras la ejecución del entorno de verificación basado en la metodología *Universal Verification Methodology* (UVM), con el módulo como dispositivo bajo verificación (DUV). La evaluación se centra en dos ejes principales: por un lado, el análisis del comportamiento funcional del IP bajo diferentes condiciones de operación y por otro, la verificación formal de propiedades clave mediante *SystemVerilog Assertions* (SVA) integradas en el entorno UVM.

Se examina el cumplimiento de las propiedades concurrentes desarrolladas y su capacidad para detectar comportamientos no deseados o inconsistentes en el diseño. Para ello, se han ejecutado distintos escenarios de prueba representativos, diseñados para cubrir las funcionalidades esenciales del módulo: inserciones con prioridad, extracciones ordenadas y situaciones límite como condiciones de saturación o vaciado total. La ejecución de estos *tests* se ha realizado con las SVA activas, utilizando tanto el enfoque API manual como la variante automática y *phase-aware*, demostrando su equivalencia a nivel de resultado funcional.

Durante las pruebas se han observado tanto ejecuciones correctas como violaciones intencionadas, con el objetivo de validar la sensibilidad de las propiedades ante condiciones anómalas. Estas violaciones permiten verificar que las *assertions* no solo están correctamente formuladas y activadas, sino que además responden con precisión ante comportamientos que comprometen la integridad funcional del IP.

La información se presenta desglosada por test ejecutado, proporcionando una visión detallada del cumplimiento de las propiedades, la trazabilidad del comportamiento funcional del diseño y las posibles incidencias detectadas. Se incluye evidencia gráfica a partir de las formas de onda y la consola de resultados del simulador, con el fin de respaldar el análisis técnico y facilitar la identificación de áreas susceptibles de mejora, tanto en el IP como en el entorno de verificación.

8.1. Análisis de resultados por test ejecutado

En esta sección se detallan los resultados obtenidos tras la ejecución de los diferentes *tests* definidos en la librería UVM. Cada uno de estos *tests* se diseñó con el objetivo de estimular funcionalidades específicas del módulo y evaluar, tanto desde una perspectiva funcional como formal, el cumplimiento de los requisitos establecidos mediante *SystemVerilog Assertions* (SVA). La ejecución de los *tests* se realizó con las *assertions* activadas, utilizando tanto la metodología de encapsulación manual (API) como la versión automática y *phase-aware*. En ambos casos, se obtuvieron resultados equivalentes en términos de verificación.

El análisis incluye la respuesta funcional del IP, la activación de propiedades concurrentes, la detección de violaciones y la validación del comportamiento esperado a través del *scoreboard*. En los apartados siguientes se expone un análisis detallado de cada test, junto con capturas de las formas de onda relevantes y la consola de *assertions*.

8.1.1. Test t_i00: Inserciones ordenadas por prioridad

El propósito de este escenario de prueba es validar el comportamiento del módulo frente a operaciones de inserción secuenciales con datos de distinta prioridad. Concretamente, se pretende verificar que el sistema implementa correctamente la política de prioridad establecida: el elemento con menor valor numérico de *rank_in*, mayor prioridad relativa, debe mantenerse accesible en la salida *rank_out, meta_out* del módulo, y el orden interno debe reflejar esta jerarquía.

Este *test* simula una condición simple pero crítica. Se insertan dos transacciones consecutivas, la primera con prioridad mínima y la segunda con prioridad máxima. El objetivo es verificar que el sistema identifica correctamente cuál de los dos elementos debe mantenerse como dato de salida más prioritario, incluso en ausencia de operaciones de extracción.

La simulación se ejecutó con ambas metodologías de encapsulación de SVA habilitadas, API manual y *automatic phase-aware*, en ejecuciones independientes. Durante la simulación se activó una única secuencia de inserciones, *insert_data_min_prio* seguida de *insert_data_max_prio*, sobre el canal de entrada. El canal de extracción permaneció inactivo, lo que permitió observar exclusivamente el comportamiento interno del módulo en estado de acumulación.

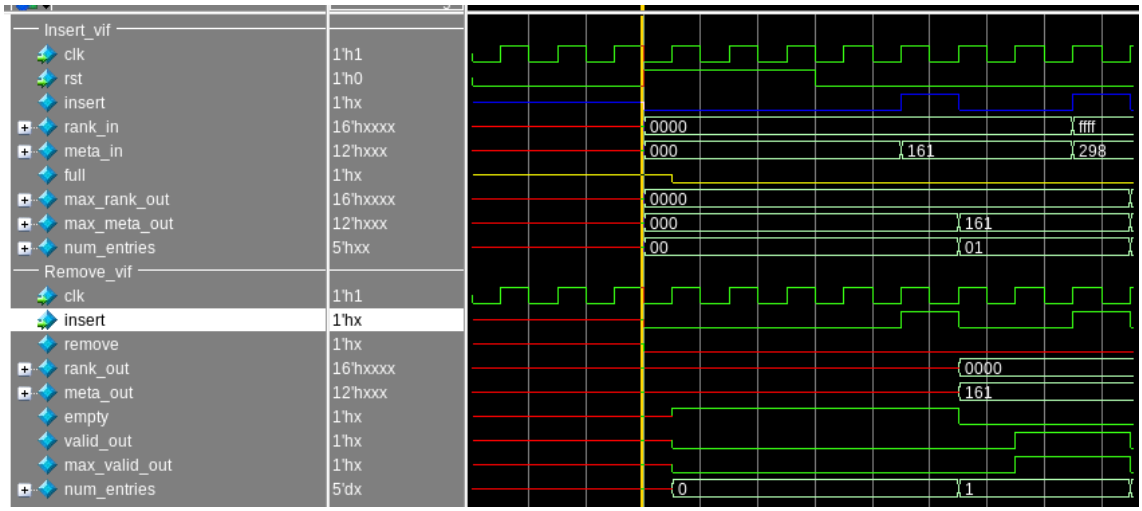

Figura 72: Formas de onda resultado de la ejecución del test_i00

La ejecución del *test* t_i00 permitió comprobar que el módulo gestionó correctamente ambas inserciones. Las señales *rank_in* y *meta_in* fueron capturadas con precisión por el monitor de entrada, y el *scoreboard* reconstruyó correctamente la estructura interna esperada.

La Figura 72 muestra las formas de ondas correspondientes al t_i00. Tras la primera inserción, la señal *empty* pasó de 1 a 0, reflejando que el módulo ya contenía un dato. Tras la segunda inserción, la señal *num_entries* se incrementó adecuadamente, y *rank_out* permaneció asociado al dato con menor valor de prioridad, evidenciando que el orden lógico se mantuvo.

Las señales *max_rank_out* y *max_valid_out* también se actualizaron, reflejando correctamente cuál era el elemento de menor prioridad presente en la estructura. Este comportamiento se considera coherente con las especificaciones del módulo, y confirma que

el mecanismo de comparación interna (basado en estructuras jerárquicas) opera como se espera.

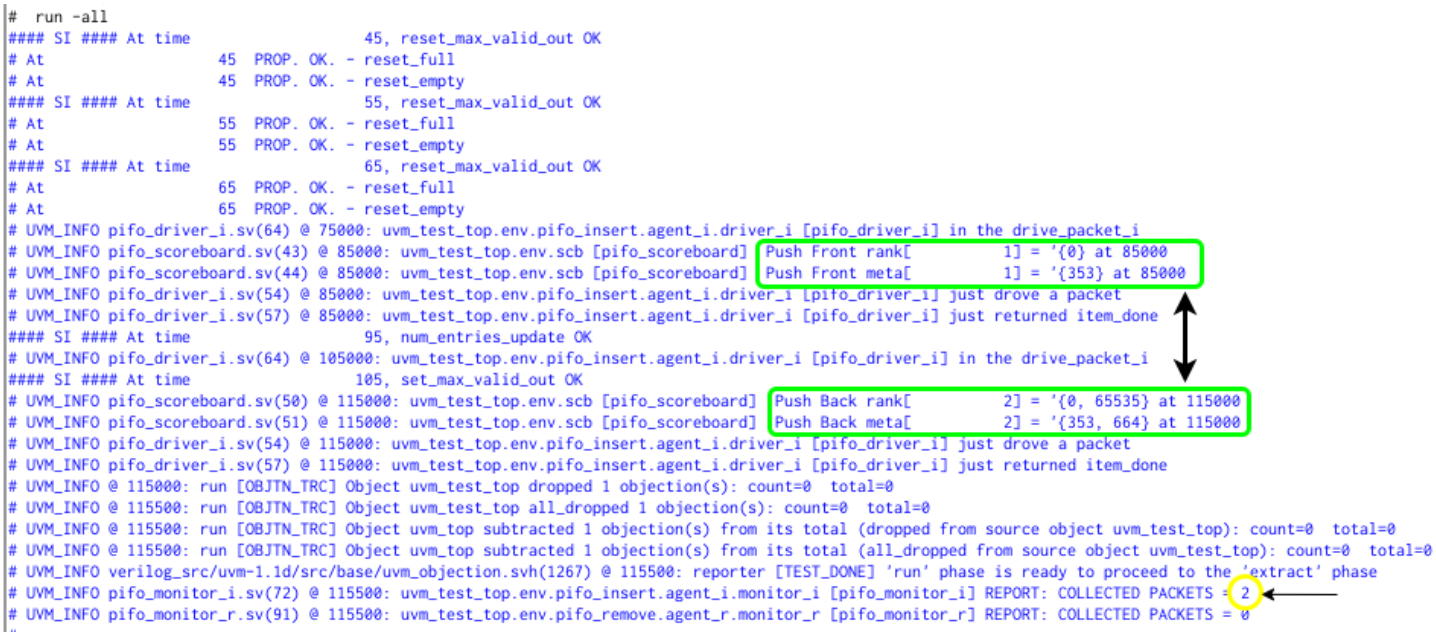

Figura 73: Log generado correspondiente al test t_i00

El *scoreboard* detectó correctamente las transacciones, como se puede observar en la Figura 73, observadas por el monitor de entrada, e interpretó el orden relativo de prioridad de los elementos. Tal como se muestra en la traza de simulación correspondiente a la imagen superior. La primera transacción, *rank* = 1, fue registrada en la cabecera de la cola (*Push Front rank*) mientras que la segunda transacción, rank = 2, fue añadida al final de la cola (*Push Back rank*), lo que confirma que el entorno reconoce correctamente la relación de prioridad entre ambos datos.

Además, los metadatos asociados a cada elemento (meta[1] = 353) y (meta[2] = 664) fueron correctamente diferenciados y registrados por el *scoreboard*, con distintos valores de *rank* en el orden correspondiente. Las operaciones del *driver* se ejecutaron sin errores y ambas transacciones alcanzaron la fase *item_done*, sin bloqueos ni interrupciones.

Como era de esperar en este *test*, el canal de salida no recogió ningún paquete *(COLLECTED PACKETS = 0)*, ya que no se realizó ninguna operación de *remove*.

Estos resultados confirman que el entorno de verificación funcional, basado en el uso de monitores y *scoreboard*, se comportó conforme a lo esperado y validó correctamente la lógica interna del DUV.

Durante la ejecución de *t_i00* se activaron un conjunto de *assertions* concurrentes contenidas en el *checker* embebido en la interfaz del DUV. Las propiedades relevantes fueron: *ap_insert_period_check, ap_insert_when_full, ap_num_entries_update, ap_full_requires_max_entries, ap_set_max_valid_out.*

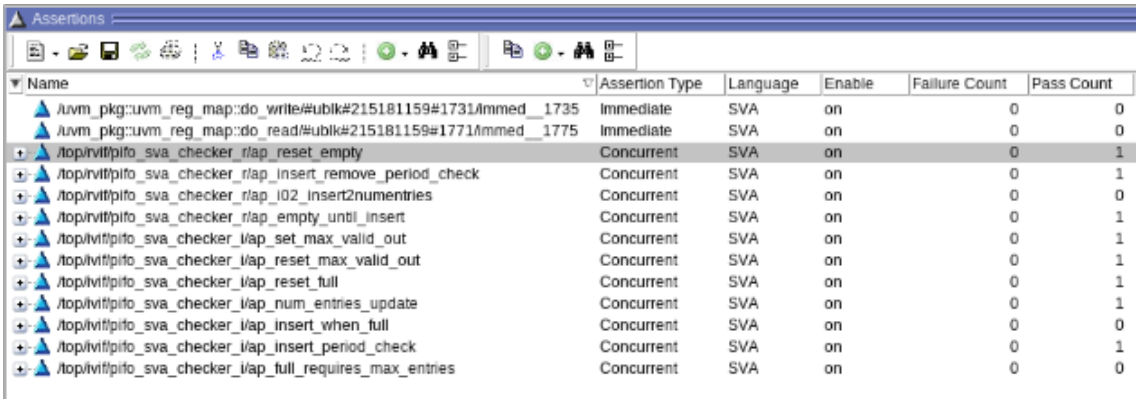

Figura 74: Resultados de las SVA implementadas tras la ejecución del test_i00

Todas ellas se activaron correctamente, se evaluaron durante la ventana temporal correspondiente, y fueron superadas sin errores, como se evidencia en la consola del simulador correspondiente a la Figura 69. Los contadores de evaluación muestran *pass count* = 1 y *failure count* = 0 para cada propiedad.

En la forma de onda capturada correspondiente a la Figura 72 se aprecia claramente:

- La señal *insert* activa durante un único ciclo de reloj (condición temporal válida).
- La evolución ascendente de *num_entries*, en línea con las inserciones realizadas.
- La no activación de *full*, lo cual cumple con la condición de capacidad.
- El valor de *max_valid_out* elevado correctamente cuando se inserta un dato de menor prioridad con mayor *rank*.

Además, todas las propiedades se formularon con `disable iff` ligado a `checks_enable_i` y *rst*, lo cual garantiza que no se activen durante la fase de *reset* ni fuera del intervalo de evaluación deseado.

No se registraron violaciones de *assertions* ni errores en la verificación funcional. El comportamiento del DUV fue consistente con su especificación y no se identificaron condiciones límites críticas. Esto sugiere una implementación sólida para este modo de operación, así como una integración correcta del entorno de verificación y de las SVA.

El *test* t_i00 ha demostrado que el módulo IP implementa correctamente su política de prioridad interna durante inserciones secuenciales. El diseño respondió de forma coherente en cuanto a propagación de señales de salida, actualización de contadores y gestión de prioridades.

Tanto el entorno funcional (*scoreboard*) como el formal (*assertions*) confirmaron la validez del comportamiento observado. La correcta activación y evaluación de las propiedades formales refuerza la fiabilidad del *checker* desarrollado e ilustra el valor de integrar SVA dentro de entornos UVM.

Este *test* representa un escenario de baja complejidad, pero esencial para la verificación funcional del diseño, y sirve como base para validar modos de operación más exigentes en los *tests* posteriores.

8.1.2.Test t_r00: Inserciones seguida de extracción

El objetivo principal de este escenario es verificar el comportamiento básico de la lógica de inserción y extracción del módulo IP, así como la correcta propagación del dato insertado hacia la salida tras una operación de eliminación. El *test* simula un flujo mínimo de operación: insertar un único elemento y extraerlo inmediatamente. Con ello, se evalúa la integridad de la ruta de datos, la actualización de las señales de estado (*empty, valid_out, num_entries*, etc.) y la consistencia de la política PIFO bajo condiciones sin conflicto.

Además, se espera que este *test* active propiedades formales relacionadas con la transición de estados (*empty → no empty → empty*), la coherencia del contador de entradas y la validez de las señales de salida tras operaciones de eliminación.

La ejecución del *test* se realizó con las SVA activadas bajo ambas metodologías de encapsulación *API* y *phase-aware*, partiendo de la clase base. La señal *checks_enable_i* fue activada durante la fase de configuración, garantizando que todas las *assertions* relevantes estuvieran habilitadas al inicio de la simulación funcional.

Durante la prueba se introdujo un único paquete en el sistema mediante una operación de *insert*, seguido inmediatamente de una operación de *remove*. Esta secuencia permite comprobar si el elemento propagado a la salida corresponde con el valor previamente insertado, y si la estructura interna responde correctamente a una operación de vaciado.

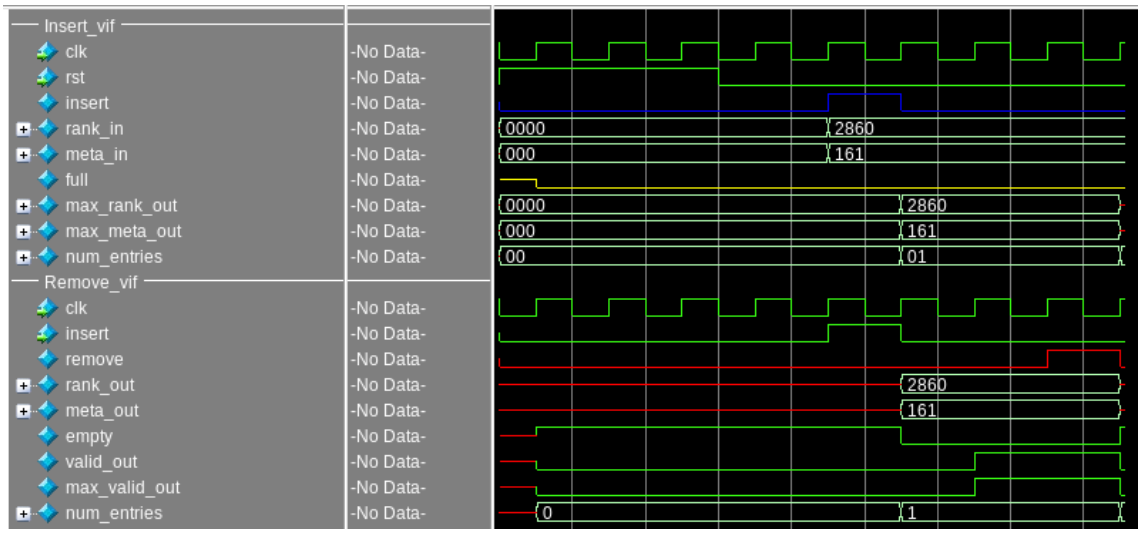

Figura 75: Forma de onda resultado de la ejecución del test_r00

La Figura 75 muestra que el comportamiento del IP fue correcto en todo momento. Tras la inserción, las señales de salida *rank_out, meta_out* y *valid_out* reflejaron el valor introducido. Concretamente, *rank_out* = 2860 y *meta_out* = 161, los cuales coinciden con los valores aplicados por el *driver*, como se aprecia en la las formas de onda correspondiente a la Figura 75.

A continuación, la operación de extracción provocó la desactivación de la señal *valid_out*, y la señal *empty* retornó a nivel alto, confirmando que el único dato presente en la estructura fue correctamente eliminado.

Figura 76: Log generado tras la ejecución del test_r00

La evolución del contador *num_entries* fue coherente. Este pasó de 0 a 1 tras la inserción, y de 1 a 0 tras la extracción. Esta secuencia confirma que el control interno del buffer y la gestión de estado funcionan adecuadamente.

La Figura 76 muestra tras la generación del test estos mensajes de consola confirman que el entorno funcional detectó y validó correctamente el comportamiento observado:

- El *scoreboard* registró la inserción del paquete *rank* = 10336, meta = 353 como *Push Front*.
- Posteriormente, se detectó su extracción como SCB-PASS, tras comparación entre el valor esperado y el observado.
- El mensaje [SCB-PASS] *Rank_out expected*: 10336, actual: 10336 indica coincidencia exacta en el campo de prioridad.
- Del mismo modo, *Meta_out expected*: 353, actual: 353 valida la integridad del dato completo.

Además, el *monitor_r* del canal de salida reportó *COLLECTED PACKETS* = 1, confirmando que el paquete fue efectivamente recogido tras la operación de extracción.

Estas evidencias demuestran que el entorno funcional, compuesto por agentes, monitores y *scoreboard*, actuó correctamente y fue capaz de validar el comportamiento esperado en un escenario simple pero esencial.

Durante esta prueba se activaron varias *assertions* concurrentes críticas para este modo de operación. Entre las más relevantes se encuentran: *ap_insert_period_check, ap_insert_when_full, ap_num_entries_update, ap_empty_until_insert, ap_set_max_valid_out, ap_reset_max_valid_out, ap_reset_empty y ap_reset_empty*.

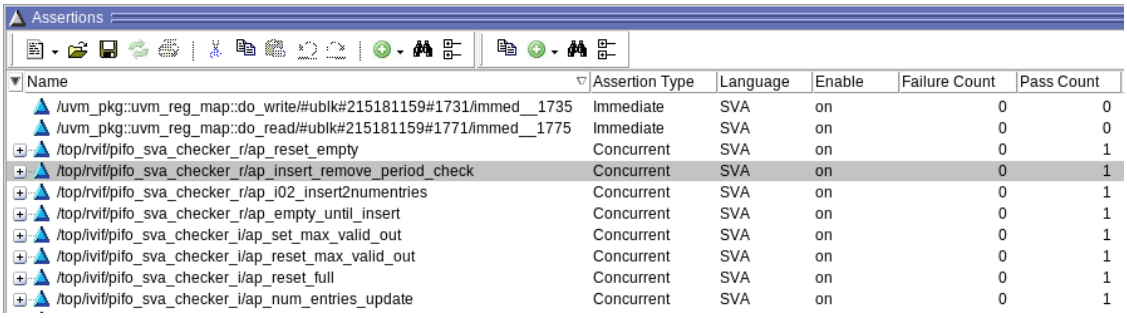

Figura 77: Resumen de las SVA ejecutadas durante el test_r00

Tal y como muestra la Figura 77 todas estas *properties* se activaron en los ciclos pertinentes y fueron evaluadas sin errores, reportando *pass count* = 1 y *failure count* = 0. Su correcta activación se debió a la propagación adecuada de *checks_enable_i*, combinada con el uso de cláusulas `disable iff` en la formulación de las *assertions*, lo que impidió activaciones espurias durante el *reset*.

Las formas de onda confirman, la transición de *empty* 1 → 0 → 1 en sincronismo con las operaciones. Así como la activación y desactivación coherente de *valid_out* y la actualización correcta de *num_entries*.

La *assertion ap_empty_until_insert* validó que la señal *empty* no se desactivó hasta una inserción válida, mientras que *ap_set_max_valid_out* verificó la propagación del dato con menor *rank* al canal de salida. Todas las *properties* mostraron un comportamiento acorde a las especificaciones formales previstas.

No se registraron errores funcionales ni formales durante la ejecución de este test. El módulo respondió correctamente, y tanto el *checker* formal como el entorno funcional confirmaron el comportamiento esperado. No se identificaron anomalías ni situaciones límite incorrectamente gestionadas.

El *test t_r00* permitió validar la funcionalidad básica del módulo en un ciclo completo de operación: inserción seguida de extracción. Las señales internas y externas evolucionaron de forma coherente con las expectativas de diseño, y el entorno UVM fue capaz de observar, registrar y verificar todos los eventos relevantes.

Las *assertions* concurrentes activadas durante este escenario contribuyeron a reforzar la confianza en el sistema, al validar condiciones de integridad estructural y transiciones de estado esenciales.

Este caso de prueba sirve como referencia para evaluar el comportamiento del diseño en contextos simples y controlados, siendo especialmente útil como base para el análisis de escenarios más exigentes, como los abordados en el *test t_ir01*.

8.1.3.Test t_ir01: Inserciones seguida de extracción

Este escenario fue diseñado para someter al módulo a condiciones de operación intensivas, combinando múltiples inserciones y extracciones de manera simultánea. El objetivo principal es validar la robustez del diseño frente a concurrencia de operaciones, asegurando la integridad del contenido del registro, la coherencia de las señales internas *empty*, *full, num_entries* y la correcta gestión de prioridad entre elementos.

Este *test* busca también verificar si el módulo puede mantener la política de prioridad bajo cambios dinámicos de contenido y detectar posibles condiciones de carrera o errores de sincronización. Desde el punto de vista formal, se espera que se activen múltiples *assertions* concurrentes, permitiendo evaluar el comportamiento del diseño frente a secuencias no deterministas.

El *test* fue ejecutado desde *base_test* con las *assertions* activadas mediante ambas

metodologías de integración *API* y *phase-aware*. La señal *checks_enable_i* se propagó correctamente durante la fase de configuración y las propiedades quedaron habilitadas tras el *reset*.

Se aplicaron múltiples transacciones en ambos canales *insert* y *remove*, generando una interacción dinámica entre los agentes del entorno. En total, se insertaron y extrajeron 16 elementos, generando múltiples transiciones de estado internas y activaciones de lógica de comparación.

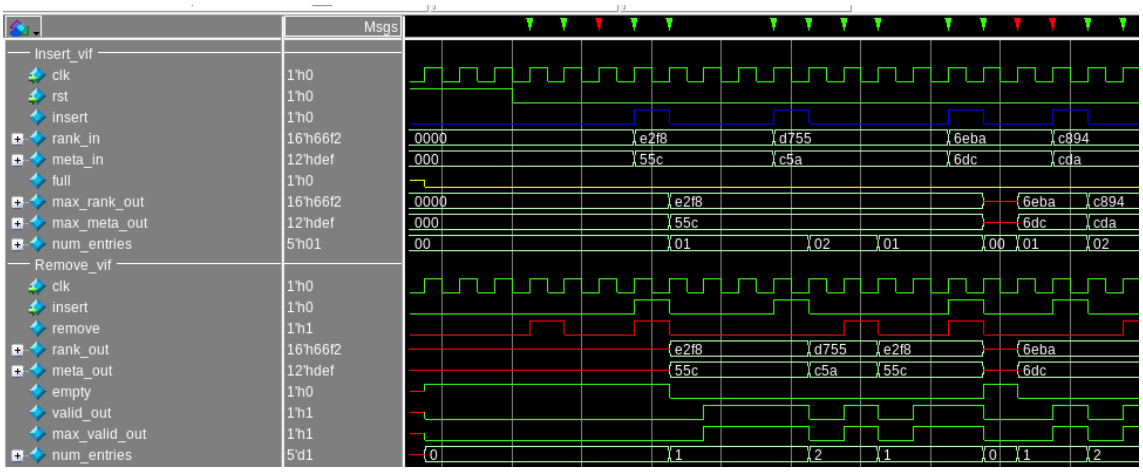

Figura 78: Forma de onda generada tras la ejecución del test_ir01

Las formas de onda correspondiente a la Figura 77 y la traza funcional correspondientes a las Figura 78 y Figura 79 muestran que el módulo procesó correctamente las operaciones concurrentes. Durante la ejecución:

- Se realizaron múltiples inserciones con diferentes valores de *rank_in*, observándose actualizaciones coherentes en *rank_out, meta_out, max_rank_out* y *valid_out*. Estas inserciones vienen reflejadas en la Figura 78 con las activaciones de la señal *insert* y la validación posterior de la señal *valid_out*.
- La señal *num_entries* fluctuó conforme a la carga y descarga de datos. Estas fluctuaciones se muestran en las formas de onda después de cada inserción y extracción. En este ultimo caso solo, solo se atienden a extracciones cuando la cola no este vacía.

- Las señales *empty* y *full* se activaron y desactivaron conforme al estado real del módulo, reflejando un control interno coherente.

Las Figuras 79 y 80 muestran los mensajes por consola generados en este test. Los mensajes mostrados a la Figura 78 permite hacer un seguimiento de las operaciones realizadas durante el test por cada inserción y extracción. La Figura 79 muestra información sobre los registros de *COLLECTED PACKETS* = 16 tanto en el monitor de inserción como en el de extracción. Estos registros confirman que no hubo pérdida de paquetes, y que la actividad funcional fue completa.

```
#  run -all
#### SI #### At time                   45, reset_max_valid_out OK
# At                   45  PROP. OK. - reset_full
# At                   45  PROP. OK. - reset_empty
#### SI #### At time                   55, reset_max_valid_out OK
# At                   55  PROP. OK. - reset_full
# At                   55  PROP. OK. - reset_empty
# UVM_INFO pifo_driver_r.sv(61) @ 65000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r]   VALID_OUT = 0
# UVM_INFO pifo_driver_r.sv(62) @ 65000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r]   insert = 0
# UVM_INFO pifo_driver_r.sv(63) @ 65000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] in the drive_packet_r
#### SI #### At time                   65, reset_max_valid_out OK
# At                   65  PROP. OK. - reset_full
# At                   65  PROP. OK. - reset_empty
# UVM_INFO pifo_scoreboard.sv(92) @ 75000: uvm_test_top.env.scb [pifo_scoreboard]  [SCB-FAIL] Empty Queue rank and meta size =          0 at: 75000
# UVM_INFO pifo_driver_r.sv(48) @ 75000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] just drove a packet
# UVM_INFO pifo_driver_r.sv(51) @ 75000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] just returned item_doen
# ** Error: ERROR ap_i02_insert2numentries
#    Time: 85 ns Started: 75 ns  Scope: top.rvif.pifo_sva_checker_r.ap_i02_insert2numentries File: pifo_sva_checker_r.sv Line: 63
# UVM_INFO pifo_driver_i.sv(64) @ 95000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] in the drive_packet_i
# UVM_INFO pifo_driver_r.sv(61) @ 95000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r]   VALID_OUT = 0
# UVM_INFO pifo_driver_r.sv(62) @ 95000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r]   insert = 0
# UVM_INFO pifo_driver_r.sv(63) @ 95000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] in the drive_packet_r
# UVM_INFO pifo_scoreboard.sv(43) @ 105000: uvm_test_top.env.scb [pifo_scoreboard]  Push Front rank[          1] = '{58104} at 105000
# UVM_INFO pifo_scoreboard.sv(44) @ 105000: uvm_test_top.env.scb [pifo_scoreboard]  Push Front meta[          1] = '{1372} at 105000
# UVM_INFO pifo_driver_i.sv(54) @ 105000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] just drove a packet
# UVM_INFO pifo_driver_i.sv(57) @ 105000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] just returned item_done
# UVM_INFO pifo_scoreboard.sv(87) @ 105000: uvm_test_top.env.scb [pifo_scoreboard]  [SCB-FAIL] Meta_out expected: 55c,  actual: 0 at 105000
# UVM_INFO pifo_scoreboard.sv(88) @ 105000: uvm_test_top.env.scb [pifo_scoreboard]  [SCB-FAIL] Rank_out expected: e2f8,  actual: 0 at 105000
# UVM_INFO pifo_driver_r.sv(48) @ 105000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] just drove a packet
# UVM_INFO pifo_driver_r.sv(51) @ 105000: uvm_test_top.env.pifo_remove.agent_r.driver_r [pifo_driver_r] just returned item_doen
```

Figura 79: Parte inicial del Log generado tras la ejecución del test_ir01

```
# UVM_INFO pifo_driver_i.sv(64) @ 765000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] in the drive_packet_i
# UVM_INFO pifo_scoreboard.sv(69) @ 775000: uvm_test_top.env.scb [pifo_scoreboard]  Insert rank[          3] = '{931, 11772, 63469} at 775000
# UVM_INFO pifo_scoreboard.sv(70) @ 775000: uvm_test_top.env.scb [pifo_scoreboard]  Insert meta[          3] = '{3674, 1870, 2278} at 775000
# UVM_INFO pifo_driver_i.sv(54) @ 775000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] just drove a packet
# UVM_INFO pifo_driver_i.sv(57) @ 775000: uvm_test_top.env.pifo_insert.agent_i.driver_i [pifo_driver_i] just returned item_done
# UVM_INFO @ 775000: run [OBJTN_TRC] Object uvm_test_top dropped 1 objection(s): count=0  total=0
# UVM_INFO @ 775500: run [OBJTN_TRC] Object uvm_test_top all_dropped 1 objection(s): count=0  total=0
# UVM_INFO @ 775500: run [OBJTN_TRC] Object uvm_top subtracted 1 objection(s) from its total (dropped from source object uvm_test_top): count=0  total=0
# UVM_INFO @ 775500: run [OBJTN_TRC] Object uvm_top subtracted 1 objection(s) from its total (all_dropped from source object uvm_test_top): count=0  total=0
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 775500: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO pifo_monitor_i.sv(72) @ 775500: uvm_test_top.env.pifo_insert.agent_i.monitor_i [pifo_monitor_i] REPORT: COLLECTED PACKETS = 16
# UVM_INFO pifo_monitor_r.sv(91) @ 775500: uvm_test_top.env.pifo_remove.agent_r.monitor_r [pifo_monitor_r] REPORT: COLLECTED PACKETS = 16
#
```

Figura 80: Parte final del Log generado tras la ejecución del test_ir01

Durante la simulación se activaron y evaluaron correctamente todas las *assertions* concurrentes. En este *test*, se detectaron fallos exclusivamente en las siguientes propiedades señaladas en la Figura 81. Esto confirma su utilidad y precisión de las *assertions*.

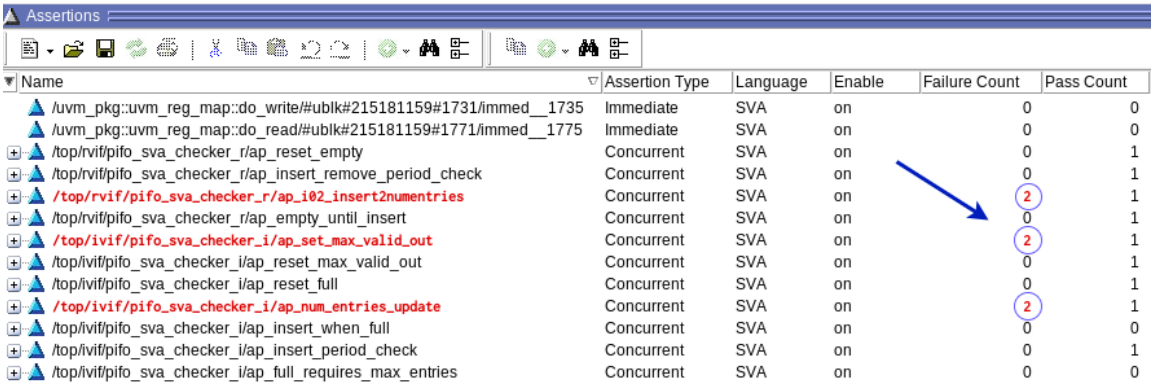

Figura 81: Resumen de las SVA ejecutadas en el test_ir01

A continuación se exponen aquellas propiedades que fallan durante la ejecucion del test.

Propiedad `ap_i02_insert2numentries`: *Failure Count = 2* Esta propiedad verifica que por cada operación *remove* se decremente *num_entries* en 1. El fallo indica que, en al menos dos ocasiones, el contador no reflejó el número esperado de entradas. La forma de onda muestra que la relación temporal entre la entrada y la actualización del contador no se respetó puntualmente. Este fenómeno se puede observar en la Figura 82, dado que el test ejecuta operaciones aleatorias, comienza haciendo dos operaciones *remove* cuando aún no se ha insertado ningún dato, por tanto, *num_entries* conserva el valor dado por el estado de reset.

Si bien esta propiedad ha fallado, el error que lo provoca hubiese saltado con la propiedad que verifica que no se acepten extracciones con la cola vacía.

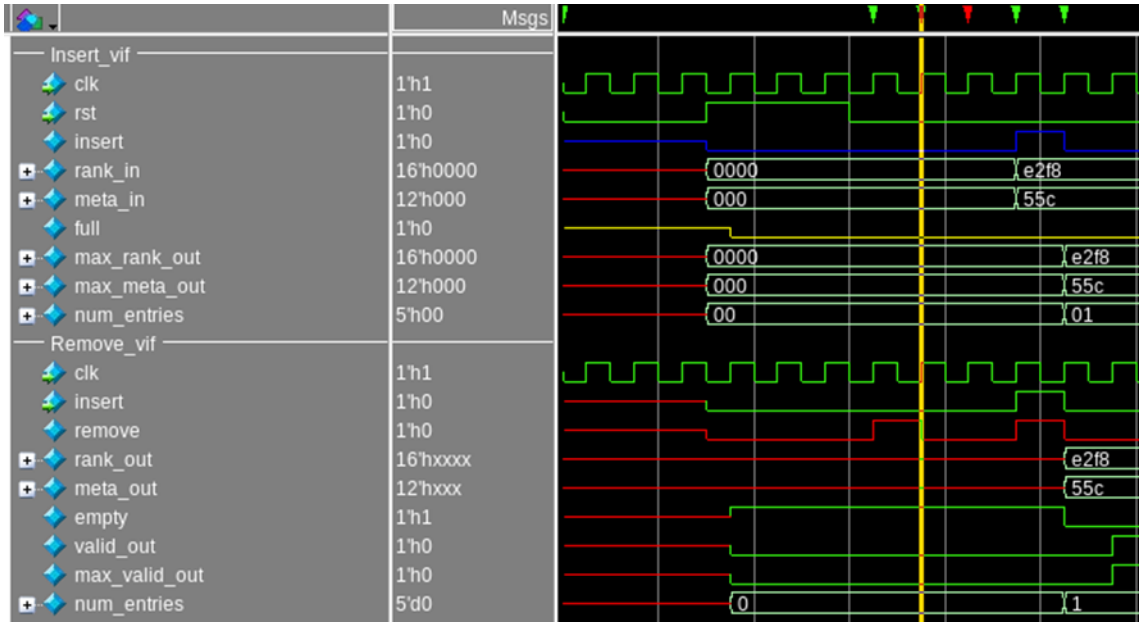

Figura 82: Evidencia ap_i02_insert2numeries

Propiedad `ap_set_max_valid_out`: *Failure Count = 2*. Esta *assertion* valida que *max_valid_out* solo se active dos ciclos después del flanco de subida de un *insert*. En la figura 83 se observa la activación de *max_valid_out* un ciclo más tarde de lo esperado, por tanto la propiedad notifica el fallo de manera correcta. La línea de marcación vertical señala el ciclo donde se espera la activación de la señal.

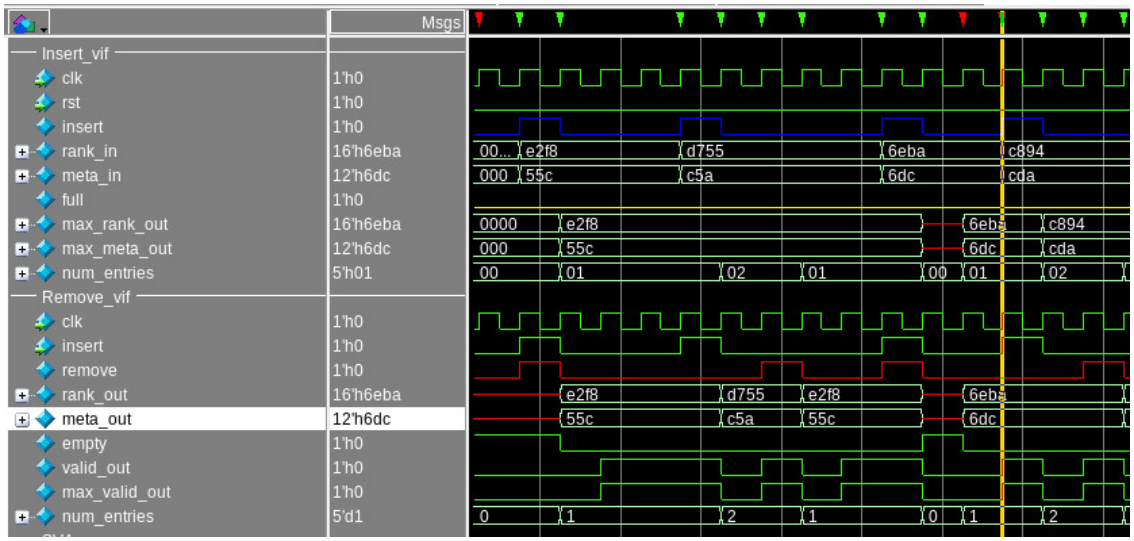

Figura 83: Evidencia num_entries_update

Propiedad `ap_num_entries_update`: *Failure Count = 2*. Esta propiedad supervisa que la señal *num_entries* se incremente en el ciclo siguiente a un flanco de subida de la señal *insert*. En la Figura 84 se observa una situación en la que, tras un *insert* válido, el valor de *num_entries* no cambia como se esperaba. En este caso, ocurre primero una operación de *remove*, seguida inmediatamente por un *insert*. El problema surge porque justo cuando se almacena el valor anterior de *num_entries*, este se decrementa debido al *remove*, y en el siguiente ciclo se incrementa debido a la operación de *insert* pendiente. Como resultado, la propiedad espera un valor de *num_entries = 2*, pero el sistema proporciona *num_entries* = 1, provocando el fallo de la propiedad.

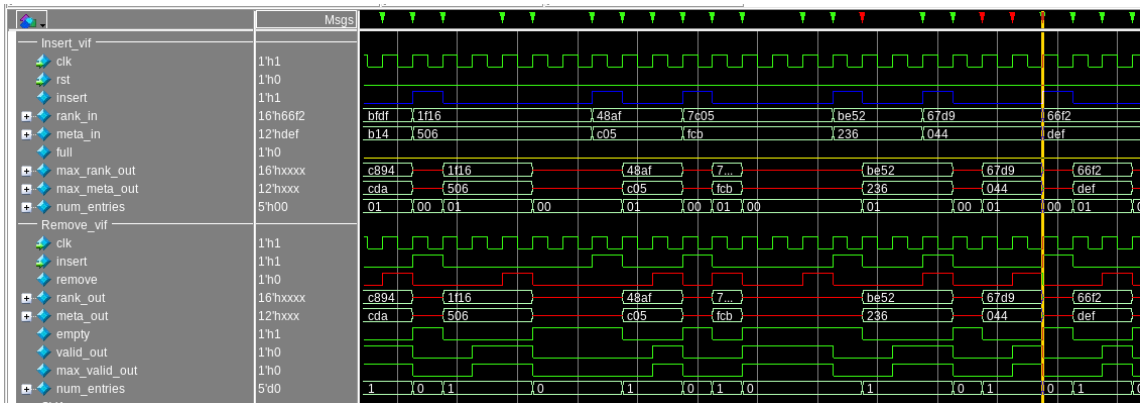

Figura 84: Forma de onda parcial del test ir_01

El resto de *properties* pasaron correctamente su evaluación, incluidas, *ap_insert_period_check, ap_insert_when_full, ap_empty_until_insert, ap_full_requires_max_entries, ap_reset_empty, ap_reset_max_valid_out* y *ap_insert_remove_period_check*.

En la Figura 81, demuestra que estas *assertions* se activaron y evaluaron correctamente en ciclos previos, lo que confirma su correcto funcionamiento en condiciones normales.

En conjunto, estos resultados reflejan que las SVA fueron precisas y eficaces, con activación controlada y detección de fallos exclusivamente cuando el comportamiento del diseño se desvió de la especificación.

El *test t_ir01* ha permitido validar el comportamiento del módulo IP bajo un escenario intensivo de operaciones concurrentes de inserción y extracción. A nivel funcional, el diseño mostró un comportamiento generalmente estable y coherente, gestionando correctamente el flujo de datos y manteniendo operativas las señales de control *valid_out, empty y num_entries*. Las formas de onda confirmaron la integridad de las rutas de datos, sin evidencias de pérdidas o errores.

Sin embargo, este test reveló ciertos comportamientos anómalos que fueron detectables mediante observación funcional y gracias al uso de *SystemVerilog Assertions*. En particular, las violaciones de las propiedades *ap_num_entries_update*, *ap_set_max_valid_out* e *ap_i02_insert2numentries* expusieron discrepancias entre el comportamiento esperado según la especificación funcional y el observado en situaciones de carga y concurrencia. Estas violaciones fueron verificadas mediante las formas de onda, descartando activaciones espúreas y confirmando que las propiedades fallaron de forma legítima.

Este caso demuestra que las SVA no solo enriquecen el entorno de verificación, sino que proporcionan una capa de diagnóstico extremadamente precisa y fiable, capaz de detectar errores sutiles de sincronización, lógica interna o actualización de estado que pueden pasar desapercibidos en entornos basados únicamente en referencias funcionales. La correcta activación, evaluación y trazabilidad de las *assertions* implementadas consolidan su utilidad

como herramienta formal, especialmente en escenarios donde se requiere verificación exhaustiva.

Por tanto, este *test* no solo ha permitido validar aspectos funcionales del IP, sino también identificar áreas de mejora en su lógica interna, reforzando la necesidad de incluir validación formal dentro de entornos UVM avanzados para lograr una cobertura más profunda y una detección temprana de defectos críticos.

9º Capítulo: Conclusiones y líneas futuras

La realización de este Trabajo Fin de Grado ha supuesto un reto técnico y formativo de gran envergadura, especialmente considerando el punto de partida. El proyecto partía de conocimientos muy limitados en verificación avanzada: no se había trabajado previamente con *SystemVerilog* más allá de *testbenchs* básicos en VHDL, no existía experiencia previa con entornos de simulación profesionales como *QuestaSim*, y tampoco se había empleado ninguna herramienta de control de versiones como GitHub. A pesar de estas condiciones iniciales, el trabajo ha conseguido no solo alcanzar los objetivos propuestos, sino también profundizar en una metodología profesional de verificación que actualmente es referencia en la industria: la *Universal Verification Methodology* (UVM), complementada con el uso de propiedades, especificadas mediante el lenguaje *SystemVerilog Assertions* (SVA).

El trabajo realizado ha permitido a la División de Sistemas Integrados del Instituto Universitario de Microelectrónica Aplicada, establecer las bases para la verificación de sistemas mediante el uso de propiedades. Esto ha sido un aspecto novedoso si se tiene en cuenta que, hasta la fecha, la única metodología de verificación utilizada por esta división se basaba en su totalidad en el uso de una verificación funcional basada, únicamente, en simulación.

Desde una perspectiva técnica, el principal logro del proyecto ha sido la implementación efectiva de un entorno de verificación funcional basado en UVM, al que se ha integrado de forma no intrusiva un conjunto de propiedades de verificación formales expresadas en *SystemVerilog Assertions*. Esta integración se ha realizado siguiendo dos metodologías inspiradas en el *paper* "*SVA Encapsulation in UVM: Enabling Phase and Configuration Aware Assertions*" (*Verilab*), permitiendo que las *assertions* sean configurables y conscientes de la fase de simulación. Esta aportación representa un ejemplo práctico y aplicado de una de las tendencias más relevantes en la verificación moderna: el uso combinado de técnicas funcionales y formales dentro de un mismo flujo estructurado.

A nivel metodológico, se ha demostrado que la encapsulación de SVA en interfaces especializadas permite preservar la modularidad del *testbench*, evitando contaminar las interfaces funcionales con lógica de verificación. Además, se ha validado que la configuración

de estas propiedades puede realizarse tanto mediante llamadas explícitas API como de forma automática *phase-aware*, siendo esta última la opción más robusta, escalable y alineada con entornos industriales complejos.

En cuanto a resultados, los *tests* ejecutados han demostrado que el entorno de verificación es capaz de detectar errores funcionales mediante *scoreboard* y mediante la activación de *assertions*. Estas últimas han mostrado su utilidad al detectar condiciones no triviales, como desincronizaciones internas, errores de conteo, o inconsistencias entre señales de estado. También se ha comprobado, mediante el análisis de formas de onda y trazas, que las propiedades desarrolladas no generan falsos positivos y sólo se activan cuando efectivamente se vulnera el comportamiento esperado del IP.

Desde el punto de vista formativo, este trabajo ha supuesto una curva de aprendizaje exigente. La necesidad de adquirir conocimientos avanzados en *SystemVerilog*, UVM, entornos de simulación *QuestaSim*, y el uso de herramientas de control de versiones como GitHub, ha requerido un esfuerzo significativo de autonomía, organización y capacidad de aprendizaje. El dominio progresivo de estos entornos, así como la aplicación efectiva de conceptos de verificación formal.

Líneas futuras

El carácter finalista de este trabajo deja dos posibilidades a la hora de plantar posibles líneas futuras: el desarrollo de un plan completo de verificación formal basado en propiedades y, por otro lado, sustituir el proceso de simulación por el uso de una herramienta de verificación formal.

Partiendo de un IP cuyas especificaciones funcionales sean conocidas, la primera propuesta consiste en desarrollar un Plan de Verificación Formal completo, en el que se detallen, para cada una de las propiedades, el tipo de propiedad que se trata; la característica a verificar; la secuencia de test para alcanzar el estado inicial, y, por último, los criterios de verificación para dicha propiedad. Este trabajo usaría la estructura de integración de las propiedades realizada en el presente Trabajo Fin de Grado, lo que permitiría centrar el trabajo en el Plan de Verificación y no en su integración en el testbench del módulo IP a verificar.

Hasta la fecha, todos los trabajos de verificación realizados en la División de Sistemas Integrados del Instituto Universitario de Microelectrónica Aplicada de la ULPGC se han basado en la simulación como técnica para la realización del proceso de verificación. En estos casos, y como herramienta, se ha utilizado, particularmente, la herramienta Questasim de Siemens. El uso de propiedades de verificación, utilizando la simulación como metodología de base para realizar el proceso de verificación, limita el alcance de las propiedades de verificación. Esto es, las propiedades se comprueban, únicamente, en los estados que se cubran a lo largo del proceso de simulación. Es decir, estas propiedades no se comprueban en el total del espacio de funcionamiento del sistema. Para poder dar un rango de cobertura del 100% de una propiedad, es necesario el uso de una herramienta de verificación formal. Estas herramientas permiten verificar matemáticamente que el diseño cumple con las especificaciones requeridas.

Hoy en día las principales empresas de diseño electrónico, como Siemens, Cadence o Synopsys, ofrecen herramientas de verificación formal para descripción de sistemas a nivel RTL. Hay que hacer hincapié en que el uso de esta metodología está enfocado en la verificación de núcleos IP de baja o mediana complejidad. Su uso en sistemas complejos no es recomendable por dos motivos: el elevado uso de recursos que utiliza este tipo de herramientas y, por otro lado, la complejidad de especificar una propiedad a este nivel.

Anexo A: Presupuesto

Para la realización del presente Trabajo Fin de Grado ha sido necesario disponer de una serie de recursos tanto humanos como materiales. La estimación económica de estos elementos permite cuantificar de forma aproximada el coste asociado al desarrollo completo del proyecto, considerando el valor que cada recurso aporta en función de su tipo, duración de uso y relevancia dentro del proceso de trabajo.

En este anexo se desglosa el presupuesto total, diferenciando entre los conceptos más relevantes, como la dedicación técnica, el equipamiento empleado, el software utilizado y los gastos derivados de la redacción, impresión y presentación del documento final. Todo ello con el fin de ofrecer una visión detallada y justificada del esfuerzo económico implícito en la elaboración de este proyecto académico.

- Coste recursos humanos
- Coste recursos hardware
- Coste de recursos software
- Coste de material fungible
- Coste de redacción de la memoria
- Derechos de visado del COITT
- Gastos de tramitación y envío
- Coste total

A.1. Recursos humanos

El coste asociado a los recursos humanos se ha estimado en función de la retribución que correspondería al perfil técnico encargado del desarrollo del presente proyecto. Se ha tomado como referencia el salario estipulado para un graduado en Ingeniería en Tecnologías de la Telecomunicación, categoría que se ajusta al nivel formativo y responsabilidades asumidas durante el desarrollo del Trabajo de Fin de Grado.

Dado que el trabajo se ha llevado a cabo en el marco académico de la Universidad de Las Palmas de Gran Canaria (ULPGC), se ha aplicado la tarifa correspondiente al personal técnico con titulación mínima de grado o equivalente, conforme a las directrices establecidas por el acuerdo de retribuciones del personal de perteneciente a la ULPGC [20], mostrado en la Tabla 4.

Tipo Personal	Retribución Mensual	Duración	Coste Total
Ingeniero técnico	760,05€	4 meses	3020,20€

Tabla 4: Coste recursos humanos

El coste final del trabajo por el tiempo empleado es de TRES MIL CUARENTA EUROS CON VEINTE CÉNTIMOS.

A.2. Recursos materiales

Durante el desarrollo del Trabajo Fin de Grado se han empleado diversos recursos materiales, incluyendo tanto equipamiento físico (hardware) como herramientas digitales (software). Para estimar el coste asociado al uso de estos recursos, se ha considerado su valor de adquisición y el periodo durante el cual han sido utilizados dentro del marco del proyecto.

El cálculo de este coste se ha realizado siguiendo un modelo de amortización lineal, en el que se reparte el valor útil del recurso a lo largo de su vida estimada. En este caso, se ha tomado como referencia una vida útil de 3 años, conforme a criterios estándar en entornos académicos y profesionales. La fórmula empleada para determinar la cuota de amortización anual es la siguiente:

$$Cuota = \frac{Valor\ de\ la\ adquisición - Valor\ residual}{Tiempo\ de\ vida\ útil}$$

Esta cuota anual se multiplica por el número de meses efectivos de uso proporcional al año, permitiendo así obtener una estimación precisa del coste imputable al proyecto.

A.2.1. Recursos hardware

En la Tabla 5 se muestran los resultados del coste total de los recursos hardware utilizados en el desarrollo de este TFG.

Equipo	Valor Adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Ordenador personal- MSI GP 63 Leopard	1500€	36 meses	41,6€	4 meses	166,4€

Tabla 5: Coste recursos hardware

El coste final de los recursos hardware empleados es de CIENTO SESENTA Y SIES CON CUARENTA CÉNTIMOS.

A.2.2. Recursos software

En la Tabla 6 se muestra el coste total de los recursos software utilizados en el desarrollo del presente Trabajo Fin de Grado. Destaca que la gran mayoría de los recursos usado han sido *OpenSource* o licenciados por la ULPGC.

Equipo	Valor de Adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
OS Microsoft Windows 10	100€	36 meses	2,7€	4 meses	10,8€
Microsoft Office	0€ Licencia ULPGC	36 meses	0€	4 meses	0€
Mozilla Firefox	0€ Licencia ULPGC	36 meses	0€	4 meses	0€
OS Linux	0€ OpenSource	36 meses	0€	4 meses	0€
QuestaSim	20.000€	36 meses	555,55€	4 meses	2222,2€
Librería UVM	0€ OpenSource	36 meses	0€	4 meses	0€
Adobe Reader	0€ OpenSource	36 meses	0€	4 meses	0€

Tabla 6: Coste recursos software

El coste final de los recursos hardware empleados es de DOS MIL DOSCIENTOS TREINTA Y TRES EUROS.

A.3. Material fungible

Los costes derivados del material fungible utilizado en este Trabajo Fin de Grado se muestran en la Tabla 7.

Concepto	Coste
Folios	10,00€
Impresión de TFG	35,00€
Encuadernado	5,00€

Tabla 7: Coste material fungible

El coste final de los recursos hardware empleados es de CINCUENTA EUROS.

A.4. Redacción del trabajo

A partir del presupuesto global del proyecto, es posible estimar el coste asociado a las tareas de redacción, edición y maquetación de la memoria final del Trabajo de Fin de Grado. Para ello, se emplea una fórmula propuesta por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT) [21], que contempla un porcentaje fijo sobre el presupuesto base, corregido mediante un coeficiente ajustado a la escala del proyecto.

La expresión empleada es la siguiente: $R = 0{,}07 * P * Cn$

Donde:

- R representa el coste estimado derivado de la redacción del documento final.
- P es el presupuesto total del proyecto, suma de recursos humanos y materiales.
- Cn es el coeficiente corrector, cuyo valor depende del intervalo presupuestario en el que se encuentra P.

En este caso, el presupuesto estimado P no supera los 30.050,00 €, por lo que el valor de Cn se establece como 1, de acuerdo con los criterios del COITT.

$$R = 0{,}07 * P * Cn = 0{,}07 * 5419{,}6 * 1 = 379{,}37\ €$$

Por tanto, el coste imputable a la redacción del trabajo asciende a TRESCIENTOS SETENTA Y NUEVE EUROS CON TRIENTA Y SIETE CÉNTIMOS.

A.5. Derechos de visado del COITT

El Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT) publica anualmente una tabla de tarifas para el cálculo de los honorarios mínimos y los derechos de visado asociados a proyectos técnicos. En el caso de proyectos de carácter general, como el Trabajo Fin de Grado, el coste del visado se determina mediante una fórmula que tiene en cuenta el presupuesto del proyecto y, en su caso, el presupuesto de ejecución de obra civil.

La fórmula propuesta por el COITT para el cálculo del coste del visado es la siguiente:

$$V = 0{,}006 * P1 * C1 + 0{,}003 * P2 * C2$$

Donde:

- V es el coste final del visado.
- P1 representa el presupuesto general del proyecto.
- C1 es el coeficiente reductor en función del presupuesto P1.
- P2 es el presupuesto correspondiente a ejecución de obra civil.
- C2 es el coeficiente corrector para P2.

En el contexto del presente Trabajo de Fin de Grado, no existe ejecución de obra civil, por lo que P2=0, y el segundo término de la ecuación queda anulado. Además, como el presupuesto total del proyecto no supera los 30.050 €, se adopta el valor de C1=1, según las tarifas recogidas por el COITT para el año vigente.

$$V = 0{,}006 * P1 * C1 + 0{,}003 * P2 * C2 = 0{,}006 * 5798{,}97 * 1 = 34{,}97€$$

Por tanto, el coste correspondiente a los derechos de visado asociados a este proyecto asciende a TREINTA Y CUATRO EUROS CON SETENTA Y NUEVE CÉNTIMOS.

A.6. Gastos de tramitación y envío

Además de los costes técnicos y materiales vinculados al desarrollo del Trabajo de Fin de Grado, es necesario contemplar una serie de gastos administrativos relacionados con su presentación y registro oficial.

De acuerdo con la referencia, los gastos de administración asociados a la tramitación de documentos técnicos ante el COITT oscilan entre 9 € y 12 €, dependiendo de si el visado se realiza en formato digital o manual, respectivamente. En este caso, dado que el proceso se realiza de forma telemática, se considera un visado digital, por lo que se aplica un gasto de administración de NUEVE EUROS (9,00 €).

Este valor cubre los costes asociados a la gestión del documento en formato electrónico, incluyendo su registro, emisión y archivo digital por parte del colegio profesional.

A.7. Presupuesto final del proyecto

Al importe total del presente Trabajo de Fin de Grado se le aplica el Impuesto General Indirecto Canario (IGIC), vigente en la Comunidad Autónoma de Canarias, que corresponde al 7 % del valor neto del presupuesto.

A continuación, se presenta la Tabla 8, resumen con todos los conceptos presupuestarios, incluyendo la aplicación del IGIC:

Concepto	Importe
Recursos humanos	3.040,20€
Recursos hardware	166,40€
Recursos software	2.233,00€
Material fungible	50,00€
Redacción del trabajo	379,37€
Derechos de visado	34,79€
Gastos de tramitación y envío	9,00€
Subtotal	5.912,76€
IGIC (7%)	413,89€
Total con IGIC	6.326,65€

Tabla 8: Coste total del proyecto

Bibliografía:

[1] «2024 Wilson Research Group IC/ASIC functional verification trend report.» Accedido: 14 de marzo de 2025. [En línea]. Disponible en: https://resources.sw.siemens.com/en-US/white-paper-2024-wilson-research-group-ic-asic-functional-verification-trend-report/

[2] «IEEE Standard for Universal Verification Methodology Language Reference Manual» Accedido: 14 de marzo de 2025. [En línea], IEEE. doi: 10.1109/IEEESTD.2020.9195920.

[3] «What is SystemVerilog?» Accedido: 8 de febrero de 2024. [En línea]. Disponible en: https://www.doulos.com/knowhow/systemverilog/what-is-systemverilog/

[4] Chipright, «Engineer - Consultant Jobs - ASIC Verification». Accedido: 8 de febrero de 2024. [En línea]. Disponible en: https://www.chipright.com/consultant-jobs/asic-verification/

[5] «Design Verification Engineer», Meta Careers. Accedido: 8 de febrero de 2024. [En línea]. Disponible en: https://www.metacareers.com/jobs/353283357374475

[6] «SystemVerilog Assertions», ChipVerify. Accedido: 10 de febrero de 2024. [En línea]. Disponible en: https://www.chipverify.com/systemverilog/systemverilog-assertions

[7] M. Litterick, «enabling phase and configuration aware assertions», 2013, Accedido: 8 de febrero de 2024. [En línea]. Disponible en: https://dvcon-proceedings.org/wp-content/uploads/sva-encapsulation-in-uvm-enabling-phase-and-configuration-aware-assertions.pdf

[8] «SystemVerilog», *Wikipedia*. Accedido: 12 de diciembre de 2024. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=SystemVerilog&oldid=1290302788

[9] G. Rao, «Know The Difference Between Verilog And System Verilog», ChipEdge VLSI Training Company. Accedido: 3 de marzo de 2025. [En línea]. Disponible en: https://chipedge.com/know-the-difference-between-verilog-and-systemverilog-2/

[10] *IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language*. Accedido: 14 de marzo de 2025. [En línea]. doi: 10.1109/IEEESTD.2024.10458102.

[11] A. Hegde, «An Overview On System Verilog Testbench», ChipEdge VLSI Training Company. Accedido: 15 de marzo de 2025. [En línea]. Disponible en: https://chipedge.com/an-overview-on-system-verilog-testbench/

[12] «UVM - Universal Verification Methodology | Siemens Verification Academy», Verification Academy. Accedido: 16 de marzo de 2025. [En línea]. Disponible en: https://verificationacademy.com/topics/uvm-universal-verification-methodology/verificationacademy.com/topics/uvm-universal-verification-methodology/

[13] «UVM Component [uvm_component]», ChipVerify. Accedido: 16 de marzo de 2025. [En línea]. Disponible en: https://www.chipverify.com/uvm/uvm-component

[14] «Caché (informática)», *Wikipedia, la enciclopedia libre*. 18 de diciembre de 2024. Accedido: 2 de abril de 2025. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Cach%C3%A9_(inform%C3%A1tica)&oldid=164194152

[15] «SystemVerilog TestBench Example - with Scb», Verification Guide. Accedido: 6 de abril de 2025. [En línea]. Disponible en: https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-with-scb/

[16] M. Zhu, K. Chen, Z. Chen, y N. Lv, «FAIFO: UAV-assisted IoT programmable packet scheduling considering freshness» Accedido: 20 de abril de 2025. [En línea], *Ad Hoc Netw.*, vol. 134, p. 102912, sep. 2022, doi: 10.1016/j.adhoc.2022.102912.

[17] «Lesson 15: Formal Verification and Model Checking», Accedido: 5 de mayo de 2025. [En línea]. Disponible en: https://btu.edu.ge/wp-content/uploads/2023/07/Lesson-15_-Formal-Verification-and-Model-Checking.pdf

[18] V. R. Cooper, *Getting started with UVM: a beginner's guide*, First edition. Austin, TX: Verilab, 2013.

[19] K. A. Meade y S. Rosenberg, *A practical guide to adopting the Universal Verification Methodology (UVM)*, 2. ed. San Jose, Calif: Cadence Design Systems, 2013.

[20] «Actualizacion tablas salariales 2024». Accedido: 10 de mayo de 2025. [En línea]. Disponible en: https://www.ulpgc.es/sites/default/files/ArchivosULPGC/investigacion/actualizacion_tablas_salariales_2024.pdf

[21] «derechos-de-visado-2020-vf.pdf». Accedido: 10 de mayo de 2025. [En línea]. Disponible en: https://www.telecos.zone/media/attachments/2020/05/21/derechos-de-visado-2020-vf.pdf