UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"ANÁLISIS, DESARROLLO Y VERIFICACIÓN DE VIP (VERIFICATION INTELLECTUAL PROPERTY) BASADO EN EL PROTOCOLO AXI-4 LITE MEDIANTE LOS LENGUAJES SYSTEMVERILOG Y PYTHON Y LAS LIBRERÍAS UVM Y PYUVM"

TRABAJO FIN DE GRADO

Septiembre – 2023

AUTOR: Francisco Javier Salido Vidal

DIRECTOR/ES: Roberto Gutiérrez Mazón

Daniel Gutiérrez Castro

AGRADECIMIENTOS

Quisiera aprovechar este pequeño hueco antes de exponer mi estudio para expresar mi gratitud a aquellos a los que, a pesar de ausentarme para dar lo mejor de mí en este trabajo, me han apoyado incondicionalmente, y han supuesto un pilar fundamental para mí este periodo, y no me cabe ninguna duda que lo seguirán siendo en el futuro.

También quiero agradecer a mi profesor de Diseño de Sistemas Electrónicos y tutor académico, Roberto Gutiérrez. Al comenzar mi etapa final en el grado, sentía que ninguna carrera laboral se ajustaba a mis necesidades y me hallaba desorientado, como en un callejón sin salida. De no ser por él y la pasión que me demostró con su asignatura, probablemente mi camino hubiera sido uno muy distinto, y mucho menos gratificante para mí. Dedicado, paciente, siempre ha estado dispuesto a regalarme valiosos consejos que espero haya sabido aprovechar, para que la redacción de este documento alcanzase las expectativas que yo mismo tenía puestos en él.

Por otro lado, todo este trabajo de investigación no hubiera sido posible sin la ayuda de mi tutor técnico y referente Daniel Gutiérrez, quien sin apenas conocerme se arriesgó, apostó por mí, y me ofreció una beca para formarme en el sector de la electrónica reprogramable. No puedo más que agradecerle por abrirme la puerta a este maravilloso mundo, en el que cada día me levanto con ganas de superarme y disfrutar aprendiendo más que un trabajo, una profesión.

Muchas gracias por ayudarme a crecer cada día. Sois parte de este trabajo, y también parte de mí.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE CÓDIGOS

1. INTRODUCCIÓN

1.1. Presentación del tema. Relevancia del ámbito de la verificación en RTL

Hay un denominador común en todos los escritos relacionados con la industria EDA desde la creación de VHDL y Verilog en la década de los 80. No importa cuando fue escrita o de donde provenga la fuente, todos los escritos que tratan sobre sistemas electrónicos hablan sobre la revolución que causaron e las tecnologías de lenguaje HDL en el diseño, síntesis e implementación RTL [1]. A pesar de ser una materia comúnmente desconocida en la sociedad actual, sobre ella se cimienta gran parte de nuestros avances tecnológicos, teniendo una gran participación en las últimas revoluciones industriales.

Esta industria ha crecido exponencialmente en complejidad, agregando capas de dificultad a los diseños. Estas capas de dificultad, a su vez, han generado paulatinamente la necesidad de crear y aplicar diversas metodologías para verificar cada vez unos diseños más versátiles y complejos con la mayor eficacia y eficiencia posible.

Es por ello por lo que la verificación toma un papel crucial en una nuestra seguridad diaria. Sería precario seguir viviendo en un mundo donde los sectores con sistemas críticos, tales como la aviación, la industria o la automoción, no fuesen verificados correctamente. Incluso generaría situaciones de peligro en acciones cuotidianas como coger un ascensor o ascender por unas escaleras mecánicas.

A pesar del peso que tiene la verificación en la sociedad actual, las metodologías de verificación actuales llevan más de una década dominando el mercado sin cambios relevantes (se expondrá con más calma en el estudio del arte), mientras que los diseños aumentan constantemente. Se vuelve imperante la necesidad de comparar las

diversas metodologías para estudiar las diferencias significativas entre ellas, analizar y potenciar los puntos fuertes cada una, además de proporcionarnos información que puede ser útil para vislumbrar el rumbo que tomará el mercado de la industria EDA en el futuro.

1.2. Objetivos del TFG

El objetivo fundamental de este TFG es el diseño de un "VIP multiframework" (por VIP multiframework nos referimos a 2 VIPs programados en diferentes librerías que intenten ser lo más similares posibles y que simulen el mismo resultado) para las librerías UVM, la librería y metodología más adoptada actualmente por los ingenieros de verificación, y pyUVM, que hereda la misma metodología que UVM aunque está soportada en Python. Tras el diseño de los VIPs se compararán los resultados para destacar los puntos fuertes y débiles de cada librería.

El diseño verificado será el bloque esclavo de la interfaz AXI4-Lite (Advanced eXtensible Interface 4 - Lite). La interfaz AXI4-Lite es una simplificación del protocolo AXI desarrollado por ARM. Se utiliza comúnmente en sistemas integrados y SoCs donde una interfaz ligera es suficiente para comunicar diversos IP Cores o periféricos entre ellos. Aporta una baja latencia y un acceso eficiente a los registros, lo que la convierte en una opción para aplicaciones de baja complejidad. Se ha escogido verificar el bloque esclavo ya que es el que aporta mayores funcionalidades automatizadas a la transacción, como veremos en el capítulo 4.

2. ESTUDIO DEL ARTE

2.1. Introducción: Revolución de los HDLs en la industria EDA

Para comprender por qué es necesario comparar las metodologías y librerías existentes en el mercado con nuevas, debemos entender el origen de los HDLs.

Son dos los lenguajes de descripción de hardware que se han establecido como estándares en la industria EDA. A primera vista, a uno podría parecerle redundante, ya que el objetivo sería tener un lenguaje único que lo uniese todo, es por ello por lo que debemos conocer los orígenes de los HDL estándares para responder a esta pregunta – VHDL y Verilog.

2.1.1. "VHDL", estricto y militar

En 1979 el Dpto. de Defensa de los Estados Unidos comenzó su programa VHSIC a diez años vista para responder a las necesidades en microelectrónica requeridas militarmente [2]. Como parte de este proyecto, se pensó en crear un HDL estándar que permitiera transferir datos de diseño y descripciones totalmente independientes de cualquier herramienta de diseño o base de datos, y que además fuese legible por máquina y con una base muy amplia. Este proyecto dio lugar a un lenguaje altamente tipificado y de manera estática, lo que garantizó la detección temprana de errores en tiempos de compilación. A este lenguaje fue bautizado como VHDL (VHSIC Hardware Description Language).

El contrato de desarrollo de VHDL comenzó en 1983 y se le adjudicó a un equipo de Intermetrics, IBM y Texas Instruments [3]. Uno de los requisitos más destacables fue el de utilizar construcciones del lenguaje ADA en la medida de lo posible, de ahí su herencia del lenguaje. La intención era convertir VHDL en un lenguaje estándar, obligando su uso para el diseño y descripción de hardware en el Dept. de Defensa. Esta estandarización tuvo su culminó en el estándar IEEE 1076-1987 [4], donde fue adoptada como el primer lenguaje HDL estandarizado.

2.1.2. "Verilog", flexible y apadrinado por los proveedores

Gateway Design Automation era una compañía IT la cual encargó a Phil Moorby a finales de 1983 desarrollar un producto patentado para la verificación/simulación, saliendo al mercado a principios de 1985. Este producto era Verilog (Verifying Logic) [3].

Las principales inspiraciones de Moorby para el desarrollo de Verilog fueron C y Pascal. Debemos tener en cuenta que los objetivos iniciales del lenguaje eran la simulación lógica y de fallos y, en menor escala, la síntesis lógica. La ambición principal del proyecto era la de permitir realizar análisis estáticos y dinámicos de temporización min-max [5].

Verilog se hizo rápidamente con una cuota de mercado significativa en un sector ya maduro de por sí con otros simuladores por parte de la competencia. Verilog presentaba un lenguaje único, que integraba tanto el diseño como el testbench (banco de pruebas), junto con otras ventajas como una mayor velocidad de simulación y mayor capacidad de diseño.

Sumado a esto, Gateway hizo relaciones comerciales con los proveedores de ASICs para que usaran Verilog como su simulador por defecto en cuanto a señalización de temporización. Al estar en contacto estrecho con los principales vendedores, Verilog tuvo información de primera mano para implementar mejoras en el gate-level timing. Al ser Verilog fruto de un lenguaje propietario, pero con fuertes conexiones con los vendedores de ASICs, pudo distanciarse de su competencia en el ámbito privado, debido a su estandarización "de facto", pero también de su competencia en el ámbito público, ya que Gateway podía adaptarse fácilmente a las necesidades del mercado sin verse anclado al proceso plurianual de la normalización del IEEE.

Debido a este auge en el uso de Verilog, varias compañías compraron la licencia del lenguaje para sus librerías de simulación y para sus herramientas de síntesis.

En 1989, Gateway fue adquirida por Cadence, que inició los trámites de estandarización que tuvieron su fin en diciembre de 1995, cuando se volvió oficialmente un IEEE estándar [3].

2.2. La guerra de las metodologías en el ámbito de la verificación

Tras analizar como tomaron forma los dos gigantes estandarizados en la industria EDA, vamos a destacar sus puntos más interesantes para el tema de investigación.

Por un lado, tenemos VHDL, el lenguaje altamente preparado para realizar diseños y descripciones, y adoptado por varias organizaciones gubernamentales de alto calibre. Por el otro, tenemos Verilog, un lenguaje adoptado por los vendedores, generando ventajas competitivas a los programadores que lo dominaran.

No obstante, recordemos que estos dos lenguajes fueron creados con el objetivo de diseñar, describir, o simular los comportamientos hardware. Se ha discutido ampliamente durante más de treinta años cual es el mejor lenguaje de diseño de hardware y para intentar ganar esa lucha se ha extendido el uso de estos dos lenguajes al ámbito de la verificación, a pesar de que ninguno de los dos fue diseñado inicialmente para generar testbenches. De hecho, el desarrollo de testbenches es la astilla que la industria EDA nunca ha logrado librarse de ella, ya que desde el comienzo ha intentado abarcar un problema de software desde el punto de vista de lenguajes hardware. Varios enfoques han abarcado la necesidad de un lenguaje para la verificación.

2.2.1. "e", innovación en las bases de la verificación

Para entender el problema que supone la verificación, debemos entender que hay diversos componentes que forman parte de un diseño, pero no por ello no siempre se deben verificar todos los aspectos o componentes, lo que requiere poder enfocar el problema de la verificación desde dos puntos de vista distintos.

En 1996, la empresa InSpec fue la que introdujo en la industria estos dos puntos: la programación orientada a objetos y la programación orientada a aspectos. Lo hizo a

través de "*e*", un lenguaje diseñado explícitamente para la verificación. Junto con él, también implementó la primera metodología de verificación, llamada *e*RM – *e* Reuse Methodology [6]. Esta metodología sentó las bases de la técnica de aleatorización restringida, que sigue usándose hoy en día.

Al ser un lenguaje totalmente nuevo, "*e*" ganó popularidad, pero tuvo unas limitaciones muy grandes. Al ser de dominio privado, había que comprar el software de Verisity (la rebautizada empresa InSpec), y ejecutar un testbench puramente en lenguaje "*e*". Además, dada la innovación que traía el lenguaje los equipos necesitaban mucha consultoría para generar sus testbenches. Finalmente, Cadence compró Verisity en el año 2000.

2.2.2. "SystemVerilog", adaptación del hardware al software

En los años posteriores al desarrollo de "*e*", otra empresa llamada *Co-Design Automation* anunció SUPERLOG. SUPERLOG se trataba de una ampliación al lenguaje Verilog para proporcionarle ciertas facilidades en el ámbito de la verificación.

La rivalidad entre "*e*" y SUPERLOG fue crucial para el ecosistema de la verificación ya que atrajo a las grandes empresas a apostar por ellas. Tanto fue así que Synopsys acabó comprando Co-Design, y donando SUPERLOG, junto con otros lenguajes como Vera al organismo de estándares Accellera (partícipe en la creación de VHDL). Accellera fusionó conceptos de Vera, SUPERLOG y el lenguaje de aserción ForSpec, dando lugar a SystemVerilog. [7]

2.2.3. "UVM", la tregua definitiva

Alrededor del año 2006, la industria EDA había pasado de no tener solución a su problema de verificación a tener tres metodologías que intentaban aspirar a ser la estándar para la verificación, estas eran:

- Universal Reuse Methodology (URM). Creada por Cadence, que compró Verisity, soportada por SystemVerilog, SystemC y "*e*".
- Verification Methodology Manual (VMM). Creada por Synopsys, que compró Co-Design, soportada solo por SystemVerilog [8].
- Advanced Verification Methodology (AVM). Creada por Mentor Graphics, escrita completamente en SystemVerilog. [6]

El objetivo de una metodología reutilizable para cualquier situación ya es de por sí complicada si tenemos en cuenta que varios ingenieros pueden abarcar el mismo problema desde puntos de vista distintos, dando lugar a resultados totalmente dispares, aunque igual de válidos. Añadido el problema de que ahora existían tres metodologías distintas, reutilizar componentes de verificación se hacía un evento casi inexistente. Esto suponía un freno absoluto al desarrollo de componentes de verificación cada vez más escalables y complejos. Las compañías, en una rivalidad constante, trataban de intentar luchar por el máximo cupo de ingenieros de verificación para que reescribieran los testbenches de las otras metodologías a la suya.

Lo que estaba claro es que esta situación no era útil para nadie. Por ello en 2008 Mentor Graphics dio un paso adelante y se alió junto a Cadence para crear Open Verification Methodology (OVM), la predecesora de la metodología dominante hoy en día.

Es curioso que Synopsys, la empresa que compró Co-Design y ofreció los recursos a Accellera para crear SystemVerilog, fue la última en unirse. Cuando Mentor Graphics y Cadence donaron OVM a Accellera, Synopsis dio el último empujón uniéndose al equipo para crear, en el IEEE 1800.2 (año 2017), el manual de referencia estándar para la Universal Verification Methodology (UVM), este manual ha sido actualizado posteriormente en 2020 [9].

2.2.4. "OSVVM" y "UVVM", el parche de la comunidad VHDL

Tras varios años bajo la imposición del Gobierno de EE. UU de describir los diseños RTL en VHDL, varios ingenieros fueron acomodados en el lenguaje en sí y se formó una gran librería de IP Cores en VHDL. Años más tarde, al levantarse la restricción de programar en VHDL, la comunidad VHDL vislumbró la guerra existente entre las metodologías de Cadence, Mentor Graphics y Synopsys. Debido a esto, intentaron encontrar una forma de realizar los testbenches en VHDL [10].

Esta situación acabó estancando los testbenches en VHDL hasta 2012, cuando Jim Lewis logró crear una metodología con aleatorización restringida (funcionalidad clave desde la creación de "*e*"), pero basada totalmente en VHDL, la Open Source VHDL Verification Methodology (OSVVM). Años más tarde, en 2015 se creó también la Universal VHDL Verification Methodology (UVVM) por Espen Tallaksen [6]. Estas dos metodologías son usadas ampliamente en la actualidad de la mano de la comunidad VHDL. Sin embargo, no existe una estandarización al respecto de estas dos librerías, en parte porque VHDL fue creada de manera muy fiel a la descripción de diseños RTL, por lo que su síntesis, implementación y, lo que más nos atañe, creación de testbenches, no son óptimas y varían su interpretación dependiendo de la herramienta CAD que se utilice.

2.3. Un enfoque apropiado para un problema de software

Una vez detectadas y analizadas las metodologías actuales, observamos como tenemos OSVVM y UVVM, fruto de la comunidad VHDL que, al ser VHDL un lenguaje más optimizado para diseño debido a su fuerte tipado, se muestra reticente a cambiar de lenguaje para la verificación de los diseños, buscando unificar las dos partes de la creación de circuitos electrónicos bajo un mismo lenguaje. En contraposición, nos encontramos con UVM, un conjugado de sus predecesores que, tras años de una cruel rivalidad, dejaron aparte sus diferencias para crear el mejor enfoque posible hasta el momento para atacar las dificultades de la verificación.

Podríamos decir que UVM es el resultado de un incrementalismo de la industria EDA que ha ido parcheando errores a medida que han ido surgiendo. Este incrementalismo persiste hoy en día, buscando dar el siguiente paso en la dirección correcta para afrontar los desafíos de verificación del futuro. Ray Salemi creó la librería pyUVM en 2020, la cual implementa los estándares IEEE 1800.2 para UVM en Python [6]. Cabe destacar que pyUVM no es una librería integral, sino que está conectada a la librería cocotb (COroutine COsimulation TestBench) escrita en 2003 (aunque el proyecto estuvo bastante abandonado hasta principios de 2019) por FOSSi Foundation (Free and Open Source Silicon Foundation) que ya permitía realizar testbenches para diseños RTL tanto en VHDL como en Verilog, aunque sin aplicar ninguna metodología.

2.3.1. "Python", el candidato a entrevistar

El punto de vista que aporta el autor de la librería es que la siguiente limitación que debemos eliminar es la de percibir el testbench como un componente hardware. Es natural pensar que, al verificar y simular el comportamiento de un diseño de hardware, deberíamos diseñar hardware que envíe señales al DUT y luego compruebe los resultados con los predichos por el testbench. Sin embargo, con la tecnología actual esto no tiene por qué ser así, es más sencillo pensar en que el DUT está conectado a dos buses funcionales que, por mediación de APIs, se conectan con nuestro software, que genera estímulos y revisa resultados, como haría un testbench.

Respecto al lenguaje de programación, la respuesta es clara: Python es el candidato idóneo. Algunos de sus puntos fuertes son:

- **Sintaxis clara y legible.**

  Python no utiliza llaves o palabras clave para delimitar bloques de código, con una mera identación estructura el código de manera limpia y facilita la comprensión del código. Esto, a su vez, reduce la posibilidad de errores.

- **Vasta cantidad de recursos.**

Gracias a la amplia gama de bibliotecas y frameworks, Python puede mutar a gusto del ingeniero de verificación. Estas frameworks pueden aumentar de manera brutal las funcionalidades del código. Como ejemplos, podemos observar “NumPy” y “Pandas”, usados para el análisis de datos, o “datetime” y “time”, que aportan funciones para trabajar con fechas y tiempos. Recordemos que incluso la librería pyUVM está cimentada en cocotb, lo que conforma un pequeño framework en sí.

- **Compatibilidad y versatilidad**

  Python es un lenguaje muy accesible en diferentes entornos ya que es altamente portátil. Gracias a esto, se pueden usar y conectar a él varios simuladores de código abierto, eliminando casi por completo la barrera de entrada de algunos simuladores, los cuales tiene un precio desorbitado para el usuario común. Algunos de estos simuladores open-source a destacar son “GHDL”, para diseños VHDL, e “Icarus Verilog”, para los diseños en Verilog.

- **Comunidad activa y soporte**

  Actualmente Python cuenta con un equipo de desarrolladores que mantiene un soporte sólido continuo, que puede verse reflejado en sus actualizaciones (Python 3.5, por ejemplo, se lanzó en 2015. Actualmente, a fecha 15 de julio de 2023, ya existe un prelanzamiento de Python 3.12).

- **Aprendizaje rápido y curva de entrada baja**

  Esta ventaja es una amalgama derivada de la sintaxis clara y la comunidad activa. La comunidad de Python aporta activamente tutoriales a la comunidad, dando mucha accesibilidad para aprender los conceptos fundamentales de forma rápida y simple. Junto con una de las sintaxis más claras actualmente, hacen que el aprendizaje de Python sea muy liviano.

3. EXPLICACIÓN DEL ESTUDIO COMPARATIVO

Sería idóneo y de mucho valor poder comparar un VIP para un mismo diseño diseñado en cada una de las librerías nombradas durante el estudio del arte. No obstante, el alcance y el tiempo predefinido de ese proyecto supera con creces al de un TFG, ya que la fase de diseño de un VIP suele variar desde meses hasta años. Es por ello, que analizaremos los pros y los contras de las librerías dominantes actualmente nombradas ya en este documento y seleccionaremos una para compararla con la nueva metodología emergente pyUVM.

Los factores para tener en cuenta serán, la media entre los porcentajes de adopción para proyectos relacionados con FPGAs e IC/ASICs y la edad generacional de la metodología.

3.1. Elección de la metodología a comparar

Los datos de la primera variable de decisión serán recopilados en el "The 2022 Wilson Research Group Functional Verification Study" [11], realizado por Harry Foster. Hemos escogido este estudio principalmente debido a su prestigio internacional. No obstante, no lo sería conocido internacionalmente si no fuese por su fiabilidad. Su intervalo de confianza es de 95 ± 3.7%.

3.1.1. Porcentajes de adopción de las metodologías para FPGAs e ICs/ASICs

Hay que destacar antes de comenzar con el análisis, que este estudio acepta múltiples respuestas, por lo que la suma de todos los porcentajes de proyectos no tiene por qué dar un cómputo de 100%, sino que puede darse el caso de utilizarse diversas metodologías para un mismo proyecto, elevando ese número.

En el mercado de los FPGAs está ligeramente dominado por UVM, aunque hubo un auge en 2022 en el uso de las librerías soportadas por VHDL. Se puede ver como cocotb está empezando a abarcar una cota de mercado, a pesar de que su desarrollo

se continuó comenzó a principios de 2019. El gráfico fue extraído del “Wilson Research Group and Siemens EDA, 2022 Functional Verification Study” [11].

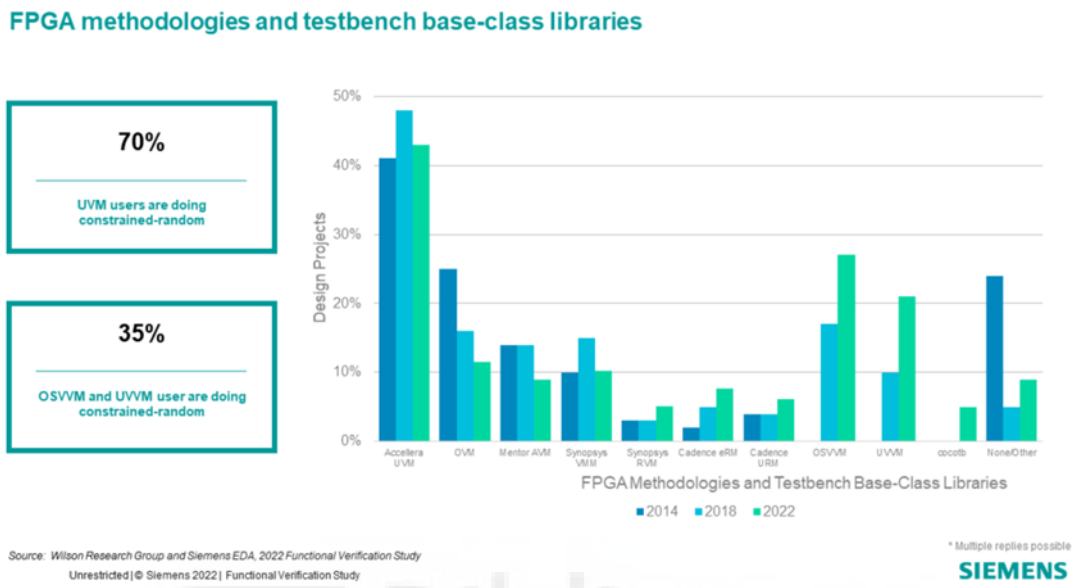

Figura 1. Gráfico de barras que representa el porcentaje de adopción de las metodologías existentes en el mercado de los proyectos relacionados con FPGAs.

Por otro lado, el mercado de los ASICs está mucho más decantado por la metodología UVM, y no vemos ese auge que tenía la comunidad de VHDL en los FPGAs. Como mención especial, de nuevo, cocotb sí que crece el mismo porcentaje que en los FPGAs, por lo que su uso podemos abstraerlo del dispositivo y atribuirlo a otros factores. Gráfico extraído también del “Wilson Research Group and Siemens EDA, 2022 Functional Verification Study” [11].

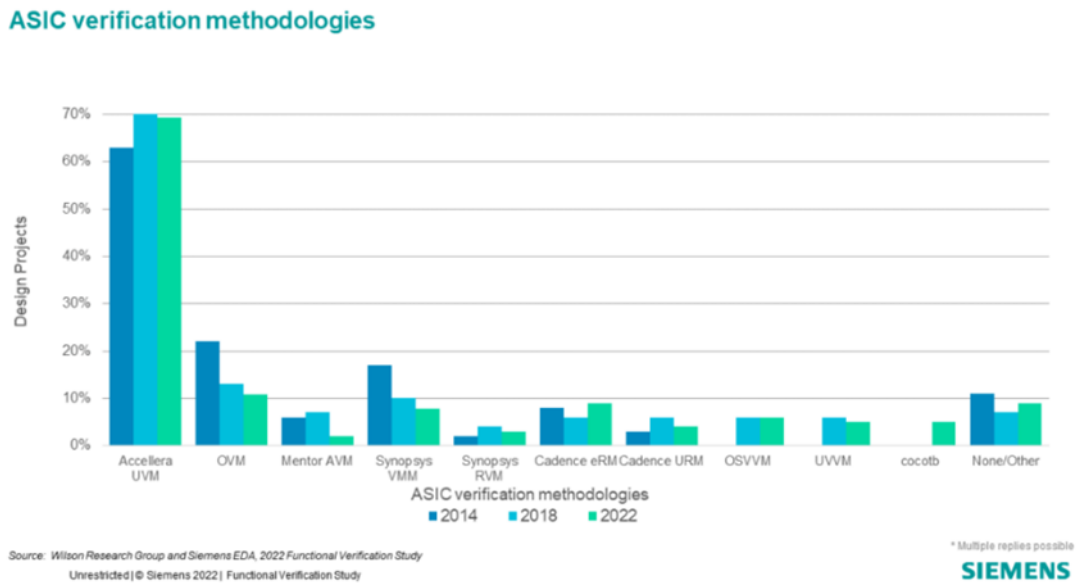

Figura 2. Gráfico de barras que representa el porcentaje de adopción de las metodologías existentes en el mercado de los proyectos relacionados con ASICs.

Es curioso ver como metodologías como OVM e incluso la rivalidad de la tríada URM, VMM o AVM siguen teniendo un cupo mercado en la adopción de testbenches tanto en FPGAs como en ASICs. Incluso *e*RM, debido probablemente a su gran librería de verificación, se mantiene por encima de ellas en 2022 en incluso crece un poco. No obstante, se trata de un cambio muy pequeño que no es para nada asociable a que la metodología esté reviviendo, sino más bien que hay proyectos en los que habrá más VIPs en la librería *e* y otros en los que, en caso de querer hacerse con *e*, deberían programarse desde los cimientos y, por tanto, se adoptará otra metodología.

A continuación, se adjunta una tabla con los porcentajes de adopción de metodologías en ASICs, FPGAs y su media según el Wilson Research [11]:

Metodología	Mercado FPGA	Mercado ASIC	Media de los dos mercados
Accellera UVM	43 %	69 %	56 %
OVM	12 %	11 %	11.5 %
AVM	8 %	2 %	5 %
VMM	10 %	7 %	8.5 %
RVM	5 %	3 %	4 %
***e*RM**	7 %	9 %	8 %
URM	6 %	4 %	5 %
OSVVM	27 %	6 %	16.5 %
UVVM	21 %	5 %	13 %
Cocotb	5 %	5 %	5 %
Other	9 %	9 %	9 %

Tabla 1. Tabla con las cotas de mercado que abarcan cada una de las metodologías, en el mercado de los FPGAs, el de los ASICs, y la media de los 2 en las metodologías encuestadas.

La industria EDA es muy característica, en la tabla de arriba podemos apreciar la resistencia al cambio. Esto es debido, como ya se ha nombrado repetidas veces, a la enorme cantidad de programas realizados en metodologías anteriores y que pueden ser medianamente reutilizados. Los proyectos de verificación tienen una duración desde meses a años, por lo que poder reutilizar partes de un proyecto anterior puede salvar mucho tiempo.

Ahora sí, podemos afirmar que más de la mitad de los proyectos de verificación del año 2022 utilizaron la metodología Accellera UVM en mayor o menor medida. De hecho, ninguna de las otras librerías destaca respecto a las demás, las únicas que despuntan son OSVVM y UVVM, pero esto es atribuido probablemente a que la comunidad VHDL busca acomodarse en el lenguaje y no pasar a otra metodología.

3.1.2. Fechas de creación de las anteriores metodologías.

Durante el estado del arte nombramos varias de las fechas de creación de diversas metodologías, aunque faltaron algunas. A continuación, se redacta una lista con las metodologías y los años en las que fueron lanzadas al mercado según el Wilson Research [11].

Metodología	Año de creación
Accellera UVM	2011
OVM	2008
AVM	2006
VMM	2005
RVM	2003
***e*RM**	2002
URM	2007
OSVVM	2016
UVVM	2015
Cocotb	2013
Other	X

Tabla 2. Año de creación de las metodologías encuestadas.

Apreciando los años de creación, se ha de valorar el apoyo que ha hecho la comunidad de ingenieros VHDL para utilizar las librerías de UVVM y OSVVM, a pesar de ser las más recientes en el mercado.

3.1.3. Toma de decisiones

A pesar del gran crecimiento que están teniendo las metodologías de OSVVM y UVVM, aún les queda un largo trecho para ser igual de influyentes que UVM, que se considera el estándar de la industria. Ésta, además, comparte los mismos estándares IEEE que pyUVM. Sería interesante comprobar en otro trabajo de investigación el nivel de similitud que logra plasmar UVVM de la metodología UVM a través de VHDL.

Por ende, elegiremos finalmente realizar dos VIPs, uno en la librería pyUVM de Python y otro en la librería UVM de SystemVerilog.

3.2. Valores sujetos a evaluación

Dado que tanto la librería UVM en SystemVerilog como pyUVM en Python cumplen los estándares IEEE de UVM, los resultados del diseño de verificación serán muy similares.

Para profundizar más en las escasas diferencias que tiene, ahondaremos en las siguientes métricas relacionadas tanto con el test como con la librería en sí:

- Velocidad de ejecución.
- Líneas de código.
- Uso relativo de CPU.
- Uso relativo de RAM.
- Existencia de soporte o librería de VIPs.
- Implementación de la técnica CRV (Constrained Random Verification).
- Compatibilidad con simuladores.
- Cobertura funcional.

Además, otros valores más abstractos tales como la barrera de entrada para los dos lenguajes, la compatibilidad con otros lenguajes y la percepción social respecto a las dos librerías también serán considerados en el capítulo 6.

4. AXI4-LITE. ADVANCED EXTENSIBLE INTERFACE 4 LITE

En este capítulo se explicará el funcionamiento tanto del bloque esclavo, que es el diseño para el cual se realizará una verificación, como del bloque maestro de la interfaz del AXI4-Lite, ya que el diseño de verificación simulará las señales que le debería enviar el maestro. El diseño del bloque esclavo a verificar ha sido proporcionado por el simulador Vivado Design Suite, que aporta una gran librería de IP cores para su uso. A grandes rasgos el diseño es adecuado para usarlo como sujeto de verificación, si bien es cierto que carece de alguna funcionalidad, además de haber sido modificado para que sea compilado por el simulador open-source GHDL, requerido para la simulación en pyUVM del que hablaremos más adelante [12].

La interfaz AXI4-Lite se compone de un esquema formado por dos bloques, el maestro y el esclavo. El funcionamiento general del AXI4-Lite se basa en el intercambio de transacciones de lectura y escritura. Una transacción de lectura es una solicitud para leer datos, mientras que una transacción de escritura es una solicitud para escribir datos. Para realizar estas solicitudes se requiere un conjunto de señales asociadas a cada solicitud, además de las señales que finalmente transporten los datos en caso de que se acepte la solicitud. Para organizar las señales de forma más clara, se agrupan en canales, que simplemente son un grupo de señales que guardan relación entre sí.

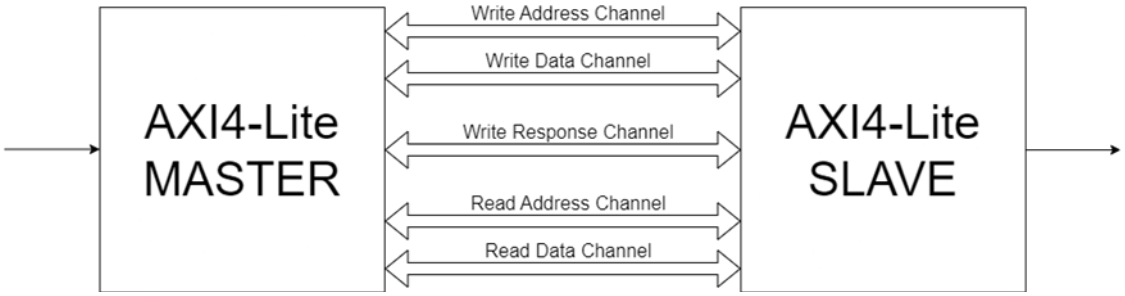

Figura 3. Esquema de bloque esclavo y maestro de la interfaz AXI4-Lite, y canales por los cuales se conectan para realizar las distintas transacciones.

En la Figura 3 se observan cinco canales, tres dedicados para la transacción de escritura y dos para la transacción de lectura.

Comenzando por el grupo de arriba, el primer canal es el “Write Address Channel”, que es el encargado de realizar enviar la dirección de escritura del dato que queremos escribir. En segundo lugar, bastante cerca al primer canal, tenemos el “Write Data Channel”, en el cual se envía el dato a escribir. Finalmente, tenemos el “Write Response Channel”, que verifica el estado de la transacción de escritura.

Los otros dos canales que conforman el grupo de abajo son los homólogos a los canales del grupo de arriba para la transacción de lectura, con el añadido que “Read Data Channel” tiene incluida la función de verificar el estado de la transacción de lectura, al ser ésta más simple.

4.1. Lista de señales

Tabla B1-1 - AXI4-Lite Interface signals

Global	Write Address Channel	Write Data Channel	Write Response Channel	Read Address Channel	Read Data Channel
ACLK	**AWVALID**	**WVALID**	**BVALID**	**ARVALID**	**RVALID**
ARESETn	**AWREADY**	**WREADY**	**BREADY**	**ARREADY**	**RREADY**
-	**AWADDR**	**WDATA**	**BRESP**	**ARADDR**	**RDATA**
-	**AWPROT**	**WSTRB**	-	**ARPROT**	**RRESP**

Tabla 3. Tabla de señales de la interfaz AXI4-Lite. Tabla extraída de AMBA AXI and ACE Protocol Specification.

A continuación, se describen las señales nombradas en la Tabla 3, extraídas de AMBA AXI and ACE Protocol Specification [13], y su función en el diseño.

4.1.1. Señales globales

Señal	Fuente	Descripción
ACLK	Fuente de reloj	Señal general del reloj, se encarga de sincronizar el diseño y registrar secuencialmente algunos procesos.
ARESETn	Fuente de reset	Señal general del reset. Reestablece los valores por defecto en caso de ser, activado. Activo a nivel bajo, se ha implementado un reset síncrono en el diseño.

Tabla 4. Señales globales de la interfaz AXI4-Lite.

4.1.2. Write Address Channel

Señal	Fuente	Descripción
AWVALID	Maestro	Indica que el canal está enviado un dato de dirección de escritura válido.
AWREADY	Esclavo	Indica que el esclavo está listo para aceptar la dirección de escritura.
AWADDR	Maestro	Contiene los datos de dirección de la transacción de escritura.
AWPROT	Maestro	Indica el privilegio y nivel de seguridad de la transacción de escritura, y si se trata de una transacción de acceso a datos o acceso a instrucciones.

Tabla 5. Señales del canal de dirección de escritura de la interfaz AXI4-Lite.

4.1.3. Write Data Channel

Señal	Fuente	Descripción
WVALID	Maestro	Indica que el canal está enviado un dato de escritura válido.
WREADY	Esclavo	Indica que el esclavo está listo para aceptar el dato de escritura.
WDATA	Maestro	Contiene los datos de dirección de la transacción de escritura.
WSTRB	Maestro	Indica que bytes de WDATA tienen datos de escritura válidos. Hay un bit de WSTRB por cada bit de WDATA.

Tabla 6. Señales del canal de escritura de la interfaz AXI4-Lite.

4.1.4. Write Response Channel

Señal	Fuente	Descripción
BVALID	Esclavo	Indica que el canal está enviado un dato de respuesta válido.
BREADY	Maestro	Indica que el maestro está listo para aceptar una respuesta de la escritura.
BRESP	Esclavo	Indica el estado de la transacción de escritura.

Tabla 7. Señales del canal de respuesta de transacción de escritura de la interfaz AXI4-Lite.

4.1.5. Read Address Channel

Señal	Fuente	Descripción
ARVALID	Maestro	Indica que el canal está enviado un dato de dirección de lectura válido.
ARREADY	Esclavo	Indica que el esclavo está listo para aceptar el dato de dirección de lectura.
ARADDR	Maestro	Contiene los datos de dirección de la transacción de lectura.
ARPROT	Maestro	Indica el privilegio y nivel de seguridad de la transacción de lectura, y si se trata de una transacción de acceso a datos o acceso a instrucciones.

Tabla 8. Señales del canal de dirección de lectura de la interfaz AXI4-Lite.

4.1.6. Read Data Channel

Señal	Fuente	Descripción
RVALID	Esclavo	Indica que el canal está enviado un dato de lectura válido.
RREADY	Maestro	Indica que el maestro está listo para aceptar el dato de lectura.
RDATA	Esclavo	Contiene los datos de dirección de la transacción de lectura.
RRESP	Esclavo	Indica el estado de la transacción de lectura.

Tabla 9. Señales del canal de lectura de la interfaz AXI4-Lite.

4.2. Funcionamiento de las transacciones

Dependiendo del tipo de transacción que realicemos, tomarán parte unos canales u otros. Como bien indican sus nombres, para una transacción de lectura los canales habilitados serán el Read Address Channel y el Read Data Channel. De igual manera, para la transacción de escritura lo que los canales asociados serán los tres restantes, el Write Address Channel, el Write Data Channel y el Write Response Channel.

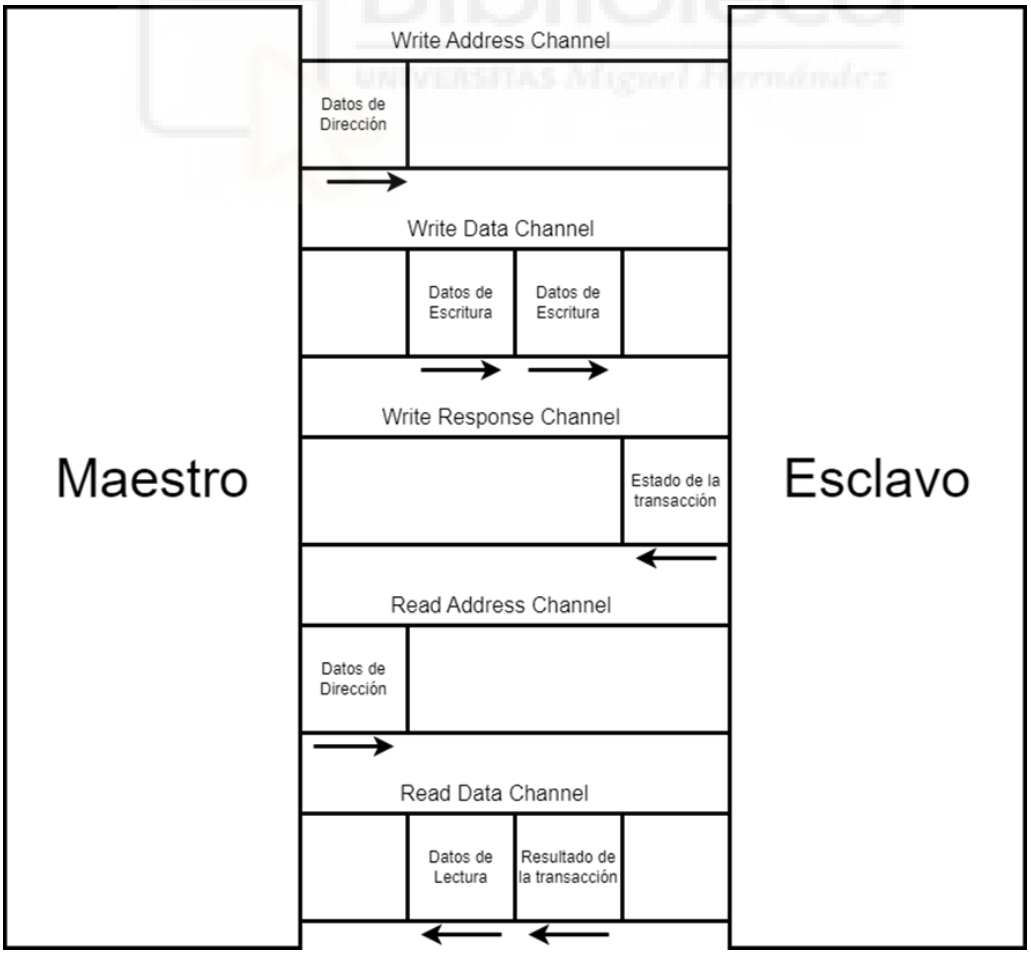

Figura 4. Funcionamiento de las transacciones de escritura y lectura de la interfaz AXI4-Lite.

4.2.1. Transacción de lectura

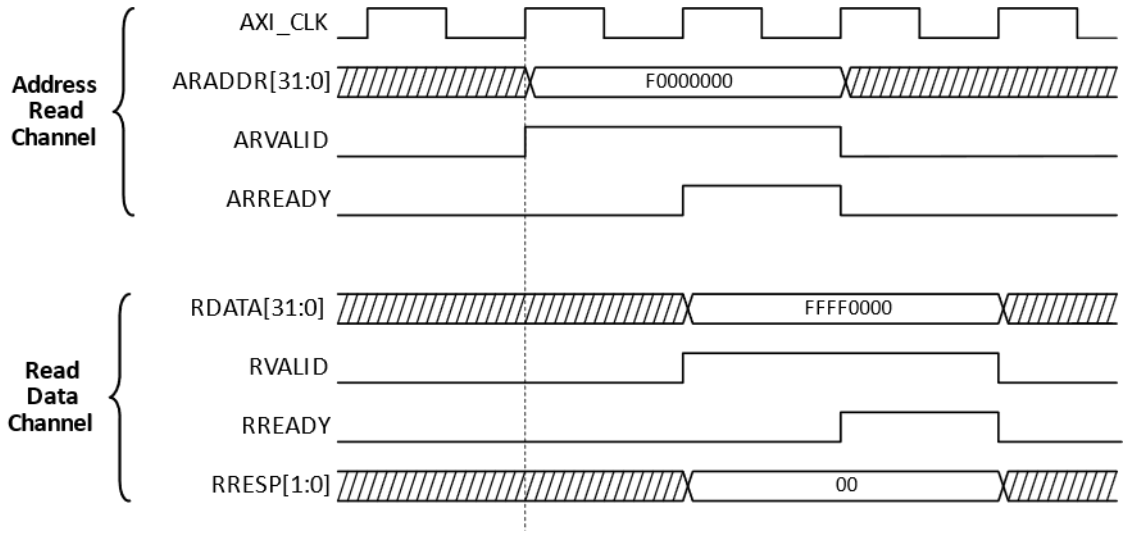

Figura 5. Forma de onda de una transacción de lectura.

Para poder explicar la transacción de lectura y posteriormente la de escritura, primero se debe comprender cómo funciona una forma de onda. Estas formas de onda son representaciones gráficas de las señales que existen dentro de un circuito. Estas señales interactúan entre sí y se comportan de manera específica dependiendo del circuito. Las formas de onda son útiles tanto para visualizar el resultado del circuito como para crear un modelo del comportamiento esperado.

Se comenzará explicando la transacción de lectura representada en la Figura 5, extraída en la web RealDigital [14], ya que es la más sencilla de comprender de las dos. La interfaz AXI4-Lite es una interfaz síncrona, sensible al flanco de subida del reloj, que en este caso es AXI_CLK. Por lo tanto, las señales actualizarán su valor cada vez que el reloj pase de un nivel bajo a un nivel alto.

La transacción comienza cuando el maestro activa la señal ARVALID y ARADDR al esclavo, indicando que el valor que se muestra por ARADDR es válido. El valor de ARADDR no tiene por qué introducirse en el mismo momento que ARVALID, pero no se tomará como válido hasta que la ARVALID esté activo. Al siguiente flanco de subida, el esclavo reconoce el valor de ARVALID activo y activa ARREADY para confirmarle que ha

recibido el valor de la dirección. El envío de la dirección se da por finalizado cuando ARVALID y ARREADY están los dos activos, a esto se le conoce como handshake de dirección de lectura.

Por otro lado, a la vez que el esclavo activa ARREADY; también envía RDATA y RRESP, y activa RVALID. El maestro, al capturar al siguiente flanco el valor alto de RVALID, activa RREADY para confirmar que el dato ha llegado correctamente, terminando el handshake de lectura de datos. Si fallara tanto el handshake de dirección de lectura como el de lectura de datos, RRESP dejaría de envíar el valor “00”, y cambiaría a otro valor, el cual ayudaría a determinar el error que ha sucedido.

La Figura 6 muestra dos diagramas de flujo, desde el punto de vista de cada bloque, que condensa lo explicado en este apartado.

Figura 6. Diagrama de flujo de la transacción de lectura desde el punto de vista del bloque maestro (a la izquierda) y el bloque esclavo (a la derecha).

4.2.2. *Transacción de escritura*

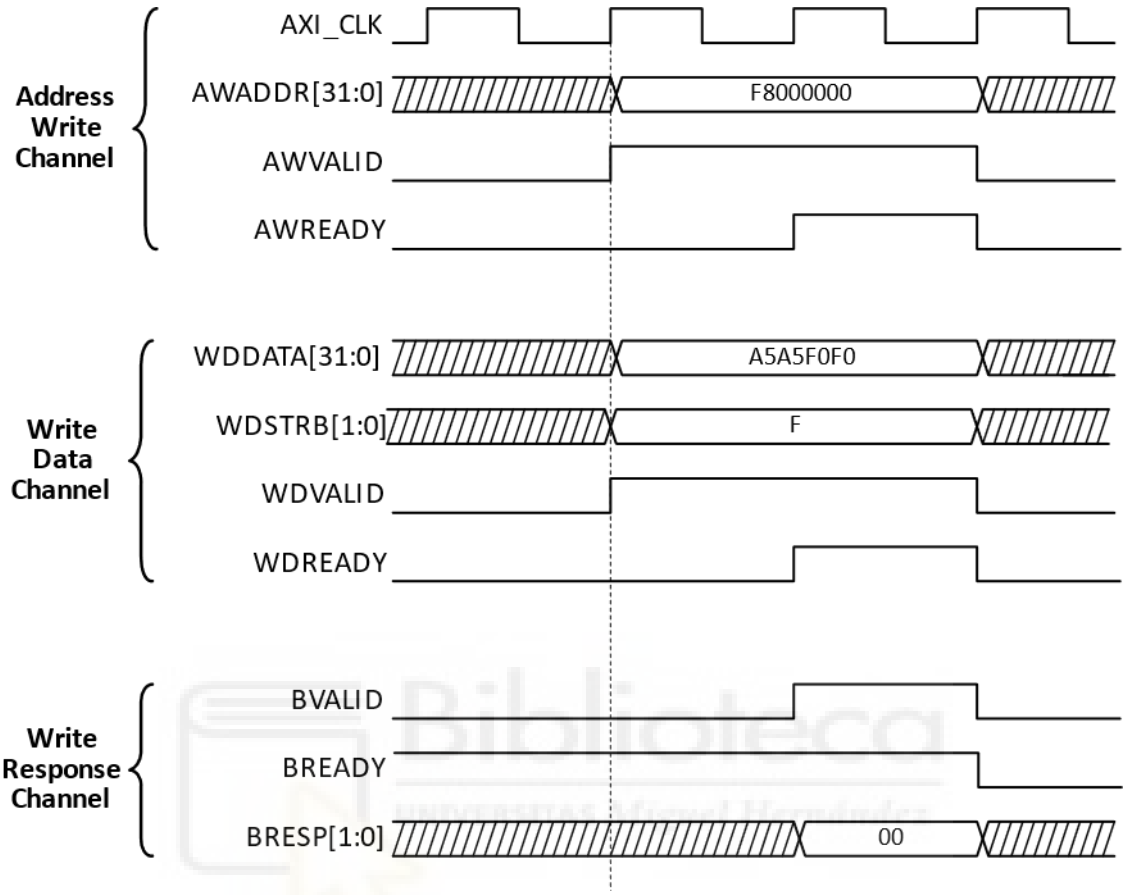

Figura 7. Forma de onda de una transacción de escritura.

La imagen obtenida en la web RealDigital [14] utiliza el prefijo WD en para las señales asociadas al Write Data Channel. No obstante, se seguirá usando la terminología WDATA, WSTRB, WVALID y WREADY en este trabajo para facilitar su comprensión y unificar términos con la transacción de escritura.

La transacción de escritura es algo más compleja para asegurar una mayor fiabilidad a la hora de escribir los datos. Comienza de nuevo con el maestro enviando datos al bloque esclavo. Los datos necesarios son AWADDR, WDATA y WSTRB. Una vez enviados los datos se activan de forma paralela AWVALID, WVALID y BREADY, las dos primeras para indicar que los valores introducidos son válidos y la última para indicar que el maestro está listo para recibir una respuesta de la transacción. De manera

análoga a la transacción de lectura, el bloque esclavo activará AWREADY y WREADY cuando haya recibido correctamente los datos, realizando el handshake de dirección de datos de escritura y de datos de escritura.

En el momento que el esclavo haya confirmado los dos handshakes de escritura, activará entonces BVALID. Esto es importante ya que en esta forma de onda no se representa, pero puede darse el caso de enviar la dirección primero y los datos después. Si el handshake de dirección de datos de escritura ha sido confirmado pero el de los datos de escritura no, BVALID no se activará. Cuando BVALID se active, el esclavo también mandará por BRESP el estado de la transacción, enviando un 00 si no ha habido errores.

La Figura 8 representa en un diagrama de flujo los pasos a seguir de los dos bloques en la transacción de escritura.

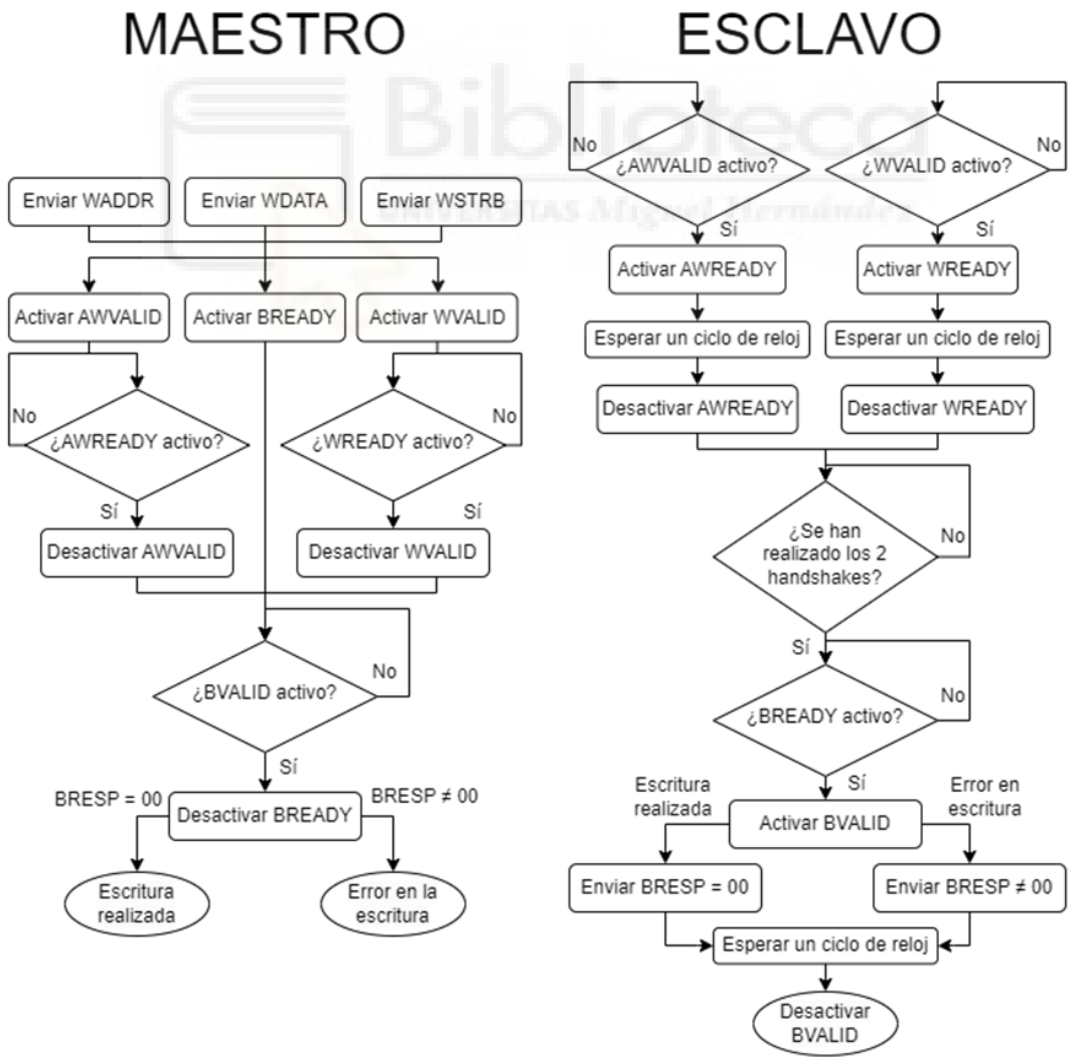

Figura 8. Diagrama de flujo de la transacción de lectura desde el punto de vista del bloque maestro (a la izquierda) y el bloque esclavo (a la derecha).

5. DISEÑO DE VERIFICACIÓN PARA AXI4-LITE

Ya se ha hablado previamente en el estado del arte que una de las razones por las que las metodologías buscan que sus programas sean reutilizables y escalables es debido a que los programas de verificación son muy complicados y costosos de realizar. Para crear una estructura que permita hacer eso, las metodologías también acaban dotadas de cierto nivel de complejidad. Por hacer una analogía, se podría decir lo mismo de un proceso industrial el cual, para automatizarlo, primero debemos entender los procesos o componentes comunes en toda automatización, como pueden ser la logística del proceso, la manipulación del producto, el almacenaje, etc. Es por ello por lo que este capítulo tendrá un subapartado al comienzo explicando, de manera conceptual, cómo estructura UVM los testbenches, familiarizando ciertos conceptos que luego se verán en el diseño de verificación.

5.1. Estructura de un testbench de UVM

Recuperando el ejemplo de un proceso de automatización, UVM se fundamenta en entender las tareas afines a todo proceso de verificación, separándolas y clasificándolas, de manera que cada parte cumpla una única labor [9]. Al estar escrito en SystemVerilog, UVM abarca los testbenches desde un punto de vista de hardware, por lo que los encargados de realizar las labores o tareas de un testbench también son componentes que, a su vez, están encapsulados en otros componentes, hasta llegar al componente principal del testbench, que es el Test en sí.

En la Tabla 10 podemos ver una tabla con cada una de las tareas esenciales que debe realizar un testbench, junto con el componente que se encarga de realizarlas.

Tarea	Componente	Descripción
Generación de estímulos	Sequencer	Genera y controla secuencias de estímulos para comprobar el comportamiento del componente a verificar.
Conexión del testbench con el DUT	Interface	Encapsula las señales del testbench para facilitar la comunicación entre el testbench y el DUT.
Envío de estímulos al DUT	Driver	Envía los estímulos generados al DUT siguiendo la secuencia definida anteriormente.
Recogida de resultados del DUT	Monitor	Captura y registra las señales de salida del DUT una vez se han enviado los estímulos.

Comparación de resultados	Scoreboard	Compara y verifica las señales de salida generadas por el DUT para validar el comportamiento del diseño.

Tabla 10. Tabla de las tareas fundamentales de un testbench, junto con el componente de UVM que las realiza.

Una vez determinados los componentes esenciales, solo queda ir añadiendo contenedores que agrupen los componentes, hasta crear una estructura única que lo unifique todo. Este enfoque de desarrollo que permite desglosarlo todo en partes más pequeñas y manejables se le conoce como programación "top-down".

La Tabla 11 recoge los componentes que permiten desglosar el Test en las tareas fundamentales.

Componente	Descripción
Agent	Contiene a los dos componentes que se conectan a la interfaz, el Driver y el Monitor. Puede ser de dos tipos, activo o pasivo. Si el agente es pasivo, significa que no se controla al DUT, por lo tanto, se desactiva el Driver. Si por el contrario es activo, el Driver controla al DUT.
Environment	Contiene al Agent, al Scoreboard y al Sequencer. Representan el entorno de verificación del diseño de forma totalmente funcional, aunque no genera nada por sí solo.
Test	Contiene al Environment y a las Sequences. Una Sequence, traduciendo al castellano, secuencias que son transmitidas al Sequencer dentro del environment, para que las acate y las siga.

Tabla 11. Tabla de los componentes restantes de la estructura de un Testbench de UVM.

Tras leer la Tabla 11 aparece el concepto Sequence, el cual no se había hablado hasta ahora. Aunque la estructura de componentes de UVM está completada, carecen de funcionamiento si no se traspasan los estímulos entre ellas. UVM desglosa también los estímulos y los divide en tres partes, llamados objetos. Estos objetos son los que se transfieren entre los distintos componentes del Test para verificar el diseño. La Tabla 12 enumera los tres objetos principales que soporta UVM.

Objeto	Descripción
Transaction	Estructura de datos y señales necesarias para conectarse y enviar información al DUT.
Sequence	Secuencia de estímulos que representan un escenario de prueba para la verificación del DUT.
Sequence_Item	Encapsula la secuencia de estímulos que representa un escenario de prueba junto con una estructura aleatoria de señales, buscando verificar casos totalmente aleatorios, mejorando la fiabilidad de la verificación.

Tabla 12. Tabla de objetos de la estructura de un testbench de UVM.

Se puede profundizar más en la estructura de un testbench de UVM. No obstante, esto escapa al alcance de este estudio, ya que el objetivo es aportar la información suficiente para comprender los testbenches que se van a realizar. Si se quiere ahondar en estos conceptos, se recomienda acceder al manual de referencia del IEEE para UVM [9].

5.2. Diseño del VIP

El objetivo de este estudio comparativo es intentar realizar un VIP multiframework, de manera que el enfoque al verificar el diseño sea el mismo. Esto permitiría supuestamente, salvando la barrera del lenguaje de programación, realizar testbenches en cualquiera de los dos lenguajes y poder pasar instantáneamente al otro. Esto llevado a la práctica no es tan sencillo, ya que a pesar de que las dos librerías utilizan la misma metodología, ambas tienen ciertos rasgos característicos que las distinguen. Para visualizar la distancia real que hay entre ellas, se procederá a explicar en primer lugar el procedimiento planteado para verificar el AXI4Lite y, posteriormente, explicaremos tanto el comportamiento que tendrán los diferentes componentes de UVM como la formación de los objetos de UVM para cada uno de los dos testbenches en los que verificaremos el diseño en base al procedimiento explicado previamente.

Para preservar el *know-how* de la empresa, las aportaciones gráficas en forma de código de los componentes de ambos testbenches serán presentadas como pseudocódigo. De esta manera se facilitará la comprensión del lector a la vez que protegerá los conocimientos técnicos aplicados por parte de la empresa.

5.2.1. Test Procedure

Uno de los puntos fuertes que destaca de las metodologías de verificación es que son reutilizables, lo que a priori puede parecer algo beneficioso para la industria. Nada más lejos de la realidad, al comenzar a reutilizar iteradas veces un diseño de verificación, su lectura se hace cada vez más y más compleja, obstaculizando a los nuevos ingenieros su compresión, sin entender qué hace realmente el testbench y, por ende, limitando su potencial para reutilizarlo o escalarlo en el futuro.

Para paliar esta situación, la industria EDA se apoya en los test procedures. Se denomina "Test Procedure" al documento que explica de forma verbalizada el enfoque desde el cual se abarca un problema de verificación, explicando los pasos a seguir de forma verbalizada. Esto soluciona casi por completo (dependiendo, por supuesto, de la calidad del test procedure) la barrera de conocimientos entre los creadores de un testbench y los miembros del equipo que se unan al proyecto más adelante y quieran entender el contexto global del mismo. Un test procedure puede tener una extensión muy variada dependiendo de la especificación que quiera abarcar, ya que puede describir tanto la estructura del test, como algunos ficheros o librerías de soporte, archivos en los que se guardará el resultado de la simulación, requisitos que cumple (si es en base a un proyecto firmado por empresas las cuales los han estipulado), etc. Para este caso, se realizará un test procedure en el que se especificará solamente el objetivo del testbench y los pasos planteados para llevarlo a cabo:

5.2.1.1. Objetivo

El objetivo del testbench, es verificar el bloque esclavo del diseño AXI4Lite, separando la transferencia de datos de la transacción en sí, y separando la transacción de escritura de la de lectura.

Para ello, se dividirá el testbench en tres tests más específicos, que abarcarán cada uno una funcionalidad distinta del diseño, además de separar también los distintos niveles de funcionalidad (siendo la transacción un nivel inferior a la transferencia de datos).

5.2.1.2. Tests

El primer test comprobará la transacción de escritura, sin comprobar si la escritura se realiza correctamente, simplemente verificando que los handshakes se realizan satisfactoriamente. Para ello, estimularemos unas señales de entrada que simulen el comienzo de una transacción de escritura. El bloque esclavo debe responder asertivamente a esos estímulos para considerarlo como test superado. En la Tabla 13 se recogen más detalladamente los resultados esperados del test.

Paso	Descripción	Resultados Esperados
1	Verificación del handshake en el Write Address Channel	• AWREADY debe tener un flanco de subida. • AWREADY debe tener un flanco de bajada.
2	Verificación del handshake en el Write Data Channel	• WREADY debe tener un flanco de subida. • WREADY debe tener un flanco de bajada.
3	Verificación del handshake en el Write Response Channel	• BVALID debe tener un flanco de subida. • BVALID debe tener un flanco de bajada. • BRESP debe ser '0' en todo momento.

Tabla 13. Tabla de pasos y resultados esperados para el test de transacción de escritura.

El segundo test comprobará la transacción de escritura, el alcance sólo será a nivel transaccional, por lo que estimularemos unas señales de entrada que simulen el comienzo de una transacción de lectura. El bloque esclavo debe responder asertivamente a esos estímulos para considerarlo como test superado. En la Tabla 14 se recogen más detalladamente los resultados esperados del test.

Paso	Descripción	Resultados Esperados
1	Verificación del handshake en el Read Address Channel	• ARREADY debe tener un flanco de subida. • ARREADY debe tener un flanco de bajada.
2	Verificación del handshake en el Read Data Channel	• RVALID debe tener un flanco de subida. • RVALID debe tener un flanco de bajada. • RRESP debe ser '0' en todo momento.

Tabla 14. Tabla de pasos y resultados esperados para el test de transacción de lectura.

El tercer y último test comprobará la transferencia completa de datos de manera íntegra, escribiendo un dato en uno de los registros y posteriormente leyéndolo. El alcance de este test será completo, por lo que el bloque esclavo debe enviar correctamente el dato introducido para considerarlo como test superado. De esta manera, verificaremos tanto la escritura en un registro como la lectura de este. En la Tabla 15 se recogen más detalladamente los resultados esperados del test.

Paso	Descripción	Resultados Esperados
1	Verificación de la transacción de escritura	• Debe realizarse el handshake en el Write Address Channel • Debe realizarse el handshake en el Write Data Channel • Debe realizarse el handshake en el Write Response Channel
2	Verificación de la transacción de lectura	• Debe realizarse el handshake en el Read Address Channel. • Debe realizarse el handshake en el Read Data Channel.
3	Verificación de los datos leídos	• Los datos leídos por RDATA deben coincidir con los escritos, que dependen de WDATA y WSTRB.

Tabla 15. Tabla de pasos y resultados esperados para el test de transferencia de datos completa.

Si alguno de los resultados esperados no llegase a cumplirse, el test en cuestión sería tomado como fallido.

5.2.2. VIP en UVM

Para llevar a cabo el procedimiento especificado en el test procedure, la estructura general del testbench deberá seguir lo esquematizado en la Figura 9.

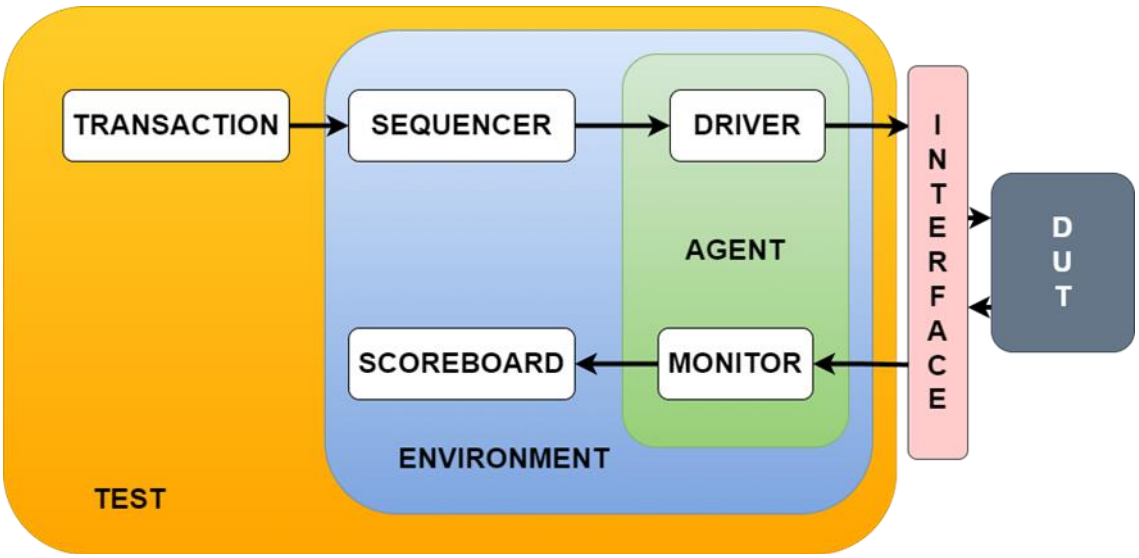

Figura 9. Diagrama con el funcionamiento del testbench diseñado en SystemVerilog mediante la librería UVM.

En la Figura 9 están plasmados los componentes que cumplirán con todas las funciones de este testbench. A continuación se procederá a explicar cada uno de los engranajes que lo forman.

5.2.2.1. Entorno

En la actualidad no hay muchas herramientas que permitan simular la librería UVM de forma gratuita. Sí que es cierto que hay algunas herramientas online como EDA Playground [15], que permiten una simulación gratuita, pero que sus funcionalidades se quedan muy cortas comparadas con las demás herramientas, dificultando mucho la simulación a gran escala con diferentes archivos. Esto la convierte más en una web de experimentos rápidos (como bien indica su nombre) que en un simulador de UVM. El único simulador relativamente decente es el que ofrece Xilinx en la versión Vivado Design Suite. En ella, permite simular la librería UVM sin coste alguno, (la versión de pago se diferencia de la gratuita en la cantidad de dispositivos en los que se puede implementar el diseño, pero la simulación tiene las mismas funcionalidades y especificaciones). Es por ello por lo que se ha escogido esta herramienta para simular el testbench, específicamente la versión Vivado Design Suite v2023.1.

Para realizar la simulación en Xilinx Vivado, se debe crear un proyecto en la herramienta. Una vez creado, se debe abrir la consola de comandos TCL y añadir todos los archivos con los diferentes "componentes" del testbench UVM con el comando "*add_files /%PATH%/<design>.vhd*", siendo %PATH% la ruta absoluta donde se localiza el archivo y <design> el diseño a añadir. Vivado, al igual que otras herramientas CAD de pago, gestionan automáticamente el orden de compilación de documentos, funcionalidad que agiliza bastante la mayoría de los proyectos (aunque se puede dar la situación de que se genere un orden de compilación erróneo, en cuyo caso se debería manipular manualmente).

A pesar de no ser una de las razones por las que se ha escogido este simulador, Vivado ofrece una GUI muy potente que nos permite realizar todas las acciones realizadas previamente inclusive hay algunas que exclusivamente se pueden realizar mediante la GUI, como el cambio de la escala de tiempo del test, acción que también ha sido realizada para obtener mayor precisión en el test. En este estudio no se quiere profundizar en la GUI de Vivado, por lo que para más información se recomienda consultar la guía de usuario de Vivado [16]. Finalmente se debe escribir el comando "*launch_simulation"*. Tras realizar la simulación, se deberá escribir el comando "*open_wave_database <waveform_file>.wd"* para abrir el archivo generado con la forma de onda, siendo <waveform_file> el nombre del documento generado por Vivado automáticamente.

5.2.2.2. Package

Para una mayor claridad en el código, se han separado en un paquete la declaración de los parámetros genéricos del diseño y el tipo de datos especial enumerado "*t_mode*". Por dato enumerado se entiende que es una señal que guarda un conjunto de elementos. SystemVerilog permite hacer datos enumerados asociándolos a un valor binario.

El diseño del Package contiene los siguientes datos:

```
parameter data_width, addr_width
special_data t_mode (no_handshake = 0, read_handshake = 1,
                     write_handshake = 2, full_transaction = 3)
```

Código 1. Pseudocódigo del package del testbench de UVM.

5.2.2.3. Transaction

La Tabla 16 detalla la lista de señales que toman un papel clave en el test para verificar los resultados esperados del test procedure.

Señal	Canal	Test	Función
AWADDR	Write Address Channel	• Transferencia de Datos	Controla la dirección en la que se guarda WDATA. Generado aleatoriamente.
AWVALID	Write Address Channel	• Transacción de Escritura • Transferencia de Datos	Comienza el handshake de escritura del Write Address Channel. Dato estimulado.
AWREADY	Write Address Channel	• Transacción de Escritura • Transferencia de Datos	Resultado esperado del test de Transacción de Escritura.
WDATA	Write Data Channel	• Transferencia de Datos	Dato de escritura en bruto. Forma parte de la comparación final del test de Transferencia de Datos. Generado aleatoriamente.
WSTRB	Write Data Channe	• Transferencia de Datos	Decide que bytes deben ser escritos y cuáles no. Forma parte de la comparación final del test de Transferencia de Datos Generado aleatoriamente.
WVALID	Write Data Channe	• Transacción de Escritura • Transferencia de Datos	Comienza el handshake de escritura del Write Data Channel. Dato estimulado.
WREADY	Write Data Channe	• Transacción de Escritura • Transferencia de Datos	Resultado esperado del test de Transacción de Escritura.
BREADY	Write Response Channel	• Transacción de Escritura • Transferencia de Datos	Comienza el handshake de escritura del Write Response Channel. Dato estimulado.
BVALID	Write Response Channel	• Transacción de Escritura • Transferencia de Datos	Resultado esperado del test de Transacción de Escritura.
BRESP	Write Response Channel	• Transacción de Escritura • Transferencia de Datos	Resultado esperado del test de Transacción de Escritura.
ARADDR	Read Address Channel	• Transferencia de Datos	Controla la dirección en la que se lee. Dato copiado de AWADDR para leer donde se ha escrito.
ARVALID	Read Address Channel	• Transacción de lectura • Transferencia de Datos	Comienza el handshake de escritura del Read Address Channel. Dato estimulado.
ARREADY	Read Address Channel	• Transacción de lectura • Transferencia de	Resultado esperado del test de Transacción de Lectura.

		Datos	
RDATA	Read Data Channel	• Transferencia de Datos	Dato leído. Forma parte de la comparación final del test de Transferencia de Datos.
RREADY	Read Data Channel	• Transacción de lectura • Transferencia de Datos	Comienza el handshake de escritura del Read Data Channel. Dato estimulado
RVALID	Read Data Channel	• Transacción de lectura • Transferencia de Datos	Resultado esperado del test de Transacción de Lectura.
RRESP	Read Data Channel	• Transacción de lectura • Transferencia de Datos	Resultado esperado del test de Transacción de Lectura.
TEST_TYPE	Global	• Transacción de Escritura • Transacción de lectura • Transferencia de Datos	Indica a los componentes del testbench cuál de los tres tests está siendo comprobado.
WDONE	Global	• Transacción de Escritura • Transferencia de Datos	Se activa si todos los handshakes de escritura se han realizado correctamente. Utilizado para verificar los resultados esperados de la Transacción de Escritura.
RDONE	Global	• Transacción de lectura • Transferencia de Datos	Se activa si todos los handshakes de lectura se han realizado correctamente. Utilizado para verificar los resultados esperados de la Transacción de Lectura.

Tabla 16. Tabla con las señales cruciales para la realización del testbench en UVM.

En el Código 2 se observan 4 tipos de señales:

- Bit: Señales sin un valor por defecto. Todas estas señales son utilizadas para asertar los handshakes. Todas excepto "*s00_axi_rdata*", que es usada por el Monitor guarda el valor leído en el registro y juega un papel clave en la verificación final del Scoreboard.
- Random: Señales que son generadas con un valor aleatorio.
- Constrained random: Señal que es aleatorizada de manera restringida. En este caso el valor de "*s00_axi_awaddr*" nunca será mayor o igual a 8. Esto se utiliza para adaptar el testbench al diseño. Si se quisiera reutilizar el VIP para otro

diseño que tuviera 32 registros, debería cambiarse este valor para poder acceder a todos.

- T_mode: Tipo de datos enumerado explicado en el capítulo 5.2.2.2.

```
object transaction ()

bits s00_axi_aresetn, s00_axi_awprot, s00_axi_awvalid, s00_axi_awready,
     s00_axi_wvalid, s00_axi_wready, s00_axi_bresp, s00_axi_bvalid,
     s00_axi_bready, s00_axi_arprot, s00_axi_arvalid, s00_axi_arready,
     s00_axi_rresp, s00_axi_rdata, s00_axi_rvalid, s00_axi_rready, wdone,
     rdone
random s00_axi_wdata, s00_axi_wstrb, s00_axi_araddr
constrained random (s00_axi_awaddr < 8)
t_mode test_type

function generate(input = uvm_transaction)
    inherit(input)
```

Código 2. Pseudocódigo de la transaction del testbench de UVM.

Este pseudocódigo refleja muy bien las señales más importantes del testbench. Por otro lado, la Transaction solo dispone de una función, la función **generate**. Esta función la veremos en la gran mayoría de componentes de ahora en adelante.

La función **generate** representa una función llamada automáticamente por UVM (y pyUVM) en el momento que se detecta este objeto. Esta función crea el objeto y le aporta las propiedades de su análogo antecesor en UVM. En este caso, el objeto transaction **hereda** (de ahí el nombre de la función interna "*inherit*") todas las propiedades de "*input*", que se puede ver en el argumento de la función que es "*uvm_transaction*". Más adelante, veremos como cada componente de UVM y pyUVM necesita como argumento su antecesor para recibir las propiedades ocultas que las librerías aportan.

5.2.2.4. Secuencias

La Figura 10 describe los cambios que se producen en el camino que recorre la Transaction desde que es generada hasta que verifica los resultados obtenidos. Estos cambios varían según qué secuencia esté activa.

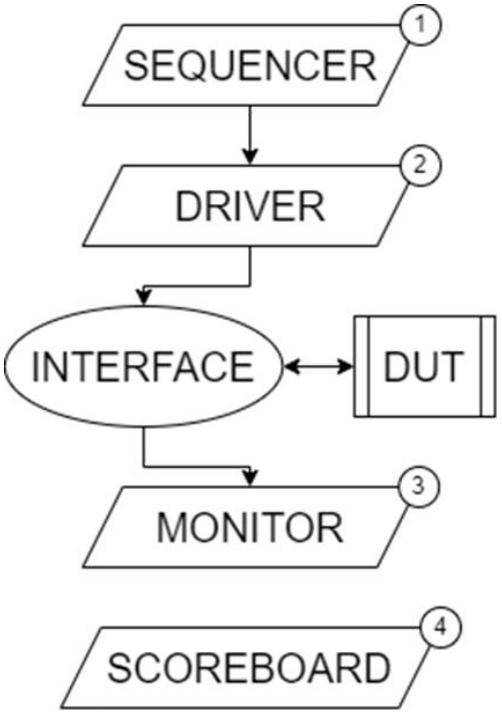

Figura 10. Workflow de la Transaction en el test de SystemVerilog.

Escritura Simple

```
object write_sequence(transaction)
transaction t

function generate(input = "uvm_sequence")
    inherit(input)

routine body()
    from 0 to 3 repeat
        send (transaction t)
        assert (transaction is randomized)
        test_type = write_handshake
        s00_axi_araddr = 0
        finish (transaction t)
```

Código 3. Pseudocódigo de la secuencia de la transacción de escritura del testbench de UVM.

- AWADDR, WDATA y WSTRB son generados.
- AWVALID, WVALID y BREADY son estimuladas en el Driver (capítulo 505.2.2.5).
- El Monitor (capítulo 5.2.2.8) activa la señal WDONE si el DUT ha devuelto las señales AWREADY, WREADY y BVALID a nivel alto.
- Si WDONE está activa, el Scoreboard (capítulo 5.2.2.9) da por superado el test.

Lectura Simple

```
object read_sequence (transaction)
transaction t
```

```
function generate(input = "uvm_sequence")
    inherit(input)

routine body()
    from 0 to 3 repeat
        send(transaction t)
        assert(transaction is randomized)
        test_type = read_handshake
        s00_axi_awaddr = 0
        s00_axi_wdata = 0
        s00_axi_wstrb = 0
        finish(transaction t)
```

Código 4. Pseudocódigo de la secuencia de la transacción de lectura del testbench de UVM.

- ARADDR es generada.
- ARVALID y RREADY son estimuladas en el Driver
- El Monitor activa la señal RDONE si el DUT ha devuelto las señales ARREADY y RVALID a nivel alto.
- Si RDONE está activa, el Scoreboard da por superado el test.

Escritura y Lectura

```
object full_sequence (transaction)
transaction t

function generate(input = "uvm_sequence")
    inherit(input)

routine body()
    from 0 to 3 repeat
        send(transaction t)
        assert(transaction is randomized)
        test_type = full_transaction
        s00_axi_araddr = s00_axi_awaddr
        finish(transaction t)
```

Código 5. Pseudocódigo de la secuencia de transferencia de datos del testbench de UVM.

- AWADDR, WDATA y WSTRB son generadas. ARADDR toma el valor de AWADDR.
- Al comienzo, el Driver realiza la estimulación de la escritura simple. Tras activarse las tres señales asociadas a la transacción de escritura, procede entonces a realizar la estimulación de la lectura simple (ARVALID y RREADY).
- El Monitor activa WDONE si han sido activadas las señales de la escritura simple. En ese momento, recoge los valores de WDATA y WSTRB. Posteriormente, activa RDONE y recoge el valor de RDATA si han sido activadas las señales de la lectura simple.

- Si algún bit de WSTRB está activo y los bits de WDATA relacionados son idénticos a los de RDATA el Scoreboard dará por superado el test si WDONE y RDONE están también activos.

5.2.2.5. Driver

```
class driver(transaction)
transaction t
axi4lite_interface vif

function generate(input = "driver")
    inherit(input)

function build(phase = uvm_phase)
    inherit (phase)
    if vif is not found: print("ERROR – Could not Access the Interface")

## Reset Logic
routine init()
    vif(s00_axi_aresetn) = 0
    vif(s00_axi_awvalid = 0
    vif(s00_axi_wdata) = 0
    vif(s00_axi_awprot) = 0
    vif(s00_axi_wstrb) = 0
    vif(s00_axi_wvalid) = 0
    vif(s00_axi_bready) = 0
    vif(s00_axi_araddr) = 0
    vif(s00_axi_arprot) = 0
    vif(s00_axi_arvalid) = 0
    vif(s00_axi_rready) = 0
    vif(s00_axi_awaddr) = 0

    wait_clk_cycle

    vif(s00_axi_aresetn) = 1
    print("DUT Initialization – Completed")

## Single Write
routine write()
    vif(s00_axi_wstrb) = t(s00_axi_wstrb)
    vif(s00_axi_awadd) = t(s00_axi_awaddr)
    vif(s00_axi_wdata) = t(s00_axi_wdata)

    wait_clk_cycle

    vif(s00_axi_awvalid) = 1
    vif(s00_axi_wvalid) = 1
    wait(vif(s00_axi_awready) and vif(s00_axi_wready)) = 1
    wait(vif(s00_axi_awready) and vif(s00_axi_wready)) = 0
    vif(s00_axi_awvalid) = 0
    vif(s00_axi_wvalid) = 0
    vif(s00_axi_bready) = 1
    wait(vif(s00_axi_bvalid)) = 1
```

```
    vif(s00_axi_bready) = 0

## Single Read
routine read()
    vif(s00_axi_araddr) = t(s00_axi_araddr)

    wait_clk_cycle

    vif(s00_axi_arvalid) = 1
    wait(vif(s00_axi_arready) and vif(s00_axi_rvalid)) = 1
    vif(s00_axi_rready) = 1
    wait(vif(s00_axi_arready) and vif(s00_axi_rvalid)) = 0
    vif(s00_axi_rready) = 0

## Main Driver Run
routine run(phase = uvm_phase)
    init()
    inifinite bucle
       get_next_item(t)
       vif(test_type) = t(test_type)

       wait_clk_cycle

       if t(test_type) = write_handshake: write()
       elsif t(test_type) = read_handshake: read()
       elsif t(test_type) = full_seq: write() and read()
           item_done()
```

Código 6. Pseudocódigo del Driver del testbench de UVM.

En el Código 6 se remarcan 4 partes importantes del testbench:

- Reset Logic: Función auxiliar que permite inicializar todas las señales del testbench.
- Single Write: Función auxiliar que envía los datos generados por la secuencia de transacción de escritura y posteriormente envía a la Interface las señales pertinentes para realizar las transacciones de escritura.
- Single Read: Función auxiliar que envía los datos generados por la secuencia de transacción de lectura y posteriormente envía a la Interface las señales pertinentes para realizar las transacciones de lectura.
- Main Driver Run: Función principal del Driver. Primero lanza la función "*init*" para reiniciar el diseño. Tras ello, pide de manera cíclica la Transaction generada, envía el "*test_type*" por la Interface (para que lo recoja el Monitor como se verá posteriormente en el capítulo 5.2.2.8) y en base a la señal "*test_type*" lanza la función "*write*", "*read*", o la combinación de ambas.

Antes de la primera marca en el pseudocódigo, aparecen otra función que, a pesar de pasar inadvertida, son esenciales para la construcción de un testbench en UVM. Esta es

la función "*build*". Esta función permite construir el componente e instanciar los subcomponentes que están encapsulados en él. Dado que es un concepto de la librería y no es el objetivo del estudio, no se profundizará más sobre esta función.

Con este pseudocódigo, se confirma que el componente Driver cumplirá los siguientes requisitos indispensables:

- El Driver deberá ser capaz de resetear el diseño al comienzo del testbench para confirmar que no ha sido manipulado previamente.
- El Driver deberá ser controlado por la señal **TEST_TYPE**, la cual, indicará al Driver si debe estimular una transacción de escritura, de lectura, o ambas de forma secuencial, para poder llevar a cabo los tres tests.
- El Driver deberá enviar los datos generados a la Interface.

5.2.2.6. Interface

El componente Interface es el conjunto de señales que tiene el testbench y el diseño, de manera que establezca un puente entre el Test y el DUT. Por ello, el componente cumple los siguientes requisitos para realizar el testbench:

- La interfaz debe habilitar al Test la estimulación del DUT para activar su funcionamiento (señales globales).
- La interfaz debe habilitar al Driver la estimulación del DUT para activar las funcionalidades que se desean verificar (transacciones y transferencia de datos).
- La interfaz debe conectar el Driver y el Monitor para poder enviar el **TEST_TYPE** utilizado tanto en la recogida de datos como en la verificación final.

A continuación, se listan en el Código 7 las señales necesarias para conectar el testbench con el componente.

```
class interface

bit s00_axi_aclk
bit s00_axi_aresetn
bit s00_axi_awaddr
bit s00_axi_awprot
bit s00_axi_awvalid
bit s00_axi_awready
bit s00_axi_wdata
```

```
bit s00_axi_wstrb
bit s00_axi_wvalid
bit s00_axi_wready
bit s00_axi_bresp
bit s00_axi_bvalid
bit s00_axi_bready
bit s00_axi_araddr
bit s00_axi_arprot
bit s00_axi_arvalid
bit s00_axi_arready
bit s00_axi_rdata
bit s00_axi_rresp
bit s00_axi_rvalid
bit s00_axi_rready
t_mode test_type
```

Código 7. Pseudocódigo de la Interface del testbench de UVM.

5.2.2.7. DUT

El DUT es el diseño del componente esclavo del protocolo AXI4-Lite, explicado en el capítulo 4. El bloque esclavo, como ya ha sido explicado anteriormente, produce las respuestas ante los estímulos generados por el maestro, que es el rol que toma el testbench. La Figura 11 y el Código 8 muestran la entidad del bloque a verificar.

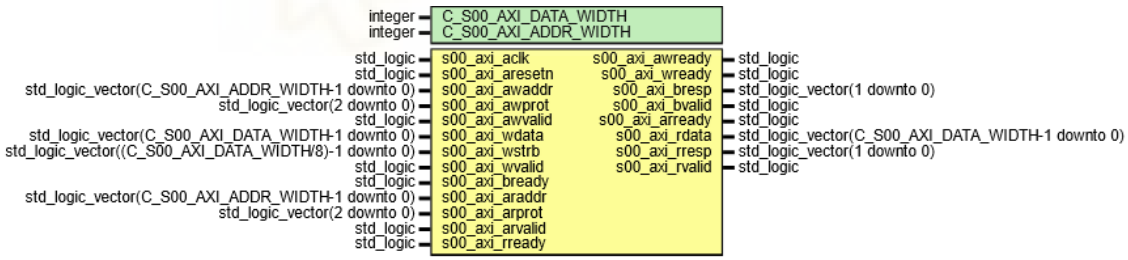

Figura 11. Entidad del componente a verificar en el testbench de UVM.

```
-------------------------------------------------------------------------------
--           Company: Intigia. S.L.
--     Author: Salido Vidal, Fco Javier
--     Comments: Salido Vidal, Fco Javier
--
--     Type:
--     Design
--
--     Description:
--     Instantiates an AXI4-Lite Slave Bus Interface in VHDL
--
--
--     Initial Comment:
--     All the signals declared and instantiated in this design are properly
```

```
--           commented in axi4lite_slave_v1_0_S00_AXI.vhd
--
-------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity axi4lite_slave_v1_0 is
      generic (
            ------------------------------------------------
            -- Generic Parameters

            C_S00_AXI_DATA_WIDTH      : integer    := 32;
            C_S00_AXI_ADDR_WIDTH      : integer    := 3
            ------------------------------------------------
      );
      port (
            ------------------------------------------------
            -- Ports of Axi Slave Bus Interface S00_AXI

            s00_axi_aclk        : in std_logic;
            s00_axi_aresetn           : in std_logic;
            s00_axi_awaddr            : in
std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
            s00_axi_awprot            : in std_logic_vector(2 downto 0);
            s00_axi_awvalid           : in std_logic;
            s00_axi_awready           : out std_logic;
            s00_axi_wdata       : in std_logic_vector(C_S00_AXI_DATA_WIDTH-1
downto 0);
            s00_axi_wstrb       : in
std_logic_vector((C_S00_AXI_DATA_WIDTH/8)-1 downto 0);
            s00_axi_wvalid            : in std_logic;
            s00_axi_wready            : out std_logic;
            s00_axi_bresp       : out std_logic_vector(1 downto 0);
            s00_axi_bvalid            : out std_logic;
            s00_axi_bready            : in std_logic;
            s00_axi_araddr            : in
std_logic_vector(C_S00_AXI_ADDR_WIDTH-1 downto 0);
            s00_axi_arprot            : in std_logic_vector(2 downto 0);
            s00_axi_arvalid           : in std_logic;
            s00_axi_arready           : out std_logic;
            s00_axi_rdata       : out std_logic_vector(C_S00_AXI_DATA_WIDTH-
1 downto 0);
            s00_axi_rresp       : out std_logic_vector(1 downto 0);
            s00_axi_rvalid            : out std_logic;
            s00_axi_rready            : in std_logic
            ------------------------------------------------
      );
end axi4lite_slave_v1_0;
```

Código 8. Entidad del componente a verificar en el testbench de UVM.

Como puede apreciarse, la entidad presenta todos los canales explicados en el capítulo 4. Los canales "*s00_axi_awprot*" y "*s00_axi_arprot*" no son utilizados en esta verificación, ya en la posterior arquitectura están desconectados, por lo que no están implementados.

5.2.2.8. Monitor

```
class monitor(transaction)
transaction t
axi4lite_interface vif
C_TIMEOUT_TIME = 10 clk cycles

function generate(input = "uvm_monitor")
    inherit(inst)

function build(phase = uvm_phase)
    inherit(phase)
    create transaction t
    if vif is not found: print("ERROR – Could not Access the Interface")

## Single Write
routine write()
    bit timeout = 1
    wait(vif(s00_axi_awready) and vif(s00_axi_wready) and
vif(s00_axi_bvalid) = 0) or wait(C_TIMEOUT_TIME)
    t(s00_axi_awaddr) = vif(s00_axi_awaddr)
    t(s00_axi_wdata) = vif(s00_axi_wdata)
    t(s00_axi_wstrb) = vif(s00_axi_wstrb)
    if C_TIMEOUT_TIME has not happened: t(wdone) = 1

## Single Read
routine read()
    bit timeout = 1
    wait(vif(s00_axi_arready) and vif(s00_axi_rvalid)) or
wait(C_TIMEOUT_TIME)
    t(s00_axi_araddr) = vif(s00_axi_arraddr)
    t(s00_axi_rdata) = vif(s00_axi_rdata)
    if C_TIMEOUT_TIME has not happened: t(rdone) = 1

## Main Monitor Run
task run(phase = uvm_phase);
    create transaction t

    wait_clk_cycle

    infinite bucle

         wait_clk_cycle

         t(wdone) = 0 and t(rdone) = 0
         t(test_type) = vif(test_type)
         if t(test_type) = write_handshake: write()
        elif t(test_type) = read_handshake: read()
        elif t(test_type) = full_transaction: write() and read()
       send_transaction_to_scoreboard(t)
```

Código 9. Pseudocódigo del Monitor del testbench de UVM.

En el Código 9 se exponen las distintas funciones que dispone el Monitor para recolectar los datos del DUT:

- Single Write: Función auxiliar espera la realización de los handshakes de escritura o un tiempo estándar definido (C_TIMEOUT_TIME). Cuando una de las dos condiciones se cumple, recoge los datos asociados con la transacción de escritura. Si no ha pasado el tiempo definido en C_TIMEOUT_TIME, activa la señal wdone para confirmar que se ha realizado correctamente la transacción.
- Single Read: Función auxiliar espera la realización de los handshakes de lectura o un tiempo estándar definido (C_TIMEOUT_TIME). Cuando una de las dos condiciones se cumple, recoge los datos asociados con la transacción de lectura. Si no ha pasado el tiempo definido en C_TIMEOUT_TIME, activa la señal rdone para confirmar que se ha realizado correctamente la transacción.
- Main Monitor Run: Función principal del Monitor. Tiene una función cíclica en la cual primero inicializa las señales "*wdone*" y "*rdone*" a 0 para evitar errores en caso de que estuvieran activas de una anterior iteración. Luego, recoge el valor del "*test_type*" enviado desde el Driver (véase Código 6) y, en función de su valor, realiza la función "*write*", "*read*", o la combinación de ambas. Una vez recogidos los datos, acaba su función enviándolos al Scoreboard.

De esta forma, el Monitor cumple la función descrita en la Tabla 10, además de algún requisito extra requerido para este test:

- El Monitor debe ser capaz de recoger datos de la Interface en sincronía con las estimulaciones generadas por el Driver.
- El Monitor será controlado por la señal **TEST_TYPE**, la cual, indicará al Monitor si debe capturar los datos una transacción de escritura, de lectura, o ambas de forma secuencial, para poder llevar a cabo los tres tests.

5.2.2.9. Scoreboard

```
class scoreboard(transaction)

transaction t
integer i
integer count

function generate(input = "uvm_scoreboard")
    inherit(input)

function build(phase = uvm_phase)
    create transaction t
```

```
## Final Comparation
function compare(transaction)
         receive_transaction_from_monitor(t)
         if t(test_type) = write_handshake:
             if wdone = 1: print("TEST PASSED")
             else: print("TEST FAILED")
        elif t(test_type) = read_handshake:
             if rdone = 1: print("TEST PASSED")
             else: print("TEST FAILED")
        elif t(test_type) = full_transaction:
             if wdone and rdone and (rdata = wdata when wstrb_bit = 1):
                print(" TEST PASSED")
            else: print("TEST FAILED")
```

Código 10. Pseudocódigo del Scoreboard del testbench de UVM.

El Scoreboard del testbench de UVM tiene solamente una función (remarcada en el Código 10, la función "*compare*") y es la de verificar que los valores recogidos por el Monitor son los esperados. Esto se realiza mediante la comparación de las señales "*wdone*", "*rdone*" y los valores almacenados en "*wdata*", "*wstrb*" y "*rdata"*, con los valores que esperados según el dato "*test_type*". Al terminar la comparación, escribe en el registro si el test se ha realizado satisfactoriamente o ha sucedido algún error.

De esta forma, el Scoreboard verifica completamente el diseño, cumpliendo uno a uno los requisitos siguientes:

- El Scoreboard será controlado por la señal **TEST_TYPE**, la cual, indicará al Scoreboard cómo debe verificar los datos recogidos por el Monitor.
- El Scoreboard deberá verificar que la transacción de escritura se ha realizado correctamente.
- El Scoreboard deberá verificar que la transacción de lectura se ha realizado correctamente.
- El Scoreboard deberá verificar que la transferencia de datos se ha realizado correctamente.

5.2.2.10. Agent

El componente Agent encapsula al Sequencer, Driver y Monitor dentro de un mismo componente para facilitar el diseño del testbench aguas arriba. Además, conecta el Sequencer con el Driver para que puedan enviarse los datos generados.

```
class agent

uvm_sequencer seqr(transaction)
driver drv(transaction)
monitor mon(transaction)

function generate(input = "uvm_agent”)
    inherit(input)

function build (phase = uvm_phase)
    inherit(phase)
    create_monitor(mon)
    create_sequencer(seqr)
    create_driver(drv)

function connect (phase = uvm_phase)
    inherit(phase)
    connect(drv;seqr)
```

Código 11. Pseudocódigo del Agent del testbench de UVM.

5.2.2.11. Environment

El componente Environment tiene la misma funcionalidad que el Agent. Encapsula el Agent y el Scoreboard en un mismo componente para facilitar el testbench. Esta encapsulación se realiza así para poder cambiar tanto la conexión con el componente a verificar como cambiar la comparación que se realiza. En caso de querer sustituir cualquiera de los dos componentes, solo hay que instanciarlo en el código del environment. De igual manera, se puede cambiar el environment entero como veremos en el capítulo 0.

```
class environment

agent a
scoreboard sco

function generate(input = "uvm_env")
    inherit(input)

function build(phase = uvm_phase);
    create_scoreboard(sco)
    create_agent(a)

function connect(phase = uvm_phase)
    connect(mon;sco)
```

Código 12. Pseudocódigo del Scoreboard del testbench de UVM.

5.2.2.12. Test

Al igual que el Enviroment hace con el Agent y el Scoreboard, el Test encapsula el Environment, y añade las secuencias requeridas para este test. Además, el Test tiene una función más importante, gestionar el procedimiento del test. Esto se ve reflejado en el Código 13, donde está remarcada la rutina "*procedure*", en la lanza los tests en el orden programado además de imprimir por el registro cuando comienza cada uno.

```
class test

environment env
write_sequence wr_seq
read_sequence rd_seq
full_sequence full_seq

function generate(input = "uvm_test”)
    inherit input)

function build(phase = uvm_phase)
    inherit(phase)
    create_environment(env)
    create_write_sequence(wr_seq)
    create_read_sequence(read_seq)
    create_full_sequence(full_seq)

## Procedure
routine procedure()
    print("===================================")
    print("Starting Write Transaction Test")
    print("===================================")
        start(wr_seq)
    print("===================================")
    print("Starting Read Transaction Test")
    print("===================================")
        start(rd_seq)
    print("===================================")
    print("Starting Data Transference Test")
    print("===================================")
        start(full_seq)
```

Código 13. Pseudocódigo del Test del testbench de UVM.

5.2.2.13. UVM Top Module

Por último, el Top Module es el módulo encargado de instanciar el Test de UVM y el diseño a verificar. Este módulo no forma parte de la librería UVM, sino que está programado en Verilog. Cabe destacar la manera tan fluida en la que Verilog incluye diseños VHDL. A esto se le conoce como diseños Mixed-Language.

```
module axi4lite_top_module()

axi4lite_interface vif

axi4lite_slave_v1_0
    #(
        .C_S00_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S00_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    )
    dut (
        .s00_axi_aclk(vif(s00_axi_aclk)),
        .s00_axi_aresetn(vif(s00_axi_aresetn)),
        .s00_axi_awaddr(vif(s00_axi_awaddr)),
        .s00_axi_awprot(vif(s00_axi_awprot)),
        .s00_axi_awvalid(vif(s00_axi_awvalid)),
        .s00_axi_awready(vif(s00_axi_awready)),
        .s00_axi_wdata(vif(s00_axi_wdata)),
        .s00_axi_wstrb(vif(s00_axi_wstrb)),
        .s00_axi_wvalid(vif(s00_axi_wvalid)),
        .s00_axi_wready(vif(s00_axi_wready)),
        .s00_axi_bresp(vif(s00_axi_bresp)),
        .s00_axi_bvalid(vif(s00_axi_bvalid)),
        .s00_axi_bready(vif(s00_axi_bready)),
        .s00_axi_araddr(vif(s00_axi_araddr)),
        .s00_axi_arprot(vif(s00_axi_arprot)),
        .s00_axi_arvalid(vif(s00_axi_arvalid)),
        .s00_axi_arready(vif(s00_axi_arready)),
        .s00_axi_rdata(vif(s00_axi_rdata)),
        .s00_axi_rresp(vif(s00_axi_rresp)),
        .s00_axi_rvalid(vif(s00_axi_rvalid)),
        .s00_axi_rready(vif(s00_axi_rready))
    )

at_start()
    vif(s00_axi_aclk) = 0

do_always()
    wait(5 ns)
    vif(s00_axi_aclk) = not vif(s00_axi_aclk)

at_start()
create_axi4lite_interface(vif)
run_test("test")
```

Código 14. Código del Top Module del testbench de UVM.

5.2.3. *VIP en pyUVM*

Para llevar a cabo el procedimiento especificado en el test procedure, la estructura general del testbench deberá seguir lo esquematizado en la Figura 12.

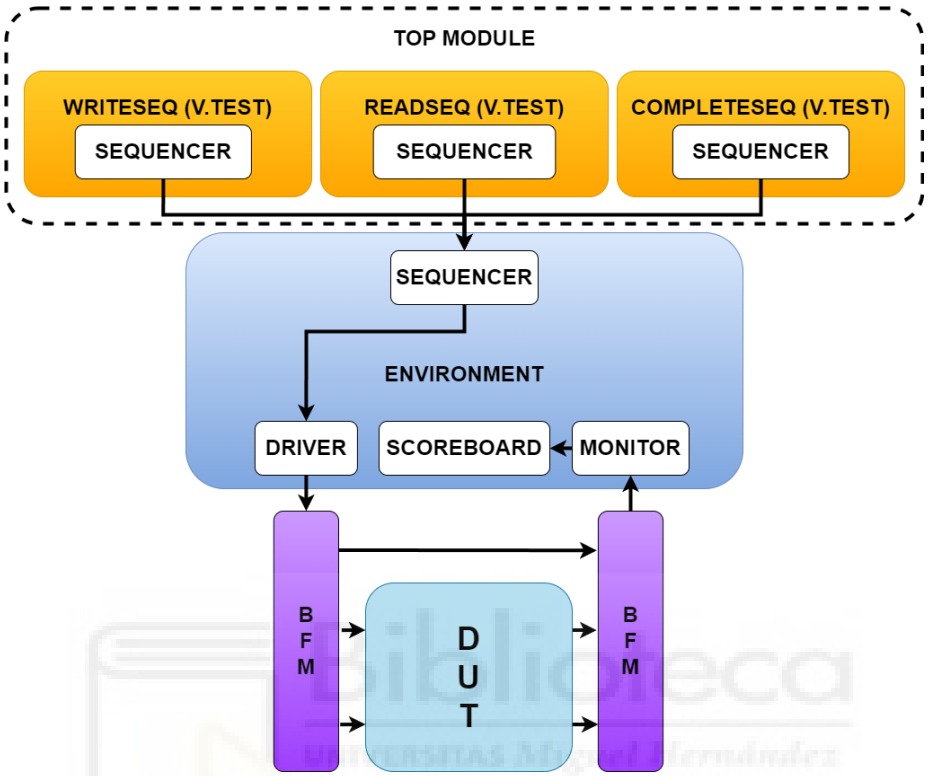

Figura 12. Diagrama con el funcionamiento del testbench diseñado en Python mediante la librería pyUVM.

Se pueden vislumbrar tres grandes cambios de Figura 12 respecto a la Figura 9. Primero, este testbench no tiene un Agent. Este hecho no se debe a que no se pueda programar, sino que las conexiones en pyUVM se definen de una forma más sencilla, haciendo más sencillo encapsular todos los módulos directamente en el Environment. Por otro lado, vemos que hay dos nuevos bloques que parecen suplantar el rol de la Interface, el BFM (Bus Functional Model). Aunque sobre el papel los bloques que forman parte del testbench programados en Python simulan ser objetos o componentes, son esencialmente funciones, y este es un punto fuerte que Python puede aprovechar para desmenuzar aún más las tareas del testbench, punto que se expondrá con más detalle al explicar las distintas funciones y componentes. El tercer y último cambio, que es el más perceptible, es que no está la figura del test, sino que en su lugar está el módulo top. Como se verá con más detalle en el capítulo 5.2.3.11, se aprovecha de nuevo la flexibilidad de pyUVM implementando las 3 secuencias como "tests virtuales". Esto simplifica el código y lo hace más atractivo para el programador.

5.2.3.1. Entorno

Al contrario que en el testbench de UVM, el entorno de pyUVM tiene una mayor complejidad. Esto es debido a que, al ser open-source, se deben conectar una amalgama de herramientas que permitan la simulación del testbench.

Para la simulación del testbench en pyUVM hemos generado un entorno virtual con el IDE Anaconda Navigator 2.4.2., que permite mediante el gestor de entornos conda (para la versión de Anaconda usada se utiliza conda 22.9.0) generar, valga la redundancia, entornos principalmente enfocados principalmente en Python. Se adjunta en la Figura 13 una tabla con la lista de los paquetes instalados en el entorno virtual.

```
(pyUVM) C:\Users\JaviSV>conda list
# packages in environment at C:\Users\JaviSV\.conda\envs\pyUVM:
#
# Name                    Version                   Build  Channel
alabaster                 0.7.12                    pypi_0    pypi
astroid                   2.12.13                   pypi_0    pypi
attrs                     22.1.0                    pypi_0    pypi
babel                     2.11.0                    pypi_0    pypi
bzip2                     1.0.8                he774522_0
ca-certificates           2023.05.30           haa95532_0
cachetools                5.2.0                     pypi_0    pypi
certifi                   2023.5.7                  pypi_0    pypi
chardet                   5.1.0                     pypi_0    pypi
charset-normalizer        2.1.1                     pypi_0    pypi
cocotb                    1.7.2                     pypi_0    pypi
colorama                  0.4.6                     pypi_0    pypi
dill                      0.3.6                     pypi_0    pypi
distlib                   0.3.6                     pypi_0    pypi
docutils                  0.19                      pypi_0    pypi
exceptiongroup            1.0.4                     pypi_0    pypi
filelock                  3.8.2                     pypi_0    pypi
find-libpython            0.3.0                     pypi_0    pypi
idna                      3.4                       pypi_0    pypi
imagesize                 1.4.1                     pypi_0    pypi
importlib-metadata        5.1.0                     pypi_0    pypi
iniconfig                 1.1.1                     pypi_0    pypi
isort                     5.10.1                    pypi_0    pypi
jinja2                    3.1.2                     pypi_0    pypi
lazy-object-proxy         1.8.0                     pypi_0    pypi
libffi                    3.4.4                hd77b12b_0
m2-base                   1.0.0                        3    msys2
m2-bash                   4.3.042                      5    msys2
m2-bash-completion        2.3                          2    msys2
m2-catgets                1.1                          3    msys2
m2-coreutils              8.25                       102    msys2
m2-dash                   0.5.8                        2    msys2
m2-file                   5.25                         2    msys2
m2-filesystem             2016.04                      4    msys2
m2-findutils              4.6.0                        2    msys2
m2-gawk                   4.1.3                        2    msys2
m2-gcc-libs               5.3.0                        4    msys2
m2-gettext                0.19.7                       4    msys2
m2-gmp                    6.1.0                        3    msys2
m2-grep                   2.22                         4    msys2
m2-gzip                   1.7                          2    msys2
m2-inetutils              1.9.2                        2    msys2
m2-info                   6.0                          2    msys2
m2-less                   481                          2    msys2
m2-libasprintf            0.19.7                       4    msys2
m2-libcatgets             1.1                          3    msys2
m2-libcrypt               1.3                          2    msys2
m2-libgettextpo           0.19.7                       4    msys2
m2-libiconv               1.14                         3    msys2
m2-libintl                0.19.7                       4    msys2
m2-liblzma                5.2.2                        2    msys2
m2-libpcre                8.38                         2    msys2
m2-libreadline            6.3.008                      8    msys2
m2-libutil-linux          2.26.2                       2    msys2
m2-libxml2                2.9.2                        3    msys2
m2-make                   4.1                          5    msys2
m2-mintty                 1!2.2.3                      2    msys2
m2-mpfr                   3.1.4                        2    msys2
m2-msys2-launcher-git     0.3.28.860c495               2    msys2
m2-msys2-runtime          2.5.0.17080.65c939c                3    msys2
m2-ncurses                6.0.20160220                 2    msys2
m2-sed                    4.2.2                        3    msys2
m2-tftp-hpa               5.2                          2    msys2
m2-time                   1.7                          2    msys2
m2-ttyrec                 1.0.8                        2    msys2
m2-tzcode                 2015.e                       2    msys2
m2-util-linux             2.26.2                       2    msys2
m2-which                  2.21                         3    msys2
m2-zlib                   1.2.8                        4    msys2
markupsafe                2.1.1                pypi_0    pypi
mccabe                    0.7.0                pypi_0    pypi
msys2-conda-epoch         20160418                     1    msys2
openssl                   3.0.9             h2bbff1b_0
packaging                 22.0                 pypi_0    pypi
pip                       23.1.2           py311haa95532_0
platformdirs              2.6.0                pypi_0    pypi
pluggy                    1.0.0                pypi_0    pypi
pygments                  2.13.0               pypi_0    pypi
pylint                    2.15.8               pypi_0    pypi
pyproject-api             1.2.1                pypi_0    pypi
pytest                    7.2.0                pypi_0    pypi
python                    3.11.3            he1021f5_1
pytz                      2022.6               pypi_0    pypi
pyuvm                     2.9.0                pypi_0    pypi
requests                  2.28.1               pypi_0    pypi
setuptools                67.8.0           py311haa95532_0
snowballstemmer           2.2.0                pypi_0    pypi
sphinx                    5.3.0                pypi_0    pypi
sphinxcontrib-applehelp   1.0.2                pypi_0    pypi
sphinxcontrib-devhelp     1.0.2                pypi_0    pypi
sphinxcontrib-htmlhelp    2.0.0                pypi_0    pypi
sphinxcontrib-jsmath      1.0.1                pypi_0    pypi
sphinxcontrib-qthelp      1.0.3                pypi_0    pypi
sphinxcontrib-serializinghtml 1.1.5            pypi_0    pypi
sqlite                    3.41.2            h2bbff1b_0
tk                        8.6.12            h2bbff1b_0
tomli                     2.0.1                pypi_0    pypi
tomlkit                   0.11.6               pypi_0    pypi
tox                       4.0.8                pypi_0    pypi
typing-extensions         4.4.0                pypi_0    pypi
tzdata                    2023c             h04d1e81_0
urllib3                   1.26.13              pypi_0    pypi
vc                        14.2              h21ff451_1
virtualenv                20.17.1              pypi_0    pypi
vs2015_runtime            14.27.29016       h5e58377_2
wheel                     0.38.4           py311haa95532_0
wincertstore              0.2                  pypi_0    pypi
wrapt                     1.14.1               pypi_0    pypi
xz                        5.4.2             h8cc25b3_0
zipp                      3.11.0               pypi_0    pypi
zlib                      1.2.13            h8cc25b3_0
```

Figura 13. Lista de paquetes instalados en el entorno virtual para simular el testbench pyUVM.

Los paquetes más destacables serían **cocotb** (soporta la librería Python y permite simular diseños descritos en VHDL y Verilog con Python), el paquete **m2** (permite el

uso de consola Linux), especialmente **m2-make** (agrega el uso de Makefiles para programar scripts en Linux, el cual se hará énfasis más adelante), **python** y **pyuvm** por razones obvias.

Al tratarse de una librería open-source, pyUVM no dispone, al menos por ahora, de una interfaz que nos facilite la generación del proyecto. La solución que ha encontrado cocotb es el uso de la herramienta Make implementada en GNU.

Gracias a los scripts con Make, se puede sincronizar la compilación de diseños y lanzar la simulación del testbench, el Código 15 muestra el script de Make:

```
CWD=$(shell pwd)
export PYTHONPATH :=
$(CWD)/../03_testbench/02_tests:$(CWD)/../03_testbench/02_tests/sequences:$(P
YTHONPATH)
COCOTB_REDUCED_LOG_FMT = False
SIM ?= ghdl
TOPLEVEL_LANG ?= vhdl
ifeq ($(shell echo $(TOPLEVEL_LANG) | tr '[:upper:]' '[:lower:]'),verilog)
      $(error "No verilog files found in this project")
else ifeq ($(shell echo $(TOPLEVEL_LANG) | tr '[:upper:]' '[:lower:]'),vhdl)
      VHDL_SOURCES =  $(CWD)/../02_design/axi4lite_slave_v1_0.vhd \

      $(CWD)/../02_design/axi4lite_slave_v1_0_S00_AXI.vhd
else
      $(error "A valid value (verilog or vhdl) was not provided for
TOPLEVEL_LANG=$(TOPLEVEL_LANG)")
endif
MODULE := pyuvm_top
TOPLEVEL := axi4lite_slave_v1_0
GENERICS+="C_S00_AXI_DATA_WIDTH=32"\
          "C_S00_AXI_ADDR_WIDTH=3"
SIM_ARGS = --vcd=waveform.vcd --wave=waveform.ghw
COCOTB_HDL_TIMEUNIT = 1ps
COCOTB_HDL_TIMEPRECISION = 1ps
GHDL_ARGS := -fsynopsys --std=08
include $(shell cocotb-config --makefiles)/Makefile.sim
include $(CWD)/cleanall.mk
```

Código 15. Script con los parámetros de simulación del testbench de pyUVM.

El Código 15 muestra principalmente de asignaciones a varios parámetros que cocotb necesita para realizar la simulación.

Los parámetros más destacables son ***SIM*** (para indicar el simulador que vamos a utilizar), ***TOPLEVEL_LANG*** (para indicar el lenguaje HDL en el que se va a simular), ***MODULE*** (para indicar el nombre del fichero top del proyecto de simulación),

TOPLEVEL (para indicar el fichero top del diseño), ***GENERICS*** (para pasarse los parámetros genéricos que requiere el diseño), ***SIM_ARGS*** (argumentos que se le pasan a la simulación, en este caso para crear las formas de onda), ***COCOTB_HDL_TIMEUNIT*** (unidad de tiempo de simulación) y ***COCOTB_HDL_TIMEPRECISION*** (unidad de precisión de simulación).

En este estudio no se va a profundizar en los parámetros que cocotb ofrece para configurar una simulación, para más información se puede acceder a la documentación de cocotb [17] en la que explica con más detalle cada una de las opciones del entorno.

Con este script Make llama a un Makefile el cual dispone la librería cocotb y lanza la simulación en Python. Faltaría solamente un editor de código para programar el testbench, para este estudio se ha utilizado VS Code v1.81.1, que además proporciona una terminal de comandos.

La secuencia de comandos para lanzar la simulación comienza activando el entorno virtual de pyUVM de conda con "*conda activate pyUVM*", posteriormente se debe situar en el directorio donde está el Makefile que se ha programado con "*cd <directorio>*" y se finalmente se debe escribir "*make*" para que se inicie el Makefile y, por ende, la simulación.

5.2.3.2. Package

La flexibilidad de Python es un punto del que ya se ha hablado previamente, y desde el comienz podemos ver reflejada esta característica. Además de los parámetros definidos en el package como en el package de UVM, se han definido dos funciones que ayudan a simplificar el código.

```
special_data test_type (no_handshake = 0, read_handshake = 1,
                        write_handshake = 2, full_transaction = 3)

data_width = 32
addr_width = 3

function axi4lite_predictor (test_type, wdata, wstrb, rdata):
   if test_type is not class(test_type): ERROR
   else:
```

```
      success = 0
      if test_type = no_handshake then (wdone = 0, rdone = 0, success = 1)
      if test_type = read_handshake then (wdone = 0, rdone = 1, success = 1)
      if test_type = write_handshake then (wdone = 1, rdone = 0, success =
         1)
      if test_type = full_transaction and rdata = wdata when wstrb = 1:
         (wdone = 1, rdone = 1, success = 1)
predicted_t = (wdone, rdone, success)
return predicted_t

function collect_binary_value:
   try_to_get_value = binary(value)
   if not: value = binary(0)
return value
```

Código 16. Pseudocódigo del package del testbench de pyUVM.

La función “predictor” devuelve los valores esperados para cada test_type, y la función “collect_bynary_value” intenta recolectar el valor introducido en la función, en caso de no poder hacerlo devuelve 0.

5.2.3.3. Sequence Item

La librería pyUVM prefiere definir el conjunto de señales que son transmitidas de bloque en bloque como un objeto que ha generado la secuencia. Tiene la misma funcionalidad que la Transaction en la librería UVM. Por ende, el pseudocódigo sigue la misma dinámica. Se puede consultar el Código 2 para tener un punto de vista más gráfico.

5.2.3.4. Secuencias

La Figura 14 describe los cambios que se producen en el camino que recorre la Transaction desde que es generada hasta que verifica los resultados obtenidos. Estos cambios varían según qué secuencia esté activa.

Los estímulos generados, los valores recogidos y la fórmula de comparación variará según el test virtual que esté activo (es decir, la secuencia). Para la generación de las 3 secuencias virtuales necesarias para este test, se ha establecido una jerarquía de secuencias.

```
class BaseSeq(uvm_sequence)
    "Base sequence for AXI 4 Lite
       SeqItem Inputs (5):
        - s00_axi_awaddr
        - s00_axi_wdata
        - s00_axi_wstrb
        - s00_axi_araddr
        - test_type"

    function main()
        repeat for 10 times:
            t = SeqItem("t", 0, 0, 0, 0, 0)
            start(t)
            get_value(t)
            finish(t)

    function get_values()
        pass
```

Código 17. Pseudocódigo de la secuencia "BaseSeq" del testbench de pyUVM.

Un aspecto que puede apreciarse en el pseudocódigo explicado es que Python permite poner descripciones a todas las funciones y clases, de forma que colocando el puntero encima del nombre te da la descripción escrita. Esto aumenta las líneas de código, pero facilita mucho la lectura de funciones, dejando un código claro y limpio.

La secuencia base desde la que se parte solo tiene la función "main", que define y lanza la transacción. Al contrario que los HDL, en Python no hay señales que deben definidas, sino objetos. Esto permite que el Sequence Item sea una tupla de 5 valores. Entre las funciones "*start*" y "*finish*" se llama a la función "*get_values*". Esta función está vacía en el Código 17, pero será reescrita en las secuencias posteriores.

```
class Single_WR(BaseSeq)
    """Single Write Transaction Sequence.
       Overrides BaseSeq"""

    function get_values()
        random t(s00_axi_awaddr), t(s00_axi_wdata), t(s00_axi_wstrb)
        t(s00_axi_araddr) = 0
        t(test_type) = 2
```

Código 18. Pseudocódigo de la secuencia de transacción de escritura del testbench de pyUVM.

```
class Single_RD(BaseSeq)
    """Single Read Transaction Sequence.
       Overrides BaseSeq"""

    function get_values()
        random t(s00_axi_araddr)
        t(s00_axi_awaddr) and t(s00_axi_wdata) and t(s00_axi_wstrb) = 0
        t(test_type) = 1
```

Código 19. Pseudocódigo de la secuencia de transacción de lectura del testbench de pyUVM.

```
class Complete_TR(BaseSeq)

    function get_values()
        random t(s00_axi_awaddr) and t(s00_axi_wdata) and t(s00_axi_wstrb)
        t(s00_axi_araddr) = t(s00_axi_awaddr)
        t(test_type) = 3
```

Código 20. Pseudocódigo de la secuencia de transferencia de datos del testbench de pyUVM.

Las secuencias **Single_WR** (Código 18), **Single_RD** (Código 19) y **Complete_**TR (Código 20) heredan las propiedades y funciones de **BaseSeq** (Código 17) y reescriben la función "*get_values*" acorde con el tipo de test virtual al que pertenece la secuencia.

```
class WriteSeq(uvm_sequence)
    """A virtual sequence that starts only a write transaction"""

    function main()
        create_sequencer(seqr)
        create_Single_WR_sequence(single_wr)
        seqr.start(single_wr)
```

Código 21. Pseudocódigo de la secuencia virtual de transacción de escritura del testbench de pyUVM.

```
class ReadSeq(uvm_sequence)
    """A virtual sequence that starts only a read transaction"""

    function main()
        create_sequencer(seqr)
        create_Single_RD_sequence(single_rd)
        seqr.start(single_rd)
```

Código 22. Pseudocódigo de la secuencia virtual de transacción de lectura del testbench de pyUVM.

```
class CompleteSeq(uvm_sequence)
    """A virtual sequence that starts a full transaction"""

    function main()
        create_sequencer(seqr)
        create_Complete_TR_sequence(complete_tr)
        seqr.start(complete_tr)
```

Código 23. Pseudocódigo de la secuencia virtual de transferencia de datos del testbench de pyUVM.

Finalmente, las secuencias virtuales **WriteSeq**, **ReadSeq** y **CompleteSeq** instancian el Sequencer y la secuencia correspondiente. Por último, lanzan el Sequence Item por el Sequencer para que el Driver la recoja.

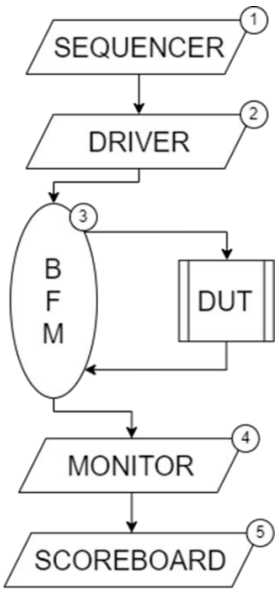

Figura 14. Workflow de la Transaction en el test de Python.

Los recorridos son los siguientes

Escritura Simple:

- AWADDR, WDATA y WSTRB son generados en el Sequencer.
- Se envían los valores generados y el TEST_TYPE (transacción de escritura) al BFM (capítulo 5.2.3.5).
- AWVALID, WVALID y BREADY son estimuladas. WDONE se activa si AWREADY, WREADY y BVALID han sido activadas.
- Se recoge el valor de WDONE del BFM.
- Si WDONE está activa, el Scoreboard (Código 27) da por superado el test.

Lectura Simple:

- ARADDR es generada.
- Se envían los valores generados y el TEST_TYPE (transacción de lectura) al BFM.
- ARVALID y RREADY son estimuladas. RDONE se activa si ARREADY y RVALID han sido activadas.
- Se recoge el valor de RDONE del BFM.
- Si RDONE está activa, se da por superado el test.

Escritura y Lectura:

- AWADDR, WDATA y WSTRB son generadas. ARADDR toma el valor de AWADDR.
- Se envían los valores generados y el TEST_TYPE (transferencia de datos) al BFM.
- Al comienzo, realiza la estimulación de la escritura simple. Tras activarse AWREADY, WREADY y BVALID entonces realiza la estimulación de la lectura simple.

  Activa WDONE si han sido activadas las señales de la escritura simple. En ese momento, recoge los valores de WDATA y WSTRB. Posteriormente activa RDONE y recoge el valor de RDATA si han sido activadas las señales de la lectura simple.
- Se recogen los valores de WDONE, RDONE, WDATA, WSTRB y RDATA del BFM.
- Si algún bit de WSTRB está activo y los bits de WDATA relacionados son idénticos a los de RDATA, se dará por superado el test si WDONE y RDONE también están activos.

5.2.3.5. Driver

Una pequeña diferencia perceptible en los componentes de pyUVM respecto de UVM es el argumento que se le pasa al componente. En UVM se pasa la Transaction como argumento al componente, mientras que en pyUVM se pasa como argumento el componente uvm_driver. Esta casi imperceptible diferencia esconde detrás una diferencia de instanciación mucho mayor.

Los componentes en UVM tienen la transaction como argumento para que el componente tenga los mismos inputs y outputs que la Transaction (recordemos que son el punto de vista de UVM son componentes). Para luego obtener las mismas características que el componente Driver de UVM, se define la función "*generate*" que le permite heredar las propiedades de su antecesor.

Por otro lado, tenemos a pyUVM. Como ya se ha nombrado en varias ocasiones, Python permite una flexibilidad excepcional con los datos. Es por ello por lo que no es necesario que el componente sea "formateado" con los inputs y los outputs del Sequence Item (capítulo 5.2.3.3). No obstante, sí que es necesario que herede las

propiedades del componente UVM, de ahí que el argumento que reciba sea su antecesor y, como vemos en el Código 24, la Transaction (que Python trata como un objeto más) se recibe a través de la conexión con el Sequencer (véase capítulo 5.2.2.11).

Las funciones visibles en el Código 24 permiten iniciar tres funciones en el BFM (Código 25) y enviar valores al BFM de manera cíclica.

```
class Driver(uvm_driver)
    """Driver for AXI 4 Lite Protocol"""

axi4lite_bfm bfm

    def bfm_start()
        bfm.init()
        bfm.init_clock()
        bfm.reset_dut()
        bfm.start_bfm()

    def run()
        bfm_start()
        infinite bucle
            t = get_next_item(t)
            driver_queue.send(t)
            item_finish(t)
```

Código 24. Pseudocódigo del Driver del testbench de pyUVM.

Tal y como se ha descrito en el apartado 5.2.3, pyUVM permite desmenuzar más las tareas. En el pseudocódigo se observa que el Driver se encarga de la función inicial de iniciar el BFM y el DUT. A partir de ese momento recibe constantemente una transacción y la envía al BFM. Como se ve, los requisitos de este Driver son más sencillos de cumplir:

- El Driver deberá ser capaz de iniciar el BFM y ordenarle resetear el diseño al comienzo del testbench para confirmar que no ha sido manipulado previamente.
- El Driver deberá enviar los datos generados al BFM.

5.2.3.6. Bus Functional Model

El BFM toma la responsabilidad de la interacción absoluta con el diseño, por ello su principal y único requisito es:

- El BFM será controlado por la señal **TEST_TYPE**, la cual, indicará al BFM si debe estimular y capturar una transacción de escritura, de lectura, o ambas de forma secuencial, para poder llevar a cabo los tres tests.

Puede parecer que el BFM tiene un papel sencillo, pero es en realidad el bloque más complejo del testbench. El código consta de tres grandes partes:

- Main Functions: En esta parte se realizan las funciones principales del BFM. Se hallan las tres funciones que son llamadas desde el Driver (Código 24): "*init_clock*", "*reset_dut*" y "*start_bfm*". Estas tres activan distintas funcionalidades del diseño: inician el reloj, inicializan las variables y activan las funciones de interacción con el BFM respectivamente.

  Estas funciones de interacción, "*driver_bfm*" y "*monitor_bfm*", se encargan del estímulo y recolección de datos con el DUT dependiendo del valor de **TEST_TYPE**. Dentro de la función "*monitor_bfm*" que se utiliza la función "*collect_binary_value"* del package (Código 16).
- Driver Handshakes: Contiene funciones auxiliares para iniciar los handshakes de escritura y lectura de manera síncrona.
- Monitor Handshakes: Contiene funciones auxiliares para asertar los handshakes de la transacción de escritura y lectura, activando las señales "*wdone*" y "*rdone*" en caso afirmativo. Además, permite también recolectar los valores necesarios para la comparación del test de transferencia de datos.

```
class BFM(Singleton)
    """BFM for AXI 4 Lite Protocol"""

C_TIMEOUT_TIME = 5 clk cycle

    def init()
        dut = cocotb.top
        driver_queue = 1
        monitor_queue = 0

## Main Functions
    def init_clock()
        c = clock(dut.s00_axi_aclk, 10, 'ns')
        c.start()

    def reset_dut()

        wait_clk_cycle

        s00_axi_aresetn, s00_axi_awvalid, s00_axi_wdata, s00_axi_awprot,
```

```
        s00_axi_wstrb, s00_axi_wvalid, s00_axi_bready, s00_axi_araddr,
        s00_axi_arprot, s00_axi_arvalid,s00_axi_rready, s00_axi_awaddr = 0

        wait_clk_cycle

        s00_axi_aresetn = 1

        def start_bfm()
            driver_bfm()
            monitor_bfm()

    def driver_bfm()
        infinite bucle

            wait_clk_cycle

            driver_queue.receive(t)
            if driver_queue.empty : skip
           else:
               if test_type = 1: driver_read_handshake()
               if test_type = 2: driver_write_handshake()
               if test_type = 3: driver_write_handshake() and
                                 driver_read_handshake()

    def monitor_bfm()
        infinite bucle

            wait_clk_cycle

            driver_queue.receive(test_type)

            wait_clk_cycle

            if driver_queue.empty : skip
           else:
               if test_type = 1: monitor_read_handshake()
                                 if ValueError: skip
                                 else: rdone = 1
               if test_type = 2: monitor_write_handshake()
                                 if ValueError: skip
                                 else: wdone = 1
               if test_type = 3: monitor_write_handshake()
                                 if ValueError: skip
                                 else: wdone = 1
                                 monitor_read_handshake()
                                 if ValueError: skip
                                 rdone = 1
                                 collect_binary_value(test_type =
                                                      dut(test_type),
                                                      wdone = dut(wdone),
                                                      rdone = dut(rdone),
                                                      wdata = dut(wdata),
                                                      wstrb = dut(wstrb),
                                                      rdata = dut(rdata))
                                 t = (test_type, wdone, rdone, wdata, wstrb,
                                      rdata)
                                 monitor_queue.send(t)

## Driver Handshakes
```

```
    def driver_read_handshake()
        wait(drv_arhdsk() and drv_rhdsk())

    def drive_rhdsk ()
        wait(dut(s00_axi_rvalid = 1))
        dut(s00_axi_rready) = 1
        wait(dut(s00_axi_rvalid = 0))
        dut(s00_axi_rready) = 0

    def drv_arhdsk()
        dut(s00_axi_arvalid) = 1
        wait(dut(s00_axi_arready) = 0)
        dut(s00_axi_arvalid) = 0

   def driver_write_handshake()
        wait((drv_awhdshk() and drv_whdshk()))
        dut(s00_axi_bready) = 1
        wait(dut(s00_axi_bvalid) = 0)
        dut(s00_axi_bready) = 0

    def drv_awhdsk()
        dut(s00_axi_awvalid) = 1
        wait(dut(s00_axi_awready) = 0)
        dut(s00_axi_awvalid) = 0

    def drv_whdsk()
        dut(s00_axi_wvalid) = 1
        wait(dut(s00_axi_wready) = 0)
        dut(s00_axi_wvalid) = 0

## Monitor Handshakes

    def monitor_read_handshake()
        wait((mon_rhdsk() and mon_arhdsk()) or C_TIMEOUT_TIME)
        if C_TIMEOUT_TIME has passed: ERROR

    def mon_rhdsk()
        wait(dut(s00_axi_rvalid) = 0)

    def mon_arhdsk()
        wait(dut(s00_axi_arready) = 0)

    def monitor_write_handshake()
        wait((mon_awhdsk() and mon_whdsk() and mon_bhdsk()) or
             C_TIMEOUT_TIME)
        if C_TIMEOUT_TIME has passed: ERROR

    def mon_awhdsk()
        wait(dut(s00_axi_awready) = 0)

    def mon_whdsk()
        wait(dut(s00_axi_wready) = 0)

    def mon_bhdsk()
        wait(dut(s00_axi_bvalid) = 0)
```

Código 25. Pseudocódigo del BFM del testbench de pyUVM.

5.2.3.7. DUT

El DUT es el mismo que el testbench de UVM, por lo que se puede obtener la información de la entidad en el capítulo 5.2.2.7.

5.2.3.8. Monitor

```
class Monitor(uvm_monitor)
    """Monitor for AXI 4 Lite Protocol"""

axi4lite_bfm bfm

    def init(uvm_parent = uvm_monitor)
        inherit(uvm_parent)

    def run()
        infinite bucle
            monitor_queue.receive(t)
            send_to_scoreboard(t)
```

Código 26. Pseudocódigo del Monitor del testbench de pyUVM.

Al igual que el Driver de pyUVM (capítulo 5.2.3.5), los requisitos del Monitor son sencillos:

- El Monitor debe ser capaz de recoger datos captados por el BFM
- El Monitor debe ser capaz de enviar los datos al Scoreboard.

Estos dos requisitos son realizados por la función "*run*" del Monitor, que recibe del BFM a través de "*monitor_queue.receive*" los valores requeridos y acto seguido los envía al Scoreboard.

5.2.3.9. Scoreboard

```
class Scoreboard(uvm_scoreboard)
    """Scoreboard for AXI 4 Lite Protocol"""

    def init(uvm_parent = uvm_scoreboard)
        inherit(uvm_parent)

    def verify()
        passed = TRUE
        receive_from_monitor(t)
            t = (test_type ,wdone ,rdone, wdata, wstrb, rdata)
            collected_t = (wdone, rdone, 1)
            predicted_t = axi4lite_predictor(test_type,wdata,wstrb,rdata)
```

```
            if predicted_t == collected_t: print("TEST PASSED")
            else: print("TEST FAILED")
                  passed = FALSE
        assert(passed)
```

Código 27. Pseudocódigo del Scoreboard del testbench de pyUVM.

De nuevo, la Scoreboard se apalanca en la flexibilidad de Python, dejando la función de predicción del valor correcto en el package. Debido a ello, el código restante de este código respecto del de UVM es mucho más liviano. Primero, recibe la transacción del Monitor. Acto seguido, la desmenuza y reconstruye tanto para guardar los valores de comparación como para enviar al "*axi4lite_predictor*" los argumentos necesarios. Una vez obtenido el valor predicho ("*predicted_t*"), lo compara con el recolectado ("*collected_t*"). Si son equivalentes, se da el test por superado utilizando la función "*assert*". Esta función comprueba si el valor introducido en la función es igual a TRUE. De no ser así salta un tipo de error único llamado "*AssertError*".

5.2.3.10. Environment

```
class environment(uvm_environment)

    def init(uvm_parent = uvm_environment)
        inherit(uvm_parent)


    def build()
        create_sequencer(seqr)
        create_driver(driver)
        create_monitor(monitor)
        create_scoreboard(scoreboard)

    def connect()
        connect(seqr,driver)
        connect(monitor,scoreboard)
```

Código 28. Pseudocódigo del Environment del testbench de pyUVM.

Al no existir el bloque Agent, el Environment encapsula todos los bloques del test. Es importante especificar las conexiones que se realizan en el Environment, ya que si se observa con detenimiento se ve que no hay conexión con el BFM. Esto es porque el BFM es un conjunto de funciones "auxiliares" que son llamadas desde el Monitor y el Driver. Se puede observar también en el Código 25 que el BFM no hereda de ningún

componente de UVM, porque no lo es. Hereda las propiedades de una clase **Singleton**. Al heredar las propiedades de una clase Singleton se restringe la creación del BFM a un solo bloque. El objetivo es definir claramente que parte del testbench se conecta de manera única con el DUT. Así pues, se pueden realizar distintos tipos de Driver que modulen el BFM o distintos tipos de Monitor que recojan los datos a placer, pero solo habrá un BFM que se conectará al DUT, evitando problemas de manipulación de datos desde diversas clases.

5.2.3.11. pyUVM Top Module

```
## Write Test
class WriteTest(uvm_test)
    """Write test for AXI 4 Lite, only write transaction is verified"""

        def init(uvm_parent = uvm_environment)
        inherit(uvm_parent)

        def build()
             create_environment(env)
             create_WriteSeq_sequence(write_test)

        def run()
             start(write_test)

## Read Test
class ReadTest(uvm_test)
    """Read test for AXI 4 Lite, only read transaction is verified"""
        def build()
             create_environment(env)
             create_ReadSeq_sequence(test)

        def run()
             start(read_test)

## Complete Test
class CompleteTest(uvm_test)
    """Full test for AXI 4 Lite, complete data transference is verified"""
        def build()
             create_nvironment(env)
             create_CompleteSeq_sequence(complete_test)

        def run()
             start(complete_test)
```

Código 29. Pseudocódigo del Top Module del testbench de pyUVM.

El objetivo principal del módulo top es instanciar los tests virtuales y lanzarlos. Se pueden distinguir perfectamente gracias a las partes remarcadas los distintos tests virtuales. En ellos se instancia el mismo entorno (podrían tener distintos entornos, pero para este testbench se realiza el mismo) y se crea el test virtual en base a la secuencia. Por último, lanzan el test en el bloque "*run*". En el registro de simulación de pyUVM (Código 31) se ve como se lanzan los tres tests virtuales en este orden, y al final se realiza un resumen.

6. RESULTADOS OBTENIDOS

Una vez simulados los dos testbenches, se han recogido tanto las métricas nombradas en el capítulo 3.2 como los resultados de la simulación mediante el registro y las formas de onda.

6.1. VIP en UVM

La velocidad de ejecución del testbench completo es de 7 segundos, el total de líneas de código asciende a 649 líneas de código.

```
## MARCA DE TIEMPO 1
launch_simulation
Command: launch_simulation
INFO: [Vivado 12-12493] Simulation top is 'vip_axi4lite_s_uvm_top'
…
Time resolution is 1 ps
open_wave_config
C:/Users/JaviSV/Desktop/TFG.git/tfg.git/UVM/AXI4LITE_S_VIP/01_project/vip_axi
4lite_s_uvm_top_behav.wcfg
source vip_axi4lite_s_uvm_top.tcl
# run 1000ns
----------------------------------------------------------------
(C) 2007-2014 Mentor Graphics Corporation
(C) 2007-2014 Cadence Design Systems, Inc.
(C) 2006-2014 Synopsys, Inc.
(C) 2011-2013 Cypress Semiconductor Corp.
(C) 2013-2014 NVIDIA Corporation
----------------------------------------------------------------
MARCA DE TIEMPO 2
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(22) @ 0: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(23) @ 0: uvm_test_top [TEST]
Starting Single Write Test
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(24) @ 0: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 75000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 115000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 155000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 195000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 235000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 275000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 315000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 355000:
```

```
uvm_test_top.e.sco [SCO] TEST PASSED
MARCA DE TIEMPO 3
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(28) @ 385000: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(29) @ 385000: uvm_test_top [TEST]
Starting Single Read Test
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(30) @ 385000: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(27) @ 395000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 435000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 475000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 515000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 555000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 595000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 635000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 675000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 715000:
uvm_test_top.e.sco [SCO] TEST PASSED
MARCA DE TIEMPO 4
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(34) @ 745000: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(35) @ 745000: uvm_test_top [TEST]
Starting Complete Transaction Test
UVM_INFO %UVM_PATH%/vip_axi4lite_s_test.sv(36) @ 745000: uvm_test_top []
==================================
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(36) @ 755000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(63) @ 825000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(63) @ 895000:
uvm_test_top.e.sco [SCO] TEST PASSED
UVM_INFO %UVM_PATH%/vip_axi4lite_s_scoreboard.sv(63) @ 965000:
uvm_test_top.e.sco [SCO] TEST PASSED
MARCA DE TIEMPO 5
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:07 ; elapsed = 00:01:49 . Memory
(MB): peak = 1382.684 ; gain = 0.000
```

Código 30. Registro de simulación del testbench de UVM.

En el registro de simulación se remarcan cinco partes importantes del testbench:

- Marca de Tiempo 1: Inicia el testbench con el comando "*launch_simulation*", reconoce el módulo top, define la escala de tiempo y abre la forma de ondas.
- Marca de Tiempo 2: Comienza el testbench verificando primero las transacciones de escritura. Se puede observar como el Scoreboard, que es el

encargado de comparar los valores recogidos del monitor con los esperados, verifica que la transacción se realiza correctamente. Aparece una variable llamada %UVM_PATH%. Esta variable no es más que la ruta absoluta hasta el archivo. Se ha introducido esta variable debido a la irrelevancia de la jerarquía de archivos del proyecto en el estudio y para simplificar el código.

- Marca de Tiempo 3: Prosigue verificando las transacciones de lectura, verificando correctamente las transacciones.
- Marca de Tiempo 4: Lanza la tercera y última secuencia, verificando la transferencia de datos al completo.
- Marca de Tiempo 5: Por último, recopila el tiempo de simulación (el tiempo de simulación por defecto de Vivado son 1000 ns) y algunos datos físicos como el tiempo de simulación real y la memoria usada.

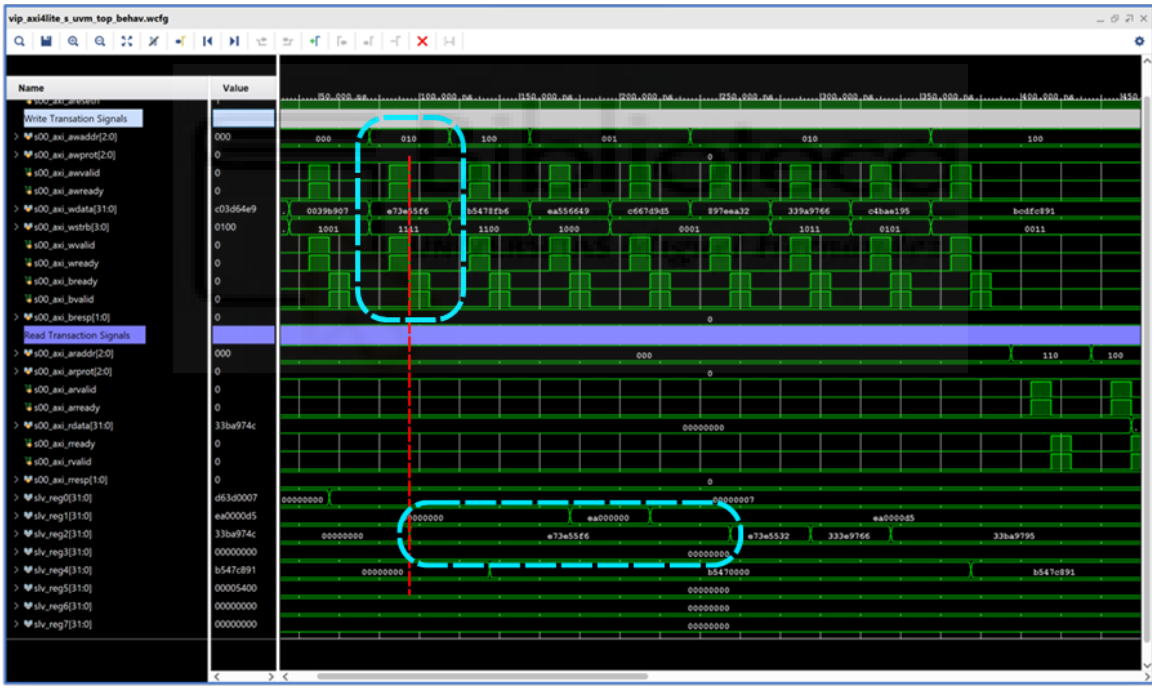

Figura 15. Forma de onda del test de Transacción de Escritura en UVM.

En la Figura 15 se pueden observar las distintas transacciones realizadas en el testbench de UVM. Esta figura específicamente refleja el test de transacción de escritura. La parte marcada con recuadros azules y una línea roja representa una transacción de escritura. Se puede ver como lo primero que se aporta a la transacción son los datos de **AWADDR**, **WDATA** y **WSTRB**. Tras ello pasan a estar activas como ya se ha visto en el Driver (capítulo 5.2.2.5) las señales **AWVALID** y **WVALID**. Tras recibir la

respuesta del DUT confirmando que se ha recibido tanto la dirección como el dato (activando **AWREADY** y **WREADY**) se puede observar como se escribe en el registro el valor correcto (se debe prestar especial atención al **WSTRB**, como en este caso todos los bits son '1', se escriben todos los valores de **WDATA** en el registro). Tras escribirse en dato en el registro se realiza el handshake de confirmación de escrita con **BREADY** y **BVALID** y se da por finalizada la transacción.

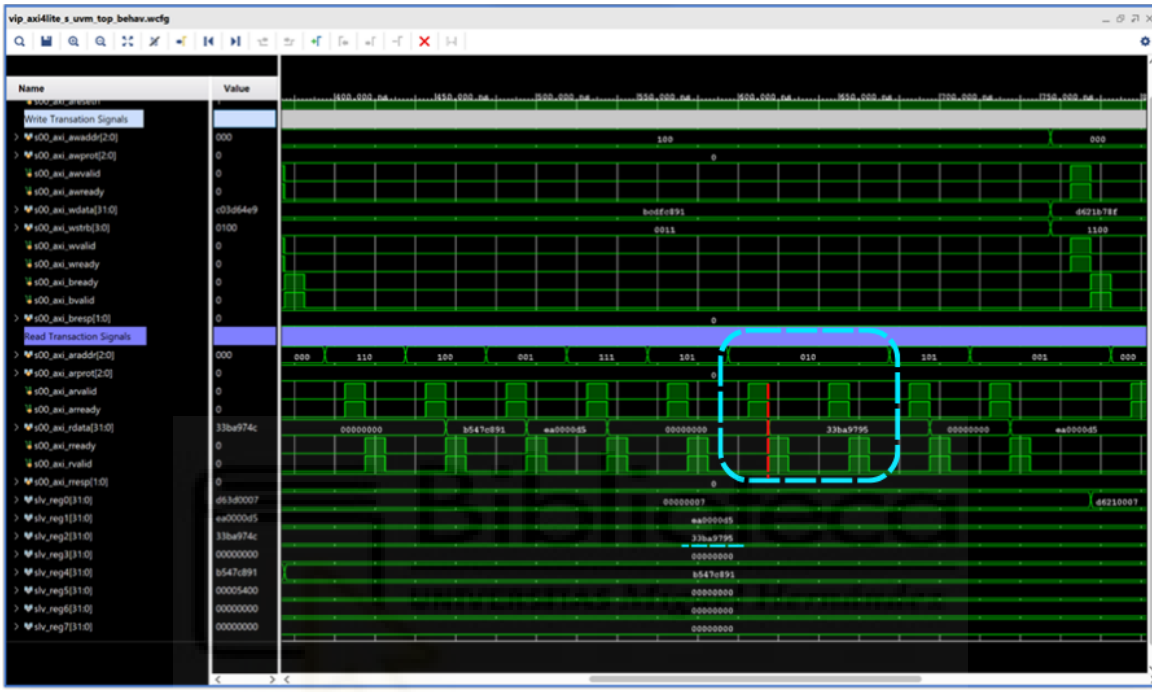

Figura 16. Forma de onda del Test de Transacción de Lectura en UVM.

Como ejemplo de transacción de lectura podemos apreciar las partes remarcadas en la Figura 16. Se puede cerciorar como se recibe primero **ARADDR**, indicando la dirección que se quiere leer, y acto seguido se manda **ARVALID** para decirle que la dirección es correcta y comenzar la transacción de escritura. El AXI4-Lite Slave responde activando **ARREADY** para informar que ha encontrado el registro y tras confirmar que las dos señales estaban activas envía el valor por **RDATA**. Para dar por finalizada la transacción se utiliza el handshake de confirmación de lectura mediante **RREADY** y **RVALID**.

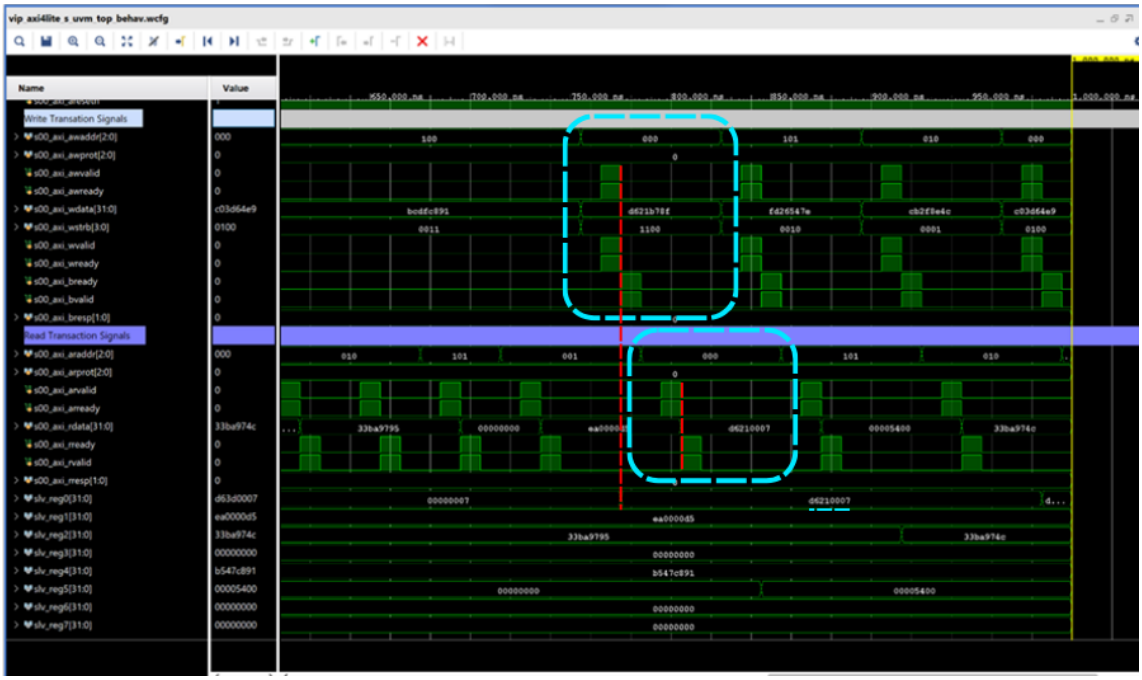

Figura 17. Forma de onda del Test de Transferencia de Datos en UVM.

Por último, la Figura 17 muestra las transferencias de datos realizadas en el test de UVM. El primer recuadro realiza una transacción de escritura. De nuevo, en el momento de confirmación de los handshakes de escritura se escribe el valor en el registro 0 (el indicado en **AWADDR**). Cuando se da por finalizada la transacción de escritura, se procede a comenzar la de lectura en el mismo registro. Este ejemplo clarifica también el uso del **WSTRB**. Si se observa antes de escribir en el registro, éste tenía el valor de "00000007" (está expresado en valor hexadecimal). Cuando se escribe en él, se con el **WSTRB** "1100", es decir, que escribe solo los primeros 4 bytes más significativos de **WDATA**, dejando los 4 bytes menos significativos. Es por ello por lo que en el registro permanece el "0007". Al leer el valor, se recoge el dato "d6210007". El Scoreboard de UVM tiene en cuenta este comportamiento, y añade la variable **WSTRB** a la comparación para verificar correctamente la transferencia de datos.

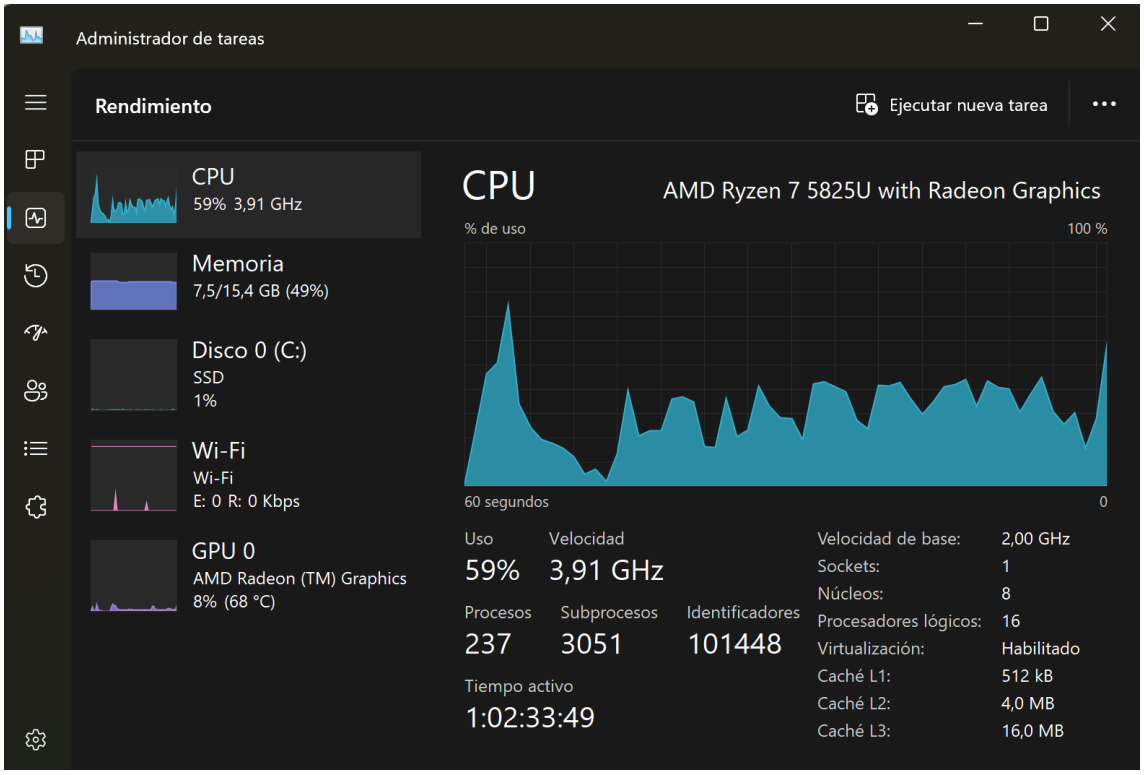

Figura 18. Gasto de CPU y RAM al simular el testbench de UVM.

Al simular el testbench de UVM, se han registrado unos valores del 59% de uso de la CPU y un 48% de uso de RAM.

6.2. VIP en pyUVM

La velocidad de ejecución del testbench completo es de 0.92 segundos, el total de líneas de código asciende a 588 líneas de código.

```
MARCA DE TIEMPO 1
make
0.00ns INFO     cocotb                                 Running on GHDL version
3.0.0 (2.0.0.r1417.g7de967c51) [Dunoon edition]
     0.00ns INFO     cocotb                                 Running tests with
cocotb v1.7.2 from C:\Users\JaviSV\.conda\envs\Intigia\Lib\site-
packages\cocotb
     0.00ns INFO     cocotb                                 Seeding Python
random module with 1694099200
     0.00ns INFO     cocotb.regression                      Found test
pyuvm_top.WriteTest
     0.00ns INFO     cocotb.regression                      Found test
pyuvm_top.ReadTest
     0.00ns INFO     cocotb.regression                      Found test
pyuvm_top.CompleteTest
MARCA DE TIEMPO 2
     0.00ns INFO     cocotb.regression                      running WriteTest
(1/3)
```

```
                                                    Trial test for
AXI 4 Lite, only write transaction
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     .._top.env.SCOREBOARD2234190719824
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   485.00ns INFO     cocotb.regression                    WriteTest passed
MARCA DE TIEMPO 3
   485.00ns INFO     cocotb.regression                    running ReadTest
(2/3)
                                                    Trial test for
AXI 4 Lite, only read transaction
   720.00ns INFO     .._top.env.SCOREBOARD2234190619728
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   720.00ns INFO     .._top.env.SCOREBOARD2234190619728
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   720.00ns INFO     .._top.env.SCOREBOARD2234190619728
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   720.00ns INFO     .._top.env.SCOREBOARD2234190619728
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
   720.00ns INFO     cocotb.regression                    ReadTest passed
   720.00ns INFO     cocotb.regression                    running CompleteTest
(3/3)
MARCA DE TIEMPO 4
                                                  Complete sequence
test for AXI 4 Lite
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     .._top.env.SCOREBOARD2234192085392
vip_axi4lite_s_scoreboard.py(39)[uvm_test_top.env.SCOREBOARD]: TEST PASSED
  1565.00ns INFO     cocotb.regression                    CompleteTest passed
```

```
  1565.00ns INFO     cocotb.regression


MARCA DE TIEMPO 5
**************************************************************************
** TEST                   STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
**************************************************************************
** pyuvm_top.WriteTest    PASS   485.00         0.02         24685.25     **
** pyuvm_top.ReadTest     PASS   235.00         0.01         18773.79     **
** pyuvm_top.CompleteTest PASS   845.00         0.03         28760.51     **
**************************************************************************
** TESTS=3 PASS=3 FAIL=0 SKIP=0  1565.00        0.92         1705.54      **
**************************************************************************
```

Código 31. Registro de simulacion del testbench de pyUVM.

En el registro de simulación se remarcan cinco partes importantes del testbench:

- Marca de Tiempo 1: Inicia el testbench con el comando "*make*", busca e inicializa el simulador, el módulo top y las secuencias de pyUVM.
- Marca de Tiempo 2: Comienza el testbench verificando las transacciones de escritura de forma satisfactoria mediante el scoreboard.
- Marca de Tiempo 3: Prosigue verificando las transacciones de lectura de forma satisfactoria.
- Marca de Tiempo 4: Lanza la tercera y última secuencia, verificando la transferencia de datos al completo.
- Marca de Tiempo 5: Por último, recopila las verificaciones de cada una de las secuencias lanzadas en el test y muestra el estado de ellas, el tiempo de simulación, el tiempo real de simulación y la velocidad a la que ha simulado.

Figura 19. Forma de onda del test de Transacción de Escritura y Lectura en pyUVM.

Al igual que en el test de UVM, podemos observar las transacciones de escritura y lectura en el test de pyUVM. Se han encontrado dos diferencias principales en las formas de onda. La primera es que el DUT no parece reconocer las señales enviadas por el BFM hasta el siguiente flanco de reloj, lo que el testbench tarda un ciclo de reloj más en hacer la transacción (aspecto que se vuelve casi irrelevante cuando se compara el tiempo de simulación real de los 2 testbenches). La segunda diferencia, es que al haber utilizado secuencias virtuales en pyUVM el testbench ha tomado como inicio del test cada vez que se lanzaba una secuencia. Esto ha producido que pyUVM inicialice los registros en cada test virtual, dejando las transacciones de lectura sin un valor que leer. No obstante, como el objetivo del test de transacción de lectura no es el de recibir un dato sino verificar el handshake, se da como superado el test.

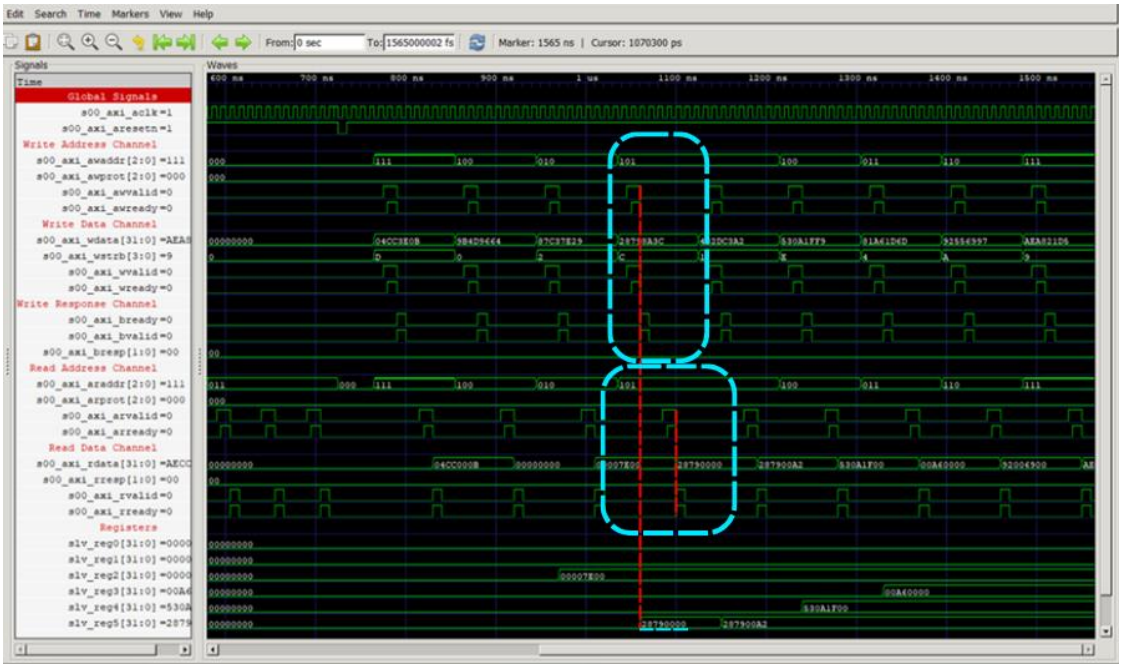

Figura 20. Forma de onda del test de Transferencia de Datos en pyUVM.

Finalmente, la forma de onda de pyUVM refleja de nuevo una transferencia de datos totalmente satisfactoria. Se puede apreciar de nuevo el ciclo que tarda el DUT en reconocer las señales enviadas por el testbench de pyUVM (se estima una pérdida de 10 nanosegundos por transacción). Este hecho se atribuye al desconocimiento del comportamiento de los deltas al simular con funciones de Python.

Se puede observar cómo tras la primera línea roja, al verificar que los valores enviados por el BFM son válidos, se escribe en el registro los bytes correspondientes (**WSTRB** aparece en hexadecimal, refiriéndose por C a “1100”). Tras realizar la transacción de escritura, se realiza la transacción de escritura sin complicaciones y se extrae el valor esperado, dando por superado el test.

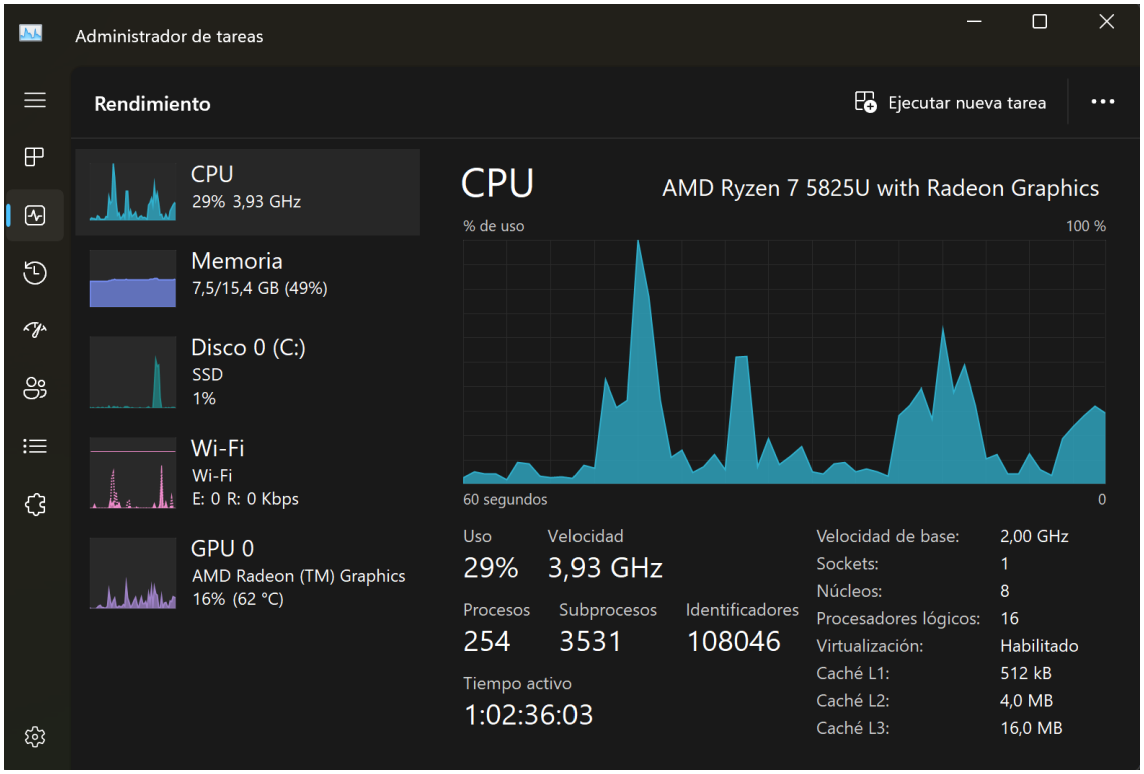

Figura 21. Gasto de CPU y RAM al simular el testbench de pyUVM.

En este estudio los dos tests fueron lanzados en el mismo PC con una diferencia de 10 minutos entre ellos para intentar igualar las condiciones de estos. Al simular el testbench de UVM, se han registrado unos valores del 29% de uso de la CPU y un 49% de uso de RAM.

7. CONCLUSIONES

7.1. Interpretación de los resultados

En cuanto a la velocidad de simulación, a pesar de la reputación de Python de ser un lenguaje lento, podemos observar al final de los dos registros de simulación como el test de pyUVM, usando el simulador open-source GHDL, apenas tiene una duración de 0,92 segundos.

Por otro lado, el test de UVM, utilizando el simulador Vivado™ ML Standard Edition (su versión gratuita, que solo limita el número de componentes en los que implementar circuitos) dura nada más y nada menos que 7 segundos.

Se aprecia claramente una inclinación hacia pyUVM en este aspecto. Y no solamente en este, sino que la cantidad de líneas de código favorece también a pyUVM (649 líneas en SystemVerilog contra 588 en Python, una diferencia de 61 líneas de código, lo que equivale a un 9,40% menos de líneas de código en pyUVM). Si bien es un testbench relativamente pequeño, hay bastantes más líneas con comentarios en Python que en SystemVerilog (dada la funcionalidad que tiene Python de poder darle una descripción a una función, clase u objeto). Por ello la diferencia descartando los comentarios podría aproximarse a 100. Si se escalase a un testbench de mayor complejidad, hay una gran probabilidad que esta diferencia escalase con él. Esto nos deja pistas sobre la facilidad de programar en Python (un aspecto bastante obvio, aunque no sea normalmente expuesto desde este punto de vista).

En cuanto al uso de CPU y de RAM, son valores bastante relativos debido a que el PC en el que han sido simulados no ha sido completamente optimizado. No obstante, no se debería despreciar la diferencia relativa que hay entre el gasto de CPU al realizar el testbench de UVM al gasto de CPU al simular en Python (una diferencia del 30%).

A la hora de afrontar el reto de plantear una verificación y buscar soporte o apoyo en librerías de código, se ha podido comprobar extensamente que la librería de UVM es inmensa en comparación a la de pyUVM.

Debido a esto, la realización de la estructura del testbench en SystemVerilog fue más sencilla. Se asocia esta conclusión a la reciente creación de la librería pyUVM, que aún no se ha dispuesto del suficiente tiempo como para haberse generado una librería que sirva de referencia para nuevos ingenieros.

En cuanto a la cobertura funcional y a la implementación de la técnica CRV, ambos cumplen satisfactoriamente todos los requisitos, permitiendo generar valores aleatorios restringiendo su valor y manipulando el DUT sin complicaciones aparentes.

Cabe destacar ciertos aspectos en los últimos valores sujetos a evaluación en el estudio. Si bien es cierto, a primera vista parecería más complejo aprender y dominar SystemVerilog para realizar testbenches. No obstante, para simular pyUVM se requiere de un entorno bastante más complejo que con UVM, el cual requiere solo de un compilador y simulador. El entorno de pyUVM requiere tener cocotb, makefiles, y otros add-ons que facilitan la programación en Python. Como bien se ha dicho, el diseño de este entorno puede resultar más caótico, pero sin duda merece la pena si tomamos en cuenta la gran cantidad de simuladores y herramientas que cocotb, y por ende pyUVM, soportan para optimizar la simulación de sus diseños [17], además de su baja barrera de entrada, el cual es más familiar a los ojos del programador al ser un lenguaje interpretado.

Puede parecer a la vista de los resultados de este estudio que la “alta” dificultad de generar un entorno óptimo para simular Python en comparación con SystemVerilog es un inconveniente, pero eso no es del todo cierto. Sería más propicio denominarlo un arma de doble filo. Python es un lenguaje mucho más versátil que SystemVerilog, conocido mundialmente en cualquier ámbito relacionado con la programación, y esto permite que se creen muchas conexiones entre distintos lenguajes y compatibilidades inesperadas.

Como se ha visto en el estudio del arte, la industria EDA se ha caracterizado por ser reticente a los cambios probablemente debido a los altos costes de los simuladores,

tales como ModelSim (a partir de 945$), Aldec Rivera-Pro (a partir de 5900$), etc. Esto sumado al trabajo que supone rehacer una librería de código tan amplia como su antecesora. Es por ello por lo que se espera que la percepción social en cuanto a pyUVM sea negativa respecto a UVM.

7.2. Limitaciones del estudio y áreas de mejora futura.

De cara a futuros estudios que quieran realizarse en este ámbito, se dejan como recomendación los siguientes temas:

- Realización del mismo estudio con un diseño con varias interfaces.

Al disponer de varias interfaces, el test debería disponer de varios Agents que puedan abarcar todas las señales a manipular y por recolectar. Estudiar los límites de pyUVM y UVM en este ámbito podrían arrojar datos interesantes relacionados con este estudio.

- Comparación de este estudio con uno análogo con VHDL

En el estudio del arte [2.2.4] se menciona como la comunidad VHDL afronta el problema de la verificación tomando una vía paralela utilizando únicamente su lenguaje de programación, dando lugar a OSVVM y UVVM. Comparar las dos metodologías, tanto por separado como conjuntamente a las dos ya comparadas en este estudio, aportarían una visión más global del sector de la verificación en la industria EDA.

- Profundizar en los distintos simuladores

Cambiando el punto de enfoque, las formas en las que los simuladores compilan los testbenches es variada y diversa. Analizar para cada metodología qué simulador se adapta más a ella sería una vertiente interesante de investigación. Esto podría realizarse programando varios diseños en distintos lenguajes, y simulándolos en la mayor cantidad de simuladores posibles, enfocándose principalmente en los aspectos físicos, como el tiempo de simulación o la RAM utilizada.

7.3. Predicción del uso de las librerías en el futuro en base al estudio realizado.

Una vez analizado el estado del arte del sector de la verificación en la industria EDA y las conclusiones extraídas de este estudio, no sería descabellado afirmar que pyUVM ha llegado para quedarse, ya que aporta nuevas oportunidades de afrontar la verificación, y es justo lo que el sector necesita.

No obstante, necesitarlo no es lo mismo que quererlo, y es por ello que pyUVM, en caso de instaurarse como una de las metodologías de verificación utilizadas en la verificación, deberá pasar por un camino de espinas, comenzando por la aceptación social de empresas y proyectos, que "arriesguen" y decidan apostar por ella.

Hay dos eventos clave que podrían desequilibrar la balanza para cualquiera de los dos lados, y estos son:

1. Restricciones (o ayudas) de las distintas compañías con simuladores para favorecer una metodología u otra.
2. Capacidad de cocotb de absorber VIPs en otros lenguajes, tanto en VHDL como en SystemVerilog.

No sería la primera vez que la industria se deja llevar por los caprichos del dinero [2.1.2] favoreciendo a uno de los dos aspirantes, siempre a beneficio del que realice la ayuda por supuesto. Como ya se ha nombrado varias veces en este estudio, los simuladores actualmente tienen unos costes desorbitados, y generan grandes beneficios para las empresas. La librería pyUVM no es a priori una amenaza, pero sí que acerca al programador a varios simuladores open-source, sacándoles un gran rendimiento, lo cual podría generar desconfianza en los simuladores actuales de pago.

Por último, y esto es más bien una presunción que una situación, es una apuesta por Python. Al ser un lenguaje flexible y en constante desarrollo y crecimiento, podría darse el caso que en un futuro cocotb o pyUVM permitieran integrar VIPs de otros

lenguajes de programación. De hecho, es algo que justamente esperaríamos que Python pudiera hacer.

Si esto sucediera, eliminaría por completo el problema del soporte, ya que se podrían reusar todos los diseños anteriormente generados, no solo en SystemVerilog, sino tal vez en VHDL o "*e*". Esto probablemente sí que supondría el fin de UVM como metodología estándar, ya que se vería absorbida por pyUVM.

TERMINOLOGÍA USADA

A

Accellera

Organización sin fines de lucro que impulsa la adopción y desarrollo de estándares de verificación y diseño de sistemas electrónicos, proporcionando un foro para la colaboración entre la industria y las comunidades de usuarios..... 18

ADA

Lenguaje de programación de alto nivel diseñado especialmente para aplicaciones críticas en tiempo real. 15

API

Application Programming Interface. Conjunto de reglas y protocolos que permite que diferentes aplicaciones y sistemas interactúen y se comuniquen entre sí de manera estandarizada. 21

ARM

Empresa británica de diseño de software y semiconductores. 14

ASIC

Application-Specific Integrated Circuit. Tipo de circuito integrado diseñado para realizar funciones específicas en aplicaciones particulares, lo que lo hace más eficiente y optimizado en comparación con soluciones genéricas..... 16

AVM

Advanced Verification Methodology 19

AXI

Advanced eXtensible Interface. Estándar de bus de interconexión de alta velocidad desarrollado por ARM, utilizado para facilitar la comunicación entre diferentes componentes. 14

B

BFM

Bus Functional Model. Representación en software de un componente de un circuito electrónico. Tiene varios buses externos y no es sintetizable..... 61

C

C

Lenguaje de programación de propósito general ampliamente utilizado, que ha sido fundamental en el desarrollo de sistemas operativos y aplicaciones de software..... 16

CAD

Computer-Aided Design. Software especializado que permite a los diseñadores crear, editar y analizar diseños y modelos digitales..... 20

cocotb

Biblioteca de código abierto en Python que permite la creación de testbenches de simulación para verificación de hardware digital mediante la combinación de lenguajes de descripción de hardware (HDL) y Python..... 21

CRV

Constrained Random Verification. Técnica para generar casos de prueba aleatorios con restricciones específicas para garantizar que los estímulos de entrada generados cumplen determinados requisitos de diseño. 28

D

Driver

E

F

G

H

I

K

M

O

P

R

S

T

U

V

W

Bibliografía

[1] R. G. Sargent and O. Balci, "History of verification and validation of simulation models," in *Winter Simulation Conference (WSC)*, Las Vegas, NV, USA, 2017.

[2] S. Olloz, L. Teres, E. Villar y Y. Torroja, VHDL - Lenguaje estándar de Diseño Electrónico, McGrawHill, 1998.

[3] S. Golson and L. Clark, "Language Wars in the 21st Century: Verilog versus VHDL - Revisited," in *SNUG Austin 2016 Proceedings*, 2016.

[4] IEEE, 1076-1987 - IEEE Standard VHDL Language Reference Manual, IEEE, 1988.

[5] P. Flake, P. Moorby, S. Golson, A. Salz and S. Davidmann, "Verilog HDL and its ancestors and descendants," *Proceedings of the ACM on Programming Languages,* pp. 1-90, 12 June 2020.

[6] R. Salemi, Python for RTL Verification, Independently pubished, 2022.

[7] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Design & Test of Computers,* pp. 18-25, July-Aug 2009.

[8] J. Bergeron, E. Cerny, A. Hunter and A. Nightingale, Verification Methodology Manual for SystemVerilog, Springer, 2005.

[9] IEEE, IEEE Standard for Universal Verification Methodology Language Reference Manual, IEEE, 2020.

[10] D. L. Perry, VHDL. Programming by Example, McGrawHill, 2002.

[11] H. Foster, "The 2022 Wilson Research Group Functional Verification Study," 10 Octubre 2022. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study/.

[12] T. Gingold, "GHDL," 2017. [Online]. Available: https://ghdl.github.io/ghdl/. [Accessed 29 Julio 2023].

[13] A. *AMBA AXI and ACE Protocol Specification,* ARM, 2003, 2004, 2010, 2011, 2013.

[14] Real Digital, "RealDigital," [Online]. Available: https://www.realdigital.org/doc/6cb4e88c145c960ad199be9fac5e9402. [Accessed 30 Julio 2023].

[15] Doulos, "EDA Playground," [Online]. Available: https://edaplayground.com/. [Accessed 05 September 2023].

[16] AMD, Vivado Design Suite User Guide: Synthesis, [Online]. Available: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis/Vivado-Design-Suite-User-Guide-Synthesis. [Accessed 05 September 2023].

[17] Open-Source, "Cocotb," 2014. [Online]. Available: https://docs.cocotb.org/en/stable/index.html. [Accessed 23 Julio 2023].