ULPGC Universidad de Las Palmas de Gran Canaria

eite escuela de ingeniería de telecomunicación y electrónica

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

“Diseño de un testbench UVM integrando IP de verificación Mentor Graphics y un IP propio, configurable y con soporte de cobertura”

Titulación:	**Grado en Ingeniería en Tecnologías de la Telecomunicación**
Mención:	**Sistemas electrónicos**
Autor:	**D. Daniel Baute Trujillo**
Tutores:	**Dr. D. Valentín de Armas Sosa**
	Dr. D. Félix Bernardo Tobajas Guerrero
Fecha:	**Julio de 2022**

AGRADECIMIENTOS

A mis tutores, por todo lo que he aprendido de ellos tanto dentro como fuera de las aulas. Son referencias en esta escuela y su dedicación es un ejemplo a seguir para todos. En especial, este TFG no sería posible sin el tiempo y esfuerzo puestos por Valentín, por quien me decidí a elegir este tema y quien me dió la oportunidad de seguir adelante. En estos últimos meses su apoyo y guía han sido la luz que ha evitado que me pierda en el camino.

A mis amigos, por nunca hacerme sentir un extraño cada vez que volvía a casa. Por todos los triunfos y derrotas que hemos vivido juntos durante tantos años. Hemos forjado algo que ni el tiempo ni la distancia puede romper y no podría estar más orgulloso de todos y cada uno de ellos, porque no podría seguir adelante sin saber que siempre tendré un lugar al que volver 1000 días después-

A mis telecos, por convertirse en mi segundo hogar. Por todos los pasos que hemos hecho juntos a lo largo de este camino, al principio imposible, pero que se ha convertido en el mejor que podría haber recorrido. Hemos hecho un equipo de "extranjeros" que ha conseguido alegrar cada uno de mis días aquí, porque no serían igual las tardes de estudio sin las salidas y paseos de recompensa ni cada mañana sin el olor a café y vainilla.

A mi familia, por creer en mí cuando ni yo mismo lo hacía. Por darme todas las oportunidades que pueden y por siempre estar en mis fallos y mis logros, porque, como dice el bueno de Rubén Blades, a pesar de los problemas, familia es familia y cariño es cariño. Por haber hecho de mí la persona que soy. Se los debo todo y espero conseguir que estén tan orgullosos de mí como lo estoy yo de ellos.

Gracias, por ser la razón por la que no me rendiré jamás.

RESUMEN

En la actualidad, la etapa de verificación ocupa un lugar fundamental en el proceso de desarrollo de sistemas *hardware* digitales. A causa del aumento de la demanda de los sistemas electrónicos y la mejora tecnológica en el sector, los sistemas desarrollados integran un mayor número de componentes y de funcionalidades diferentes, lo que resulta en que se diseñen con una estructura más compleja. Debido a esto, el proceso de verificación se vuelve, a su vez, más complejo y, por lo tanto, el tiempo consumido a lo largo de dicho proceso aumenta. Como solución a estos inconvenientes surge la metodología *Universal Verification Methodology* (UVM), basada en el lenguaje de descripción y verificación *hardware SystemVerilog*.

El objetivo principal de este Trabajo Fin de Grado (TFG) consiste en el diseño de un *testbench* UVM utilizando dos IP de verificación distintos desarrollados con la metodología UVM. El primero está proporcionado por *Mentor Graphics, a Siemens Business,* y utiliza el protocolo de comunicación AMBA AXI4, siendo totalmente configurable. El segundo consiste en un IP de verificación propio funcionando como esclavo en modo reactivo correspondiente al protocolo de comunicación *Wishbone* en su versión B4.

En primer lugar, se realizará un estudio en profundidad del funcionamiento del protocolo AXI4 y del protocolo *Wishbone B4*, además de los fundamentos de la metodología UVM y el IP de verificación de *Mentor Graphics*. En segundo lugar, se estudiará en profundidad el dispositivo a verificar (DUV), con el fin de entender su funcionamiento. Finalmente, se desarrollará el *testbench* UVM que permitirá la verificación del correcto comportamiento de dicho dispositivo.

ABSTRACT

Nowadays, the verification stage occupies a fundamental place in the development of digital hardware systems. Because of the increasing demand for electronic systems and the technological improvement associated with the sector, the developed systems integrate a greater number of components and different functionalities, which results in a more complex structure in their design. Due to this, the verification process becomes more complex and time-consuming throughout this process increases. As a solution to these problems, the Universal Verification Methodology (UVM) arises. This methodology is based on the SystemVerilog hardware description and verification language.

The main objective of this Final Degree Project consists of the design of a UVM testbench using two different verification IP developed with the UVM methodology. The first one is provided by Mentor Graphics, a Siemens Business, and uses the AMBA AXI4 communication protocol, being fully configurable. The second consists of a proprietary verification IP running as a slave in reactive mode corresponding to the Wishbone communication protocol in its B4 version.

Firstly, an in-depth study of the AXI4 and Wishbone B4 protocols will be carried out, as well as the basics of the UVM methodology and the Mentor Graphics verification IP. Secondly, the device to be verified (DUV) will be studied in depth in order to understand how it works. Finally, the UVM testbench will be developed to verify the correct behaviour of the device.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ACRÓNIMOS

ABV	*Assertion-Based Verification*
AHB	*Advanced High-performance Bus*
AMBA	*Advanced Microcontroller Bus Architecture*
APB	*Advanced Peripheral Bus*
API	*Application Programming Interfaces*
ARM	*Advanced RISC Machine*
ASIC	*Application-Specific Integrated Circuit*
AVM	*Advanced Verification Methodology*
AXI3	*Advanced eXtensible Interface 3*
AXI4	*Advanced eXtensible Interface 4*
BFM	*Bus Functional Model*
COITT	Colegio de Ingenieros Técnicos de Telecomunicación
DPI	*Direct Programming Interface*
DSI	Diseño de Sistemas Integrados
DUV	*Device Under Verification*
ECTS	*European Credit Transfer and Accumulation System*
EDA	*Electronic Design Automation*
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
FIFO	*First In First Out*
FPGA	*Field-Programmable Gate Array*
GITT	Grado en Ingeniería en Tecnologías de la Telecomunicación
HDL	*Hardware Description Language*
HVL	*Hardware Verification Language*
I2C	*Inter-Integrated Circuit*
IGIC	Impuesto General Directo Canario
IP	*Intelectual Property*
IUMA	Instituto Universitario de Microelectrónica Aplicada
OOP	*Object-Orieted Programming*
OVM	*Open Verification Methodology*

QVIP	*Questa Verification Intellectual Property*
PCIe	*Peripheral Component Interconnect Express*
RRHH	Recursos Humanos
RTL	*Register-Transfer Level*
SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
SV	*SystemVerilog*
TFG	Trabajo Fin de Grado
TLM	*Transaction-Level Methodology*
UART	*Universal Asynchronous Receiver-Transmitter*
ULPGC	Universidad de Las Palmas de Gran Canaria
USB	*Universal Serial Bus*
UVC	*Universal Verification Component*
UVM	*Universal Verification Methodology*
VHDL	*Very High-Speed Integrated Circuit Hardware Description Language*
VIP	*Verification Intellectual Property*

MEMORIA

Capítulo 1: INTRODUCCIÓN

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

1.1. ANTECEDENTES

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

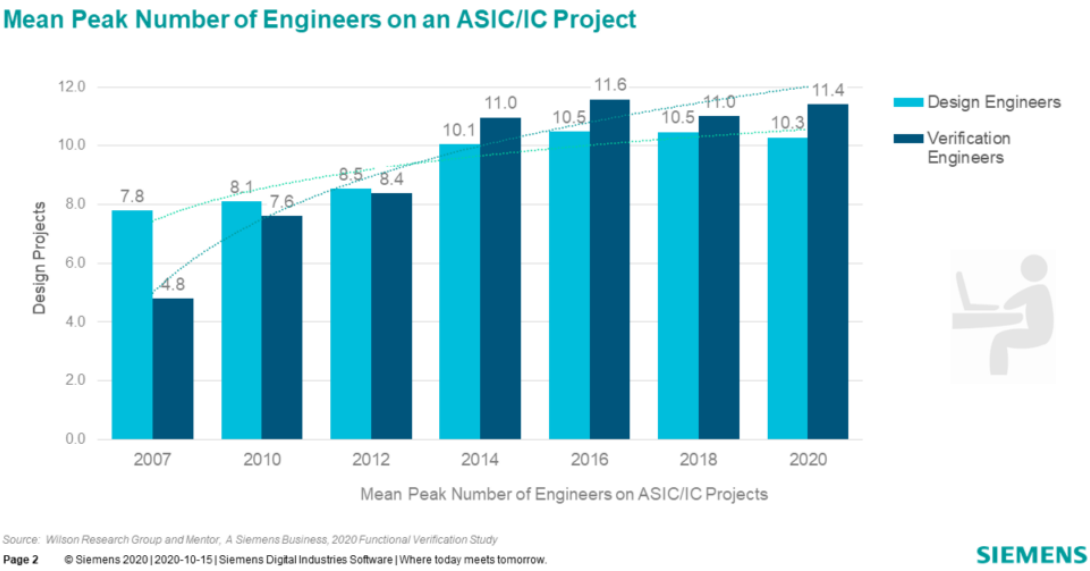

Figura 1 Pico medio de ingenieros en proyectos de circuitos integrados [1]

Actualmente, el desarrollo de sistemas electrónicos integrados se caracteriza por su gran complejidad, como pueden ser los Sistemas en Chip (*System on Chip*, SoC), en los que encontramos distintos bloques IP dentro de un mismo chip. Esta situación plantea la necesidad de encontrar un método de verificación distinto a los convencionales, basados en el análisis de formas de onda, para optimizar el flujo de trabajo. Esta necesidad ha provocado el auge de las metodologías de verificación. Sin embargo, las metodologías de verificación existentes en el mercado hace una década eran, en su mayoría, propietarias. Esta característica limita sin duda el avance de cualquier metodología. Este motivo impulsó la creación de una metodología de verificación de código abierto, como es UVM.

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

Entre las principales ventajas de adoptar UVM como metodología de verificación funcional, se puede destacar la generación de componentes y objetos reutilizables, requiriendo un mayor esfuerzo en las primeras etapas de la verificación, pero con la capacidad de ser capaz de adaptarse a nuevos sistemas. Esto conlleva a que UVM como metodología y SV como lenguaje sean los más utilizados actualmente, acaparando más del 70% de los proyectos [4].

En el mercado laboral actual, la importancia de UVM para el perfil de ingeniero de verificación es muy elevada, siendo su conocimiento y la experiencia en su uso requisitos obligatorios para la mayoría de los puestos de trabajo relacionados con este sector, en los que no

hay propuesta en las que no se soliciten estas características [5] . Esto sucede en empresas como *Meta*, antiguo *Facebook*, que pide como requisito para el puesto de ingeniero de verificación tener una experiencia mínima de más de dos años en SV y UVM [6].

Hasta la fecha, la división de Diseño de Sistemas Integrados (DSI) perteneciente al Instituto Universitario de Microelectrónica Aplicada (IUMA) ha realizado una variedad de *testbench* para distintos diseños. En su último TFG, se realizó un *tesbench* de un *crossbar* con interfaz AXI4 mediante el uso de los IP de verificación de *Mentor Graphics*. En el presente Trabajo Fin de Grado se propone la realización del *testbench* de un adaptador de protocolos que convierte el protocolo AXI4 (*Advanced eXtensible Interface* 4) a *Wishbone*. Esta necesidad surge debido a que la plataforma a utilizar en este TFG, *QuestaSim* de *Mentor Graphics*, no dispone de ningún IP de verificación compatible con este último protocolo. La solución propuesta a esta situación es utilizar los IP de verificación AXI4 proporcionados por *Mentor Graphics*, desarrollar un IP propio compatible con el protocolo *Wishbone* e integrarlo en el *testbench* UVM a desarrollar.

1.2. OBJETIVOS

El objetivo principal de este TFG es diseñar un *testbench* haciendo uso de los IP de verificación de *Mentor Graphics* para el protocolo AXI4, integrando un IP de verificación propio con protocolo *Wishbone*. La plataforma de *Mentor Graphics* proporciona IP de verificación para diferentes protocolos. Sin embargo, no proporciona ningún IP para el protocolo *Whisbone*. Al mismo tiempo, y teniendo en cuenta que la división DSI ya tiene experiencia en el manejo de los IP de verificación, se propone añadir a este IP, por un lado, un componente de cobertura para medir el grado de verificación del protocolo y, por otro, desarrollar el IP de manera que sea configurable.

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

O1. Conocer y comprender el funcionamiento del protocolo AMBA AXI4, necesario para poder realizar las tareas de comprobación, a nivel de interfaz, del correcto funcionamiento del IP en su interfaz AXI4.

O2. Estudiar en profundidad los conceptos necesarios de la metodología UVM, principalmente aquellos relacionados con la integración del IP de verificación y la descripción de secuencias de operaciones sobre el IP a verificar.

O3. Estudiar y comprender el IP de verificación de *Mentor Graphic*s, así como como entender el módulo IP que se va a verificar, lo que va a permitir describir los test necesarios para comprobar su correcto funcionamiento.

O4. Desarrollar el componente IP de verificación para la interfaz *Wishbone*, añadiendo las características de configuración y de cobertura.

O5. Realizar un *testbench* UVM para la verificación del adaptador de protocolo, integrando los IP de verificación de *Mentor Grap*hics para AXI4 y el IP de verificación para *Wishbone*.

1.3. PETICIONARIO

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

1.4. ESTRUCTURA DEL DOCUMENTO

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

- **Capítulo 1. Introducción.** En este primer capítulo se exponen los antecedentes y necesidades que dan lugar a la realización de este Trabajo Fin de Grado. Además, se indican los objetivos que perdigue este TFG, el peticionario y la estructura del presente documento.
- **Capítulo 2. Interfaces AXI y *Wishbone*.** En este segundo capítulo se profundizará en los conceptos básicos relacionados con los protocolos AXI4 y *Wishbone* debido a que son los protocolos de comunicación que interconectarán el dispositivo a verificar (*Device Under Verification*, DUV) con el entorno de verificación. Se analizarán los factores claves de los protocolos. Para el análisis del protocolo AXI4, se expondrán los canales y las señales asociadas a los mismos, el protocolo de *handshake* y la configuración de las transacciones. Para el análisis del protocolo *Wishbone*, se aportará la especificación de la interfaz y el funcionamiento de los ciclos del bus.
- **Capítulo 3. Metodología UVM.** En este tercer capítulo se analiza en profundidad la metodología UVM y el entorno de verificación que se crea a partir de esta metodología. Se describirán los antecedentes y se detallarán las características principales de esta metodología, además de sus principios de funcionamiento y los componentes que forman un entorno de verificación.
- **Capítulo 4. Diseño a verificar.** Durante este capítulo se analizarán los conceptos claves del dispositivo a verificar durante la realización del presente Trabajo Fin de Grado. Además, se realizará y ejecutará un *testbench ad-hoc* para simular y analizar el comportamiento de este dispositivo.
- **Capítulo 5. Desarrollo del entorno de verificación UVM.** En este quinto capítulo se mostrará y explicará en detalle el entorno de verificación UVM desarrollado para el dispositivo elegido. También se mostrarán los resultados obtenidos de estimular de manera controlada el DUV en este entorno.
- **Capítulo 6. Conclusiones y líneas futuras.** Una vez completados los objetivos establecidos inicialmente, se recogen las conclusiones obtenidas a partir del desarrollo de este Trabajo Fin de Grado. Por último, se analizarán las futuras ampliaciones que puedan surgir a partir de este TFG.

Capítulo 2: INTERFACES AXI Y *WISHBONE*

Este capítulo está dividido en dos apartados importantes para la posterior comprensión del funcionamiento del DUV elegido. Por un lado, se encuentra el protocolo de comunicación AXI4 y, en el otro, el protocolo *Wishbone B4*. Se comentarán las principales características que definen estas interfaces y sus modos de comunicación.

2.1. INTERFAZ AXI4

La interfaz AXI4 (*Advanced eXtensible Interface 4*) forma parte de la cuarta generación de las especificaciones del bus AMBA (*Advanced Microcontroller Bus Architecture*) de ARM (*Advanced RISC Machine*), cuya principal función es la interconexión de bloques funcionales en un *System on a Chip* (SoC).

Es una interfaz paralela de alto rendimiento, síncrona y multimaestro orientada a sistemas de altas prestaciones funcionando a alta frecuencia. Este protocolo es adecuado para diseños de alto ancho de banda y baja latencia, proporciona flexibilidad en la implementación de arquitecturas de interconexión, cumple con los requisitos de interfaz de una amplia gama de componentes y es compatible con versiones anteriores de las interfaces AHB (*Advanced High-performance Bus*) y APB (*Advanced Peripheral Bus*) [7].

2.1.1. CANALES Y SEÑALES

La comunicación entre sistemas que use el protocolo AXI se establece a través de cinco canales independientes. Cada canal cuenta con sus propias señales, exceptuando dos que se repiten en todos ellos, las señales de control `VALID` y `READY`. Debido a que en un extremo de la comunicación hay un sistema maestro y, en el otro, un sistema esclavo, estas señales comunes son

las encargadas de gestionar las comunicaciones, mediante un protocolo de *handshake* entre ellos [8]. A continuación, se ofrece una breve descripción de cada uno de los canales.

- *Read Address channel*. Canal de dirección de lectura que establece la configuración y dirección para una transacción de lectura mediante las señales mostradas en la Tabla 1.

Tabla 1 Señales asociadas al canal Read Address

Señales	Fuente	Descripción
ARID	Maestro	Identificador ID de lectura. Este identificador es la etiqueta de identificación para el grupo de señales de la dirección de lectura.
ARADDR	Maestro	Dirección de lectura. Proporciona la dirección de la primera transferencia en una transacción de lectura en modo *Burst* o ráfaga.
ARLEN	Maestro	Longitud de la ráfaga. Proporciona el número exacto de transferencias en una ráfaga. Esta información determina el número de transferencias de datos asociados con la dirección proporcionada.
ARSIZE	Maestro	Tamaño de la ráfaga. Este bus indica el tamaño de cada transferencia en la ráfaga.
ARBURST	Maestro	Indica el tipo de ráfaga. Junto con la información de tamaño determinan cómo se calcula la dirección para cada transferencia dentro de la ráfaga.
ARVALID	Maestro	Señal `VALID` para las direcciones de lectura. Esta señal indica una dirección de lectura válida en el bus `ARADDR`, así como toda la información de control asociada a la misma.
ARREADY	Esclavo	Señal `READY` para las direcciones de lectura. Esta señal indica que el esclavo puede aceptar una nueva operación de lectura.

- *Read Data channel*. Canal de datos de lectura cuya función es enviar los datos solicitados por el maestro desde el sistema esclavo. Además, incluye una señal que indica el estado de la transferencia. La Tabla 2 recoge las señales de este canal.

Tabla 2 Señales asociadas al canal Read Data

Señales	Fuente	Descripción
RID	Esclavo	Este bus contiene la etiqueta de identificación para la lectura en curso generada por el esclavo. Esta etiqueta es utilizada por el esclavo como identificador de transacciones.
RDATA	Esclavo	Bus de datos de lectura.

RRESP	Esclavo	Este bus indica el estado de la transacción de lectura.
RLAST	Esclavo	Esta señal indica la finalización de la transferencia de datos de lectura en modo ráfaga.
RVALID	Esclavo	Esta señal valida los datos de lectura presentes en el bus `RDATA`, así como la información asociada a la misma.
RREADY	Maestro	Esta señal indica que el maestro puede aceptar nuevos datos de lectura.

- *Write Address channel*. Canal de dirección de escritura en el que se establece la dirección de escritura y la configuración de la transacción, utilizando las señales descritas en la Tabla 3.

Tabla 3 Señales asociadas al canal Write Address

Señales	Fuente	Descripción
AWID	Maestro	Este bus contiene la etiqueta de identificación para el grupo de señales de la dirección de escritura en curso. Esta etiqueta es utilizada por el maestro como identificador de transacciones.
AWADDR	Maestro	Este bus contiene la dirección de la primera transferencia en una transacción de escritura en modo ráfaga.
AWLEN	Maestro	Este bus proporciona el número exacto de transferencias en una ráfaga. Esta información determina el número de transferencias de datos asociadas con la dirección validada.
AWSIZE	Maestro	Este bus indica el tamaño de cada transferencia dentro de la ráfaga.
AWBURST	Maestro	Con este bus se indica el tipo de ráfaga. Junto con la información de tamaño, determina cómo se calcula la dirección para cada transferencia dentro de la ráfaga.
AWVALID	Maestro	Esta señal indica que la dirección presente en `AWADDR` es válida, así como toda la información de control asociada a la misma.
AWREADY	Esclavo	Esta señal indica que el esclavo puede aceptar una nueva dirección.

- *Write Data channel*. Canal de escritura de datos que se encarga de enviar los datos desde el sistema maestro al esclavo, utilizando las señales que se muestran en la Tabla 4.

Tabla 4 Señales asociadas al canal Write Data

Señales	Fuente	Descripción
WDATA	Maestro	Bus de datos de escritura.
WSTRB	Maestro	Señal de validación de cada byte presente en el bus de datos `WDATA`.
WLAST	Maestro	Esta señal indica la finalización de la transferencia de datos de escritura en modo ráfaga (*burst*).
WVALID	Maestro	Esta señal indica que hay datos válidos en el bus `WDATA`.
WREADY	Esclavo	Esta señal indica que el esclavo puede aceptar los datos de escritura.

- *Write Response channel*. Canal de respuesta de escritura que envía el reconocimiento del esclavo al maestro una vez completada la escritura. Las señales que intervienen en este canal se reflejan en la Tabla 5.

Tabla 5 Señales asociadas al canal Write Response

Señales	Fuente	Descripción
BID	Esclavo	Etiqueta de identificación del canal de respuesta. Esta señal identifica la respuesta de la escritura realizada.
BRESP	Esclavo	Esta señal indica el estado de la transacción de escritura.
BVALID	Esclavo	Señal `VALID` de la respuesta enviada. Esta señal valida la respuesta enviada por el bus `BRESP`.
BREADY	Maestro	Señal `READY` del canal de respuesta de escritura. Esta señal indica que el maestro puede aceptar una respuesta de escritura.

La arquitectura de comunicación de estos canales se muestra en la Figura 2. En ella, se ve en qué sentido se transmite la información para cada canal. Primero, en los dos de lectura y, luego, en los tres de escritura.

Figura 2 Arquitectura de comunicación AXI [8]

Cabe destacar que estos canales los forman más señales. Sin embargo, se han mostrado únicamente las más relevantes para el uso del protocolo y aquellas que influyen en el funcionamiento del DUV escogido en este Trabajo Fin de Grado. Para conocer en profundidad el protocolo, se recomienda hacer uso de la bibliografía citada en este documento [7].

2.1.2. Protocolo de *handshake* en AXI4

Como se mencionó anteriormente, los cinco canales de comunicación AXI utilizan las señales `VALID` y `READY` para realizar el protocolo de *handshake* que les permite transferir direcciones, datos y configuración. Este protocolo hace que tanto maestro como esclavo puedan controlar la velocidad de transmisión y recepción de datos, lo que significa que ambos pueden verse como solicitante y solicitado.

El *handshake* funciona de la siguiente manera. Por un lado, el extremo que envía los datos, fuente, activa la señal `VALID` para indicar que la información que transmite está disponible. Por otro lado, el extremo que recibe los datos, destino, activa la señal `READY` cuando esté preparado para recibir la información. La transferencia se produce en el momento en que ambas señales estén a nivel alto a la vez.

La Figura 3 muestra los tres casos en los que se puede dar este protocolo. En T1, se produce el caso en el que la fuente está preparada para enviar la información, pero espera a que el destino autorice la transacción. El caso contrario se produce en T2, cuando el destino está esperando preparado para recibir la información una vez la valide la fuente. Por último, en T3 se encuentra el caso en el que ambas señales se activan a la vez, con una comunicación instantánea.

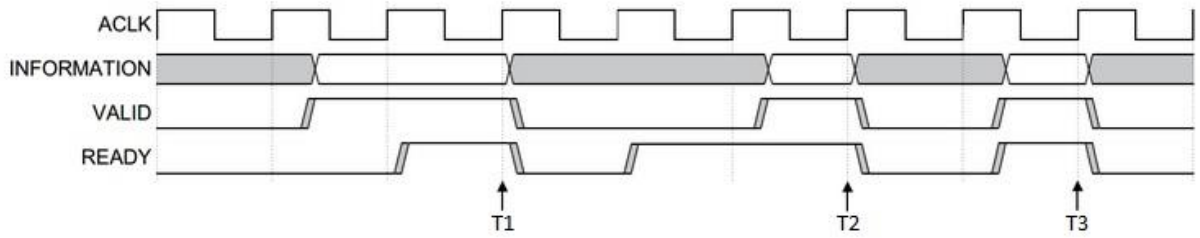

Figura 3 Protocolo de *handshake* AXI4[8]

Un aspecto importante dentro de la especificación AXI es que la señal `VALID` de un componente no debe depender nunca de la señal `READY` de otro. `READY` puede esperar por un `VALID`, pero no está obligado a hacerlo. Estas reglas evitan que ocurran bloqueos. Si ambas señales dependen la una de la otra, es normal ver que ninguna suele estar activa, ya que se están esperando entre ellas.

2.1.3. Configuración y Respuesta de las transacciones

Desde el punto de vista de la verificación, unos de los aspectos más importantes del protocolo AXI es la caracterización de las transacciones. En este TFG se contemplará la configuración mediante los parámetros `BURST`, `SIZE`, `LEN` y `ID`. Estos parámetros se establecen durante la fase de direcciones, tanto en las transacciones de lectura como de escritura. El uso de estos parámetros permite verificar que la transacción llega al destino manteniendo los mismos valores. Estos parámetros han sido definidos de forma detallada en la Tabla 1 y la Tabla 3.

En los siguientes apartados se presentarán los distintos valores que pueden adoptar estos parámetros. Cabe destacar que, como estos parámetros se ajustan en los canales de direcciones (*address*), se utilizará la nomenclatura *AxSIGNAL.* De esta manera, se engloba la señal tanto para escritura como para lectura.

Además, se profundizará en el bus `RESP` tanto de lectura como de escritura, definido en la Tabla 2 y Tabla 5. De esta manera se mostrará los distintos estados de la transacción que puede indicar el canal de respuesta.

2.1.3.1. Parámetro BURST

El protocolo AXI define tres tipos de ráfagas (*burst*). La Tabla 6 muestra la codificación para cada tipo de *burst* de la señal presentada como *AxBURST*, explicados a continuación.

Tabla 6 Codificación del parámetro AxBURST

AxBURST[1:0]	Tipos de *burst*
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reservado

- **FIXED:** Ráfaga fija. La dirección es la misma para todas las transferencias de la ráfaga. Además, los bits de validación de cada byte son constantes para todos los pulsos de la ráfaga. Sin embargo, los bytes reales que tiene `WSTRB` activados pueden diferir para cada escritura de la ráfaga. Este tipo de ráfaga se utiliza para accesos repetidos a la misma ubicación, como ocurre al cargar o vaciar una FIFO.

- **INCR:** Ráfaga incremental. En una ráfaga incremental, la dirección de cada transferencia en la ráfaga es un incremento de la dirección de la transferencia anterior. El valor de incremento depende del tamaño de la transferencia. A modo de ejemplo, la dirección para cada transferencia en una ráfaga con un tamaño de cuatro bytes es la dirección anterior más cuatro. Este tipo de ráfaga se utiliza para acceder a la memoria secuencial normal.

- **WRAP:** Ráfaga de envoltura. Este tipo de ráfaga tiene cierto parecido con la ráfaga incremental anteriormente explicada, pero cuenta con una diferencia clave: la dirección se envuelve a una dirección más baja si se alcanza un límite de dirección superior. Tiene ciertas restricciones, como que la dirección de comienzo debe estar alineada con el tamaño (SIZE) de cada transferencia y que la longitud (LEN) de la ráfaga debe de ser de 2, 4, 8 o 16 transferencias. Este tipo de ráfaga se utiliza, por ejemplo, para accesos a la línea de datos de una memoria caché.

El direccionamiento de los datos para este protocolo se realiza por *bytes*. Esto significa que en los modos *incremental* y *wrapped*, si se parte de una dirección `0x000` en hexadecimal, las siguientes serán `0x004`, `0x008`, `0x00C` y así sucesivamente, en el caso de transferencias de 32 bits.

2.1.3.2. PARÁMETRO `SIZE`

El parámetro `SIZE` especifica el número máximo de bytes que se pueden enviar por transferencia, tanto por el canal de escritura como por el de lectura. En la Tabla 7 se indica la forma de codificar los campos de configuración de este parámetro según las necesidades del sistema. El tamaño de cualquier transferencia no debe superar el ancho del bus de datos de ninguno de los agentes que intervienen en dicha transacción.

Tabla 7 Codificación del parámetro SIZE

AxSIZE[2:0]	Bytes por transferencia
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

2.1.3.3. PARÁMETRO LEN

El parámetro LEN marca el número exacto de transferencias en una ráfaga. El protocolo AXI3 sólo soporta una longitud de 1 a 16 transferencias para todos los tipos de ráfagas. En cambio, AXI4 amplía este límite de manera significativa para el modo incremental de la ráfaga, dejando enviar de 1 a 256 transferencias, aunque en el resto de los modos mantiene el límite marcado por el protocolo AXI3. La longitud de la ráfaga, *N_Transferencias*, se define como:

***N_Transferencias* = *AxLEN* [7:0] + 1**

2.1.3.4. PARÁMETRO ID

El parámetro ID se utiliza para indicar el identificador de la transacción. De esta manera, si un maestro está compuesto por varios sub-bloques, este identificador podría indicar cuál de ellos inició la transferencia. El protocolo AXI4 no tiene restricciones de orden entre las transacciones de lectura y escritura, es decir, pueden completarse en cualquier orden, incluso si el valor de la señal *ARID* (propio de una lectura) es el mismo que el valor de la señal *AWID* (utilizado en una escritura).

La anchura de los campos del parámetro *ID* de las transacciones está definida en la especificación del protocolo AXI4. Sin embargo, se recomienda los siguientes anchos de campo *ID* de transacción:

- Los componentes maestros se deben configurar con un campo *ID* de hasta cuatro bits.
- Para los números de puerto maestro en la interconexión se añaden hasta cuatro bits adicionales al parámetro *ID*.
- Con los componentes esclavos se deben implementar ocho bits de soporte en el campo *ID* de transacción.

Si el maestro solo admite una interfaz ordenada, es aceptable la vinculación de un valor constante al campo *ID* de la transacción. En el caso de los esclavos que no usan la información de ordenación y procesan todas las transacciones en orden, se puede añadir la funcionalidad del parámetro *ID* sin necesidad de cambiar la funcionalidad básica del esclavo.

2.1.3.5. CANAL DE RESPUESTA

El protocolo AXI proporciona una señalización de respuesta tanto para transacciones de lectura como para transacciones de escritura. Para la lectura, la información de respuesta se envía desde el sistema esclavo en el *Read Data channel*, mediante la señal `RRESP`. En una transacción de escritura el esclavo se comunica a través de la señal `BRESP` del *Write Response channel*. En la Tabla 8 se muestra la codificación de las señales de respuesta. A continuación, se explicarán los distintos tipos de respuesta del protocolo.

Tabla 8 Codificación de las señales de respuesta

RRESP[1:0] BRESP[1:0]	**Respuesta**
0b00	OKAY
0b01	EXOKAY
0b10	SLVERR
0b11	DECERR

- **OKAY:** Acceso normal con éxito. Este valor indica una de las siguientes opciones: el éxito de un acceso normal, el fallo en un acceso exclusivo o un acceso exclusivo en un esclavo que no permite este acceso. OKAY es la respuesta para la mayoría de las transacciones.
- **EXOKAY:** Acceso exclusivo con éxito.
- **SLVERR:** Transacción fallida. Para simplificar las operaciones de monitorización del sistema, se recomienda que la respuesta de error se use solo en condiciones de fallo del esclavo. Estas condiciones pueden ser rebasar un *buffer*, tamaño de transacción inválido, escritura en direcciones de solo lectura o condición de tiempo agotado en el esclavo, entre otras.
- **DECERR:** Fallo en la conexión. Indica que la interconexión no puede decodificar con éxito un acceso al esclavo.

En una transacción de escritura, una sola respuesta sirve para la ráfaga completa y no para cada dato transferido dentro de la ráfaga. En el caso de una transacción de lectura, el esclavo puede enviar distintas respuestas en una ráfaga. Por ejemplo, en una ráfaga de 16 transferencias, el esclavo puede responder un OKAY para 15 de ellas y un SLVERR para una de ellas.

El protocolo especifica que se debe actuar frente a todas las transferencias de datos, incluso si ocurre un error. El resto de la ráfaga no debe cancelarse si el esclavo da una sola respuesta de error.

2.2. INTERFAZ *WISHBONE B4*

La interfaz *Wishbone* es una metodología de diseño flexible creada por *Silicore Corporation* utilizada en núcleos IP de semiconductores, sobre todo en proyectos de *OpenCores*. Su propósito es fomentar la reutilización del diseño al aliviar los problemas de integración de SoC. Esto se logra

mediante la creación de una interfaz común entre los núcleos de IP y mejora la portabilidad y la confiabilidad del sistema [9].

Existen varias versiones del protocolo, siendo las más destacables las versiones *B3* y *B4*. La versión *B3* fue creada en el año 2002 y permite comunicaciones básicas del protocolo. La versión *B4* es la más nueva y evolución natural de la anterior y fue desarrollada en 2010. Esta versión permite ciclos de comunicación avanzados y más eficientes que su predecesora. Por esta razón y por su uso en el DUV, en este TFG se va a trabajar con la versión *Wishbone B4*.

El protocolo no tiene canales separados como AXI4. Sin embargo, desde un punto de vista general, el bus *Wishbone* tiene los dos canales de comunicación que se muestran en la Figura 4. Por un lado, las transacciones de escritura y lectura se envían por el canal de peticiones (*request*). Por el otro, el canal de respuesta (*response*) devuelve el reconocimiento de la transacción. Además, el canal *request* puede ser parado por el esclavo activando la señal *Request stall.* Esto no se aplica al canal *response* [10].

Figura 4 Canales generales del protocolo *Wishbone versión B4*[10]

2.2.1. ESPECIFICACIÓN DE LA INTERFAZ *WISHBONE B4*

Para profundizar más en el protocolo se ha elegido explicar *Wishbone* desde la perspectiva del maestro. Esto influye en el sentido de las señales, si son salidas (*O*) o entradas (*I*). La Tabla 9 muestra las señales más relevantes para el uso del protocolo y aquellas que influyen en el funcionamiento del DUV escogido en este Trabajo Fin de Grado. Para conocer en profundidad el protocolo, se recomienda hacer uso de la bibliografía citada en este documento [9].

Tabla 9 Señales *Wishbone B4*

Señales	Fuente	Descripción
CYC_O	Maestro	Señal de ciclo. Cuando está activa, indica que hay un ciclo de comunicación válido en progreso. La señal se mantendrá activa el tiempo suficiente para cubrir todas las transacciones del ciclo. Por ejemplo, durante un ciclo de transferencia pueden enviarse varios datos distintos. Esta señal se activa desde el primer dato transferido y se mantiene a nivel alto hasta que el último termina la transferencia.

STB_O	Maestro	La señal de *strobe* indica un ciclo de transferencia con datos válidos. Se usa para validar otras señales como SEL_O. El sistema esclavo activará las señales ACK_I o ERR_I en respuesta a cada activación de esta señal.
WE_O	Maestro	Señal *Write Enable* que indica si el ciclo actual es de lectura o escritura. La señal estará a nivel bajo para lecturas y a nivel alto para escrituras.
ADDR_O	Maestro	Esta señal se usará para enviar la dirección donde realizar la lectura o escritura.
DATA_O	Maestro	Bus de datos de escritura. Esta señal solo funciona para datos de escritura debido a que es una salida.
SEL_O	Maestro	Señal de selección que indica dónde se espera que estén los datos válidos en un ciclo de lectura, y dónde se encuentran los datos válidos de un ciclo de escritura. Cada selección se corresponde con un byte del bus de datos.
STALL_I	Esclavo	Esta señal indica que el sistema esclavo no puede aceptar más transacciones mientras esté activa. Esto conlleva que el maestro se mantenga en espera hasta que el esclavo la desactive.
ACK_I	Esclavo	Señal de reconocimiento. Cuando se activa, indica la finalización de un ciclo normal del bus.
DATA_I	Esclavo	Bus de datos de lectura. Al contrario que DATA_O, esta señal solo funciona para datos de lectura debido a que es una entrada.
ERR_I	Esclavo	Esta señal indica una finalización anormal del ciclo.

2.2.2. CICLOS DE BUS DE LA INTERFAZ WISHBONE

Una vez vistas las señales del protocolo, en este apartado se mostrarán los ciclos de comunicación más importantes y la diferencia entre los distintos tipos de ciclos. Antes de empezar con la explicación y para entender mejor la relación con el DUV seleccionado en este TFG, se realizará un cambio en la nomenclatura de las señales respecto a la especificación de este. La Tabla 10 muestra los nuevos términos de las señales. Si bien la nomenclatura cambia, la nueva nomenclatura mantiene intacto el significado de cada una de las señales, siendo muy sencilla su interpretación.

Tabla 10 Cambio de nomenclatura señales *Wishbone*

Término Especificación	Nueva nomenclatura
CYC_O	o_wb_cyc
STB_O	o_wb_stb
WE_O	o_wb_we
ADDR_O	o_wb_addr
DATA_O	o_wb_data
SEL_O	o_wb_sel
STALL_I	i_wb_stall
ACK_I	i_wb_ack
DATA_I	i_wb_data
ERR_I	i_wb_err

2.2.2.1. CICLO ESTÁNDAR DEL PROTOCOLO *WISHBONE*

La especificación *Wishbone* define dos tipos de interacción. El primero es conocido como *Wishbone classic* y está presente en las versiones *B3* y *B4* del protocolo. En este modo clásico, el maestro debe esperar hasta que el esclavo active la señal de reconocimiento (*ACK*) antes de volver a enviar información por el canal de *request*. En la Figura 5 se muestra el funcionamiento de esta versión. Se observa la falta de señales como *stall* o *strobe* en este modo, ya que se utiliza la falta de *ACK* de parte del sistema esclavo como método de espera.

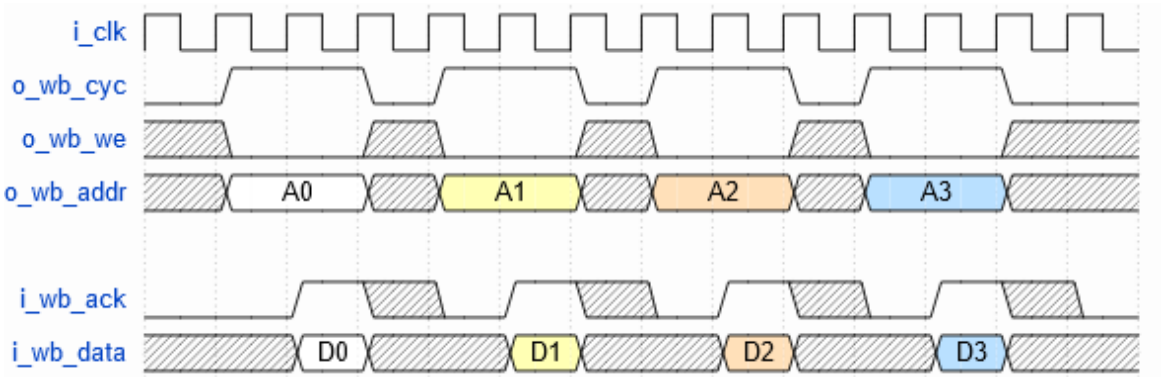

Figura 5 *Wishbone classic*

Una consecuencia de usar la implementación *classic* es que se necesita un mínimo de tres ciclos de reloj por transacción, uno para activar el *request*, otro para devolver el *response* y un último ciclo para que el maestro reciba la respuesta y acabe la transacción. Este comportamiento es un problema a la hora de añadir componentes síncronos al circuito, ya que retrasaría la transacción y ralentizaría no solo el bus, sino la velocidad de reloj en general.

2.2.2.2. CICLO *PIPELINED* DEL PROTOCOLO *WISHBONE*

Como solución al problema anterior, aparece el segundo modo de comunicación. Se conoce como *Wishbone pipelined* y está presente en la versión *B4* del protocolo. Este modo *pipelined* actúa de manera distinta al clásico. Su principal diferencia es que es capaz de mandar varios *request* sin tener que esperar a que el esclavo mande el `ACK` de cada uno. El funcionamiento se muestra en la Figura 6. En esta figura, se puede observar el uso de las señales `stall` y `strobe`. La señal `stall` es necesaria para que el sistema esclavo detenga el envío de información por el canal *request*, ya que, en este modo, `ACK` no bloquea la transacción. La señal `strobe` es necesaria para que el esclavo conozca la cantidad de `ACK` que debe responder, siendo uno por cada ciclo de reloj que `strobe` esté activa. Además, mientras `strobe` y `we` tienen relevancia únicamente durante el envío de datos, el ciclo, analizado por la señal `cyc`, no termina hasta recibir el último reconocimiento por parte del sistema esclavo.

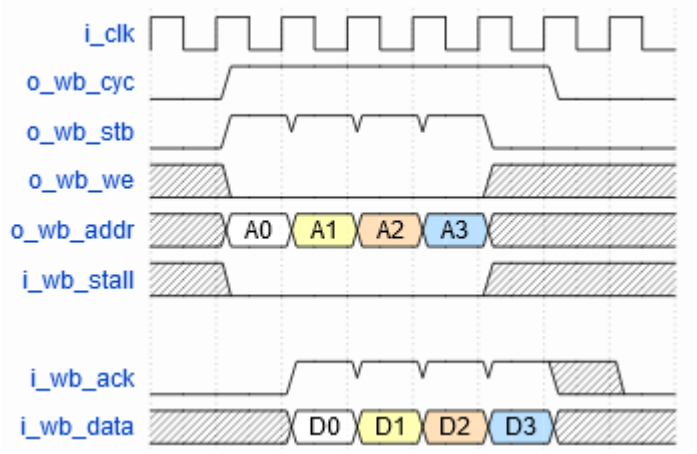

Figura 6 *Wishbone pipelined*

El direccionamiento de datos en este protocolo es diferente al anterior. En vez de direccionar al *byte*, se direcciona a la palabra. Esto representa que, si se parte de la misma dirección, `0x000`, las siguientes direcciones serán `0x001`, `0x002`, `0x003` y así sucesivamente.

La Figura 7 muestra la condición de espera por nivel alto de la señal de `stall`. En caso de estar activa, los canales se mantienen con su valor previo el tiempo necesario hasta que `stall` pase a nivel bajo, donde continúan con la transacción. El número de ciclos consecutivos en los que esta señal puede estar activa es configurable.

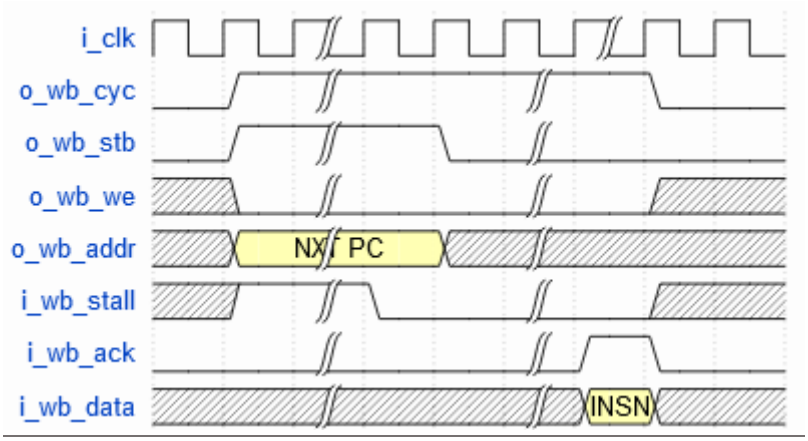

Figura 7 *Wishbone* pipelined con ciclo de `stall`

Este modo permite una comunicación más rápida y eficiente. Si utilizamos `N` como una variable numérica, el modo clásico puede aspira a una comunicación de `N` transacciones en `3*N` ciclos de reloj. Sin embargo, el modo *pipelined* permite alcanzar una velocidad de `N` transacciones en `N+1` ciclos de reloj. Cabe destacar que el DUV de este TFG utiliza una interfaz *Wishbone pipelined*.

Capítulo 3: METODOLOGÍA UVM

En este capítulo se presentan las características principales de la metodología UVM (*Universal Verification Methodology*), utilizada en este TFG. En primer lugar, se hará una introducción destacando las características principales de la metodología UVM. Tras esto, se realizará una descripción propia de la metodología. Se comentarán los componentes que forman un entorno de verificación funcional UVM, indicando su funcionamiento, y el lenguaje de verificación hardware que se utiliza. Además, se explicarán ciertas características particulares de algunos componentes esenciales para el desarrollo de este TFG. Con estos aspectos, se dispondrán de los conocimientos necesarios para entender los beneficios del uso de UVM frente a otras metodologías de verificación, además de comprender el motivo de su creación y de su relevancia en la actualidad.

3.1. INTRODUCCIÓN A LA METODOLOGÍA UVM

UVM es un estándar del consorcio *Accellera System Initiative* que fue desarrollado mediante el trabajo cooperativo de los principales fabricantes y usuarios de herramientas EDA (*Electronic Design Automation*) y, por esto, está soportado por los principales simuladores existentes en el mercado. Cuenta con la ventaja de que sus librerías son de código abierto.

En el proceso de verificación de sistemas electrónicos adquiere especial relevancia llevar un flujo de trabajo común, ya que es poco productivo que cada diseñador o ingeniero de verificación perteneciente a un equipo, afronte la verificación de un bloque de manera diferente. Esto llevaría a que cada *testbench*, a nivel de bloques, fuese distinto de los demás. Esta metodología de verificación soluciona este problema gracias al uso de mismas plantillas para la verificación de distintos módulos.

La metodología UVM se creó sobre la sólida base existente gracias a la metodología OVM (*Open Verification Methodology*), desarrollado por *Mentor Graph*ics, *Siemens Company* y *Cadence*

Design Systems. Además, tiene características propias de AVM (Ad*vanced Verification Methodology*), creada también por *Mentor Graphics*.

A partir de esta base y con continuos avances en la verificación, UVM se ha convertido en una metodología potente y flexible con la que es posible implementar entornos de verificación reusables, escalables e interoperables. Sus características principales se listan a continuación.

- Como se ha comentado, cabe destacar la reutilización y el modularidad de la metodología UVM, ya que el entorno de verificación está organizado en distintos niveles de abstracción y está diseñado con diferentes componentes: *UVM Environment*, *UVM Agent*, *UVM Driver, UVM Sequencer*, *UVM Monitor*, entre otros.
- El *testbench* se mantiene separado de la jerarquía real del entorno de verificación, lo cual permite la reutilización de los estímulos en diferentes unidades o proyectos.
- Debido a la inherente popularidad de UVM, la biblioteca de clases básicas es compatible con la mayoría de los simuladores del mercado. Por lo tanto, no existe dependencia a un simulador específico.
- A través de la metodología y una biblioteca de código, ofrece la posibilidad de dividir de manera limpia el entorno de verificación en un conjunto de ítems de datos (estímulos y respuestas) y componentes de verificación (*Universal Verification Component*, UVC). Con esto se consigue estructurar y organizar los objetos y las funcionalidades del entorno de verificación de forma más sencilla, así como incrementar su reusabilidad.
- Esta metodología cuenta con un mecanismo de fábrica de componentes y objetos que simplifica su uso y modificación.
- Mediante un sistema de mensajes e informes de errores, facilita el proceso de análisis y depuración.
- Un factor clave es la flexibilidad que proporciona esta metodología para la generación de estímulos en forma de secuencias, ya que ofrece gran control y capacidad al ingeniero de verificación. Por lo tanto, permite el minucioso control de los flujos de datos que son enviados al DUV como estímulos.

Todas estas características han hecho que UVM se haya consolidado como la metodología de verificación ASIC/IC (*Application-Specific Integrated Circuit*) y FPGA (*Field-Programmable Gate Array*) más utilizada en la actualidad, como muestra la Figura 8.

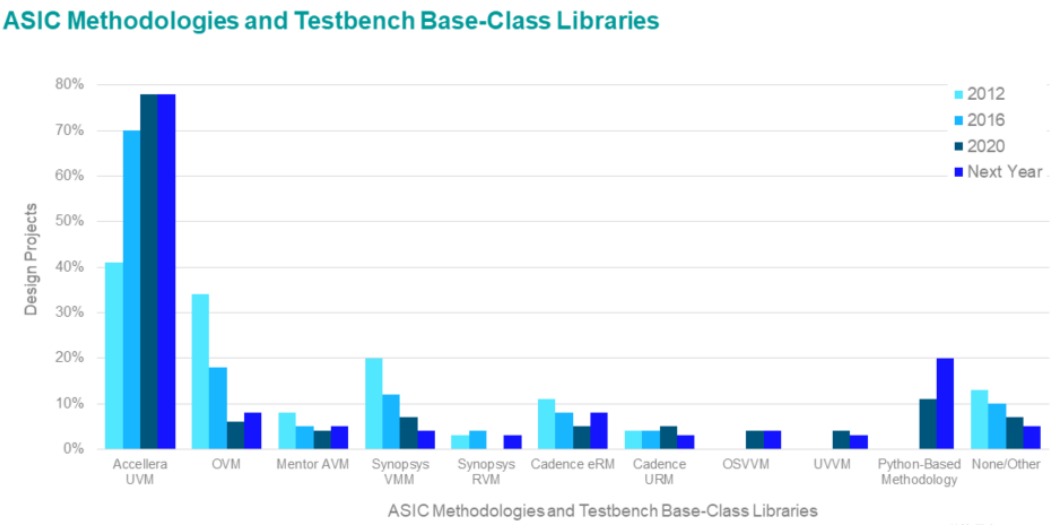

Figura 8 Uso de metodologías de verificación ASIC/IC [4]

3.2. LENGUAJE DE DESCRIPCIÓN HARDWARE *SYSTEMVERILOG*

SystemVerilog es un estándar universal para la descripción hardware versátil, pues permite su uso para procesos de verificación, además de para diseño de sistemas electrónicos. Este lenguaje proviene del estándar *Verilog-2001*, pero combina conceptos de otros lenguajes como VHDL, C y C++. Su funcionalidad abarca tanto la de lenguaje de descripción hardware (*Hardware Description Language* - HDL) como de la de lenguaje de verificación hardware (*Hardware Verification Language* - HVL)[11]. La Figura 9 refleja estas funciones. La metodología UVM va de la mano del lenguaje *SystemVerlilog*.

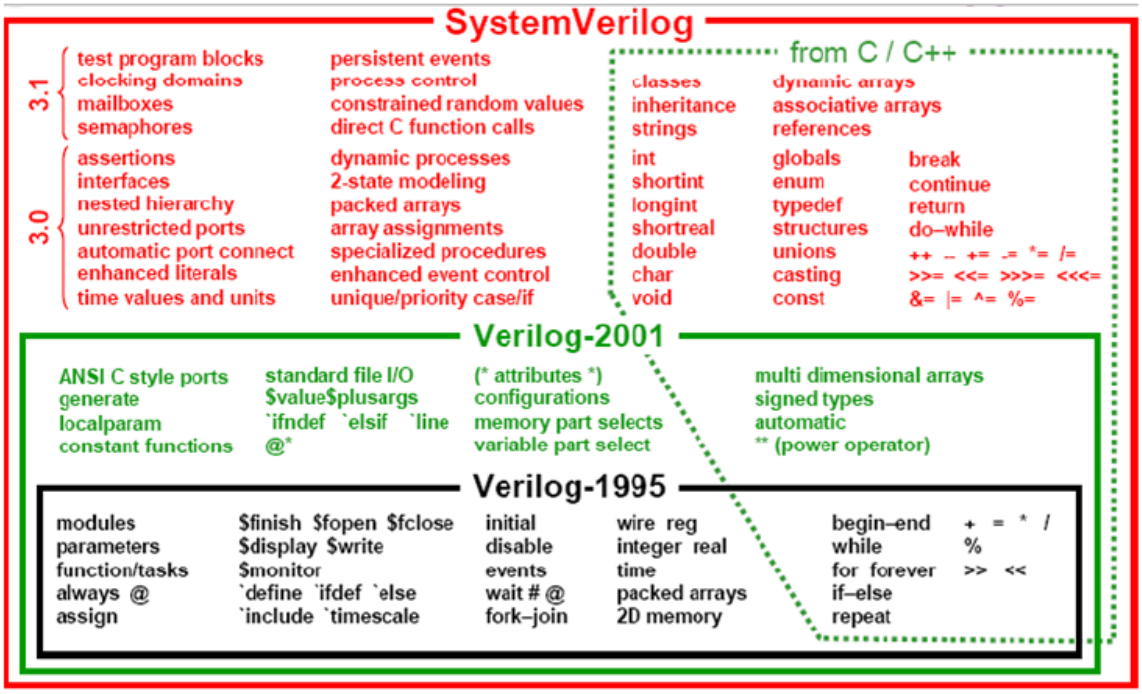

Figura 9 Funcionalidades *SystemVerilog*[11]

Como evoluciona del lenguaje de descripción hardware *Verilog*, *SystemVerilog* mantiene las características básicas para la implementación de sistemas electrónicos. Sin embargo, el principal propósito de este lenguaje está ligado a la verificación de estos diseños. Las razones más destacadas son las siguientes:

- El uso de clases, que permite utilizar técnicas de la programación orientada a objetos (*Object-Oriented Programming* - OOP). Esta característica es muy importante para el desarrollo de entornos de verificación de mayor complejidad. Una clase es un conjunto de variables y rutinas que definen el comportamiento del objeto que se va a referenciar. En *SystemVerilog*, estos objetos son elementos dinámicos que permiten el modelo de herencia simple y que pueden ser parametrizados (función básica de C++). Resulta muy útil el concepto de herencia de clases, ya que permite la reutilización del código. De esta manera, una subclase que hereda de cierta clase solo tendrá que implementar algunos métodos adicionales.
- El uso de paquetes (*packages*) para compartir código entre distintos módulos. Un paquete incluye declaraciones y definiciones que se agrupan bajo un nombre común, el del propio paquete. Con estas declaraciones se pueden incluir tipos, constantes, funciones, tareas, clases, etc.
- Permite el uso de *assertions* (propiedades) y de mecanismos para realizar de forma automática las medidas de cobertura. El concepto de *assertions* se relaciona con aquellas sentencias booleanas colocadas en un punto específico del *testbench* y se comprueba durante la fase de simulación. Además, soporta la verificación ABV (*Assertion-Based Verification*)[12] y define un mecanismo de cobertura y de adquisición de datos de manera flexible.
- La utilización de diferentes tipos de datos, tanto estáticos como dinámicos.

- Una característica clave es la posibilidad de aleatorizar los datos, lo cual es sumamente importante para la metodología UVM cuando se estimula el DUV. Además, esta aleatorización se puede realizar teniendo en cuenta algunas restricciones.
- Para la comunicación de distintos elementos permite el uso de interfaces. Otro factor importante para UVM es que soporta las comunicaciones TLM, que también permite la reutilización de los entornos de verificación en distintos niveles de abstracción.
- La interfaz DPI (*Direct Programming Interface*), que permite referenciar funciones descritas en C de forma directa en el código *SystemVerilog*.

Por estas características, entre otras, *SystemVerliog* se ha consolidado en el mercado actual como lenguaje de verificación hardware por excelencia. La Figura 10 muestra la evolución de los últimos años del uso de los distintos lenguajes de verificación para diseños ASIC, aunque es válida también para diseños con FPGA.

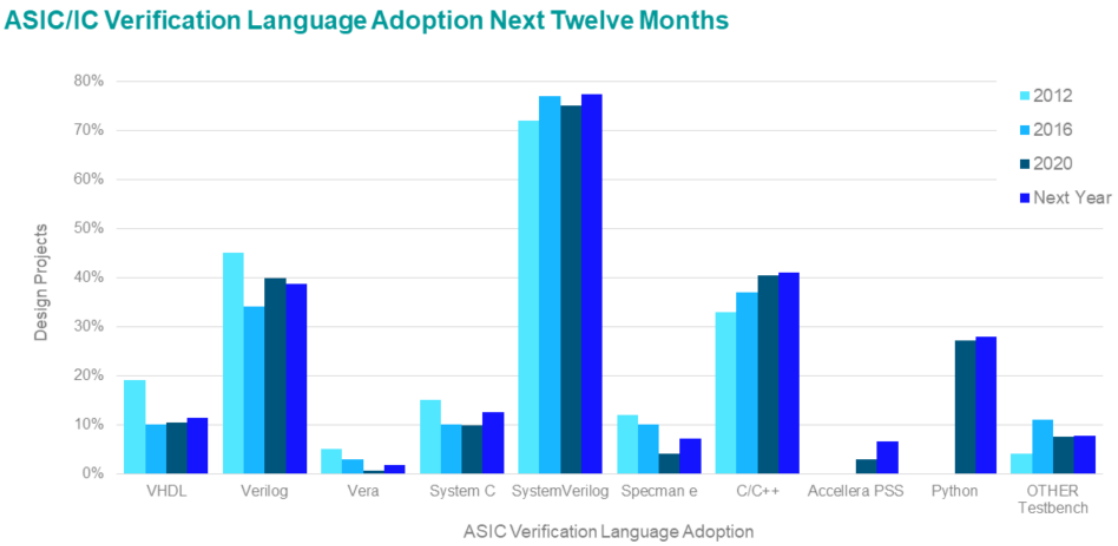

Figura 10 Evolución del uso de lenguajes de verificación *hardware*[4]

3.3. Conceptos generales de la metodología UVM

En lo que resta del presente capítulo se profundizará en los aspectos más relevantes de esta metodología y se comentarán algunas características del lenguaje de descripción hardware que se utiliza para el desarrollo de cualquier proyecto basado en UVM.

3.3.1. Biblioteca de clases UVM

Para construir un entorno de verificación basado en UVM son necesarios una serie de elementos activos, como los componentes (*drivers*, *sequencers*, *monitors*, etc.) y otros elementos pasivos, como las transacciones (objetos de clase que contienen datos reales a enviar al sistema). Todos estos elementos se crean a partir de un conjunto de clases básicas propias de la metodología UVM [13]. Estos componentes pueden encapsularse e instanciarse jerárquicamente, dando lugar a un *testbench*, y son controlados a través de un conjunto de fases (*phases*) para inicializar, ejecutar

y completar cada uno de los test. A continuación, se van a explicar los tres grupos principales de la biblioteca de clases, que se muestran en la Figura 11.

- **`uvm_object:`** deriva de la clase base `uvm_void` y actúa como clase padre en la jerarquía de clases UVM. Esto significa que el resto de clases heredan sus métodos y propiedades. Su principal función es proporcionar al resto de clases de métodos para realizar operaciones comunes y básicas para ellos.
- **`uvm_component:`** es la clase padre de todos los componentes que integran el entorno de verificación UVM. Además de heredar las características de `uvm_object`, esta clase incorpora nuevos métodos y la clase *Factory*. *Factory* permite crear nuevos componentes y otros objetos basados en una configuración especifica. También, se encarga de definir las fases que crean, conectan y caracteriza a los distintos componentes del entorno
- **`uvm_sequence` y `uvm_sequence_item`:** Son las clases que generan estímulos y agrupan las respuestas del DUV que se desea verificar.

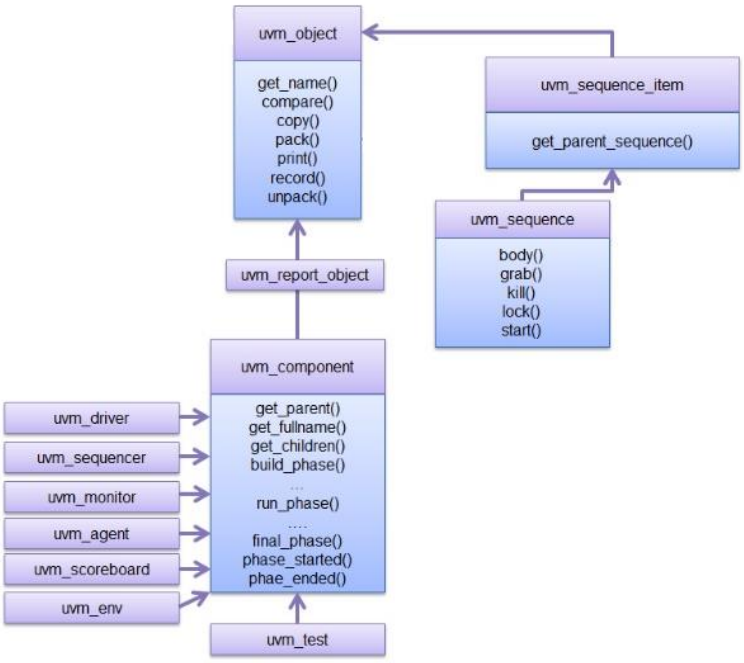

Figura 11 Jerarquía parcial de la biblioteca de clases de UVM [13]

Los objetos pertenecientes a las subclases que derivan de `uvm_component` son estáticos y forman parte de la jerarquía del *testbench* durante toda la simulación al ser creados durante las fases iniciales. Los objetos que derivan de la clase `uvm_sequence` y la clase `uvm_sequence_item` son dinámicos que se crean, utilizan y eliminan en tiempo de ejecución.

3.3.2. Mecanismo de fases en UVM

Una simulación dentro de un entorno UVM consiste en la ejecución en secuencia de una serie de fases. Antes de la ejecución de un test, se debe asegurar que el entorno de verificación se ha creado en su totalidad. Además, deben estar creados todos los elementos y componentes del

sistema para después realizar el correcto conexionado entre ellos. Por esta razón, UVM define una serie de fases que actúan como mecanismo de sincronización en la ejecución de un *testbench* [14]. La Figura 12 muestra el orden de ejecución secuencial de las distintas fases.

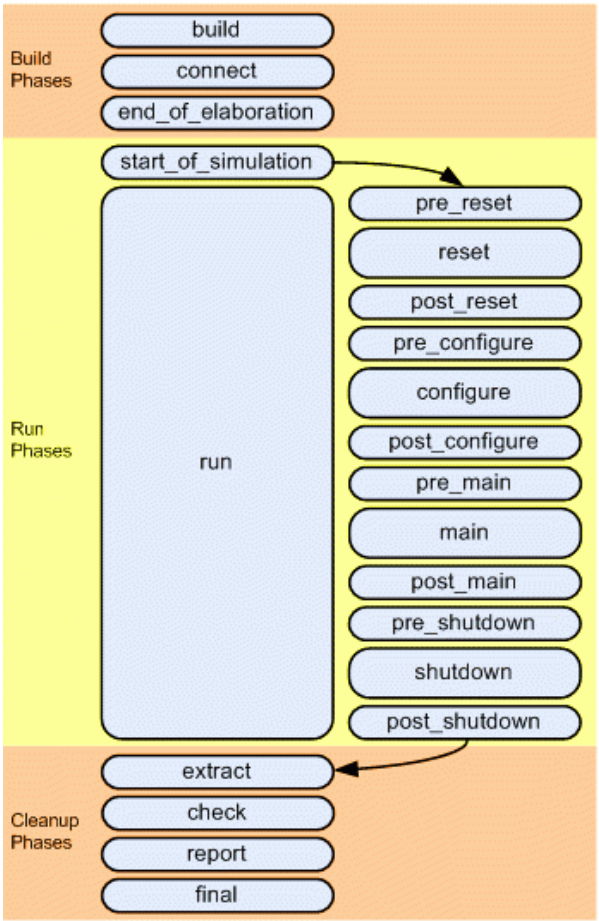

Figura 12 Fases UVM y orden de ejecución[14]

En un entorno de verificación estándar, las fases más relevantes y utilizadas son las siguientes:

- **`build_phase()`.** Esta es la primera fase del flujo de ejecución. Consiste en la creación de los componentes UVM y su jerarquía, siguiendo una filosofía *top-down*, es decir, la jerarquía se construirá desde el nivel más alto al más bajo.
- **`connect_phase()`.** La función de esta fase es realizar el conexionado de los componentes creados en la `build_phase()`. En este caso, se sigue una filosofía *bottom-up*, donde se conectarán primero los componentes de más bajo nivel hasta llegar al nivel más alto de la jerarquía.
- **`end_of_elaboration_phase()/start_of_simulation_phase()`.** Estas fases permiten mostrar en pantalla cierta información de relevancia tras la creación y conexionado de los componentes, como puede ser la topología final de la jerarquía del entorno o información de configuración.
- **`run_phase()`.** En esta fase se describe el comportamiento de los distintos componentes que constituyen el entorno de verificación. Dichos componentes implementan esta fase

activando tareas que se ejecutan de forma paralela en el tiempo, haciendo uso de una estructura *fork-join*. Esta estructura permite emular la ejecución paralela de las tareas llamadas dentro de la misma. Como se puede observar en la Figura 12, esta fase está compuesta por múltiples subfases. Para el desarrollo de un *test* de complejidad baja o media, muchas de esas subfases no resultan de utilidad, por lo que el código correspondiente se escribe directamente en la fase principal `run_phase()`. Es la única fase que consume tiempo de ejecución, lo que implica que se deben incluir todos los eventos que se quieran ejecutar durante el tiempo de simulación. Además, es necesario determinar la finalización de esta fase. Para ello, esta fase define tres métodos diferentes:

  - **`raise_objection()`.** Señaliza el inicio de la ejecución de la fase `run_phase()` en un componente.
  - **`drop_objection()`.** Señaliza el momento en el que la fase `run_phase()` finaliza en un componente. Cuando se invoca este método en cualquier componente que haya utilizado previamente el método `raise_objection()`, se da por terminada la simulación en el componente y se prepara la ejecución de la fase de procesamiento. Una vez terminada la simulación en todos los componentes, el sistema comienza la fase de procesamiento.
  - **`set_drain_time()`.** Este método establece un intervalo de tiempo a esperar una vez finalizada la simulación.

- **`Extract_phase()/check_phase()/report_phase()`.** Estas fases se pueden utilizar para recopilar información de cobertura y resultados de la simulación. Esto permite determinar el estado del sistema y depurar los resultados obtenidos.

3.3.3. Mecanismo de Factory

La metodología UVM utiliza el mecanismo *Factory* para registrar los objetos y componentes del entorno por tipos. Dichos objetos y componentes deben estar registrados en la *Factory* y se crearán a partir de ahí, posibilitando jerarquías de componentes configurables. Además, el mecanismo *Factory* permite la sustitución de objetos existentes por cualquier objeto heredable de la clase superior sin necesidad de modificar el código. Esto permite una mejora de flexibilidad del *testbench*.

La operación de registrar objetos y componentes en la *Factory* de UVM puede realizarse de diferentes formas. La manera más común es mediante el uso de dos macros de UVM específicas para esta tarea. Estas macros son `` `uvm_object_utils `` y `` `uvm_component_utils ``, a las que se les pasará como parámetro el nombre del objeto o componente que se quiera registrar. En caso de que el objeto o componente a registrar sea parametrizable, se utilizarán las macros `` `uvm_object_param_utils `` y `` `uvm_component_param_utils ``, respectivamente.

Estas macros implementan, primero, la función `get_type_name()`, que devolverá el objeto o componente como una variable de tipo *string*. Tras esto, ejecutan la función `create()` como constructor y después registran el objeto o componente en la *Factory* utilizando la variable tipo *string* generada al comienzo de la macro.

3.3.4. Mecanismo de mensajes

Un aspecto necesario de un entorno de verificación consiste en hacer uso de un mecanismo de mensajes. Cuando se ejecuta la simulación de un entorno, se reportan mensajes informativos o de errores que son de gran utilidad para la depuración del sistema. La metodología UVM aporta un conjunto de macros que permiten la notificación de estos mensajes. Estas macros son las siguientes:

- `` `uvm_info (string ID, string MESSAGE, int VERBOSITY). ``
- `` `uvm_warning (string ID, string MESSAGE). ``
- `` `uvm_error (string ID, string MESSAGE). ``
- `` `uvm_fatal (string ID, string MESSAGE). ``

El campo *ID* es una etiqueta que se proporciona para identificar el mensaje en el registro. El segundo argumento, *MESSAGE*, es una variable de tipo *string* con el mensaje que se desea imprimir en pantalla. El argumento *VERBOSITY* indica el nivel de filtro de información que se desea imprimir, que cambiará la cantidad de información que se va a mostrar. Los niveles de *VERBOSITY* en orden de menor a mayor nivel de filtrado son `UVM_NONE`, `UVM_LOW`, `UVM_MEDIUM`, `UVM_HIGH`, `UVM_FULL` y `UVM_DEBUG`. La única macro que permite este argumento es `` `uvm_info ``, aunque si no se rellena, se le asignará un valor por defecto. El resto de macros poseen un nivel de filtrado con el valor `UVM_NONE` que no se puede modificar por el usuario.

3.3.5. Modelado y comunicación TLM

La metodología UVM aporta un conjunto de interfaces y canales de comunicación TLM (*Transaction-Level Modeling*), que permiten la conexión a nivel de transacciones de distintos componentes en el entorno de verificación. Esta comunicación a nivel de transacciones es necesaria en diseños complejos que requieran un entorno de verificación complejo con una estimulación más elaborada. Por esta razón, se necesita un nivel de abstracción muy elevado. Para ello, se habilita la comunicación TLM que permite la comunicación entre componentes que se encuentran a distintos niveles de abstracción y que implementan una misma interfaz.

Con la finalidad de enviar y recibir transacciones, hay disponibles puertos de comunicación TLM *ports* y TLM *exports*. Por un lado, los TLM *ports* son los encargados de especificar todos los métodos que se pueden utilizar en la comunicación y de iniciar peticiones de transacción. Por otro lado, los TLM *exports* implementan los métodos definidos por los TLM *ports*. Comúnmente, durante la fase `connect_phase()` estos puertos se conectan a un único puerto mientras se realiza la construcción del entorno de verificación, invocando al método `connect()`. Esta conexión se realizaría después de la creación de los componentes UVM y antes de que se realice la simulación.

En determinados procesos, se podría requerir que parte de la información se condujese por varios componentes, no solo dos, límite que imponen los puertos TLM *ports* y TLm *exports*. Por esta razón, la metodología UVM genera los puertos TLM *analysis ports* y TLM *analysis exports*. Un puerto TLM *analysis port* permite la conexión de varios puertos TLM *analysis exports*, permitiendo una transmisión *broadcast* mediante el uso del método `write()`.

3.4. ENTORNO DE VERIFICACIÓN UVM

Un entorno de verificación UVM permite la posibilidad de conseguir un mayor control en la generación de estímulos en el DUV y la recepción de las señales de salida, de las que se obtiene los datos para su posterior análisis y comparación con sus valores esperados. Este entorno UVM está formado por diversos componentes de verificación organizados en una jerarquía específica [15]. Esta jerarquía se determina según la relación entre estos componentes, es decir, en qué forma y orden se referencian unos dentro de otros.

La Figura 13 muestra un modelo de entorno UVM básico que sirve de referencia jerárquica. Se parte de un módulo principal, denominado *top*, donde se referencia el sistema que se quiere verificar y el entorno de verificación. Para que ambos se comuniquen y se pueda gestionar los estímulos y salidas del DUV, se conectan a una interfaz virtual, *interface,* que hace de canal de comunicación. Dentro del entorno UVM, el nivel más alto es el UVM *Test,* que contiene los componentes UVM *Environment* necesarios. Dentro del componente UVM *Environment* se encuentran, entre otros, los componentes de verificación UVM *Agent*. Estos últimos componentes representan la unidad básica, a nivel de reusabilidad, de cualquier entorno de verificación. Por su parte, un componente UVM *Agent* está, a su vez, compuesto por un componente UVM *Sequencer*, un componente UVM *Driver* y un componente UVM *Monitor*. Todos ellos integrados en el componente UVM *Agent*.

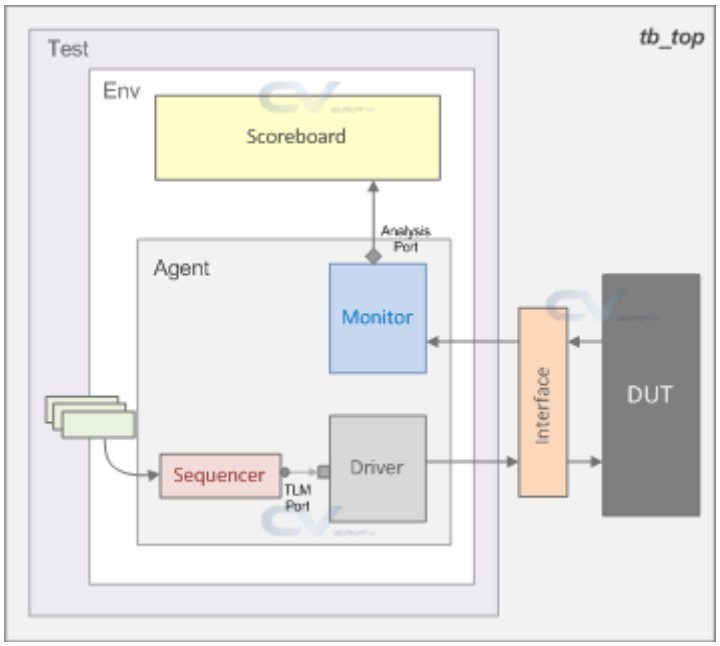

Figura 13 Modelo básico de un entorno de verificación UVM[15]

3.4.1. COMPONENTE UVM AGENT

Un UVM *Agent* es un componente que se deriva a partir de la clase `uvm_agent` y agrupa, en un nivel de abstracción mayor, un componente UVM *Sequencer*, un componente UVM *Driver* y un componente UVM *Monitor*. Cada uno de estos componentes cumple una función específica. El

componente UVM *Sequencer* se encarga de controlar el flujo de estímulos que le llegan en forma de secuencia. Estas secuencias de datos se definen como un objeto UVM *object* y, por su relevancia en el desarrollo de este TFG, se describirán en un apartado dedicado. El componente UVM *Driver* conduce estas secuencias hacia la interfaz del DUV, transformando los estímulos en señales. Por último, el componente UVM *Monitor* es el encargado de muestrear las señales que se encuentran en la interfaz del dispositivo. La Figura 14 muestra la representación gráfica de un componente UVM *Agent* común.

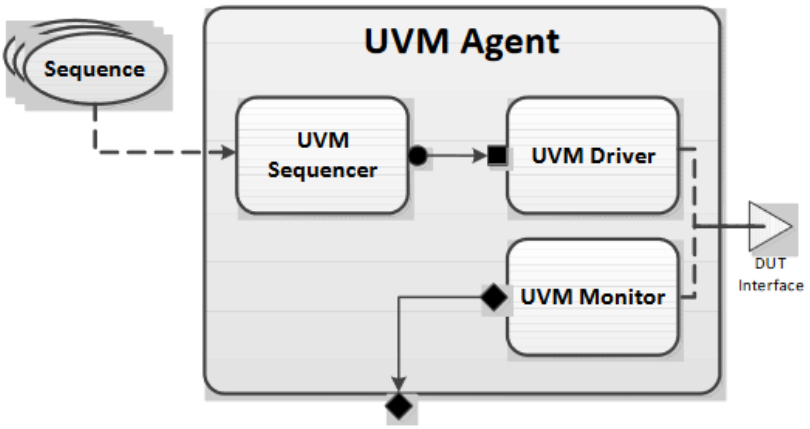

Figura 14 Estructura de un componente UVM *Agent*

Un componente UVM *Agent* se puede configurar como un elemento activo o pasivo. En el caso de estar configurado como activo, el UVM *Agent* tiene la función de conducir los estímulos generados en el test hacia el DUV. Por el contrario, un componente UVM *Agent* pasivo solamente monitoriza las señales de la interfaz del dispositivo. Por esta razón, un componente UVM *Agent* pasivo no necesita ni un componente UVM *Sequencer* ni un componente UVM *Driver.* Es capaz de cumplir su función únicamente con un UVM *Monitor*. La Figura 15 muestra la diferencia entre la configuración activa y pasiva de este componente.

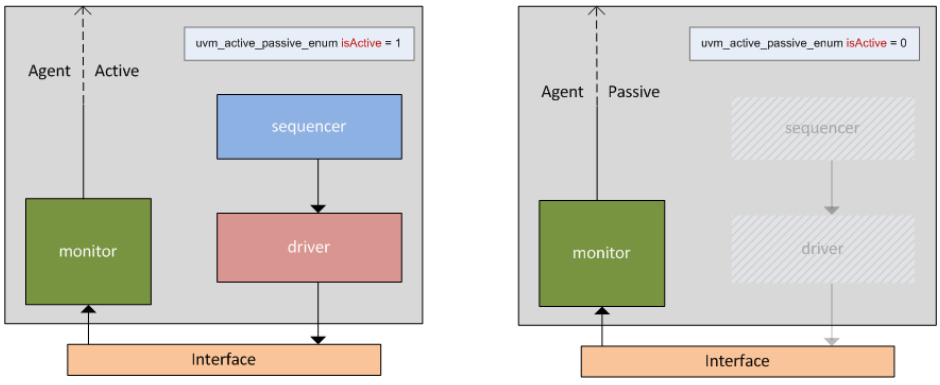

Figura 15 Diferencia entre UVM *Agent* activo y pasivo

Debido a que de manera predeterminada son de tipo activo, para configurar un UVM *Agent* en modo pasivo es necesario modificar una variable llamada `is_active`, definida dentro de su clase padre, de tipo `uvm_active_passive_enum`. Como se puede observar en la figura

anterior, al darle el valor 1 ó `UVM_ACTIVE`, el agente estará en modo activo y con el valor 0 ó `UVM_PASSIVE` se configurará en modo pasivo.

3.4.1.1. Componente UVM Sequencer

Un componente UVM *Sequencer* es un componente que se deriva de la clase `uvm_sequencer`. Esto es debido a que esta clase posee la funcionalidad requerida para realizar la comunicación entre este componente y el UVM *Driver*. El componente UVM *Sequencer* recibe transacciones de datos y las envía al componente UVM *Driver* para su ejecución. Esta comunicación se realiza mediante un protocolo de *handshake*. Este protocolo es muy importante para la gestión temporal de eventos en los componentes y se explicará en profundidad más adelante.

La creación y gestión de objetos dentro del componente UVM *Sequencer* es muy limitado, reduciendo la flexibilidad y reutilización del código. Para solucionar este problema, la complejidad de este proceso se supera mediante el uso de secuencias que, al igual que el protocolo de *handshake*, tienen su propio apartado.

3.4.1.2. Componente UVM Driver

Un componente UVM *Driver* es un componente que se deriva de la clase `uvm_driver`. La función de este componente es recibir las transacciones enviadas por el UVM *Sequencer* y conducirlas hasta la interfaz conectada con el DUV. Para ello, el componente UVM *Driver* solicita las transacciones al componente UVM *Sequencer* a través de un puerto TLM (*Transaction-Level Methodology*), para luego transformarlas en señales a nivel RTL (*Register-Transfer Level*) y aplicarlas sobre la interfaz.

En cuanto a esta comunicación con el DUV mediante una interfaz cabe destacar que no se trata de una interfaz física. Esto es debido a que al ser estática no se puede referenciar dentro de una clase que funciona de manera dinámica, como es UVM *Driver*. Por ello, se hace uso de una interfaz virtual, generada mediante un objeto de tipo *interface* en *SystemVerilog*.

3.4.1.3. Componente Monitor

Un UVM *Monitor* es un componente que se deriva de la clase `uvm_monitor` y se encarga de monitorizar la actividad de las señales de la interfaz conectada al DUV. De esta forma, este componente utiliza la información recogida y la exporta a través de un puerto TLM para ponerlo a disposición del resto de componentes del entorno de verificación.

Además, el componente UVM *Monitor* es capaz de realizar otro tipo de operaciones, como tareas de comprobación y cobertura que se utilizan, por ejemplo, para garantizar que se cumple con el plan de verificación establecido, acciones de mensajería en consola o almacenamiento de archivos. Sin embargo, lo más común es delegar este tipo de tareas en otros componentes.

En el caso de que se diseñe un UVM *Monitor* que incluye funcionalidades complejas, se puede realizar una completa abstracción desde el nivel de señales, separando el muestreo de señales del resto de funcionalidades del componente. Para ello, se introduce un nuevo componente, denominado UVM *Collector*, que se limitaría a captar la información que se envía a través de la interfaz y a generar transacciones para el componente UVM *Monitor*.

3.4.2. Componente UVM Environment

Un componente UVM *Environment* es un componente que se deriva de la clase `uvm_env` y se comporta como un contenedor. Este componente se encuentra en el segundo nivel de la jerarquía del entorno de verificación UVM y su función es la de incluir e interconectar componentes, como los explicados anteriormente, e incluso otros componentes UVM *Environment*, dependiendo de la complejidad del diseño. Este componente es totalmente necesario para cumplir con la reusabilidad del código. Esto es debido a que el componente UVM *Environment* posee las propiedades de configuración que permiten personalizar la topología del entorno de verificación y el comportamiento de los UVC (*Universal Verification Component*) que integra, pudiendo enfocarse a diversas tareas de verificación.

3.4.3. Componente UVM Test

Un componente UVM *Test* es un componente que se deriva de la clase `uvm_test` y su función es englobar el entorno de verificación UVM y todos los componentes que lo forman. Este componente es el nivel más alto de este entorno y es el primero en ser creado cuando se ejecuta la fase de `build_phase()`, para luego descender por la jerarquía definida.

Comúnmente, se crea una librería de test que consiste en una colección de pruebas que estimulan un DUV de diferentes maneras. Cuando se definen estas librerías, se suele desarrollar un test básico, denominado *base test*, que se deriva de `uvm_test` y contiene la referencia al componente UVM *Environment* principal y otros elementos comunes. A partir de este se derivan el resto de test que pueden tener diferentes características como la configuración del entorno de verificación o la selección de diferentes secuencias a ejecutar en los componentes UVM *Sequencer*. Estas secuencias también se suelen recoger en una librería para cubrir cualquier comportamiento del DUV y organizar la verificación.

3.4.4. Componente Top

El módulo *Top*, también denominado UVM *Testbench*, es del tipo *module* en *SystemVerilog* y se encarga de referenciar y configurar todos los componentes de verificación, interfaces y DUV. Esto lo convierte en un contenedor estático que contiene la función que genera el entorno de verificación UVM en su totalidad y el DUV. Al ser un componente estático permite, con una única compilación, la ejecución de varios test. El módulo *Top* es muy relevante en la metodología UVM y se van a comentar algunas de sus funciones a continuación:

- Crear y referenciar una o varias interfaces virtuales que permitirán la conexión entre el DUV y el entorno de verificación UVM. Estas interfaces definirán cada una de las señales del DUV a las que se quiera tener acceso durante la verificación. Además, se conectarán físicamente las señales de los puertos del DUV con sus homólogos en las interfaces virtuales.
- Generar las señales de reloj y de *reset*, además de cualquier señal común para todo el conjunto. La generación de la señal de reloj puede variar dependiendo de las necesidades del sistema. El módulo *Top* deberá adaptarse al número señales de reloj necesarias, así como a las diferentes frecuencias que puedan requerir.
- Incluir las interfaces virtuales creadas en la base de datos. Esto permite que ciertos componentes del entorno, como el UVM *Driver* o el UVM *Monitor*, tengan acceso a estas interfaces para poder interactuar con el DUV, introducir estímulos y monitorizar sus salidas.
- Realizar la llamada al método `run_test()`. Este método inicia la ejecución de las diferentes fases de la metodología UVM, provocando en última instancia el inicio de la simulación del test.

3.4.5. SECUENCIAS UVM Y PROTOCOLO DE *HANDSHAKE* SEQUENCER-DRIVER

Como se comentó en el apartado 3.4.1.1. , las secuencias juegan un papel muy importante dentro del entorno de verificación, en especial a la hora de la flexibilidad y reutilización del código, así como el envío y recepción de transacciones. Además, el UVM *Sequencer* y el UVM *Driver* se comunican mediante un protocolo de *handshake* muy importante para controlar la simulación. Junto a estos aspectos, en este apartado se abordará una forma de configurar las secuencias diferente a los modos activo y pasivo estándar.

3.4.5.1. SECUENCIAS UVM

Las secuencias UVM son objetos dinámicos que heredan de la clase `uvm_sequence`. No forman parte de la jerarquía de componentes, sino que tienen su propia jerarquía, pudiendo incluir unas secuencias dentro de otras. Estos objetos contienen los elementos de información y datos para generar estímulos en el DUV y crear distintas situaciones que alteren el comportamiento del dispositivo. Las secuencias se ejecutan en el UVM *Sequencer* y se envían, a través de este componente, al componente UVM *Driver*, generando los estímulos principales durante el tiempo de simulación, `run_phase`. Comúnmente, estos objetos tienen un comportamiento activo, comenzando ellos los estímulos en el DUV e iniciando las transacciones.

La metodología UVM proporciona dos macros que permiten crear, aleatorizar y enviar las secuencias que se les pasa por parámetros, que son `` `uvm_do `` y `` `uvm_do_with ``. La principal diferencia entre ambas macros es que la segunda permite incluir restricciones en la aleatorización de los campos de una transacción.

3.4.5.2. PROTOCOLO DE HANDSHAKE SEQUENCER-DRIVER

Para realizar la comunicación entre el UVM *Sequencer* y el UVM *Driver* se debe cumplir un protocolo de *handshake* que permita cumplir con esta tarea en el orden correcto [16]. La Figura 16 muestra de forma gráfica el protocolo de *handshake* entre los dos componentes. La comunicación comienza con una secuencia llegando al UVM *Sequencer* y descomponiéndose en una lista de objetos de transacción que serán enviados al componente UVM *Driver*. Este componente convierte estos *ítems* en señales conectadas al DUV. Este proceso se repite continuamente hasta el fin de la simulación.

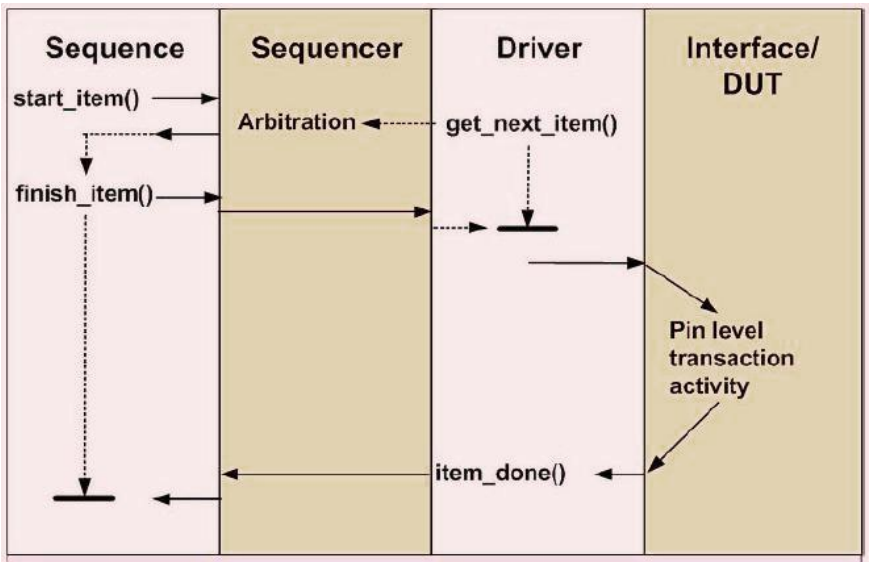

Figura 16 Protocolo de *hadshake Sequencer-Driver*[16]

Desde el punto de vista del UVM *Sequencer*, para enviar una transacción hasta el UVM *Driver* se siguen, en orden, los pasos descritos a continuación:

- Se crea el ítem de la transacción con el debido identificador usando el mecanismo de *Factory*.
- Se llama a la función `start_item(transaction_item_handle)`. Esta llamada bloquea al componente UVM *Sequencer* hasta que se garantiza el acceso de la secuencia y la transacción al componente UVM *Driver*.
- Se aleatoriza la transacción con las restricciones deseadas, si las hubiera. En este punto la transacción está preparada para ser usada por el UVM *Driver*.
- Se llama a la función `finish_item(transaction_item_handle)`. Al igual que la anterior, esta función es bloqueante y su propósito es esperar a que el componente UVM *Driver* transfiera los datos de la transacción relacionados con el protocolo.

En el lado del componente UVM *Driver* también se realizan una serie de procedimientos para completar la comunicación, enumerados en los siguientes puntos:

- Se declara el *ítem* de la transacción con su identificador.

- Se llama a la función `get_next_item(transaction_item_handle)`. Por defecto, el identificador se denomina `req` y está definido en la clase padre. Esta función bloquea el procesamiento hasta que el objeto de transacción `req` esté disponible en la solicitud del componente UVM *Sequencer*. Tras esto, la función `get_next_item()` devuelve un puntero al objeto.
- Posteriormente, el UVM *Driver* completa su protocolo de transferencia haciendo uso de la interfaz virtual.
- Por último, se llama a la función `item_done` o `item_done(rsp)`. Esto indica al UVM *Sequencer* que se ha completado el proceso. La función `item_done` es no bloqueante y se puede procesar con o sin argumento. Si se quiere o espera una respuesta por el UVM *Sequencer* o por la secuencia, se llama a `item_done(rsp)`, siendo `rsp` el identificador del objeto de transacción con la respuesta del UVM *Driver*. Este identificador está definido en la clase padre, por lo que no es necesario que el usuario lo defina.

3.4.5.3. Secuencias Reactivas en UVM

En la mayoría de los protocolos se pueden clasificar a los componentes como activos, que inician las transacciones en el bus, o pasivos, que responden a estas transacciones. Sin embargo, existe una alternativa en la que se ejecutan estímulos que reaccionan a las actividades o estados del DUV.

Desde el punto de vista de la verificación, un componente reactivo es un sistema esclavo que conduce las señales de respuesta solo cuando ya ha sido iniciada una transacción por el maestro en el DUV [17]. Es un componente activo capaz de afectar al estado y flujo de la simulación al conducir ciertas señales con valores y tiempos específicos. Las secuencias y los componentes ligados a ellas, UVM *Sequencer* y UVM *Driver*, pueden configurarse para que actúen con una naturaleza reactiva.

Generar transacciones en maestros proactivos es un proceso sencillo. Sin embargo, implementar esclavos reactivos conlleva una mayor complejidad, principalmente debido a que sus secuencias deben generarse automáticamente dependiendo de las transacciones que pueden llegar desde el DUV en cualquier momento. La implementación puede complicarse aún más si, por ejemplo, el esclavo contiene algún componente de almacenamiento. En la Figura 17 y la Figura 18, se muestran el comportamiento de un maestro proactivo y un esclavo reactivo, respectivamente.

En el primer caso, el estímulo es explícitamente iniciado por las secuencias, acción 1, dando como resultado el envío de un *request* desde el maestro proactivo al DUV, acción 2. Tras esto, el DUV genera la respuesta (*response*) en concordancia con el protocolo, acción 3.

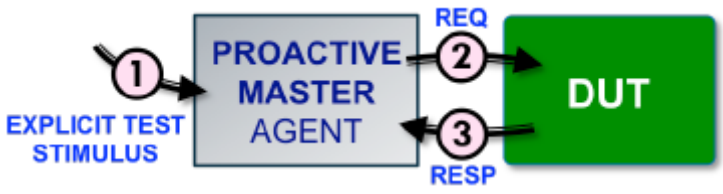

Figura 17 Comportamiento de un agente proactivo [17]

En el segundo caso, el DUV es provocado a generar un *request* como resultado de algún estímulo remoto, acción 1. Tras esta acción, el DUV genera una petición de *request*, acción 2. El agente esclavo reactivo, tras recibir la petición de *request* por parte del DUV, genera el correspondiente *response* en concordancia con el protocolo y los requisitos de la verificación, acción 3.

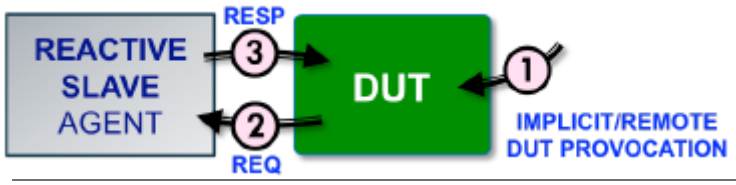

Figura 18 Comportamiento reactivo [17]

Como se ha comentado anteriormente, cuando el DUV actúa como maestro de la interfaz y el componente de verificación tiene que generar respuestas basadas en las solicitudes enviadas, el componente de verificación actúa como un esclavo reactivo. Existen varias formas de desarrollar un componente de verificación reactivo. En la mayoría de los casos, el DUV probablemente espera que los valores escritos sean retenidos por el esclavo. Por esta razón, se requiere que el esclavo reactivo incluya algún tipo de memoria. La Figura 19 muestra un modelo de los componentes de verificación de un esclavo reactivo con memoria. La arquitectura no es muy diferente a la de un agente proactivo. Sin embargo, en este caso hay comunicación adicional entre el UVM *Monitor* y el UVM *Sequencer* para generar secuencias tras reaccionar a las peticiones del DUV. Al no saber cuándo va a iniciar la comunicación el DUV, se necesita ejecutar un tipo de secuencia diferente. Además, tanto el UVM *Monitor* como el UVM *Sequencer* tienen acceso a una memoria donde almacenar los valores escritos por el DUV y recogerlos cuando se requieran.

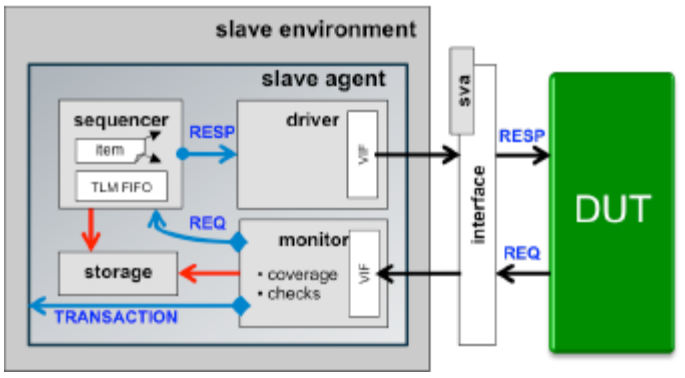

Figura 19 Modelo de esclavo reactivo con memoria [17]

Además, mientras que la infraestructura se ha complicado, el protocolo de *handshake* entre el componente UVM *Driver* y el UVM Sequencer se ha vuelto más sencillo. La Figura 20 muestra este protocolo, donde se puede observar que la operación del componente UVM *Driver* es idéntica a la del modelo básico.

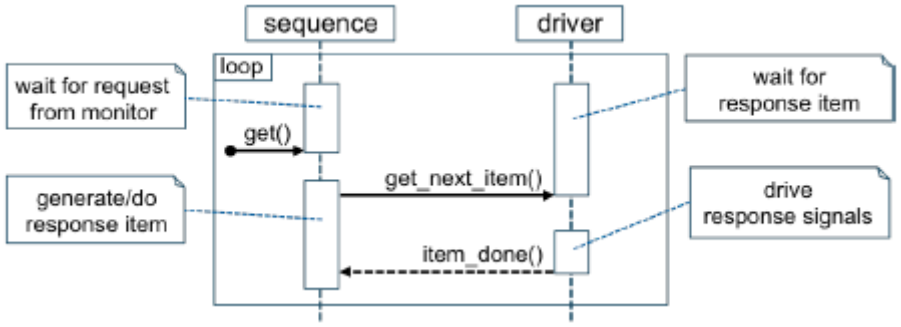

Figura 20 Modelo reactivo básico del protocolo *Sequencer-Driver* [17]

Otra alternativa para implementar un esclavo reactivo es mediante el uso de secuencias vacías (*dummy*). En esta implementación, el componente UVM *Driver* decodifica la petición del DUV y se comunica dos veces con el componente UVM *Sequencer* por cada transacción. La primera, para enviar la petición recibida y la segunda para recibir la respuesta generada. La principal ventaja de esta configuración alternativa es que la arquitectura es idéntica a la de un maestro proactivo añadiendo la comunicación adicional, como muestra la Figura 21.

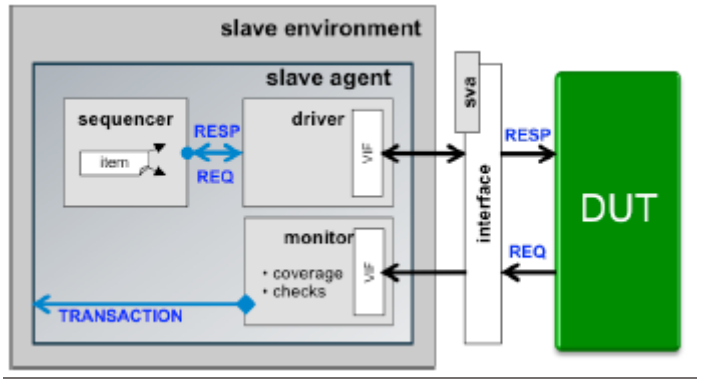

Figura 21 Modelo de esclavo reactivo con *dummy* [17]

En este caso, la interacción entre el componente UVM *Sequencer* y el componente UVM *Driver* es más complicada. La Figura 22 muestra esta comunicación. Se comienza con la generación de una transacción *dummy* que al llegar al componente UVM *Driver* no estimula el DUV. El componente UVM *Driver* espera a recibir una transacción del DUV, la decodifica y envía la información a la secuencia, que utilizará estos datos para actuar en consecuencia. A partir de la información recibida, la secuencia selecciona los datos a enviar al componente UVM *Driver* a través del componente UVM *Sequencer*. De esta manera, se repite el ciclo mientras dure la simulación.

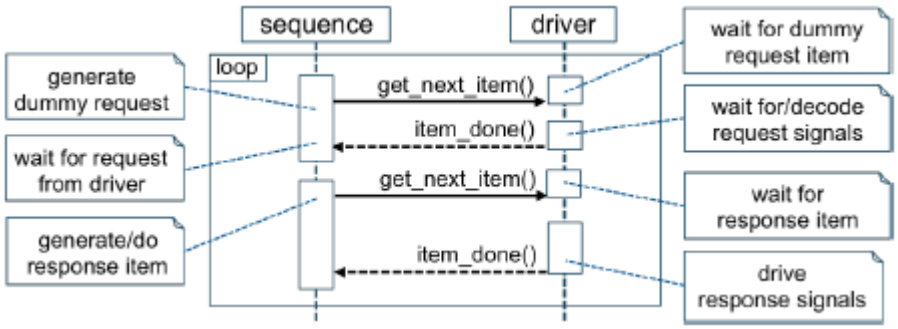

Figura 22 Modelo reactivo con transacción *dummy* del protocolo *Sequencer-Driver* [17]

Aun teniendo una comunicación interna más compleja, esta alternativa es la elegida para ser desarrollada en este TFG debido a su sencillez y la facilidad de su implementación.

3.5. Módulo Questa Verification IP de *Mentor Graphics*

Es importante resaltar que en la realización del presente Trabajo Fin de Grado se ha hecho uso de los componentes de verificación que ofrece el desarrollador *Mentor Graphics*. El motivo de su uso no es otro que el de usarlos para activar la interfaz AXI4 del diseño a verificar. Para su estudio, se ha utilizado el trabajo desarrollado por el estudiante Álvaro Moreno Florido en su Trabajo Fin de Grado [18].

En este punto se profundizará, en primer lugar, en los conceptos básicos que definen un IP de verificación de *Mentor Graphics*, denominado QVIP (*Questa Verification Intellectual Property*), con interfaz AXI4. Así mismo, se expondrá la funcionalidad que desempeña en la etapa de verificación, tanto de módulos IP como de SoC, que incluyan una interfaz de un protocolo de comunicación compatible con la interfaz utilizada por el módulo QVIP. Además, se analizarán tanto las ventajas como las desventajas asociadas al uso de estos módulos en la etapa de verificación de sistemas hardware digitales

El uso de estos componentes de verificación requiere el uso de la herramienta `qvip_configurator`, proporcionada por la compañía. Hay que indicar que, debido a la limitación temporal en el desarrollo de un TFG, no se ha considerado su uso y se ha decidido partir del trabajo desarrollado por Álvaro Moreno en su TFG, haciendo uso del entorno de verificación generado en su TFG. Aun así, se ha considerado oportuno resumir los principios básicos en el uso de esta herramienta para poder entender, a la hora de presentar el entorno desarrollado, los aspectos relacionados con el uso de estos módulos de verificación.

3.5.1. Principios Básicos para la Generación de Módulos QVIP

El uso de los módulos de verificación de *Mentor Graphics*, o QVIP, están pensados para su integración con diseños elaborados en *SystemVerilog* o VHDL. Su diseño está enfocado para simplificar su integración en entornos de verificación que siguen la metodología UVM. La familia de módulos QVIP para el protocolo AMBA proporciona una serie de API (*Application Programming*

Interfaces), denominados modelos funcionales de bus (*Bus Functional Model*, BFM) con funcionalidad completa para todos los modelos de uso de este protocolo. Estos modelos, generalmente, soportan todo tipo de estímulos sobre la interfaz para la que han sido desarrollados. Además, proporcionan una biblioteca de secuencias que se pueden utilizar para la estimulación del DUV. Esto facilita al ingeniero de verificación el alcanzar fácilmente una amplia cobertura en los diferentes escenarios de verificación.

Los módulos QVIP permiten verificar con facilidad las interfaces de un bus estándar, tal es el caso de la interfaz AXI4. Este hecho reduce el esfuerzo de los ingenieros durante la etapa de verificación del DUV, proceso clave durante el desarrollo de cualquier sistema electrónico. Esta reducción tiene un efecto directo en la etapa de generación de un *testbench*, así como en la etapa de diseño de los test a aplicar al DUV. Esto significa que el ingeniero de verificación puede centrarse, principalmente, en la etapa de *debug* de los errores encontrados. Teniendo en cuenta que esta etapa ocupa hasta un 41% del tiempo de verificación [19], tal y como se muestra en la Figura 23 su uso es fácilmente justificable.

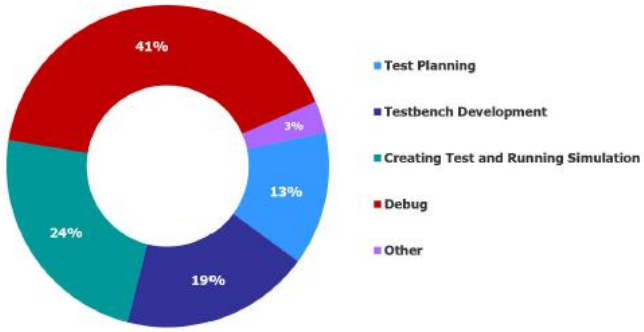

Figura 23 Tiempo de uso de las diferentes tareas de verificación [19]

3.5.2. Características de los módulos QVIP de *Mentor Graphics*

Los módulos de verificación de *Mentor Graphics* se ofrecen para la mayor parte de interfaces estándares de buses de comunicación. La Figura 24 muestra el juego completo de interfaces disponibles. Entre todas esas interfaces destacan, por ejemplo, protocolos como AXI, AHB, PCIe, Ethernet, USB y otros buses series como SPI, UART o I2C. La paleta de protocolos ofrecida por los módulos de verificación, QVIP, ha sido un factor clave para entender el uso cada vez mayor de esta herramienta, ya que se adapta con mayor facilidad a las necesidades existentes en el mercado. No obstante, la oferta de este tipo de módulos en el mercado, no se ajusta únicamente a los módulos de verificación ofrecidos por *Mentor Graphics* y a día de hoy son varias las empresas que los ofrecen. Entre ellas, se destacan los VIP de *Cadence* que, además, ofrece un catálogo de modelos de memoria [20] y los *Synopcys* VC *Verification* IP, de *Synopsys* [21], entre otros.

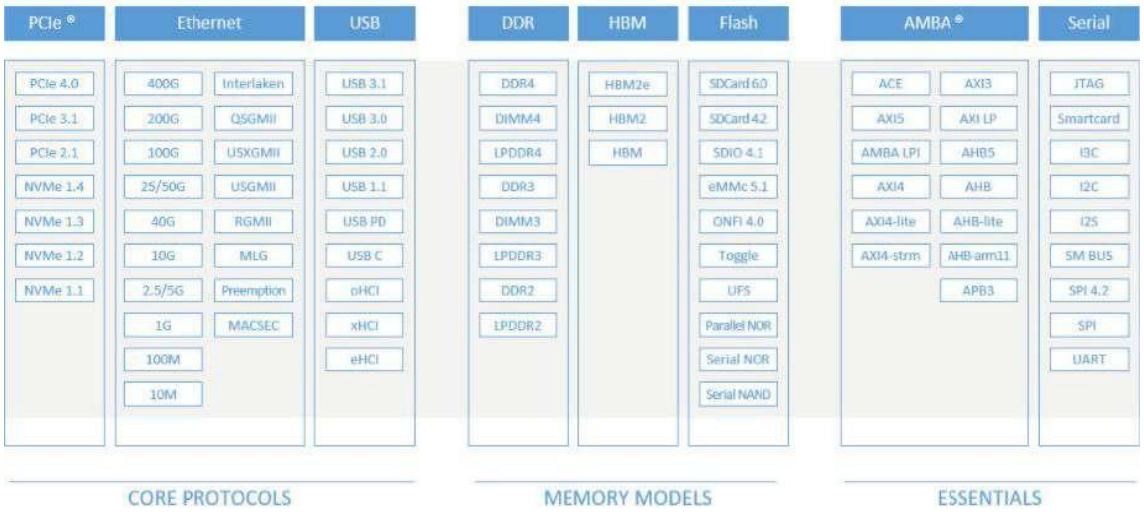

Figura 24 Protocolos de comunicación compatibles con el *Questa Verification* IP

A continuación, se detallan las características más importantes que justifican el creciente uso de los componentes de verificación. Estas características no son propias de un único desarrollador, estando presente en la práctica totalidad del catálogo de componentes de verificación del mercado.

- Todos los componentes son compatibles con diseños elaborados en *SystemVerilog* o VHDL, siendo estos los lenguajes predominantes y más extendidos en el diseño y verificación de sistemas hardware digitales.
- Los modelos ofrecidos son compatibles con los principales simuladores existentes en el mercado. Si bien es cierto que cada suministrador adapta, de mejor manera, sus VIP a su propio simulador.
- Además, los modelos pueden funcionar en los principales sistemas operativos más utilizados en la actualidad, como son Linux y Windows. Una característica común es que todos los modelos, independientemente del suministrador, precisan de una licencia para el uso de los QVIP suministrados.
- Todos los suministradores de estos modelos, además de los componentes de verificación, ofrecen una herramienta que facilita su uso. En el caso de *Mentor Graphics*, esta herramienta, tal y como se ha comentado, es la `qvip_configurator`.
- La herramienta `qvip_configurator` acelera el montaje del entorno de verificación ya que genera parte del *testbench* de manera automática, lo que ayuda en gran medida en la tarea de montar el entorno de verificación.
- El abanico de protocolos de comunicación ofrecido abarca casi todos los protocolos estándares de comunicación existentes en el mercado.
- En términos de reusabilidad, hay que destacar que todos los *test* realizados para un diseño con una interfaz definida, por ejemplo, la interfaz AXI4, son reutilizables siempre que se mantenga dicha interfaz. Esto permite reutilizar dichos test cuando el diseño se integre en un sistema más complejo (reusabilidad vertical).
- Por último, el uso de componentes de verificación proporciona una depuración intuitiva, ya que están implementados para visualizar las comunicaciones a nivel de transacciones, y no de señales únicamente. Además, estos componentes permiten la generación de archivos de depuración en varios niveles de abstracción.

Capítulo 4: DISEÑO A VERIFICAR

Una vez se ha presentado la metodología UVM y los protocolos asociados a las interfaces a utilizar, en este capítulo se va a estudiar el módulo IP que se utilizará como DUV sobre el que se realizará el entorno de verificación UVM. Para este TFG, se ha realizado una búsqueda de un diseño de terceros que cumpliese con la condición de tener al menos dos tipos de interfaces diferentes. La primera interfaz debería ser AXI4. La segunda, además de no ser AXI4, debería ser capaz de tener un comportamiento reactivo. Finalmente, se ha decidido elegir un módulo que realiza la función de conversor de AXI4 a *Wishbone B4*.

4.1. MÓDULO AXI4 MASTER TO WISHBONE

El sistema a verificar en este TFG es un módulo IP diseñado por la compañía *Gisselquist Technology.* Forma parte del proyecto *WB2AXIP* en el que se han desarrollado distintos puentes entre protocolos [22]. Este módulo tiene como principal función la conversión de comandos de escritura y lectura entre sistemas con distintas interfaces, sin alterar la velocidad de transmisión de ambos buses.

4.1.1. CARACTERÍSTICAS BÁSICAS DEL CONVERSOR

Este conversor, llamado `axim2wbsp` [23], hace la función de puente entre un sistema maestro con protocolo AXI4 y un esclavo que utiliza un protocolo *Wishbone* de tipo *pipelined*, siguiendo la arquitectura mostrada en la Figura 25.

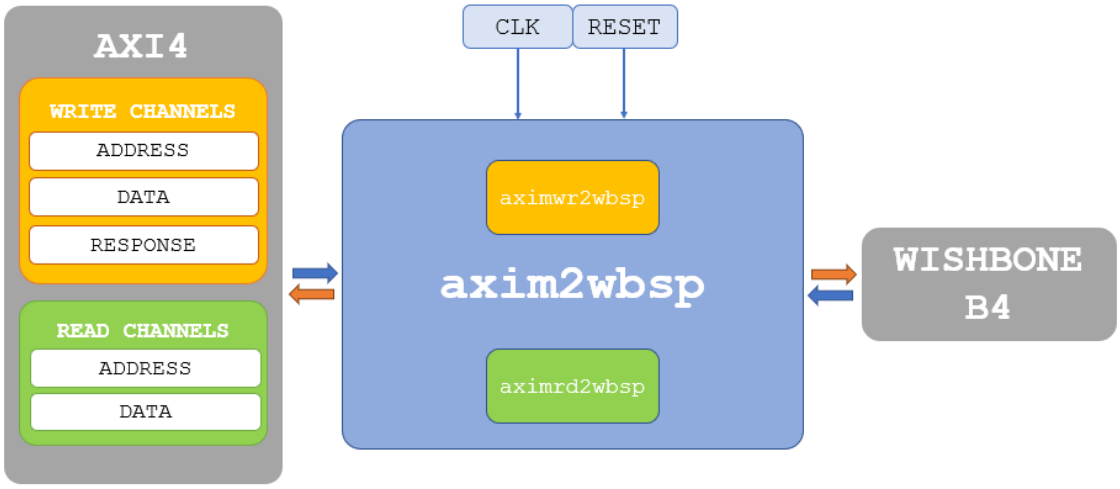

Figura 25 Arquitectura del módulo `axim2wbsp`

Este particular procesador está separado en dos mitades, una para escritura y otra para lectura. Ambas partes se pueden usar de manera independiente, generando distintas entradas para un *crossbar Wishbone*, o combinarlas en un mismo bus donde se ordenarán según su prioridad. Para asegurar su funcionamiento, se han separado las tareas de conversión en otros módulos más específicos. El módulo `aximwr2wbsp` es el encargado de transformar los comandos de escritura, utilizando las señales de direccionamiento y datos AXI y reflejándolos en señales *Wishbone*. Con el mismo papel, pero para la lectura, se hace uso del módulo `aximrd2wbsp`.

Además, ambos buses utilizan señales de control para la comunicación. El maestro necesita respuestas que confirmen que la comunicación ha sido correcta. Por esta razón, el módulo utiliza las señales de control del protocolo *Wishbone* para generar las respuestas que requiere el maestro para cada tipo de transacción.

4.2. Interfaces del procesador

Al ser capaz de realizar la conversión entre dos protocolos, el módulo dispone de dos interfaces correspondientes a cada extremo del conversor. Por un lado, dispone de una interfaz AXI4 actuando como maestro, que separa sus señales en varios canales. Por otro lado, hay una interfaz *Wishbone pipelined*, en la que el sentido de los datos marca la diferencia entre una escritura y una lectura. Ambas interfaces comparten señal de reloj. Sin embargo, la señal *reset* de la interfaz *Wishbone* es generada por el módulo `axim2wbsp` a partir de la señal de *reset* global. Estas señales son opuestas entre sí. Cuando el *reset* maestro está a nivel alto, se genera una señal de *reset* a nivel bajo para el módulo esclavo.

4.2.1. INTERFAZ AXI4 DEL CONVERSOR

Para interpretar los comandos del maestro, la interfaz AXI4 del módulo IP dispone de las señales necesarias para gestionar dicho protocolo. Estas señales están separadas globalmente en dos grupos, las necesarias para la escritura y las propias de la lectura. Ambas partes utilizan tramas con un tamaño de 32 bits para datos y 28 bits para direcciones de manera predeterminada. En la Figura 26 se indican estos parámetros. Además, se muestran las señales del protocolo AXI4 utilizadas para los comandos de escritura. Estas señales están divididas en tres canales. Entre las líneas 53 a 63 se encuentran las señales correspondientes al canal de dirección de escritura, entre las líneas 67 a 71 se muestran las señales utilizadas en el canal de escritura de datos y, por último, entre las líneas 75 y 78 se detallan las señales relacionadas con el canal de respuesta.

```
47              parameter C_AXI_ID_WIDTH = 2, // The AXI id width used for R&W
48              parameter  C_AXI_DATA_WIDTH  = 32,// Width of the AXI R&W data
49              parameter  C_AXI_ADDR_WIDTH     = 28,   // AXI Address width
50
51              // AXI write address channel signals
52              // {{{
53              input   wire                    S_AXI_AWVALID,
54              output  wire                    S_AXI_AWREADY,
55              input   wire    [C_AXI_ID_WIDTH-1:0]    S_AXI_AWID,
56              input   wire    [C_AXI_ADDR_WIDTH-1:0]  S_AXI_AWADDR,
57              input   wire    [7:0]           S_AXI_AWLEN,
58              input   wire    [2:0]           S_AXI_AWSIZE,
59              input   wire    [1:0]           S_AXI_AWBURST,
60              input   wire    [0:0]           S_AXI_AWLOCK,
61              input   wire    [3:0]           S_AXI_AWCACHE,
62              input   wire    [2:0]           S_AXI_AWPROT,
63              input   wire    [3:0]           S_AXI_AWQOS,
64              // }}}
65              // AXI write data channel signals
66              // {{{
67              input   wire                    S_AXI_WVALID,
68              output  wire                    S_AXI_WREADY,
69              input   wire [C_AXI_DATA_WIDTH-1:0]   S_AXI_WDATA,
70              input   wire [C_AXI_DATA_WIDTH/8-1:0] S_AXI_WSTRB,
71              input   wire                    S_AXI_WLAST,
72              // }}}D
73              // AXI write response channel signals
74              // {{{
75              output  wire                    S_AXI_BVALID,
76              input   wire                    S_AXI_BREADY,
77              output  wire [C_AXI_ID_WIDTH-1:0] S_AXI_BID,
78              output  wire [1:0]              S_AXI_BRESP,
```

Figura 26 Parámetros e Interfaz AXI4: Señales de escritura

La Figura 27 muestra que, para el caso de las señales de lectura, se distinguen dos grandes grupos. El primero, correspondiente a las líneas de la 83 a la 93, forma el canal de direcciones de lectura. El otro grupo lo constituye el canal de datos de lectura, que integra las señales de respuesta y datos, y se lista entre las líneas 97 a 102.

```
80              // }}}
81              // AXI read address channel signals
82              // {{{
83              input   wire                    S_AXI_ARVALID,
84              output  wire                    S_AXI_ARREADY,
85              input   wire [C_AXI_ID_WIDTH-1:0]   S_AXI_ARID,
86              input   wire [C_AXI_ADDR_WIDTH-1:0] S_AXI_ARADDR,
87              input   wire    [7:0]           S_AXI_ARLEN,
88              input   wire    [2:0]           S_AXI_ARSIZE,
89              input   wire    [1:0]           S_AXI_ARBURST,
90              input   wire    [0:0]           S_AXI_ARLOCK,
91              input   wire    [3:0]           S_AXI_ARCACHE,
92              input   wire    [2:0]           S_AXI_ARPROT,
93              input   wire    [3:0]           S_AXI_ARQOS,
94              // }}}
95              // AXI read data channel signals
96              // {{{
97              output  wire                    S_AXI_RVALID,  // Rd rslt valid
98              input   wire                    S_AXI_RREADY,  // Rd rslt ready
99              output  wire [C_AXI_ID_WIDTH-1:0]   S_AXI_RID, // Response ID
100             output  wire [C_AXI_DATA_WIDTH-1:0] S_AXI_RDATA,// Read data
101             output  wire                    S_AXI_RLAST,   // Read last
102             output  wire [1:0]              S_AXI_RRESP,   // Read response
```

Figura 27 Interfaz AXI4: Señales de lectura

4.2.2. INTERFAZ *WISHBONE* DEL CONVERSOR

En el extremo del *bridge* en el que se encuentra el esclavo, hay una interfaz *Wishbone B4 pipelined* que facilita la conexión. En la Figura 28 se muestran las señales utilizadas de este protocolo. Entre las líneas 116 y 122 se encuentran las señales de salida y entre las líneas 123 y 126 se indican las señales de entrada. Algunas de estas señales están parametrizadas al igual que en la interfaz anterior.

```
107             // Wishbone signals
108
109             output  wire                            o_reset,
110             output  wire                            o_wb_cyc,
111             output  wire                            o_wb_stb,
112             output  wire                            o_wb_we,
113             output  wire [(AW-1):0]                 o_wb_addr,
114             output  wire [(C_AXI_DATA_WIDTH-1):0]   o_wb_data,
115             output  wire [(C_AXI_DATA_WIDTH/8-1):0] o_wb_sel,
116             input   wire                            i_wb_stall,
117             input   wire                            i_wb_ack,
118             input   wire [(C_AXI_DATA_WIDTH-1):0]   i_wb_data,
119             input   wire                            i_wb_err
```

Figura 28 Interfaz *Wishbone*

4.3. *TESTBENCH* DEL CONVERSOR

Para poder comprobar el correcto funcionamiento del diseño, se hace el uso de un *testbench* que permita comprobar las características funcionales del sistema. En este caso, con la distribución de este diseño no existía ninguno a nivel RTL, por lo que fue necesario desarrollar un *testbench ad hoc* para realizar esta comprobación. Este *testbench* se realizó en lenguaje *Verilog*. Para realizar este *testbench* fue necesario estudiar los principios básicos del lenguaje *Verilog* para poder referenciar el diseño y, además, generar de forma muy dirigida los estímulos a aplicar.

4.3.1. IMPLEMENTACIÓN DEL *TESTBENCH*

El *testbench* desarrollado, llamado `axim2wbsp_tb`, comienza con la referencia al módulo original, `axim2wbsp`, como se muestra en la Figura 29 a partir de la línea 48. Para ello, se describen todas las señales necesarias para conectarlas a las entradas y salidas del sistema. De esta manera, se puede estimular al diseño para que genere las respuestas correspondientes.

```
146 // TB declaration
147 module axim2wbsp_tb;
148
149    // DUT Instantiation
150    axim2wbsp #(
151                .C_AXI_DATA_WIDTH (C_AXI_DATA_WIDTH),
152                .C_AXI_ADDR_WIDTH (C_AXI_ADDR_WIDTH)
153                )
154                axim2wbsp(
155                       // AXI Global clk and reset
156                       .S_AXI_ACLK (S_AXI_ACLK), // System clock
157                       .S_AXI_ARESETN (S_AXI_ARESETN),
158
159                       // AXI write address channel signals
160                       .S_AXI_AWVALID(S_AXI_AWVALID),
161                       .S_AXI_AWREADY (S_AXI_AWREADY),
162                       .S_AXI_AWID( S_AXI_AWID),
163                       .S_AXI_AWADDR (S_AXI_AWADDR),
164                       .S_AXI_AWLEN( S_AXI_AWLEN),
165                       .S_AXI_AWSIZE( S_AXI_AWSIZE),
166                       .S_AXI_AWBURST(S_AXI_AWBURST),
```

Figura 29 *Testbench*: Referencia al módulo original

Una vez realizada la referencia comienza la parte principal del *testbench*, en la que se estimulan las señales de entrada para obtener una respuesta a la salida. Las señales más sencillas a generar son las de `clk` y `reset`. La señal de `clk` se genera mediante un proceso `always` con una lista sensible correspondiente al tiempo `T/2`, siendo `T` el período de la señal de reloj. Por su parte, la señal de `reset` se genera usando un proceso `initial` donde se inicializa a nivel alto para, posteriormente, generar un pulso a nivel bajo.

El resto de las señales son las relacionadas con ambos protocolos. Cada parte se ha implementado de manera independiente. Para estimular las señales correspondientes al protocolo AXI se han definido tareas relacionadas con los distintos canales de escritura y lectura. La Tabla 11 muestra un listado de las tareas implementadas que facilitan y simplifican la continua variación y complejidad del protocolo. A continuación, se presentará el código de algunas de estas tareas.

Tabla 11 *Testbench*: Tareas AXI4

Tarea	Entradas	Descripción
`AW_reset`	-	Espera a que se active la señal `reset` para establecer un valor predeterminado en las señales de los canales de dirección y datos de escritura.
`AW_waddr`	`delay`, `AWADDR`, `AWBURST`,	Tras esperar un tiempo establecido por `delay`, asigna el resto de las entradas a las señales `S_AXI_AWADDR`,

	AWLEN, AWSIZE	S_AXI_AWBURST, S_AXI_AWLEN y S_AXI_AWSIZE, respectivamente. Posteriormente, gestiona el protocolo VALID-READY explicado en el capítulo 2.1.2.
W_data	delay, AWLEN, AWSIZE	Espera un tiempo delay y realiza un bucle, de tamaño marcado por AWLEN, para generar las escrituras de datos pedidas. Utiliza el valor de AWSIZE para variar la señal WSTRB. Todas las escrituras utilizan la relación VALID-READY comentada en la tarea anterior. Al mismo tiempo, los datos escritos se guardan en una memoria.
W_response	-	Espera a que se active la señal S_AXI_BVALID y pone a nivel alto la señal S_AXI_BREADY durante un tiempo.
AR_reset	-	Mismo funcionamiento que **AW_reset** para lectura.
AR_raddr	delay, ARADDR, ARBURST, ARLEN, ARSIZE	Mismo funcionamiento que **AW_waddr** para el canal de lectura.
R_data	delay, ARLEN, ARSIZE,	Tarea que gestiona el canal de lectura de datos. Sigue el mismo procedimiento que la tarea **W_data**, pero depositando los datos previamente almacenados en memoria. Activa la señal S_AXI_RREADY durante los pulsos de lectura de datos.

La Figura 30 muestra el código de una de las tareas de *reset*, en este caso AW_reset. Su funcionamiento es sencillo. Comienza en la línea 307, esperando un flanco de subida de la señal S_AXI_RESETN, que corresponde con su desactivación. Acto seguido, establece las señales de control y parámetros más relevantes a un valor predefinido para empezar correctamente con una transacción de escritura, como se refleja de la línea 308 a la línea 315.

```
304   //Tasks
305   task AW_reset;
306      begin
307         @(posedge S_AXI_ARESETN);
308         S_AXI_AWVALID <= 1'b0;
309         S_AXI_WVALID  <= 1'b0;
310         S_AXI_WLAST   <= 1'b0;
311         S_AXI_AWBURST <= 2'h0;
312         S_AXI_AWLOCK <= 1'b0;
313         S_AXI_AWCACHE <= 4'h0;
314         S_AXI_AWPROT <= 3'b000;
315         S_AXI_AWQOS <= 4'b0;
316      end
317   endtask // AW_reset
318
```

Figura 30 *Testbench*: tarea AXI *AW_reset*

Tras el *reset* de los canales de escritura, vendría el direccionamiento. Este proceso lo realiza la tarea `AW_waddr`, como se indica en la Figura 31. Tras esperar un tiempo, `delay`, recoge los datos pasados por parámetros al llamar a la tarea, líneas 321 a 324, y los coloca en las señales correspondientes, líneas 328 a 331. En ese momento, activa la señal de `S_AXI_VALID` y espera a que se produzca el protocolo de *handshake* antes de desactivarla, como se muestra de la línea 332 a la línea 337.

```
319   task AW_waddr;
320      input integer delay;
321      input reg [C_AXI_ADDR_WIDTH-1:0] AWADDR;
322      input reg [1:0]                  AWBURST;
323      input reg [7:0]                  AWLEN;
324      input reg [2:0]                  AWSIZE;
325      begin
326         repeat(delay) @(posedge S_AXI_ACLK);
327         // AW Channel
328         S_AXI_AWADDR  <= AWADDR;
329         S_AXI_AWBURST <= AWBURST;
330         S_AXI_AWLEN   <= AWLEN;
331         S_AXI_AWSIZE  <= AWSIZE;
332         S_AXI_AWVALID <= 1'b1;
333
334         do @(posedge S_AXI_ACLK);
335         while (S_AXI_AWREADY == 1'b0);
336
337         S_AXI_AWVALID <= 1'b0;
338      end
339   endtask // AW_waddr
```

Figura 31 *Testbench*: tarea AXI *AW_waddr*

La tarea `W_data` es la encargada de gestionar el *Write Data channel* con el código que se muestra en la Figura 32. Primero, como se muestra entre las líneas 350 y 354, establece la variable `WSTRB` con el valor que debería llevar el *strobe* según el *SIZE* de la transacción. Tras esperar un *delay*, comienza un bucle que se repetirá las veces que se haya marcado en la entrada `AWLEN`, línea 358. Este bucle comienza introduciendo un dato aleatorio en un *array* dinámico, línea 359, lo que permitirá el almacenamiento de datos, actuando como una memoria del sistema esclavo. En esta ocasión, la dirección `0x400`, donde se guarda el primer dato, es la correspondiente a una dirección `0x1000` de AXI para *Wishbone*. Esta dirección se guarda en una variable llamada ADDR, como se indica en la línea 357. Entre las líneas 360 y 365, se introduce el dato almacenado y el *strobe* definido anteriormente en las señales `S_AXI_WDATA` y `S_AXI_WSTRB`. Se comprueba si es el último dato a transmitir y se espera al protocolo de *handshake*. Una vez realizado, se repite el bucle actualizando la dirección donde se guardará el dato en la memoria hasta que se transfieran todos los datos pedidos.

```
341   task W_data;
342      input integer delay;
343      input reg [7:0]                  AWLEN;
344      input reg [2:0]                  AWSIZE;
345      reg [C_AXI_DATA_WIDTH/8-1:0]     WSTRB;
346      reg [C_AXI_ADDR_WIDTH-1:0]       ADDR;
347      integer i;
348      begin
349      // Strb select
350      case(AWSIZE)
351        3'd0: WSTRB = 1;
352        3'd1: WSTRB = 3;
353        3'd2: WSTRB = 15;
354      endcase // case (AWSIZE)
355      // WData Channel
356      repeat(delay) @(posedge S_AXI_ACLK);
357         ADDR = 400; //memAddr;
358      for (i= 0; i< AWLEN + 1; i = i+1) begin
359         slvmem[ADDR+i] = $random;
360         S_AXI_WVALID <= 1'b1;
361         S_AXI_WDATA  <= slvmem[ADDR];
362         S_AXI_WSTRB  <= WSTRB;
363         S_AXI_WLAST  <= (i == AWLEN);
364         do @(posedge S_AXI_ACLK);
365         while (S_AXI_WREADY == 1'b0);
366      end
367         S_AXI_WVALID <= 1'b0;
368         S_AXI_WLAST  <= 1'b0;
369      end
370   endtask // W data
```

Figura 32 *Testbench*: tarea AXI *W_data*

En relación con la interfaz Wishbone, las señales que se pueden generar son las relacionadas con la respuesta que daría un esclavo que use este protocolo. Por esta razón, la gestión de esta interfaz se ha realizado usando una serie de procedimientos `always` que están, continuamente, analizando las señales generadas por el sistema para actuar en consecuencia. La Figura 33 muestra el código *Verilog* relacionado con estos procedimientos, en los que se manejan las señales `i_wb_stall`, `i_wb_ack`, `i_wb_data` y `i_wb_err`.

- **Generación de la señal `i_wb_stall`**: Para generar una señal `i_wb_stall` que vaya modificándose en cada ciclo de reloj, se ha utilizado un vector temporal, denominado `rstall`, e inicializado con un determinado patrón, línea 274. En cada ciclo de reloj, el vector va rotando, línea 276. De esta manera, se utiliza el bit [0] de este vector para generar la señal `i_wb_stall`, tal y como se muestra en la línea 283.
- **Generación de la señal `i_wb_ack`**: Para esta señal, se utilizará un registro denomidnado `i_wb_ack_r`. Este registro, cuando no esté `i_wb_stall` activo, tendrá el mismo valor que `o_wb_stb`, línea 287. La señal `i_wb_ack` utilizará el valor que tenga este registro siempre y cuando no esté `i_wb_stall` activo, línea 291.
- **Generación de la señal `i_wb_data`**: Para generar esta señal y que tenga concordancia con las tareas del protocolo AXI, se utilizará el *array* dinámico `slvmem`. De esta manera, cada vez que se establezca una dirección en la señal `o_wb_stb`, se enviará por `i_wb_data` el valor almacenado en la memoria con esa dirección, como se muestra en la línea 277.
- **Generación de la señal `i_wb_err`**: En este caso, se ha decidido no utilizar esta señal para facilitar la comprobación del protocolo. Por este motivo, esta señal se ha puesto fija a `'b0`, tal y como se muestra en la línea 278.

```
266   // Whisbnone Interface
267
268   always @(posedge S_AXI_ACLK) begin
269      if(!S_AXI_ARESETN) o_wb_stb_r <= 1'b0;
270      else o_wb_stb_r <= o_wb_stb;
271   end
272
273    always @(posedge S_AXI_ACLK) begin
274      if (!S_AXI_ARESETN) rstall <= 16'b0100_0110_0010_1001;
275      else begin
276         rstall <= {rstall[0],rstall[15:1]};
277         i_wb_data  <= slvmem[o_wb_addr];
278         i_wb_err   <= 1'b0;
279      end
280   end
281
282   always @(*) begin
283      i_wb_stall = rstall[0];
284   end
285
286   always @(posedge S_AXI_ACLK) begin
287      if(!i_wb_stall) i_wb_ack_r <= o_wb_stb;
288   end
289
290   always @(*) begin
291      i_wb_ack = #1 i_wb_ack_r & !i_wb_stall;
292   end
293
```

Figura 33 *Testbench*: procedimientos para la gestión del protocolo *Wishbone*

4.3.2. SIMULACIÓN DEL *TESTBENCH* DESARROLLADO

Una vez implementado el *testbench*, es posible simularlo y comprobar su funcionamiento. Para esta función, se utiliza un archivo `Makefile` que permite la compilación del *testbench,* abrir la herramienta de simulación y cargar las formas de onda de manera sencilla a través de la consola. La herramienta utilizada para llevar a cabo la simulación es *QuestaSim* de *Mentor Graphics, a Siemens Bussiness,* en su versión *2019.2*. En los siguientes apartados, se simulan distintas transacciones separadas según el tipo de ráfaga que envíe el maestro. De esta manera se puede estudiar las situaciones más relevantes que ofrece el módulo.

4.3.2.1. EJECUCIÓN DE UNA TRANSACCIÓN AXI4 EN MODO *FIXED*

Entre las distintas opciones para simular este tipo de transacción, se ha optado por enviar una transacción simple. A continuación, se va a analizar, a nivel gráfico, el comportamiento del módulo para gestionar este proceso. El procedimiento consiste en enviar un dato de tamaño completo a una dirección y posteriormente se lee en esa dirección el dato enviado, para comprobar su consistencia.

- **Operación de escritura**

La Figura 34 muestra el comienzo de la transacción de escritura. En esta figura se destacan las señales del canal de dirección de escritura, etiquetado como `AW Channel`. La señal `AWADDR` indica la dirección de escritura `0x1000`. La señal `AWBURST`, al estar a 0, indica el tipo de

transacción *FIXED.* Las señales `AWLEN` y `AWSIZE` muestran que se trata de un único dato y que tiene el tamaño completo. Además, se observa cómo el protocolo de *handshake* se gestiona correctamente, ya que la transacción finaliza cuando `AWVALID` y `AWREADY` están ambas activas.

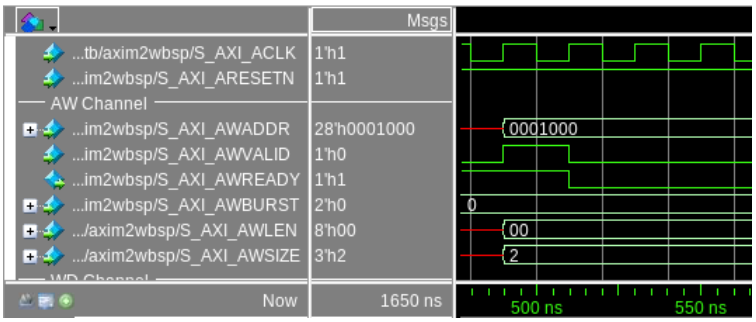

Figura 34 Simulación de transacción tipo *FIXED*: `AW Channel`

Tras completar el direccionamiento, se lleva a cabo el envío de datos. En la Figura 35 se muestra el canal de datos de escritura, etiquetado como `WD Channel`, donde se lleva a cabo la gestión del protocolo de *handshake* y se envía el dato `0x12153524` por el bus `WDATA`. Las señales `WLAST` y `WSTRB` indican que ese dato termina la transacción y que son válidos todos sus *bytes*.

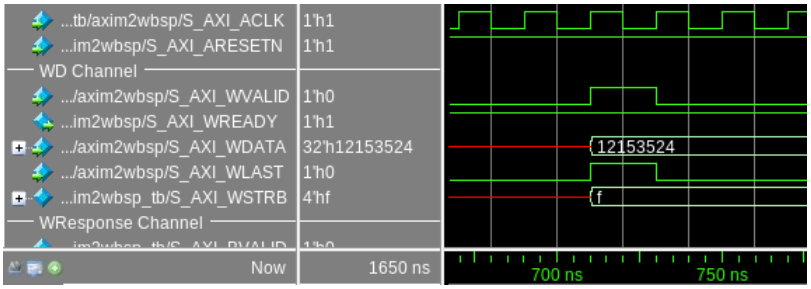

Figura 35 Simulación de transacción tipo *FIXED*: `WD Channel`

Una vez enviados la dirección y el dato a través de la interfaz, el conversor gestiona esta información y la envía a través del intefaz *Wishbone*, tal como se muestra en la Figura 36. Las señales de esta figura se dividen en dos partes.

En el apartado etiquetado como `O Whisbone` se encuentran las señales de salida del módulo. La dirección mostrada en `o_wb_addr`, `0x400`, es diferente a la dirección recibida en la interfaz AXI4, `0x1000`. Esto es debido al distinto tipo de direccionamiento de cada protocolo, explicado en anteriores apartados. Algo parecido sucede con el dato `0x24351512` que aparece en el bus `o_wb_data`. Este dato llega desordenado debido a que el protocolo AXI ordena los datos a partir del *byte* menos significativo (convenio *Little Endian*), mientras que el protocolo *Wishbone* los ordena a partir del *byte* más significativo (convenio *Big Endian*). Por último, la señal `o_wb_sel` indica que todos los *bytes* de datos son válidos, lo que coincide con la información proporcionada en la interfaz AXI4.

Las últimas tres señales, correspondientes a la parte `I Wishbone`, engloba las señales `i_wb_stall`, `i_wb_ack` e `i_wb_err` que se han configurado en el *testbench* y funcionan según lo planeado. Tal y como se muestra en el instante t1, al activarse la señal `o_wb_strb` y no estar activa la señal `i_wb_stall`, la señal de reconocimiento, `i_wb_ack` se activa, tal y como describe el protocolo. Por último, en el ciclo t2 se muestra como al activarse la señal de reconocimiento, el módulo finaliza el ciclo desactivando la señal `o_wb_cyc`.

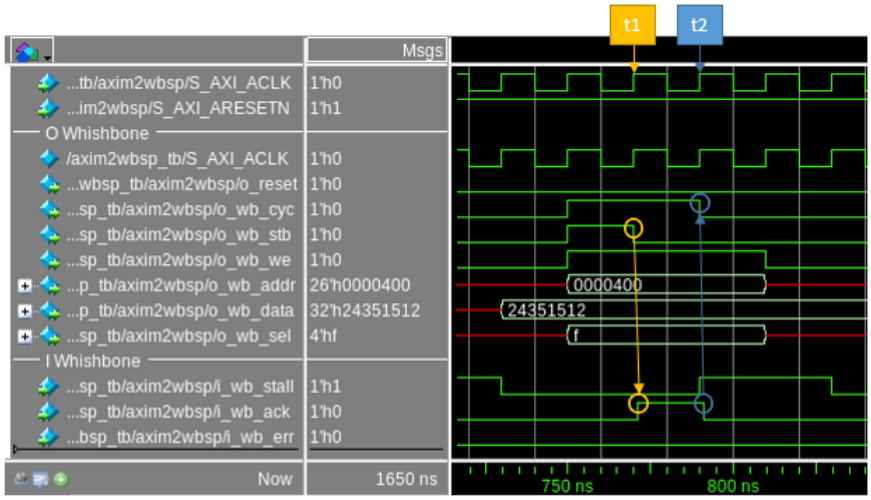

Figura 36 Simulación de transacción tipo *FIXED*: Escritura *Wishbone*

La gestión del protocolo *Wishbone* genera automáticamente una respuesta en la interfaz AXI4. La Figura 37 muestra la respuesta generada debido a la transacción enviada en la figura anterior. En el instante t1, el DUV valida, señal `S_AXI_BVALID` activa, la respuesta presente en el bus `S_AXI_BRESP`, y espera al instante t2, en el que el maestro acepta la respuesta generada. Indicar en este caso, que la respuesta ha sido OKAY.

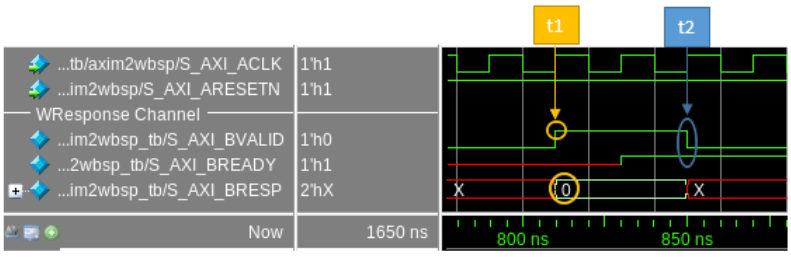

Figura 37 Simulación de transacción tipo *FIXED*: *Write Response Channel*

- **<u>Operación de lectura</u>**

El siguiente paso en esta simulación es el proceso de lectura. Para ello, al igual que para la escritura, el proceso comienza con el canal de direccionamiento (`AR Channel`), como se muestra en la Figura 38. Como se busca finalizar el proceso de escritura/lectura iniciado en la etapa anterior,

la dirección utilizada en este caso es la misma que se envió anteriormente en el canal *Address Write channel*. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura en las señales correspondientes. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

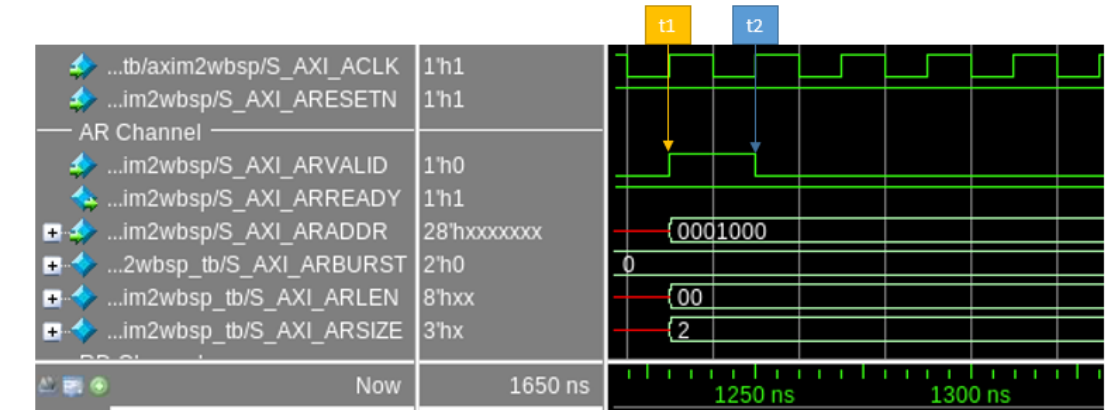

Figura 38 Simulación de transacción tipo *FIXED*: AR Channel

Al tratarse de una lectura, sería el esclavo *Wishbone* el encargado de suministrar el dato pedido debiendo el DUV solicitar, a través de la interfaz *Wishbone*, el dato deseado. La Figura 39 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso 0x400, al tratarse de una dirección a palabra. A través del bus `i_wb_data` el esclavo envía los datos al maestro. En el instante t2, la señal que valida la dirección, `o_wb_stb`, no puede desactivarse ya que el esclavo ha señalizado el ciclo anterior como un ciclo de *stall*, señal `i_wb_stall` activa. Esto provoca que el maestro contenga todos los parámetros hasta el instante t3, donde verifica que ha terminado el ciclo de *stall*. Al mismo tiempo, el esclavo activa la señal de reconocimiento, `i_wb_ack`, lo que provoca que en el instante t4 el maestro dé por finalizado el ciclo, desactivando la señal `o_wb_cyc`. Es importante destacar que una vez finalizado el ciclo de *stall*, el esclavo deposita en el bus de datos `i_wb_data` el dato solicitado por el DUV, instante t2.

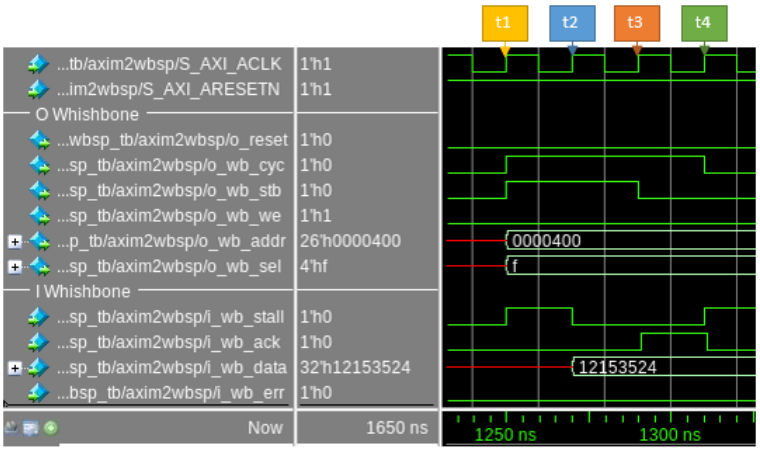

Figura 39 Simulación de transacción tipo *FIXED*: Lectura *Wishbone*

Para terminar con la lectura, el conversor realiza el cambio de protocolo en sentido contrario, respondiendo al maestro AXI con las señales adecuadas como muestra la Figura 40. En este caso, al igual que ocurría en el *Write Response channel*, el protocolo de *handshake* se realiza al revés. En este caso, es el DUV quien inicia el protocolo y el maestro quien indicaría con la señal `RREADY` que está preparado para recibir los datos validados por el DUV haciendo uso de la señal `RVALID`. Hay que destacar que, en este sentido, el dato proporcionado por el esclavo es exactamente el dato recibido a través de la interfaz *Wishbone*, esto es `0x12153524`, que, como se observa, se ha almacenado usando el convenio *Big Endian*.

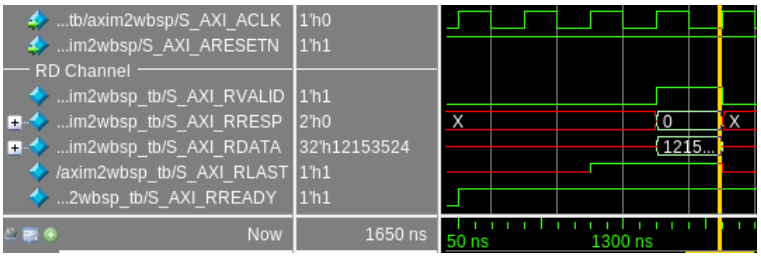

Figura 40 Simulación de transacción tipo *FIXED*: RD Channel

4.3.2.2. EJECUCIÓN DE UNA TRANSACCIÓN AXI4 EN MODO *INCREMENTAL*

El caso de una transacción tipo *incremental* es distinto al anterior. El motivo principal es que la dirección no se mantiene fija, sino que se incrementará a nivel de *bytes*. En esta ocasión, se va a simular una escritura incremental de longitud 6 palabras, con un ancho de palabra de 16 bits, por lo que se validará solo la mitad de sus *bytes*.

La escritura comienza en el instante t1 con el estímulo en las señales de `AW Channel`, como muestra la Figura 41. El bus `AWADDR` indica que se parte de la dirección `0x1000`. En esta ocasión, el parámetro `AWBURST` se establece a `0x1`, el parámetro `AWLEN` a `0x5` y el parámetro `AWSIZE` a `0x1`. Esto conlleva que la escritura sea de tipo *incremental* y que se van a enviar seis datos de 32 bits de los que son válidos la mitad. En el instante t2, y debido a que la señal de READY está activa, finaliza el inicio de la transacción.

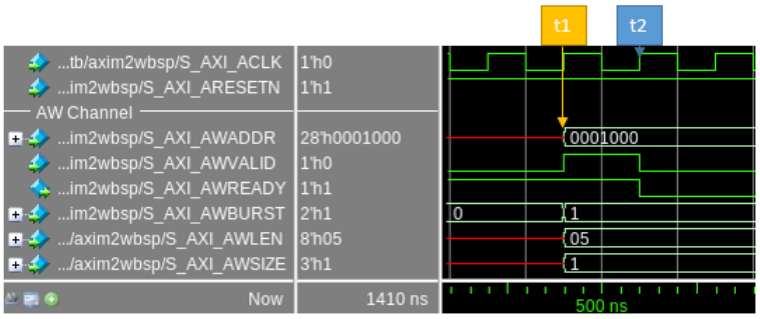

Figura 41 Simulación de transacción tipo *INCR*: AW Channel

La Figura 42 muestra las señales asociadas al canal *Write Data channel*. En esta figura, además del protocolo de *handshake*, se observan los seis datos enviados a través del bus `WDATA`, como se había programado en el canal anterior. Además, en concordancia con la programación del tamaño de palabra, la señal `WSTRB` tiene el valor `0x03`, que indica que son válidos los dos bytes menos significativos. Con el sexto dato, presente en el bus en el instante t4 y cuyo valor es `0x46DF998D`, se activa la señal `WLAST`, indicando que es el último elemento enviado.

Hay que indicar que el sistema mantiene invariables los datos en el canal en los instantes t1, t2 y t3, al no estar preparado el DUV para recibir nuevos datos. Este hecho, tal y como ya se ha comentado, se señaliza desactivando la señal READY

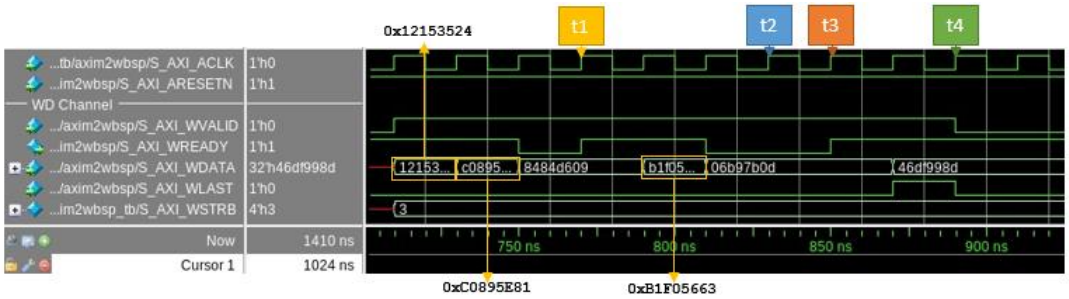

Figura 42 Simulación de transacción tipo *INCR*: WD Channel

La Figura 43 presenta las señales de respuesta generadas por el conversor en la parte de la interfaz *Wishbone*. El comportamiento de las señales `o_wb_cyc`, `o_wb_stb` y `o_wb_we` es sencillo y se activan el tiempo necesario para que se tramiten todos los datos. La gran diferencia con el apartado anterior está en las direcciones y los datos. En relación con las direcciones, se observa que la señal `o_wb_addr` se incrementa consecutivamente, desde el valor `0x400` al valor `0x402`. Este incremento sucede cada vez que se completa una escritura de 32 bits de datos de cada dirección. Como en cada escritura únicamente se validan 16 bits, se necesitan dos escrituras para cada dirección. La validación de los datos en la interfaz AXI4 se realizó sobre los dos *bytes* menos significativos. En el caso de la interfaz *Wishbone*, y debido al cambio de convenio, la validación establecida por la señal `o_wb_sel` será `0x0C`, lo cual es correcto. Los datos se confirman gracias a la activación de la señal `i_wb_ack` para cada dato.

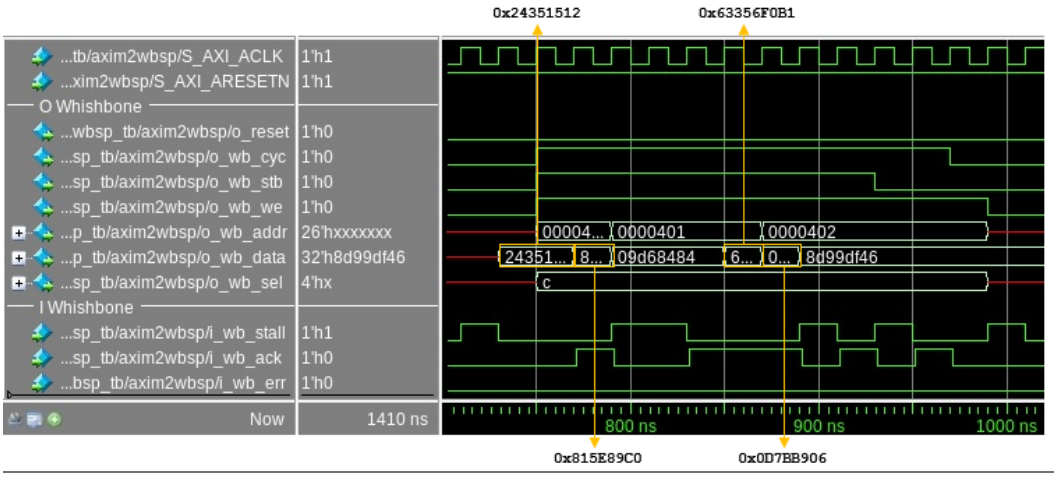

Figura 43 Simulación de transacción tipo *INCR*: Señales *Wishbone*

Una vez reconocidos todos los datos, la gestión del protocolo *Wishbone* genera automáticamente una respuesta en la interfaz AXI4. La Figura 44 muestra la respuesta generada en el canal *Write Response channel* debido a la transacción enviada en la figura anterior. En el instante t1, el DUV valida, señal `S_AXI_BVALID` activa, la respuesta presente en el bus `S_AXI_BRESP`, y espera al instante t2, en el que el maestro acepta la respuesta generada. Indicar en este caso que la respuesta ha sido OKAY.

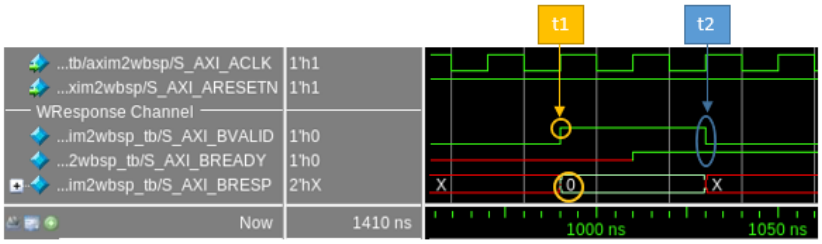

Figura 44 Simulación de transacción tipo *INCR*: *Write Response Channel*

Capítulo 5: DESARROLLO DEL ENTORNO DE VERIFICACIÓN UVM

El objetivo principal de este TFG es el desarrollo completo de un *testbench* UVM para verificar la funcionalidad de un DUV proporcionado por terceros. En este capítulo se abordará dicho desarrollo, en el que destaca la creación del IP de verificación necesario para completar el proceso de verificación del DUV seleccionado. Además de la estructura del entorno de verificación, en este capítulo se muestran los resultados obtenidos al finalizar el desarrollo del entorno, debido a que en este TFG no se aborda en mayor profundidad las distintas formas de simular el comportamiento del DUV y sus resultados correspondientes.

5.1. ESTRUCTURA DEL ENTORNO DE VERIFICACIÓN UVM

Como se ha mostrado en capítulos anteriores, para el desarrollo del entorno de verificación UVM se realizará una estructura y jerarquía de los componentes necesarios, comenzando por aquellos que se encuentran a más bajo nivel hasta los de más alto nivel. Para el caso de los componentes de bajo nivel, se mostrará el desarrollo de aquellos relacionados con la interfaz *Wishbone* del DUV. Estos son el componente UVM *Agent* y todos los componentes que integra. Los componentes relacionados con la interfaz AXI4 del módulo a verificar no se desarrollan en este TFG, ya que se partirá del QVIP realizado por Álvaro Moreno en su TFG, como se comenta en el apartado 3.5. Sin embargo, se comentará y se detallará el uso que se ha realizado de los mismos. Se destacará, principalmente, cómo a partir de estos QVIP se han generado los estímulos necesarios para activar la interfaz AXI4.

Debido al tiempo disponible y que no son relevantes para alcanzar el objetivo de este TFG, los componentes encargados de las tareas de monitorización se desarrollaron sin implementar ninguna funcionabilidad en su fase `run_phase`. En su lugar, se diseñaron a nivel de *black-box* para facilitar el conexionado con el resto de los componentes.

5.1.1. TRANSACCIONES

La transacción `wb_item` es la utilizada por los componentes encargados de la gestión del protocolo *Wishbone* del DUV. En este caso, para realizar el protocolo de *handshake* secuencia-*driver* se utilizan transacciones de este tipo. La Figura 45 muestra la implementación de esta transacción, que deriva del objeto `uvm_sequence_item` de UVM. Su función es actuar como contenedor de variables que se utilizarán para estimular y registrar las señales *Wishbone* del DUV, como se ha explicado en apartados anteriores. Además, se registra el objeto y las variables en la *Factory* de UVM.

```
2  class wb_item extends uvm_sequence_item;
3
4     localparam AXI_LSBS = $clog2(AXI4_RDATA_WIDTH)-3;
5     localparam AW = AXI4_ADDRESS_WIDTH - AXI_LSBS;
6     bit [(AW-1):0]                 addr_o;
7     rand bit [(AXI4_RDATA_WIDTH-1):0]   data_i;
8     bit [(AXI4_RDATA_WIDTH-1):0]   data_o;
9     bit                            cyc_o;
10    bit                            stb_o;
11    bit [(AXI4_RDATA_WIDTH/8-1):0] sel_o;
12    rand bit                       stall_i;
13    rand bit                       ack_i;
14    rand bit                       err_i;
15    bit                            we_o;
16    bit                            dummy;
17
18    `uvm_object_utils_begin(wb_item)
19       `uvm_field_int(addr_o, UVM_DEFAULT)
20       `uvm_field_int(data_i, UVM_DEFAULT)
21       `uvm_field_int(data_o, UVM_DEFAULT)
22       `uvm_field_int(stb_o, UVM_DEFAULT)
23       `uvm_field_int(sel_o, UVM_DEFAULT)
24       `uvm_field_int(stall_i, UVM_DEFAULT)
25       `uvm_field_int(ack_i, UVM_DEFAULT)
26       `uvm_field_int(err_i, UVM_DEFAULT)
27       `uvm_field_int(we_o, UVM_DEFAULT)
28    `uvm_object_utils_end
29
30    function new(string name = "wb_item");
31       super.new(name);
32    endfunction: new
33
34 endclass: wb item
```

Figura 45 Transacción `wb_item`

5.1.2. INTERFAZ VIRTUAL

Para poder estimular y recibir respuestas del DUV, es necesario que el entorno de verificación UVM haga uso de una interfaz virtual que realice el conexionado entre el componente UVM *Agent* y el módulo DUV.

La Figura 46 muestra el código de la interfaz virtual *Wishbone* desarrollada, que se denomina `wb_if` y se encuentra en el archivo `wb_if.sv`. Entre las líneas 8 y 17 se puede observar la declaración de las variables relacionadas con las señales del protocolo *Wishbone* de las que se hace uso en el DUV. Además, se muestran algunos parámetros de la interfaz que configuran estas señales. Los parámetros definidos se utilizan para determinar el ancho del bus de datos y direcciones. Estos se han definido de forma local, líneas 5 y 6, y hacen uso de los parámetros

genéricos del *testbench*, que se importan del paquete `test_params_pkg` (línea 3). Este paquete contiene los parámetros generales necesarios para la simulación del *testbench*, como el tamaño del bus de direcciones, `AXI4_ADDRESS_WIDTH`, o el bus de datos de lectura, `AXI4_RDATA_WIDTH`, entre otros.

```
1
2  interface wb_if (input logic clk, rst_n);
3     import test_params_pkg::*; //
4
5     localparam AXI_LSBS = $clog2(AXI4_RDATA_WIDTH)-3;
6     localparam AW = AXI4_ADDRESS_WIDTH - AXI_LSBS;
7
8     logic                             cyc_o;
9     logic                             stb_o;
10    logic                             we_o;
11    logic [(AW-1):0]                  addr_o;
12    logic [(AXI4_RDATA_WIDTH-1):0]    data_o;
13    logic [(AXI4_RDATA_WIDTH/8-1):0] sel_o;
14    logic                             stall_i;
15    logic                             ack_i;
16    logic [(AXI4_RDATA_WIDTH-1):0]    data_i;
17    logic                             err_i;
18
19 endinterface: wb_if
```

Figura 46 Interfaz virtual `wb_if`

5.1.3. Componente Agent

El componente UVM *Agent* desarrollado, que tiene por nombre `wb_agent` y se implementa en el archivo `wb_agent.svh`, se encarga de agrupar otros componentes UVM que interactúan con una interfaz específica, en este caso con la interfaz `wb_if` definida.

La Figura 47 muestra la implementación en UVM del componente UVM *Agent*. En primer lugar, se define el componente, heredado de la clase padre `uvm_agent`, línea 2. Posteriormente, se debe definir el comportamiento del agente, activo o pasivo. Esto se realiza mediante el uso de la variable `is_active`, línea 3. Iniciando esta variable al valor `UVM_ACTIVE`, se indica que este componente tendrá carácter activo, línea 3. Esta parametrización permite que al ejecutar la fase `build_phase` se implementen los tres componentes que agrupa, el componente UVM *Sequencer,* el componente UVM *Driver* y el componente UVM *Monitor*, líneas 17 a 27. Esto permite configurar dichos componentes para que se pueda hacer uso de las secuencias reactivas, convirtiendo al componente `wb_agent` en un agente esclavo reactivo. Por último, durante la fase `connect_phase`, se interconectan los componentes UVM *Sequencer* y UVM *Driver* a través de puertos TLM, tal y como se muestra entre la línea 29 y la línea 33.

```
class wb_agent extends uvm_agent;
   protected uvm_active_passive_enum is_active = UVM_ACTIVE;

   wb_sequencer sequencer;
   wb_driver    driver;
   wb_monitor   monitor;

   `uvm_component_utils_begin(wb_agent)
      `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
   `uvm_component_utils_end

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

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if(is_active == UVM_ACTIVE) begin
         sequencer = wb_sequencer::type_id::create("sequencer", this);
         driver = wb_driver::type_id::create("driver", this);
      end

      monitor = wb_monitor::type_id::create("monitor", this);

      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
   endfunction: build_phase

   function void connect_phase(uvm_phase phase);
      if(is_active == UVM_ACTIVE)
         driver.seq_item_port.connect(sequencer.seq_item_export);
      `uvm_info(get_full_name( ), "Connect stage complete.", UVM_LOW)
   endfunction: connect_phase
endclass: wb_agent
```

Figura 47 Componente `wb_agent`

A continuación, se profundizará en el desarrollo de los componentes que se construyen y conectan en el componente `wb_agent`, exceptuando el componente UVM *Monitor*. Esto es debido a que en este último caso el componente *Monitor* actúa como *black-box*, diseñado como un componente vacío que favorece el conexionado del resto de componentes.

5.1.3.1. COMPONENTE UVM *SEQUENCER*

El componente UVM *Sequencer* se denomina `wb_sequencer` en el archivo `wb_sequencer.svh` y su implementación se muestra en la Figura 48. La funcionalidad de este componente, en este caso, es muy básica y se implementa íntegramente en su clase padre por lo que únicamente se registra en la *Factory* y hereda las funciones de su clase padre, con las que es capaz de gestionar las transacciones. Aunque su implementación resulte sencilla, puede llegar a ser el componente más complicado de implementar, sobre todo cuando se ejecutan varias secuencias en paralelo sobre el mismo.

```
class wb_sequencer extends uvm_sequencer #(wb_item);

   `uvm_sequencer_utils(wb_sequencer)

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

Figura 48 Componente `wb_sequencer`

5.1.3.2. UVM *DRIVER*

El componente UVM *Driver*, denominado `wb_driver` dentro del archivo `wb_driver.svh`, se encarga de estimular al DUV y, al tener un comportamiento reactivo, envía la respuesta a las secuencias con la información obtenida a partir del análisis de la interfaz del DUV.

En la Figura 49 se puede observar la implementación del componente `wb_driver`. Sus dos funciones principales son la fase `build_phase` y la fase `run_phase`. En la fase `build_phase`, el componente accede a la base de datos usando el método `get` de la clase `uvm_config_db` para obtener la interfaz `wb_if` que utilizará para la comunicación con el DUV, líneas 11 a 16. En caso de no existir dicha interfaz, la generación del *testbench* se aborta al ejecutar la macro `uvm_fatal` que, además, saca el mensaje de error asociado a este evento. En la fase `run_phase` se llama a dos tareas para que se ejecuten por hilos paralelos en tiempo de simulación, la tarea `reset()` y la tarea `get_and_drive()`. Para su ejecución en paralelo, dichas tareas se han definido dentro de una estructura *fork-join* de *SystemVerilog*.

```
2  class wb_driver extends uvm_driver #(wb_item);
3     virtual wb_if vif;
4
5     `uvm_component_utils(wb_driver)
6
7     function new(string name, uvm_component parent);
8        super.new(name, parent);
9     endfunction: new
10
11    function void build_phase(uvm_phase phase);
12       super.build_phase(phase);
13       if(!uvm_config_db#(virtual wb_if)::get(this, "", "wb_if", vif))
14          `uvm_fatal("NOVIF", {"virtual interface must be set for: ", get_full_name( ), ".vif"})
15       `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
16    endfunction
17
18    virtual task run_phase(uvm_phase phase);
19       super.run_phase(phase);
20       fork
21          reset( );
22          get_and_drive( );
23       join
24    endtask: run_phase
25
```

Figura 49 Componente `wb_driver`

La tarea `reset()` se muestra en la Figura 50 y su funcionamiento consiste en establecer las señales de salida del agente esclavo con un valor predeterminado tras detectar un flanco de bajada de la señal de *reset* de la interfaz. Debido al uso de la sentencia `forever` (línea 28), esta tarea está continuamente esperando la detección de un *reset* del sistema para actuar en consecuencia.

```
26    virtual task reset( );
27       `uvm_info(get_type_name( ), "Resetting signals ... ", UVM_LOW)
28       forever begin
29          @(negedge vif.rst_n);
30          vif.ack_i = 1'b0;
31          vif.stall_i = 1'b0;
32          vif.err_i = 1'b0;
33       end
```

Figura 50 Tarea `reset()` del componente `wb_agent`

Debido a la complejidad de su implementación y su relevante papel junto a las secuencias para el funcionamiento reactivo del agente, la tarea `get_and_drive()` se explicará en profundidad más adelante.

5.1.4. COMPONENTE QVIP AXI4

Tal y como se ha venido comentando, la interfaz AXI4 del DUV en el presente Trabajo Fin de Grado será estimulada haciendo uso de los QVIP de *Mentor Graphcis*. En este caso se usará un QVIP AXI4 en modo maestro, proporcionado por el citado proveedor EDA. Este QVIP es altamente configurable y su integración en un entorno UVM es inmediata.

La Figura 51 muestra el código de diseño de un QVIP AXI4. Estos QVIP se modulan mediante el *token* `module` de *Verilog*. En este caso particular, el QVIP tiene 8 parámetros y se denomina `axi4_master`. Estos parámetros son fácilmente interpretables a partir de su definición. Así, por ejemplo, el parámetro `ADDR_WIDTH` especifica el ancho del bus de direcciones; el parámetro `RDATA_WIDTH` especifica el ancho del bus de datos de lectura. Sin embargo, hay que hacer una especial mención al parámetro `IF_NAME`. Este parámetro especifica el nombre de la interfaz virtual a la cual estará conectado este módulo maestro. Este parámetro deberá ser establecido por el módulo `top`, ya que si no se establece se producirá un error fatal y se abortará la ejecución del *testbench*. Tal y como se muestra en la línea 34 de la citada figura, este parámetro no se inicializa (realmente se inicializa a un valor "`null`"), por lo que, de no ser inicializado, se producirá el error mencionado.

```
16 // Input Parameters:
17 // ADDR_WIDTH      - Indicates address bus width.
18 // RDATA_WIDTH     - Indicates read data bus width.
19 // WDATA_WIDTH     - Indicates write data bus width.
20 // ID_WIDTH        - Indicates ID width.
21 // USER_WIDTH      - Indicates user signals bus width.
22 // REGION_MAP_SIZE - Indicates region map size
23 // IF_NAME         - Indicates interface name for which this module
24 //                   will be instantiated.
25
26 // (inline source)
27 module axi4_master #(// mentor configurator value range { 8 64 }
28           ADDR_WIDTH  = 32, // mentor configurator value values { 8 16 32 64 128 256 512 1024 }
29           RDATA_WIDTH = 32, // mentor configurator value values { 8 16 32 64 128 256 512 1024 }
30           WDATA_WIDTH = 32, // mentor configurator value range { 1 30 }
31           ID_WIDTH    = 4,  // mentor configurator value range { 1 64 }
32           USER_WIDTH  = 4,  // mentor configurator value range { 1 16 }
33           REGION_MAP_SIZE = 16,
34           string IF_NAME = "null",
35           string PATH_NAME ="uvm_test_top*")
36          (
37          // clock and reset signals
38          input                             ACLK,
39          input                             ARESETn,
40          // write address channel signals
41          output                            AWVALID,
42          output [ADDR_WIDTH - 1:0]         AWADDR,
43          output [2:0]                      AWPROT,
44          output [3:0]                      AWREGION,
45          output [7:0]                      AWLEN,
46          output [2:0]                      AWSIZE,
47          output [1:0]                      AWBURST,
48          output                            AWLOCK,
```

Figura 51 Código QVIP AXI4

A partir de la línea 37 se describen todos los puertos de entrada y salida del módulo. Estos puertos corresponden a todas y cada una de las señales del protocolo AXI4. El conexionado de todos estos puertos con las señales del DUV a verificar se deberá realizar en el módulo `top`.

La Figura 52 muestra otro fragmento de la definición del módulo QVIP. En este fragmento se observa cómo el módulo `axi4_master` hace uso de la clase `uvm_config_db`, más en concreto del método `set` de esta clase, para almacenar en la base de datos el puntero a la interfaz virtual definida, tal y como se muestra en las líneas 168 a 171. El tercer parámetro de este método corresponde a la denominación de la interfaz virtual.

```
164   initial begin
165     // Put the interface into the UVM configuration structure,
166     // for retrieval in the UVM test build method
167
168     uvm_config_db #(axi4_if_t)::set(null,
169                                     PATH_NAME,
170                                     IF_NAME,
171                                     axi4_if);
172     // Master end should be TLM
173     axi4_if.axi4_set_master_abstraction_level(1'b0,1'b1);
174   end
175 endmodule
```

Figura 52 Definición del QVIP AXI4 en la base de datos UVM

5.1.5. COMPONENTE *ENVIRONMENT*

Una vez desarrollado el agente reactivo, se unirá con el QVIP y ambos componentes se englobarán en un mismo componente, el componente UVM *Environment*. Este componente UVM *Environment* se denomina `qvip_axim2wbsp_env` y se ha implementado en el archivo `qvip_axim2wbsp_env.svh`. Este componente actúa como contenedor. La Figura 53 muestra el componente `qvip_axim2wbsp_env`, que tiene como función principal la función `build_phase`, que se implementa de manera externa. Hay que destacar que este componente hace uso de un objeto `qvip_axim2wbsp_env_config` para la configuración del componente UVM *Agent* suministrado para *AXI4*, línea 10. En esta línea se muestra la declaración de este objeto de configuración.

```
 8 class qvip_axim2wbsp_env extends uvm_component;
 9    `uvm_component_utils(qvip_axim2wbsp_env)
10    qvip_axim2wbsp_env_config cfg;
11    // Agent handles
12
13    axi4_master0_agent_t axi4_master0;
14    wb_agent wbagent;
15
16    function new
17      (
18       string name = "qvip_axim2wbsp_env",
19       uvm_component parent = null
20       );
21       super.new(name, parent);
22    endfunction
23
24    extern function void build_phase (uvm_phase phase);
25
26 endclass: qvip_axim2wbsp_env
```

Figura 53 Componente qvip_axim2wbsp_env

La fase `build_phase` se muestra en la Figura 54 y su papel es crear los componentes UVM *Agent*, tanto el suministrado para la interfaz AXI4 como el desarrollado para *Wishbone*.

Al comienzo de la función se comprueba si el objeto de configuración definido está inicializado o, si por el contrario, no está inicializado sino que es `null`. En este último caso, si no está inicializado, se accede a la base de datos a través del método `get` de la clase `uvm_config_db`, para buscar la cadena `env_config` y obtener así el manejador del objeto de configuración. Este manejador se asigna a la variable `cfg` definida en la clase. En caso de no encontrar la cadena `env_config` en la base de datos se abortará la creación del *testbench*.

Posteriormente, en la línea 36 se crea el agente asociado al interfaz AXI4 mediante el uso de la función `create` de la *Factory*.

En la línea 37, se asigna a dicho agente el objeto de configuración obtenido desde la base de datos. Esta asignación se realiza usando el método `set_mvc_config`.

En la línea 39, y usando la misma función `create`, se crea el agente asociado al interfaz *Wishbone*.

```
28 function void qvip_axim2wbsp_env::build_phase (uvm_phase phase);
29    if ( cfg == null )
30      begin
31         if ( !uvm_config_db #(qvip_axim2wbsp_env_config)::get(this, "", "env_config", cfg) )
32           begin
33              `uvm_error("build_phase", "Unable to find the env config object in the uvm_config_db")
34           end
35      end
36    axi4_master0 = axi4_master0_agent_t::type_id::create("axi4_master0", this );
37    axi4_master0.set_mvc_config(cfg.axi4_master0_cfg);
38
39    wbagent = wb_agent::type_id::create("wb_agent", this );
40
41 endfunction: build_phase
```

Figura 54 Fase `build_phase` del componente `qvip_axim2wbsp_env`

5.1.6. SECUENCIAS

Las transacciones que estimulan las entradas del DUV se generan en los objetos UVM *Sequence*, también llamados secuencias, que se introducen en los componentes UVM *Sequencer* a partir del componente UVM *Test*. En este caso, las secuencias que estimulan al DUV se separan según las interfaces del mismo. Además, su implementación es distinta debido a que unas tienen un comportamiento proactivo y otras un comportamiento reactivo. La explicación de las secuencias reactivas para el agente esclavo *Wishbone* irá acompañada del desarrollo de la tarea `get_and_drive()` del componente `wb_driver`, como se comentó previamente.

5.1.6.1. SECUENCIAS AXI4

Para la estimulación de la interfaz AXI4 del DUV utilizado en este Trabajo Fin de Grado, se ha hecho uso de la librería de secuencias que el proveedor *Mentor Graphics* pone a disposición con los modelos QVIP. Hay que destacar que, si bien se dispone de esta librería, el ingeniero de verificación debe desarrollar las secuencias particulares, ya que aquellas solo sirven de secuencias base. En este Trabajo Fin de Grado se creó una secuencia base, denominada `qvip_axim2wbsp_vseq_base`, cuyo código se muestra en la Figura 55.

```
2 // File: qvip_axim2wbsp_vseq_base.svh
3 //
4 // Generated from Mentor VIP Configurator (20190214)
5 // Generated using Mentor VIP Library ( 2019_1.1 : 02/24/2019:07:17 )
6 //
7 class qvip_axim2wbsp_vseq_base extends mvc_sequence;
8     `uvm_object_utils(qvip_axim2wbsp_vseq_base)
9     // Handles for each of the target (QVIP) sequencers
10
11    mvc_sequencer axi4_master0;
12    function new
13    (
14        string name = "qvip_axim2wbsp_vseq_base"
15    );
16        super.new(name);
17    endfunction
18
19    task body;
20    endtask: body
21
22 endclass: qvip_axim2wbsp_vseq_base
```

Figura 55 Secuencia `qvip_axim2wbsp_vseq_base`

El análisis de este código muestra:

- En la línea 7 se define la secuencia, derivándola de la clase base `mvc_sequence`, proporcionada por *Mentor Graphics*.
- En la línea 8 se registra esta secuencia básica en la *Factory* de UVM para poder ser utilizada posteriormente.
- En la línea 11 se referencia el componente `axi4_master0`, que será el componente QVIP a utilizar en el módulo *top* para estimular la interfaz AXI4. Hay que resaltar que la variable `axi4_master0`, no es más que un manejador de tipo `mvc_sequencer` que deberá ser inicializado antes de ejecutar la secuencia. Esta inicialización se realiza en el componente test que será explicado en un título posterior.
- Las líneas 12 a 15 muestran la definición del constructor `new` de este objeto.
- Por último, las líneas 19 y 20 muestran la definición de la tarea `body` de este objeto. Al tratarse de una clase base, resulta normal que esta tarea quede vacía, siendo implementada en las tareas que deriven de esta.

Tras la definición de la secuencia base, se describe una secuencia, denominada `qvip_axim2wbsp_vseq_seq1`, encargada de excitar la interfaz AXI4 del DUV. Se trata de una secuencia proactiva que envía transacciones a la interfaz AXI4 actuando en modo maestro. Analizando el código presentado en la Figura 56, se muestra:

- En la línea 7 se define la secuencia, derivándola de la clase base `qvip_axim2wbsp_vseq_base`.
- En la línea 8 se registra esta secuencia básica en la *Factory* de UVM para poder ser utilizada posteriormente.
- Entre las líneas 10 y 16 se definen los siete tipos de transacciones que podrán utilizarse en la descripción de la secuencia. Estos tipos son: transacciones tipo *FIXED*, transacciones tipo *INCR*, transacciones tipo *WRAP* y una transacción tipo *EXCL*. Para los tres primeros tipos, se definen dos transacciones, una de escritura y otra de lectura. Las definiciones de estos tipos usan los tipos bases proporcionados por *Mentor Graphics* para las transacciones asociadas al protocolo AXI4. Por otro lado, la definición de estos

tipos facilita la creación de cualquier tipo de secuencia, tal y como se mostrará en la descripción de la tarea `body` de esta secuencia.

- Las líneas 18 a 23 muestran la definición del constructor `new` de este objeto.
- Por último, la línea 25 referencia la declaración de la tarea `body` de este objeto. En este caso, debido a que la extensión de la misma puede ser considerable, se ha decidido declararla como una tarea externa a la clase.

```
 7 class qvip_axim2wbsp_vseq_seq1 extends qvip_axim2wbsp_vseq_base;
 8    `uvm_object_utils(qvip_axim2wbsp_vseq_seq1)
 9
10    typedef axi4_fixed_wr_deparam_seq  fixed_wr_t;
11    typedef axi4_fixed_rd_deparam_seq  fixed_rd_t;
12    typedef axi4_incr_wr_deparam_seq   incr_wr_t;
13    typedef axi4_incr_rd_deparam_seq   incr_rd_t;
14    typedef axi4_wrap_wr_deparam_seq   wrap_wr_t;
15    typedef axi4_wrap_rd_deparam_seq   wrap_rd_t;
16    typedef axi4_excl_rw_deparam_seq   excl_rw_t;
17
18    function new
19      (
20       string name = "qvip_axim2wbsp_vseq_seq1"
21       );
22       super.new(name);
23    endfunction
24
25    extern task body;
26
27 endclass: qvip_axim2wbsp_vseq_seq1
```

Figura 56 Secuencia `qvip_axim2wbsp_vseq_seq1`

La Figura 57 muestra la primera parte del código correspondiente a la implementación de la tarea de la secuencia comentada anteriormente. Analizando el código se muestra:

- Entre las líneas 32 y 37 se crean las siete posibles transacciones a utilizar durante la ejecución de la transacción. Cada una de estas transacciones se crean usando el método `create` asociado al mecanismo de *Factory*.
- En la línea 39 se llama a la función `body` de la clase padre. Si bien el método `body` de su clase padre no implementa ninguna función, es normal realizar su llamada ya que facilita su reutilización en otros escenarios.
- A partir de la línea 40 se describe el comportamiento de la secuencia a la hora de manejar a la interfaz AXI4. En este caso, el resto del código se divide en dos grandes bloques, uno de escritura y el otro de lectura.
- Al bloque de escritura se describe a partir de la línea 42, con la inicialización de una transacción de escritura, denominada `incr_wr`, y a la que se le asocia el componente UVM *Sequencer* `axi4_master0` mediante la función `set_sequencer`.
- Posteriormente, en la línea 43 se fija a `0` el valor del campo `ID`, correspondiente al parámetro `AWID` del protocolo AXI4.
- En la línea 44 se aleatoriza el resto de los campos de la transacción de escritura mediante el uso del método `randomize`. En este caso, la aleatorización restringe los valores de dirección, campo `addr`, y de longitud de los datos, variable `size` del campo `wr_data`. Estos dos valores se fijan a `'h4500` y `16`, respectivamente.

- Una vez inicializada la transacción, esta se ejecuta sobre el componente UVM *Sequencer* asignado mediante el uso del método `start` definido en la transacción base. Este paso se muestra en la línea 48. Este método quedará bloqueado hasta que el DUV responda que ha recibido la transacción enviada.
- Una vez recibida la respuesta por parte del DUV a la transacción de escritura enviada, se procede con la etapa de lectura. Esta etapa se describe a partir de la línea 51, con la inicialización de una transacción de escritura, denominada `incr_rd`, y a la que se le asocia el componente UVM *Sequencer* `axi4_master0` mediante la función `set_sequencer`.
- Posteriormente, en la línea 53 se fija a `0` el valor del campo `ID`, correspondiente en este caso al parámetro `ARID` del protocolo AXI4.
- En la línea 54 se fija a `16`, coincidiendo con el tamaño de la transacción de escritura, la longitud de los datos a leer mediante el campo `rd_bytes`.
- Una vez inicializada la transacción, esta se ejecuta sobre el componente UVM *Sequencer* asignado mediante el uso del método `start` definido en la transacción base. Este paso se muestra en la línea 57. Este método quedará bloqueado hasta que el DUV responda que ha recibido la transacción enviada.

```
29 task qvip_axim2wbsp_vseq_seq1::body;
30
31    // Creates sequences
32    fixed_wr_t    fixed_wr    = fixed_wr_t::type_id::create("fixed_wr");
33    fixed_rd_t    fixed_rd    = fixed_rd_t::type_id::create("fixed_rd");
34    incr_wr_t     incr_wr     = incr_wr_t::type_id::create("incr_wr");
35    incr_rd_t     incr_rd     = incr_rd_t::type_id::create("incr_rd");
36    wrap_rd_t     wrap_rd     = wrap_rd_t::type_id::create("wrap_rd");
37    wrap_wr_t     wrap_wr     = wrap_wr_t::type_id::create("wrap_wr");
38
39    super.body();
40
41    //master0 write
42    incr_wr.set_sequencer(axi4_master0);
43    incr_wr.id = 0;
44    if(!incr_wr.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
45      `uvm_error(this.get_full_name(),"Randomisation failure");
46
47    `uvm_info("SEQ", "Running inc_wr", UVM_LOW)
48    incr_wr.start(axi4_master0);
49
50
51    incr_rd.set_sequencer(axi4_master0);
52    incr_rd.id = 0;
53    incr_rd.addr= 32'h0000_4500;
54    incr_rd.rd_bytes = 16;
55
56    `uvm_info("SEQ", "Running inc_rd", UVM_LOW)
57    incr_rd.start(axi4_master0);
```

Figura 57 Primera parte de la tarea `body` de `qvip_axim2wbsp_vseq_seq1`

La Figura 58 muestra la segunda parte del código correspondiente a la implementación de la tarea de la secuencia comentada anteriormente. En este segundo fragmento se utilizan tres transacciones de tipo *FIXED*, dos de lectura y una de escritura. Con esto se pretende comprobar que los datos leídos no son siempre los mismos y se modifican en función de la dirección. La secuenciación de las operaciones es la que se lista a continuación. Debido a que las operaciones

son las mismas que las descritas para las transacciones de tipo *INCR*, sólo se detallarán los aspectos más relevantes.

- Líneas 59 a 64. Escritura de `16` bytes en modo *FIXED* en la dirección `'h1000`.
- Líneas 66 a 72. Escritura de `16` bytes en modo *FIXED* en la dirección `'h2000`. En este caso, y al no especificar los datos a escribir, estos se asignarán de forma aleatoria debido al uso del método `randomize`.
- Líneas 74 a 80. En este código se ejecuta una transacción de lectura *FIXED* en la dirección `'h2000`. En este caso, al coincidir, tanto la dirección como la longitud de la secuencia, con la secuencia anterior, los datos leídos deberán ser los mismos que los escritos anteriormente.

```
59    fixed_rd.set_sequencer(axi4_master0);
60    fixed_rd.id = 0;
61    fixed_rd.addr= 32'h0000_1000;
62    fixed_rd.rd_bytes = 16;
63    `uvm_info("SEQ", "Running fix_rd", UVM_LOW)
64    fixed_rd.start(axi4_master0);
65
66    fixed_wr.set_sequencer(axi4_master0);
67    fixed_wr.id = 0;
68    if(!fixed_wr.randomize() with {addr == 32'h0000_2000; wr_data.size == 16;})
69      `uvm_error(this.get_full_name(),"Randomisation failure");
70
71    `uvm_info("SEQ", "Running fix_wr", UVM_LOW)
72    fixed_wr.start(axi4_master0);
73
74    fixed_rd.set_sequencer(axi4_master0);
75    fixed_rd.id = 0;
76    fixed_rd.addr= 32'h0000_2000;
77    fixed_rd.rd_bytes = 16;
78
79    `uvm_info("SEQ", "Running fix_rd", UVM_LOW)
80    fixed_rd.start(axi4_master0);
```

Figura 58 Segunda parte de la tarea `body` de `qvip_axim2wbsp_vseq_seq1`

Por último, la Figura 59 muestra la tercera y última parte del código correspondiente a la implementación de la tarea de esta secuencia. En este tercer fragmento se utilizan dos transacciones de tipo *FIXED*, la primera, e *INCR*, la segunda. Lo que se pretende en este caso es generar una escritura de 16 bytes en una dirección fija y leer, a partir de esa dirección, usando una transacción que vaya incrementando las direcciones de lectura. La secuenciación de las operaciones es la que se lista a continuación. Debido a que las operaciones son las mismas que las descritas para las transacciones, sólo se detallarán los aspectos más relevantes.

- Líneas 82 a 88. Escritura de `16` bytes en modo *FIXED* en la dirección `'h3000`. Se hace uso del método `randomize` para la aleatorización de los campos de la transacción.
- Líneas 90 a 96. Lectura de `16` bytes en modo *INCR* a partir de la dirección `'h3000`. El campo `ID`, así como la dirección de comienzo de la lectura y el tamaño de la transacción se han mantenido igual que en el caso de la transacción anterior.

```
82    fixed_wr.set_sequencer(axi4_master0);
83    fixed_wr.id = 0;
84    if(!fixed_wr.randomize() with {addr == 32'h0000_3000; wr_data.size == 16;})
85      `uvm_error(this.get_full_name(),"Randomisation failure");
86
87    `uvm_info("SEQ", "Running fix_wr", UVM_LOW)
88    fixed_wr.start(axi4_master0);
89
90     incr_rd.set_sequencer(axi4_master0);
91    incr_rd.id = 0;
92    incr_rd.addr= 32'h0000_3000;
93    incr_rd.rd_bytes = 16;
94
95    `uvm_info("SEQ", "Running inc_rd", UVM_LOW)
96    incr_rd.start(axi4_master0);
97
98
99 endtask: body
```

Figura 59 Tercera parte de la tarea `body` de `qvip_axim2wbsp_vseq_seq1`

5.1.6.2. SECUENCIAS *WISHBONE*

En el caso de la interfaz *Wishbone* del DUV, se ha creado una librería de secuencias para estimular dicha interfaz. En este Trabajo Fin de Grado, al igual que en la interfaz AXI4, se parte de una secuencia base, denominada `base_seq`, de la que heredarán el resto de las secuencias y cuya implementación se muestra en la Figura 60.

```
 2 class base_seq extends uvm_sequence #(wb_item);
 3    `uvm_object_utils(base_seq)
 4
 5    localparam DATA_WIDTH = 32;
 6
 7    // Define memory storage
 8    bit [DATA_WIDTH-1:0] mem[*];
 9    bit [DATA_WIDTH-1:0] datar;
10    bit                  ack;
11    bit                  stall;
12    bit [7:0]            stall_vec;
13    int                  num_seq;
14    int                  num_seq_rec;
15    bit                  dummy;
16
17    function new(string name = "base_seq");
18       super.new(name);
19    endfunction:new
20
21    virtual              task body( );
22    endtask: body
23
24 endclass: base_seq
```

Figura 60 Secuencia `base_seq` para la interfaz *Wishbone*

El análisis del código que se puede observar en la figura anterior muestra:

- En la línea 2 se define la secuencia, que deriva de la clase base proporcionada por UVM y denominada `uvm_sequence`. Además, se indica que se utilizará transacciones del tipo `wb_item`.

- En la línea 3 se registra esta secuencia básica en la *Factory* de UVM para poder ser utilizada posteriormente.
- Entre las líneas 5 y 15, se define un parámetro que ajusta el ancho del bus de datos y se crean una serie de variables que serán utilizadas posteriormente en el resto de secuencias. Estas variables están relacionadas con la configuración de la secuencia.
- Las líneas 17 a 19 muestran la definición del constructor `new` de este objeto.
- Por último, las líneas 19 y 20 muestran la definición de la tarea `body` de este objeto. Al tratarse de una clase base, resulta normal que esta tarea quede vacía, siendo implementada en las tareas que deriven de esta.

Una vez definida la secuencia base, se ha implementado una secuencia denominada `simple_sequence`. Esta secuencia es una secuencia reactiva que se encarga de recibir las transacciones que le llegan del DUV y reaccionar en consecuencia estimulando la interfaz *Wishbone*. El comportamiento reactivo de la secuencia se ha implementado siguiendo el modelo reactivo con transacción *dummy*, explicado en el apartado 3.4.5.3. . Además, para el funcionamiento reactivo de la secuencia es vital el uso del protocolo de *handshake* explicado en el apartado 3.4.5.2. y, en este caso, representado en la Figura 22. Por esta razón también se explicará la tarea `get_and_drive` definida en el componente `wb_driver` mostrado anteriormente. Para que la explicación resulte lo más clara y fluida posible, en primer lugar, se comentará la implementación de la secuencia y, en segundo lugar, la implementación de la tarea `get_and_drive`. En ambos casos se hará alusión a métodos o aspectos tanto del componente *driver* como de la secuencia con el fin de exponer la comunicación entre ellos mediante el protocolo de *handshake*.

La Figura 61 muestra la primera parte del código correspondiente a la implementación de la secuencia `simple_sequence`. Analizando el código se muestra:

- En la línea 27 se define la secuencia derivándola de la clase base `base_seq`.
- En la línea 28 se registra esta secuencia en la *Factory* de UVM para poder ser utilizada posteriormente.
- Por último, entre las líneas 29 y 31 se muestra la definición del constructor `new` de este objeto.

```
27 class simple_sequence extends base_seq;
28    `uvm_object_utils(simple_sequence)
29    function new(string name = "simple_sequence");
30       super.new(name);
31    endfunction:new
32
```

Figura 61 Encabezado de la secuencia `simple_sequence` para la interfaz *Wishbone*

Una vez definido el constructor comienza la implementación de la tarea `body` de la secuencia. La Figura 62 muestra la primera parte del código de esta tarea, que al analizarlo muestra:

- En la línea 35 se crea un objeto de tipo `wb_item` denominado `req`, que corresponde a una transacción inicialmente vacía.
- En la línea 36 se asigna un nivel alto a la variable `dummy` que se utilizará más adelante para diferenciar entre la transacción *dummy* y el resto de transacciones.

- En la línea 38 se muestra información por pantalla mediante la macro `uvm_info que indica la dirección inicial de la transacción.
- A partir de la línea 40 y hasta el final de la tarea, se hace uso de una estructura while de *SystemVerilog* que repite el código que se encuentra implementado dentro de la estructura mientras se cumpla una condición inicial. En este caso, la condición implica que no se saldrá de la estructura mientras el número de transacciones recibidas (num_seq_rec) sea menor al número de transacciones total que se espera recibir (num_seq). Este último parámetro está definido en la secuencia base y será configurado por el test antes de su ejecución.
- Entre las líneas 41 y 43 se realiza el primer paso de la ejecución de la secuencia, en el que se realiza la llamada al método **start_item**, utilizando el objeto req creado anteriormente, para iniciar por parte de la secuencia el protocolo de *handshake* con el componente UVM *Driver*. Este método es bloqueante hasta que el componente UVM *Driver* haga uso del método **get_next_item**. Además, se muestra por pantalla que se va a realizar la llamada a este método.
- Por último, entre las líneas 45 y 56 se encuentra una fase de identificación de la transacción recibida, según sea una transacción de escritura o lectura. Mientras la variable dummy esté a nivel alto, no se realiza este análisis ya que indica que la secuencia está gestionando la transacción *dummy* para poder recibir la primera transacción con información por parte del componente UVM *Driver*. Esto sucede una única vez al principio de la ejecución de la secuencia. Una vez se ha recibido la primera transacción por parte del componente UVM *Driver*, como se mostrará posteriormente, se ejecuta este análisis.
    - En la línea 46 se analiza si la transacción es de escritura si cumple la condición de no ser *dummy* y que tanto las señales we_o y stb_o de la respuesta del wb_driver estén a nivel alto. También se comprueba que no haya *stall* activo en ese momento, pero para esta secuencia la señal stall estará en todo momento a nivel bajo.
    - En la línea 47, si se cumple la condición anterior, se almacena el dato recibido en la respuesta del *driver* en una dirección específica, también contenida en la respuesta del *driver*, del array utilizado como memoria. Esta respuesta del *driver* será analizada posteriormente.
    - En la línea 48 se utiliza la macro `uvm_info para mostrar que se ha realizado un ciclo de escritura en la dirección deseada. Además, se muestra la dirección recibida y el dato escrito.
    - En la línea 52 se comprueba si se trata de una transacción de escritura, de la misma forma que en la línea 46, pero con la señal we_o a nivel bajo.
    - En la línea 53, si se cumple la condición anterior, se recoge el dato de la dirección de memoria especificada en la respuesta del *driver*. En caso de no haber un dato válido en esa dirección, devuelve un dato sin inicializar.
    - En la línea 48 se utiliza la macro `uvm_info para mostrar que se ha realizado un ciclo de lectura de la dirección deseada. Además, se muestra la dirección recibida y el dato leído.

```
33    virtual task body( );
34       // Creates the transaction
35       req = wb_item::type_id::create("req");
36       dummy = 1'b1;
37
38       `uvm_info("SEQUENCE-Dbg", $sformatf("Initial rsp.adr_o=0x%0h", req.addr_o), UVM_MEDIUM)
39
40       while(num_seq_rec < num_seq) begin
41          // Step-1 A call to start_item method
42          `uvm_info("SEQUENCE-Dbg", $sformatf("About to call start_item()"), UVM_MEDIUM)
43          start_item(req);
44
45          // If Write cycle, write data in memory
46          if (!dummy && rsp.we_o && rsp.stb_o && !rsp.stall_i) begin
47             mem[rsp.addr_o] = rsp.data_o;
48             `uvm_info("SEQUENCE-Dbg", $sformatf("Write Cycle mem[0x%0h] = 0x%0h",
49                                                 rsp.addr_o, mem[rsp.addr_o]), UVM_MEDIUM)
50          end
51          // If Read cycle, read data from memory if exists
52          else if (!dummy && !rsp.we_o && rsp.stb_o && !rsp.stall_i) begin
53             datar = (mem.exists(rsp.addr_o)) ? mem[rsp.addr_o] : 'x;
54             `uvm_info("SEQUENCE-Dbg", $sformatf("Read Cycle mem[0x%0h] = 0x%0h",
55                                                 rsp.addr_o, datar), UVM_MEDIUM)
56          end
```

Figura 62 Primera parte de la tarea `body` de la secuencia `simple_sequence`

La Figura 63 muestra la segunda parte del código correspondiente a la implementación de la tarea `body` de la secuencia comentada anteriormente. En este segundo fragmento se comenta el resto de la ejecución de la secuencia, así como la interacción de esta con el protocolo de *handshake* y, por lo tanto, con el componente UVM *Driver* denominado `wb_driver`. Al analizar del código se puede observar:

- A partir de la línea 58 se describe el segundo paso de la ejecución de la secuencia, la aleatorización del objeto `req`.
- En la línea 59, si no se trata de la transacción *dummy*, se asigna el valor de la señal `stb_o` obtenida de la respuesta del *driver* a la variable `ack`, que se utilizará para generar la señal `ack_i`.
- En las líneas 61 y 62 se realiza la aleatorización con restricciones de la transacción `req`. Estas restricciones son la señal `data_i`, que utilizará el dato leído en memoria en caso de lectura, la señal `ack_i`, que utilizará el valor del registro comentado en la línea anterior, y la señal `stall_i`, que en esta secuencia estará a nivel bajo en todo momento.
- En la línea 63 se hace uso de la macro `` `uvm_info `` para mostrar por pantalla que se ha realizado la aleatorización de la transacción con las restricciones especificadas.
- Entre las líneas 67 y 69 se implementa el siguiente paso del protocolo, correspondiente al envío del objeto `req`. Para ello se hace la llamada al método **`finish_item`** al que se le pasa como argumento la transacción `req`. Además, se utiliza la macro `` `uvm_info `` para indicar que se realiza esta llamada. El método `finish_item` es bloqueante y se mantiene a la espera de que el *driver* envíe esta transacción a través de la interfaz del DUV y, finalmente, ejecute el método `item_done`. En este caso, el componente *driver* realizará la llamada a este método devolviendo un objeto del mismo tipo que `req` denominado `rsp`. Este objeto, o transacción en este caso, contendrá todos los campos necesarios para volver a generar una nueva transacción a enviar al componente *driver*.

- Entre las líneas 72 y 76 se describe el cuarto y último paso de la ejecución de la secuencia. Este paso corresponde a la obtención de la transacción generada por el componente *driver*. Para ello, en la línea 73 se hace uso del método **`get_response`**, que utiliza el objeto `rsp` como transacción recibida. Este método es bloqueante y finalizara una vez el compnente *driver* envíe la respuesta haciendo uso del método **`item_done`**. Además, se utiliza la macro `` `uvm_info `` para indicar que se va a llamar a este método como para mostrar que se ha recibido una respuesta y el valor de sus señales.
- En la línea 78 se establece el valor de la variable `dummy` a nivel bajo, indicando que se acabó el ciclo *dummy*.
- Por último, entre las líneas 81 y 84 se incrementa el valor de la variable `num_seq_rec` si las señales `stb_o` y `stall_i` están a nivel bajo, lo que indica que se ha ejecutado una fase válida dentro del ciclo *Wishbone*. Además, se muestra por pantalla que se ha realizado este incremento mediante la macro `` `uvm_info ``.

```
58          // Step-2 A call to randomize
59          ack = (dummy) ? 1'b0 : rsp.stb_o;
60
61          assert(req.randomize( ) with {data_i == datar; ack_i == local::this.ack;
62                stall_i == 1'b0;});
63          `uvm_info("SEQUENCE-Dbg",
64                $sformatf("Tx randomize with data_i=0x%0h, ack_i=%0b, stall_i=%0b",
65                            req.data_i, req.ack_i, req.stall_i), UVM_MEDIUM)
66
67          // Step-3 A call to finish_item
68          `uvm_info("SEQUENCE-Dbg", $sformatf("About to call finish_item()"), UVM_MEDIUM)
69          finish_item(req);
70
71          // Step-4 A call to get_response
72          `uvm_info("SEQUENCE-Dbg", $sformatf("About to call get_response()"), UVM_MEDIUM)
73          get_response(rsp);
74          `uvm_info("SEQUENCE-Dbg",
75                $sformatf("Get response with we_o=%0b, addr_o=0x%0h, data_o=0x%0h, data_i=0x%0h,",
76                            rsp.we_o, rsp.addr_o, rsp.data_o, rsp.data_i), UVM_MEDIUM)
77
78          dummy = 1'b0;
79
80          // Update num_seq_rec
81          if (!rsp.stb_o && !rsp.stall_i) begin
82             num_seq_rec++;
83             `uvm_info("SEQUENCE-Dbg", $sformatf("Increasing num_seq_rec"), UVM_MEDIUM)
84          end
85
86       end
87    endtask: body
```

Figura 63 Segunda parte de la tarea `body` de la secuencia `simple_sequence`

Una vez vista la secuencia `simple_secuence`, la Figura 64 muestra la primera parte de la implementación de la tarea `get_and_drive`, que realiza la parte del protocolo *handshake* correspondiente al componente `wb_driver`. Esta tarea es esencial para que el comportamiento reactivo de la secuencia sea el adecuado. El resto del código relacionado con el componente UVM *Driver* y su explicación se encuentran en el apartado 5.1.3.2. . Analizando el código se muestra:

- En la línea 37 se muestra que el resto del código de la tarea se encuentra dentro de una estructura `forever` de *SystemVerilog*. Esta estructura provoca que el código que contiene se repita indefinidamente. Esto conlleva que el componente `wb_driver` esté ejecutando esta tarea durante todo el tiempo de simulación. Este comportamiento

es lógico si se tiene en cuenta que este componente describe el comportamiento de un hardware asociado a una interfaz del DUV.

- En la línea 38 se espera a que se desactive la señal de *reset*, `rst_n`, de la interfaz virtual `vif` a la que está conectada el componente *driver*.
- En la línea 39 se encuentra una estructura *while* de *SytemVerilog*. Esta estructura repite el código que contiene siempre que se cumpla la condición de que la señal `rst_n` de la interfaz `vif` no se encuentre a nivel alto.
- Entre las líneas 40 y 44 se describe el primer paso de la ejecución de la tarea.
- En la línea 40 se muestra por pantalla que se va a realizar la llamada al método `get_next_item` mediante la macro `` `uvm_info ``.
- En la línea 41 se realiza la llamada al método **`get_next_item`**. Este método inicia el protocolo de *handshake* desde el punto de vista del componente *driver* y permite a la secuencia comenzar con el paso de aleatorización de la transacción. Este método es bloqueante y espera a que la secuencia ejecute el método **`finish_item`** y envíe la transacción que será recibida en el parámetro `req` de la llamada. Este parámetro, tal y como se ha comentado, está definido en la clase base proporcionada por UVM. Hay que destacar que la comunicación secuencia-*driver* se realiza a través del puerto TLM denominado `seq_item_port` y definido, también, en la clase base.
- En las líneas 42 y 43 se realiza la clonación del objeto `req` en el objeto `rsp`, que será el objeto (transacción) a enviar de vuelta a la secuencia. Esta operación se realiza dentro de la función `cast` con la finalidad de comprobar que ambos objetos son del mismo tipo. Si se realiza con éxito, se muestra en pantalla que se va a conducir la transacción recibida, indicando el valor de alguna de sus señales. En el caso de ocurrir un error en la clonación, se procede a abortar la simulación mediante el uso de la macro `` `uvm_fatal `` indicando el error en cuestión.
- En la línea 46 se produce el siguiente paso en la ejecución de tarea mediante la llamada a la tarea `drive_packet`. Este método se encarga de transformar la transacción recibida y convertirla en señales a aplicar al DUV a través de su interfaz virtual.

```
36    virtual task get_and_drive( );
37       forever begin
38          @(negedge vif.rst_n);
39          while(vif.rst_n != 1'b1) begin
40             `uvm_info("DRIVER-Dbg", $sformatf("About to call get_next_item()"), UVM_MEDIUM)
41             seq_item_port.get_next_item(req);
42             if(!$cast(rsp, req.clone())) `uvm_fatal("DRIVER", "Cast to rsp failed")
43              `uvm_info("DRIVER-Dbg", $sformatf("Driving tx data_i=0x%0h, ack_i=%0b,
   stall_i=%0b",
44                         req.data_i, req.ack_i, req.stall_i), UVM_MEDIUM)
45
46             drive_packet(req, rsp);
```

Figura 64 Primera parte de la tarea `get_and_drive` del componente `wb_driver`

La tarea `drive_packet` se muestra en la Figura 65. Esta tarea se encarga de que el *driver* complete el protocolo de transferencia con el DUV haciendo uso de la interfaz virtual. Analizando el código se puede observar:

- En la línea 57 se muestra que la tarea tiene dos argumentos. El primero corresponde al objeto `req` recibido de la secuencia en el paso anterior y el segundo a la respuesta que genera esta tarea con la información obtenida de la interfaz virtual.

- En la línea 58 se declara un objeto de tipo `wb_item` denominado `rsp_i`, que servirá de objeto intermedio para evitar la modificación de la transacción recibida.
- En la línea 59 se realiza la clonación de `req` en `rsp_i`. Esta clonación se realiza dentro de la función `cast` lo que permite comprobar la compatibilidad de tipo. Si ocurre algún error, se aborta la simulación mediante la macro `` `uvm_fatal `` indicando el error encontrado.
- Entre las líneas 60 y 62 se estimula varias señales de la interfaz virtual con los valores recibidos en la transacción `req` de entrada. En este caso, las señales activadas serán el bus de datos `data_i`, la señal `ack_i` y la señal `stall_i`, que son las señales del protocolo *Wishbone* que debe proporcionar el *testbench*.
- En la línea 63 se espera a que la señal `cyc_o` de la interfaz virtual esté activa. Esto es debido a que, tal y como se comentó al presentar el protocolo *Wishbone*, cada transferencia dentro de un ciclo *Wishbone* se valida con la señal `cyc_o`.
- Entre las líneas 64 y 69 se recogen los valores de distintas señales enviadas a través de la interfaz *Wishbone por el DUV,* mediante la interfaz virtual y se depositan en el objeto `rsp_i`.
- En la línea 70 se copian los valores del objeto `rsp_i` al objeto `rsp` igualando sus manejadores. Este último objeto será la respuesta de la presente tarea.

```
57    virtual task drive_packet(wb_item req, output wb_item rsp);
58       wb_item rsp_i;
59       if(!$cast(rsp_i, req.clone())) `uvm_fatal("DRIVER", "Cast to rsp failed")
60       vif.data_i    <= req.data_i;
61       vif.ack_i     <= req.ack_i;
62       vif.stall_i   <= req.stall_i;
63       do @(posedge vif.clk); while(!vif.cyc_o);
64       rsp_i.addr_o = vif.addr_o;
65       rsp_i.data_o = vif.data_o;
66       rsp_i.we_o  = vif.we_o;
67       rsp_i.cyc_o = vif.cyc_o;
68       rsp_i.stb_o = vif.stb_o;
69       rsp_i.stall_i = vif.stall_i;
70       rsp = rsp_i;
71
72
73    endtask
```

Figura 65 Tarea `drive_packet` del componente `wb_driver`

Tras utilizar la tarea `drive_packet`, queda el último paso para finalizar un ciclo de ejecución de la tarea `get_and_drive` y, por ende, el protocolo de *handshake* desde el punto de vista del componente *driver*. La Figura 66 muestra la segunda parte de la implementación de la tarea `get_and_drive`. Al realizar el análisis del código se puede observar:

- En la línea 47 se muestra la llamada al método `set_id_info` mediante el manejador de la respuesta, `rsp`. Esto se debe a la necesidad de la respuesta de obtener el ID de la transacción recibida. Si la respuesta enviada a la secuencia, a través del *sequencer*, no contiene el mismo identificador que la transacción recibida, el componente *sequencer* abortará la ejecución del *testbench*.
- En la línea 49 se envía por pantalla la información correspondiente a la llamada al método `item_done` con una respuesta que contiene los valores especificados.
- Para finalizar, en la línea 52 se realiza la llamada al método `item_done` con la respuesta `rsp` para que la secuencia sea capaz de utilizar esta información y actuar en

consecuencia, como se ha comentado anteriormente. Este método finaliza el ciclo del protocolo de *handshake* por parte del *driver* por lo que el siguiente paso es repetir el proceso haciendo uso del método `get_next_item` si se cumplen las condiciones adecuadas.

```
47              rsp.set_id_info(req);
48
49              `uvm_info("DRIVER-Dbg", $sformatf("About to call item_done(rsp) with we_o=%0b,
    adr_o=0x%0h, dat_o=0x%0h, ack_i=%0b, stall_i=%0b",
50                          rsp.we_o, rsp.addr_o, rsp.data_o, rsp.ack_i, rsp.stall_i), UVM_MEDIUM)
51
52              seq_item_port.item_done(rsp);
53          end
54      end
55    endtask: get_and_drive
```

Figura 66 Segunda parte de la tarea `get_and_drive` del componente `wb_driver`

5.1.7. Componente Test

El componente Test es el primer componente que se ejecuta en la simulación en el momento en el que el módulo *Top* ponga en marcha el mecanismo de fases. Para el desarrollo de este entorno de verificación UVM, se han organizado y agrupado los test en una librería denominada `qvip_axim2wbsp_test_lib.svh`. Para explicar el funcionamiento de este componente, es necesario mencionar el uso de las secuencias en los test. Las secuencias utilizadas serán aquellas ya explicadas en los apartados anteriores.

5.1.7.1. Test base

En primer lugar, se ha creado un *test* base del que heredarán el resto de test las funciones que contiene. La Figura 67 muestra el test base, denominado `qvip_axim2wbsp_test_base`. En este caso, la mayoría de las funciones son externas, excepto la tarea `run_phase` que, en el caso de este test base, únicamente declara un tiempo de drenaje. Su definición se muestra en las líneas 26 a 29, donde, mediante el uso de la función `set_drain_time` del mecanismo de fases, establece un tiempo de drenaje para el resto de test. Este tiempo de drenaje, `drain_time`, mantiene la simulación activa durante un tiempo determinado una vez han terminado todos los procesos que se quería ejecutar. Este hecho es conveniente ya que muchos sistemas tienen una determinada latencia entre los estímulos de entrada y los datos a la salida.

```
 4 class qvip_axim2wbsp_test_base extends uvm_test;
 5   `uvm_component_utils(qvip_axim2wbsp_test_base)
 6   // QVIP Configuration objects = As defined in the qvip_axim2wbsp_params_pkg
 7
 8   axi4_master0_cfg_t axi4_master0_cfg;
 9
10   // Environment configuration object
11   qvip_axim2wbsp_env_config env_cfg;
12
13   // Environment component
14   qvip_axim2wbsp_env env;
15   function new
16     (
17      string name = "qvip_axim2wbsp_test_base_test_base",
18      uvm_component parent = null
19      );
20      super.new(name, parent);
21   endfunction
22
23   extern function void build_phase(uvm_phase phase);
24   extern function void end_of_elaboration_phase(uvm_phase phase);
25   extern function void init_vseq(qvip_axim2wbsp_vseq_base vseq);
26   virtual   task run_phase(uvm_phase phase);
27      //Drain time
28      phase.phase_done.set_drain_time(this, 25);
29   endtask: run_phase
30
31 endclass: qvip_axim2wbsp_test_base
```

Figura 67 Test `qvip_axim2wbsp_test_base`

Las funciones externas de este test son las que gestionan la fase `build_phase`, la fase `end_of_elaboration_phase` y la función `init_vseq`, y cuyas implementaciones se muestran en la Figura 68.

En la fase `build_phase` se realizan los siguientes pasos:

- **Línea 208.** Se crea el objeto de configuración del entorno. Este objeto, del tipo `qvip_axim2wbsp_env_config`, será el encargado de configurar el entorno creado.
- **Línea 209.** La inicialización de los parámetros de configuración del entorno se realiza mediante la función `initialize` del objeto creado. Esta función, básicamente, inicializa todos los rangos de direcciones del mapa de direcciones a utilizar por el agente asociado a la interfaz AXI4.
- **Línea 211.** En esta línea se crea el objeto de configuración asociado al agente `axi4_master0` que será el agente encargado de gestionar la interfaz AXI4.
- **Línea 212.** En caso de que el agente `axi4_master0` no haya sido registrado en la base de datos, se provocara un error grave mediante el uso de la macro `` `uvm_fatal `` y se abortará la creación del *testbench*.
- **Línea 218.** En esta línea se pasan las reglas de direcciones del entorno al objeto de configuración del agente.
- **Línea 219.** Se asocia el manejador del objeto de configuración del agente `axi4_master0` al manejador correspondiente del objeto de configuración del entorno. Este paso es crítico ya que asocia ambos objetos de configuración.
- **Línea 222.** Una vez creados y enlazados los objetos de configuración, se crea el entorno, denominado `env` y se inicializa el manejador `env.cfg` con el objeto de configuración ya creado previamente, línea 223.

La fase `end_of_elaboration_phase` tiene, en este caso, como único cometido imprimir en pantalla la topología seguida en el *testbench* creado. Esta fase está pensada para realizar acciones una vez creado el *testbench*, al finalizar la fase `build_phase` y antes de comenzar la simulación, fase `run_phase`.

Por último, la función `init_vseq` se utiliza para inicializar el componente UVM *Sequencer* que utilizará la secuencia a ejecutar en la interfaz AXI4. En este caso particular, línea 233, se inicializa el *sequencer* `axi4_master0` de la secuencia con el *sequencer* definido en el entorno y asignado al agente `axi4_master0`. De esta manera, cualquier test derivado de este *test* base no debe preocuparse en inicializar el *sequencer* sobre el que se ejecutara la secuencia. Al haber una única interfaz AXI4 y, por ende, una única interfaz, solo se realiza una única asignación.

```
206 function void qvip_axim2wbsp_test_base::build_phase
207   (uvm_phase phase);
208    env_cfg = qvip_axim2wbsp_env_config::type_id::create("env_cfg");
209    env_cfg.initialize();
210
211    axi4_master0_cfg = axi4_master0_cfg_t::type_id::create("axi4_master0_cfg" );
212    if ( !uvm_config_db #(axi4_master0_bfm_t)::get(this, "", "axi4_master0",
213         axi4_master0_cfg.m_bfm) )
214      begin
215         `uvm_error("build_phase",
216         "Unable to get virtual interface axi4_master0 for axi4_master0_cfg from uvm_config_db")
217      end
218    axi4_master0_config_policy::configure(axi4_master0_cfg, env_cfg.addr_map);
219    env_cfg.axi4_master0_cfg = axi4_master0_cfg;
220
221    // Once the agent configuration objects are done build the env
222    env = qvip_axim2wbsp_env::type_id::create("env", this);
223    env.cfg = env_cfg;
224 endfunction: build_phase
225
226 function void qvip_axim2wbsp_test_base::end_of_elaboration_phase
227   (uvm_phase phase);
228    uvm_top.print_topology();
229 endfunction
230
231 function void qvip_axim2wbsp_test_base::init_vseq
232   (qvip_axim2wbsp_vseq_base vseq);
233    vseq.axi4_master0 = env.axi4_master0.m_sequencer;
234 endfunction: init_vseq
```

Figura 68 Funciones externas del *test* `qvip_axim2wbsp_test_base`

5.1.7.2. TEST DERIVADOS

Una vez implementado el test base, se comienza el desarrollo del resto de *test*. El código de los diferentes test es muy parecido en todos ellos, cambiando solo aspectos relacionados con la secuencia que ejecutan. Esto es lógico ya que la implementación del test funcional que se quiere realizar radica, principalmente, en la secuencia que ejecuta. La Figura 69 muestra la primera parte de la implementación de un test denominado `qvip_axim2wbsp_simple_test`, que representa el caso en el que el esclavo *Wishbone* responde en tiempo real, sin errores ni pausas, a cualquier petición del maestro AXI4. En esta figura, se puede observar cómo este test, al heredar de `qvip_axim2wbsp_test_base`, utiliza todas sus funciones y no modifica ninguna excepto la fase `run_phase`, en la que se profundizará más adelante.

```
35 class qvip_axim2wbsp_simple_test extends qvip_axim2wbsp_test_base;
36    `uvm_component_utils(qvip_axim2wbsp_simple_test)
37
38    function new(string name, uvm_component parent);
39       super.new(name, parent);
40    endfunction: new
41
42    function void build_phase(uvm_phase phase);
43       super.build_phase(phase);
44    endfunction: build_phase
45
46    function void end_of_elaboration_phase(uvm_phase phase);
47       super.end_of_elaboration_phase(phase);
48    endfunction: end_of_elaboration_phase
49
50    function void init_vseq(qvip_axim2wbsp_vseq_base vseq);
51       super.init_vseq(vseq);
52    endfunction: init_vseq
```

Figura 69 Test `qvip_axim2wbsp_simple_test`

La fase `run_phase` es la más compleja del test y se muestra en la Figura 70. Para su explicación se destacan dos partes bien diferenciadas.

La primera parte del código ejecuta las acciones necesarias para definir y preparar las secuencias utilizadas durante la ejecución del test. Estas acciones se engloban entre las líneas 56 y 86. A continuación se detallan dichas acciones:

- **Líneas 56 a 59.** En este fragmento se definen las variables y manejadores para la creación de las secuencias. El manejador para la secuencia reactiva se declara en la línea 59, denominado `wbseq`.
- **Líneas 67 a 70.** En este bloque se analizan los argumentos de llamada al simulador para obtener todas aquellas secuencias especificadas con el token `+SEQ`. Estas serán las secuencias a ejecutar en la interfaz AXI4.
- **Líneas 75 a 86.** En este `foreach` se comprueba, en primer lugar, que las secuencias especificadas están registradas en la *Factory*, tal y como muestra las líneas 77 y 79. Ya en la línea 83 se comprueba, además, que el objeto creado para la secuencia es del tipo adecuando mediante el uso de la función `cast`.

  En todas estas comprobaciones, el código ejecuta la macro `` `uvm_fatal `` en caso de detectar que ha habido un error grave en la creación de la secuencia.

La segunda parte del código, implementada entre las líneas 88 a 108, contiene el código utilizado para la ejecución de las diferentes secuencias. A continuación, se definen dichas acciones:

- **Línea 89.** En esta línea se activan los *objections* para indicar que se ha comenzado la ejecución de un test. Esto se realiza mediante el uso del método `raise_objection` de la clase `phase`. Este mecanismo permite que la fase `run_phase` no se termine mientras haya algún *objection* activo. Al finalizar de ejecutar las secuencias del test, se desactivarán los *objections* usando el método `drop_objection` de la clase `phase`, tal y como se muestra en la línea 108.
- **Línea 91.** Se ejecuta una estructura *fork_join* propia de *SystemVerilog*. Esta estructura permite la ejecución en paralelo de varios hilos. En este caso, se declaran dos hilos, cada uno de ellos ejecuta una serie de acciones englobadas entre los *tokens begin-end*.
- **Primer hilo.** Se engloba entre las **líneas 92 a 96**. En este hilo se crea la secuencia asociada a la interfaz *Wishbone*, de tipo `simple_sequence`, ya explicado en

apartados anteriores. En la línea 94 se establece el número de transacciones a enviar, a través del uso de variable `num_seq` de dicha secuencia. Finalmente, en la línea 95, se comienza la ejecución de la secuencia haciendo uso del método `start`. Hay que hacer notar cómo al método `start` se le pasa como parámetro el componente UVM *Sequencer* asociado a la interfaz *Wishbone* que, en este caso, corresponde con el *sequencer* definido en el agente `wbagent` del componente `env` creado en el presente test.

- **Segundo hilo.** Se engloba entre las **líneas 97 a 105**. Usando un *foreach*, este hilo va ejecutando una a una cada una de las secuencias especificadas en la línea de comandos a la hora de simular el *testbench*. En primer lugar, se inicializa la secuencia mediante el método `init_vseq` ya comentado para, en segundo lugar, ejecutarla mediante el método `start`.

Finalizada la ejecución de los dos hilos anteriores, se desactivarán los *objection* dando por finalizada la fase `run_phase` de este test.

```
54    virtual task run_phase(uvm_phase phase);
55
56       string seq_names[$];
57       qvip_axim2wbsp_vseq_base vseq[];
58       // Reactive slave sequence
59       simple_sequence wbseq;
60
61       uvm_cmdline_processor clp;
62       uvm_factory factory = uvm_factory::get();
63       clp = uvm_cmdline_processor::get_inst();
64
65       super.run_phase(phase);
66
67       `uvm_info("TEST", {"Starting run_phase"}, UVM_LOW)
68       if (clp.get_arg_values("+SEQ=", seq_names) == 0)
69         `uvm_fatal("TEST/SEQ/ARG_MISSING",
70                    "You must specify at least one sequence to run using the +SEQ plusarg")
71
72       vseq = new[seq_names.size()];
73
74       // Qualify that each +SEQ specifies a sequence...
75       foreach (seq_names[i]) begin
76          uvm_object obj;
77          obj = factory.create_object_by_name(seq_names[i]);
78
79          if (obj == null)
80            `uvm_fatal("TEST/SEQ/NOT_IN_FACTORY",
81                       {"Sequence '",seq_names[i],"' not found in factory.",
82                        " Was it declared with a `uvm_*_utils macro?"})
83          if (!$cast(vseq[i], obj))
84            `uvm_fatal("TEST/SEQ/NOT_A_SEQ",
85                       {"Sequence '",seq_names[i],"' is not a sequence"})
86       end // foreach (seq_names[i])
87
88       // Run each sequence in turn...
89       phase.raise_objection(this, "Starting sequences");
90
91       fork
92          begin
93             wbseq = simple_sequence::type_id::create("wbseq");
94             wbseq.num_seq = 7;
95             wbseq.start(env.wbagent.sequencer);
96          end
97          begin
98             foreach (vseq[i]) begin
99                qvip_axim2wbsp_vseq_base seq = vseq[i];
100               `uvm_info("TEST/SEQ/RUN",
101                         {"Running sequence '",seq_names[i],"'"}, UVM_LOW)
102               init_vseq(seq);
103               seq.start(null);
104            end
105         end
106      join
107
108      phase.drop_objection(this, "Completed sequences");
109      // (end inline source)
110
111   endtask:run_phase
```

Figura 70 Fase `run_phase` del test `qvip_axim2wbsp_simple_test`

5.1.8. Módulo Top

El módulo *Top* está dividido en dos partes, una denominada `hdl_qvip_axim2wbsp` y otra denominada `hvl_qvip_axim2wbsp`. Esta configuración se basa en el modelo que se muestra en la Figura 71. En esta implementación, el fragmento HDL actúa a bajo nivel como sección estática del módulo *Top* en la que se referencia el DUV y las interfaces virtuales y se generan las señales principales que controla el DUV. Además, se registran las interfaces virtuales en la base de

datos de UVM. Por otra parte, el fragmento HVL actúa a alto nivel de manera dinámica. Se encarga de la ejecución del test, lo que conlleva generar el entorno de verificación UVM descrito.

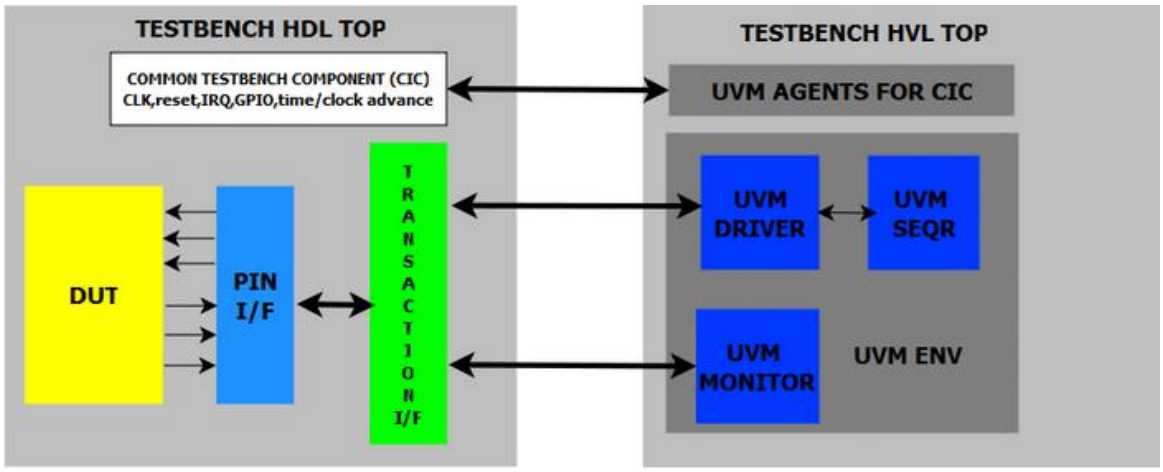

Figura 71 Modelo de implementación del módulo Top

La Figura 72 muestra la primera parte de la implementación del módulo `hdl_qvip_axim2wbsp`. Esta implementación comienza con la importación de los parámetros del resto de componentes del entorno de verificación, así como el paquete UVM y las macros de esta metodología. Antes de referenciar el DUV y las interfaces virtuales se generan los nodos que se utilizarán para conectar sus señales. Cabe destacar la línea 17, correspondiente a la implementación del nodo `o_wb_reset`, siendo este el único nodo que es necesario generar de manera externa a la interfaz virtual *Wishbone*. A partir de la línea 19 se definen todas y cada una de las variables asociadas a cada una de las señales de la interfaz AXI4 y de la interfaz *Wishbone*.

```
 7 module hdl_qvip_axim2wbsp;
 8    import uvm_pkg::*;         // UVM Package. From Installation
 9 `include "uvm_macros.svh"
10    import qvip_axim2wbsp_params_pkg::*; //
11    import test_params_pkg::*; //
12
13
14    wire                                                default_clk_gen_CLK;
15    wire                                                default_reset_gen_RESET;
16
17    wire                                                o_wb_reset;
18
19    wire                                                axi4_master_AWVALID;
20    wire [axi4_master0_params::AXI4_ADDRESS_WIDTH-1:0]  axi4_master_AWADDR;
21    wire [2:0]                                          axi4_master_AWPROT;
22    wire [3:0]                                          axi4_master_AWREGION;
23    wire [7:0]                                          axi4_master_AWLEN;
24    wire [2:0]                                          axi4_master_AWSIZE;
25    wire [1:0]                                          axi4_master_AWBURST;
```

Figura 72 Primera parte del módulo `hdl_qvip_axim2wbsp`

El siguiente paso consiste en la generación de las señales de reloj y *reset* comunes para todo el entorno y las referencia al DUV. En la Figura 73 se puede observar la implementación de estas señales entre las líneas 66 y 74 y la referencia al DUV a partir de la línea 89. En la línea 66 se

referencia el módulo `default_clk_gen` encargado de generar la señal de reloj del sistema. Asimismo, en la línea 70 se referencia el módulo `default_reset_gen`, encargado de la generación de la señal de *reset* del sistema. En la línea 78 se referencia el DUV pasándole como parámetro, línea 79, el ancho del bus de datos del sistema. A partir de la línea 83 se añaden todas y cada una de las conexiones de las señales de la interfaz del DUV con las variables equivalentes definidas anteriormente. Por razones de optimizar y acortar esta memoria, en la figura solo se muestran algunas de las señales correspondientes a la interfaz AXI4.

```
65   // Deafult clk and reset generator
66   default_clk_gen default_clk_gen
67     (
68      .CLK(default_clk_gen_CLK)
69      );
70   default_reset_gen default_reset_gen
71     (
72      .RESET(default_reset_gen_RESET),
73      .CLK_IN(default_clk_gen_CLK)
74      );
75
76
77   // DUT Instantiation
78   axim2wbsp #(
79               .C_AXI_DATA_WIDTH (AXI4_WDATA_WIDTH)
80               )
81   axim2wbsp(
82             // AXI Global clk and reset
83             .S_AXI_ACLK    (default_clk_gen_CLK), // System clock
84             .S_AXI_ARESETN (default_reset_gen_RESET),
85
86             // AXI write address channel signals
87             .S_AXI_AWVALID (axi4_master_AWVALID),
88             .S_AXI_AWREADY (axi4_master_AWREADY),
89             .S_AXI_AWID    (axi4_master_AWID),
90             .S_AXI_AWADDR  (axi4_master_AWADDR),
91             .S_AXI_AWLEN   (axi4_master_AWLEN),
92             .S_AXI_AWSIZE  (axi4_master_AWSIZE),
93             .S_AXI_AWBURST (axi4_master_AWBURST),
```

Figura 73 Segunda parte del módulo `hdl_qvip_axim2wbsp`

La Figura 74 destaca la referencia a las señales del protocolo *Wishbone* del DUV, que se conectan a la interfaz virtual `wb_if`, que se mostrará posteriormente, a excepción de la señal de *reset*.

```
133            // Wishbone signals
134            // We'll share the clock and the reset
135            .o_reset       (o_wb_reset),
136            .o_wb_cyc      (wb_if.cyc_o),
137            .o_wb_stb      (wb_if.stb_o),
138            .o_wb_we       (wb_if.we_o),
139            .o_wb_addr     (wb_if.addr_o),
140            .o_wb_data     (wb_if.data_o),
141            .o_wb_sel      (wb_if.sel_o),
142            .i_wb_stall    (wb_if.stall_i),
143            .i_wb_ack      (wb_if.ack_i),
144            .i_wb_data     (wb_if.data_i),
145            .i_wb_err      (wb_if.err_i)
```

Figura 74 Tercera parte del módulo `hdl_qvip_axim2wbsp`

Una vez referenciado el DUV, comienza la referencia a las interfaces virtuales. En primer lugar, se encuentra el caso del módulo denominado `axi4_master`, representado en la Figura 75. Este módulo lo proporciona el QVIP de *Mentor Graphics* y su función es implementar la interfaz virtual AXI y registrarla en la base de datos. Para ello, se le pasa por parámetros aquellos que determinan el tamaño de los buses, como el bus de datos o el bus de direcciones, y se conectan las señales del módulo `axi4_master` a los nodos generados en el módulo *Top*. Además, este módulo tiene dos parámetros específicos para la generación de la interfaz y su posterior registro, como muestran las líneas 171 y 172. Estos parámetros son `IF_NAME`, que indica el nombre con el que se registrará la interfaz AXI4, y `PATH_NAME`, que se utilizará en el registro y permite determinar el nivel de jerarquía base y definir la prioridad de la interfaz respecto al resto de componentes de la jerarquía.

```
162    // AXI4 Masters Instantiation
163    axi4_master
164      #(
165        .ADDR_WIDTH(axi4_master0_params::AXI4_ADDRESS_WIDTH),
166        .RDATA_WIDTH(axi4_master0_params::AXI4_RDATA_WIDTH),
167        .WDATA_WIDTH(axi4_master0_params::AXI4_WDATA_WIDTH),
168        .ID_WIDTH(axi4_master0_params::AXI4_ID_WIDTH),
169        .USER_WIDTH(axi4_master0_params::AXI4_USER_WIDTH),
170        .REGION_MAP_SIZE(axi4_master0_params::AXI4_REGION_MAP_SIZE),
171        .IF_NAME("axi4_master0"),
172        .PATH_NAME("uvm_test_top")
173        )
174    axi4_master
175      (
176       .ACLK(default_clk_gen_CLK),
177       .ARESETn(default_reset_gen_RESET),
178       .AWVALID(axi4_master_AWVALID),
179       .AWADDR(axi4_master_AWADDR),
180       .AWPROT(axi4_master_AWPROT),
181       .AWREGION(axi4_master_AWREGION),
182       .AWLEN(axi4_master_AWLEN),
```

Figura 75 Cuarta parte del módulo `hdl_qvip_axim2wbsp`

Por último, se realiza la referencia a la interfaz virtual *Wishbone* `wb_if`, explicada en apartados anteriores, tal y como se muestra en la Figura 76. Para ello, se utiliza la señal de reloj general y la señal `o_wb_reset` como señales de reloj y *reset* de la interfaz. Además, entre las líneas 231 y 234 se registra esta interfaz en la base de datos de UVM, utilizando el método `set` de la clase `uvm_config_db`.

```
226    //WishBone Interface Instantiation
227
228    wb_if wb_if(.clk(default_clk_gen_CLK), .rst_n(o_wb_reset));
229
230
231    initial
232      begin
233         uvm_config_db#(virtual wb_if)::set(uvm_root::get( ) , "*" , "wb_if", wb_if);
234      end
235
236
237 endmodule: hdl_qvip_axim2wbsp
```

Figura 76 Quinta parte del módulo `hdl_qvip_axim2wbsp`

Por otro lado, el módulo `hvl_qvip_axim2wbsp` se implementa en el código mostrado en la Figura 77. Este módulo importa el paquete de UVM y el archivo `test_packages.svh`, que incluye todos los parámetros y componentes necesarios para la generación del entorno de verificación. Entre las líneas 12 y 16 se realiza la llamada al método `run_test`, que comienza la ejecución del mecanismo de fases de UVM, lo que deriva, en último lugar, en la ejecución del *testbench*.

```
 7 module hvl_qvip_axim2wbsp;
 8     import uvm_pkg::*;
 9
10     `include "test_packages.svh"
11
12     initial
13     begin
14 //        `uvm_info("TOP",{"Running rus_test()"}, UVM_LOW)
15        run_test();
16     end
17
18 endmodule: hvl_qvip_axim2wbsp
```

Figura 77 Módulo `hvl_qvip_axim2wbsp`

5.2. RESULTADOS

Una vez desarrollado el entorno de verificación UVM, en este capítulo también se recogen los resultados obtenidos tras la ejecución de los test que se han contemplado en este Trabajo Fin de Grado. Se visualizarán los resultados de forma gráfica, mostrando las formas de onda de las señales del DUV en función del tiempo. Al igual que para el *testbench ad-hoc*, se utilizará la herramienta *QuestaSim*. Los test que se ejecutarán varían en la secuencia utilizada para el agente *Wishbone*. Las pruebas consistirán en enviar, en direcciones diferentes, tres escrituras con los distintos tipos de ráfaga que proporciona el protocolo AXI4 y realizar las tres lecturas correspondientes, comprobando que se ha leído el mismo dato escrito anteriormente. Esto se implementa en la secuencia correspondiente al agente AXI4 del entorno, denominada `qvip_axim2wbsp_vseq_seq1`, explicada en el apartado 5.1.6.1. .

5.2.1. TEST `QVIP_AXIM2WBSP_SIMPLE_TEST`

En este caso se utilizará el test `qvip_axim2wbsp_simple_test` que inicia la secuencia `simple_sequence`, ambos explicados respectivamente en los apartados 5.1.7.2. y 5.1.6.2. . La principal característica de la secuencia *Wishbone* radica en que las señales de *stall* y *err* estarán en todo momento desactivadas, por lo que el agente esclavo reactivo estará disponible en cualquier instante de la simulación.

- **Operación de escritura**

La Figura 78 muestra la primera transacción de escritura desde el punto de vista de la interfaz AXI4. En el instante t1, tanto en el canal `AWADDR` como en el canal `WDATA` se comienza con el estímulo de las señales y su validación. El bus `AWADDR` indica que se parte de la dirección `0x1000`. En esta ocasión, el parámetro `AWBURST` se establece a `0x0`, el parámetro `AWLEN` a `0x1` y el parámetro `AWSIZE` a `0x2`. Esto conlleva que la escritura sea de tipo *fixed* y que se van a enviar dos datos distintos de 32 bits. En el bus `WDATA` se establece el valor del primer dato y en el bus `WSTRB` se indica que se utilizarán todos sus *bytes*.

En el instante t2, se lleva a cabo el protocolo de *handshake* del canal AWADDR que, al estar activa la señal de *READY*, su duración será de un solo ciclo de reloj. Además, se establece el segundo dato a transmitir, cuyo valor es `0x4DCBCD24` indicando que es el último al activar la señal `WLAST`.

En el instante t3, se termina la transacción desde la interfaz AXI4 al finalizar el protocolo de *handshake* del canal `WDATA` con el último dato.

Una vez se termine de gestionar la transacción desde la interfaz *Wishbone* como se mostrará a continuación, el DUV validará, activando la señal `BVALID`, la respuesta presente en el bus `BRESP`, instante t4, y espera al instante t5, en el que el maestro acepta la respuesta generada. Indicar en este caso, que la respuesta ha sido OKAY.

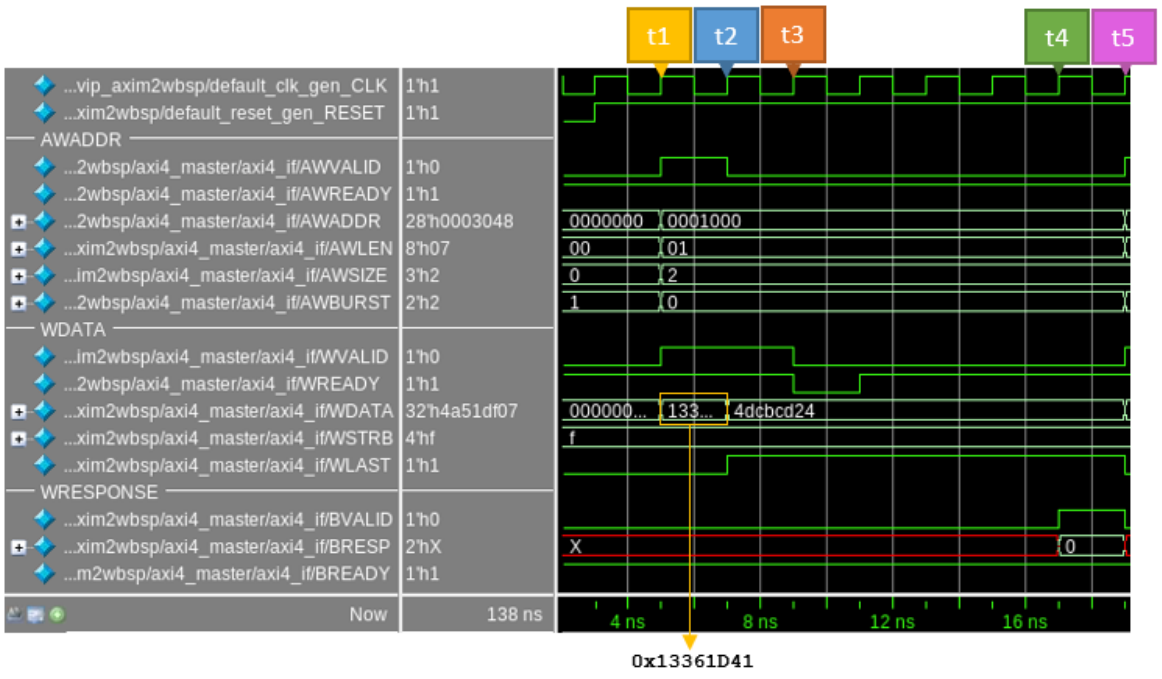

Figura 78 Simulación test qvip_axim2wbsp_simple_test: Parte 1

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 79. En el instante t1 comienza el ciclo de la transacción de escritura, señal `cyc_o` y señal `we_o` activas, y se validan los datos enviados mediante la señal `stb_o` activa y el valor `0xF` en el bus

sel_o, en concordancia con la señal WSTRB. Además, el DUV envía la dirección 0x400 por el bus addr_o y el dato 0x411d3613 por el bus data_o. Esta variación de la dirección y datos respecto a los valores en la interfaz AXI4 es propia del comportamiento del sistema y se ha explicado en apartados anteriores.

A partir del instante t2, comienza el reconocimiento de los datos mediante la activación de la señal ack_i. En el instante t3 se termina la validación del último dato enviado, 0x24CDCB4D, al desactivarse la señal stb_o. Sin embargo, no será hasta el instante t4 cuando finaliza el ciclo, señal cyc_o desactivada, instante en el que se reconoce el último dato enviado y se desactiva la señal ack_i. Esto conlleva a la respuesta generada en el canal WRESPONSE, como se mostró anteriormente.

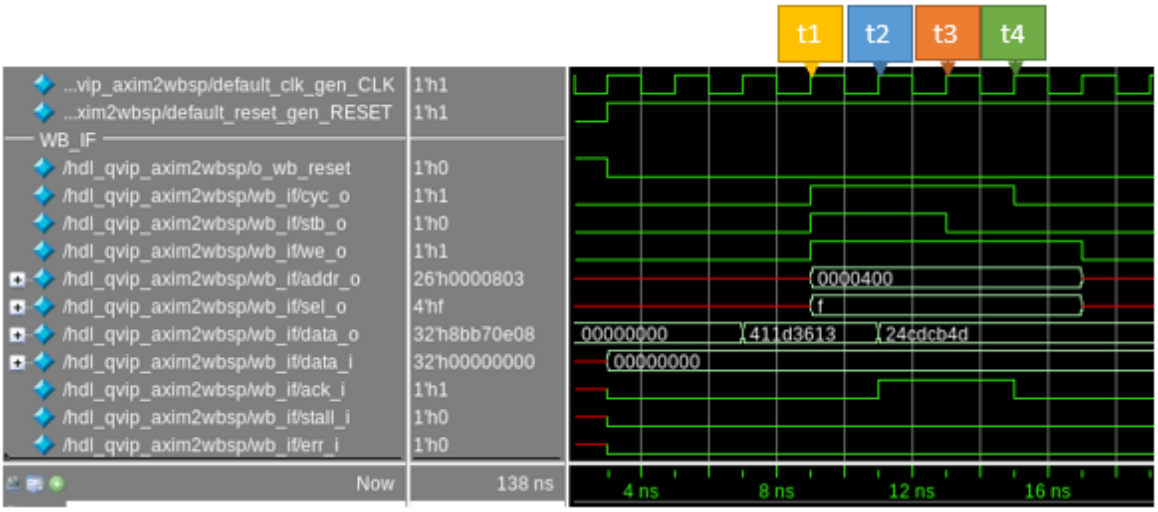

Figura 79 Simulación test qvip_axim2wbsp_simple_test: Parte 2

La Figura 80 muestra la segunda transacción de escritura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en el caso anterior con variaciones en los ajustes de la transacción. En el instante t1 se indica en el bus AWADDR que se parte de la dirección 0x2000. En esta ocasión, el parámetro AWBURST se establece a 0x1 y el parámetro AWLEN a 0x3. Esto conlleva que la escritura sea de tipo *incremental* y que se van a enviar cuatro datos distintos de 32 bits. En el transcurso de la transacción la señal WREADY se desactiva, instante t2, por lo que se mantienen los datos pausados esperando a que se reactive la señal, instante t3. En el instante t4 finaliza el envío de la transacción de escritura al DUV desde la interfaz AXI4, comenzando con el proceso en la interfaz *Wishbone*. Una vez finaliza este proceso, se lleva a cabo el envío de la respuesta por el canal WRESPONSE, que ha sido OKAY como se muestra en el instante t5.

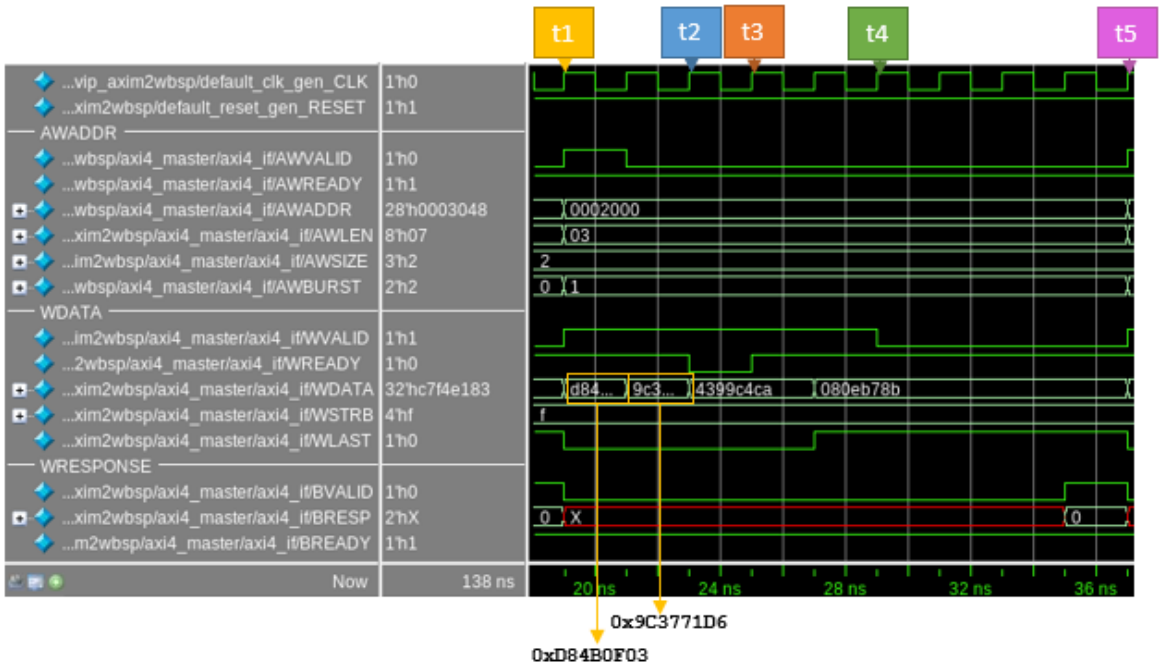

Figura 80 Simulación test qvip_axim2wbsp_simple_test: Parte 3

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 81. En el instante t1 comienza el ciclo de la transacción de escritura y se validan los datos enviados. En relación con las direcciones, se observa que la señal `addr_o` se incrementa consecutivamente, desde el valor `0x800` al valor `0x803`. Este incremento sucede cada vez que se completa una escritura de 32 bits de datos de cada dirección. Al validarse los 32 bits de cada dato, solo es necesaria una escritura para cada dirección. En el instante t2 finaliza del ciclo, instante en el que se reconoce el último dato enviado. Esto conlleva a la respuesta generada en el canal `WRESPONSE`, como se mostró anteriormente.

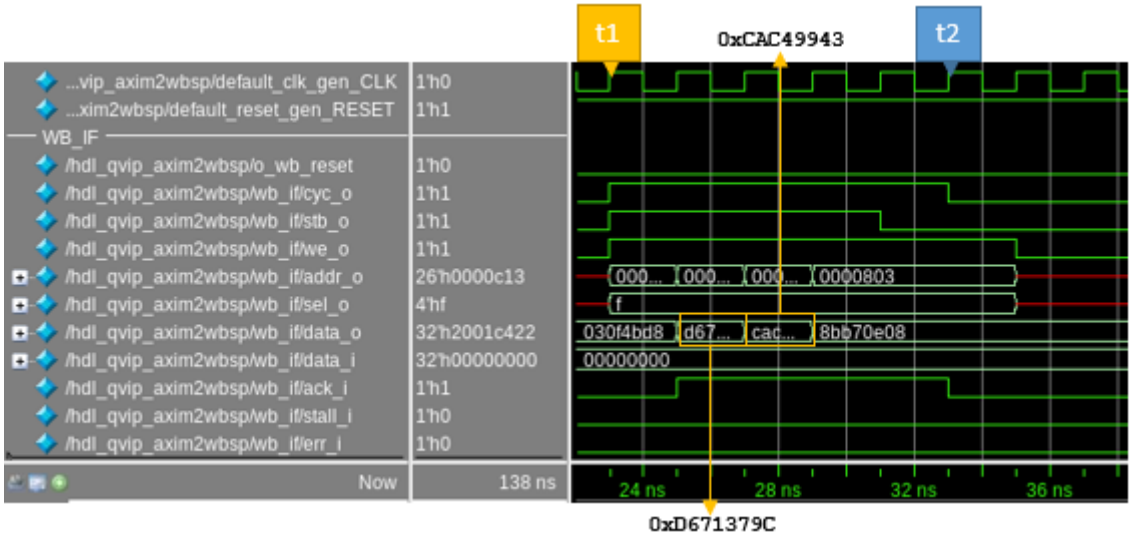

Figura 81 Simulación test qvip_axim2wbsp_simple_test: Parte 4

La Figura 82 muestra la tercera y última transacción de escritura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en los casos anterior, exceptuando las variaciones en la configuración de la transacción. En el instante t1 se indica en el bus `AWADDR` que se parte de la dirección `0x3048`. En esta ocasión, el parámetro `AWBURST` se establece a `0x2` y el parámetro `AWLEN` a `0x7`. Esto conlleva que la escritura sea de tipo *wrapped* y que se van a enviar ocho datos distintos de 32 bits. La elección de la dirección y longitud se han elegido para satisfacer las necesidades de este tipo de ráfaga y de esta forma poder visualizarlo correctamente. En el transcurso de la transacción la señal `WREADY` se desactiva, instante t2, por lo que se mantienen los datos pausados esperando a que se reactive la señal, instante t3. En el instante t4 finaliza el envío de la transacción de escritura al DUV desde la interfaz AXI4, comenzando con el proceso en la interfaz *Wishbone*. Una vez finaliza este proceso, se lleva a cabo el envío la respuesta por el canal `WRESPONSE`, que ha sido OKAY como se muestra en el instante t5.

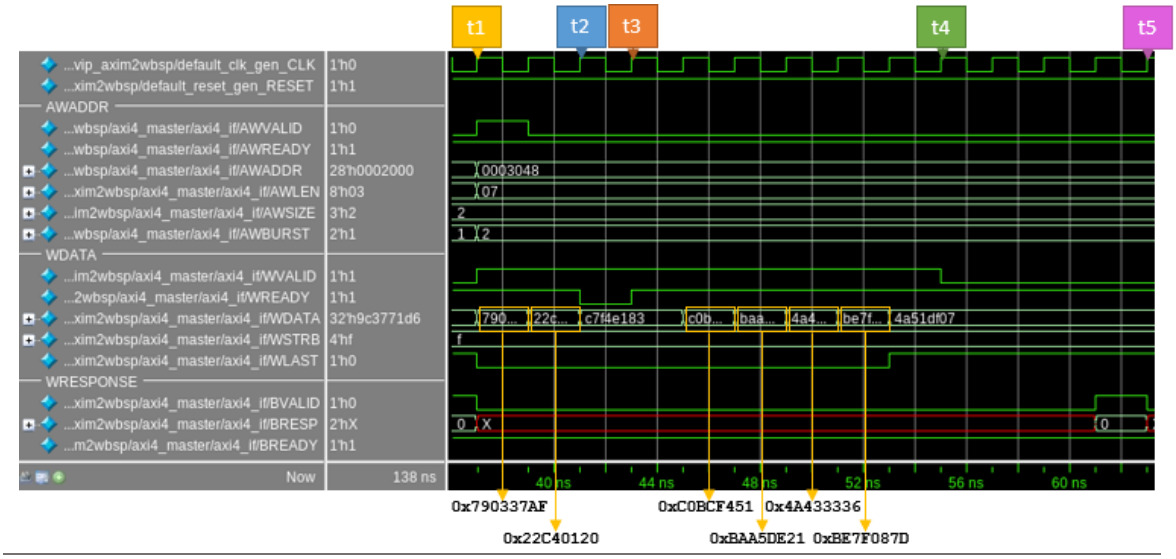

Figura 82 Simulación test qvip_axim2wbsp_simple_test: Parte 5

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 83. En el instante t1 comienza el ciclo de la transacción de escritura y se validan los datos enviados. En relación con las direcciones, se observa que la señal `addr_o` se incrementa consecutivamente, desde el valor `0xC12` al valor `0xC17` hasta el instante t2, en el que ocurre el efecto *wrapped* de este tipo de ráfaga. Cuando ocurre este efecto, se pasa de la dirección `0xC17` a la dirección `0xC10` y, por último, a la dirección `0xC11`, lo que confirma el efecto de envoltura. Este incremento sucede cada vez que se completa una escritura de 32 bits de datos de cada dirección. Al validarse los 32 bits de cada dato, solo es necesaria una escritura para cada dirección. En el instante t3 finaliza del ciclo, instante en el que se reconoce el último dato enviado. Esto conlleva a la respuesta generada en el canal `WRESPONSE`, como se mostró anteriormente.

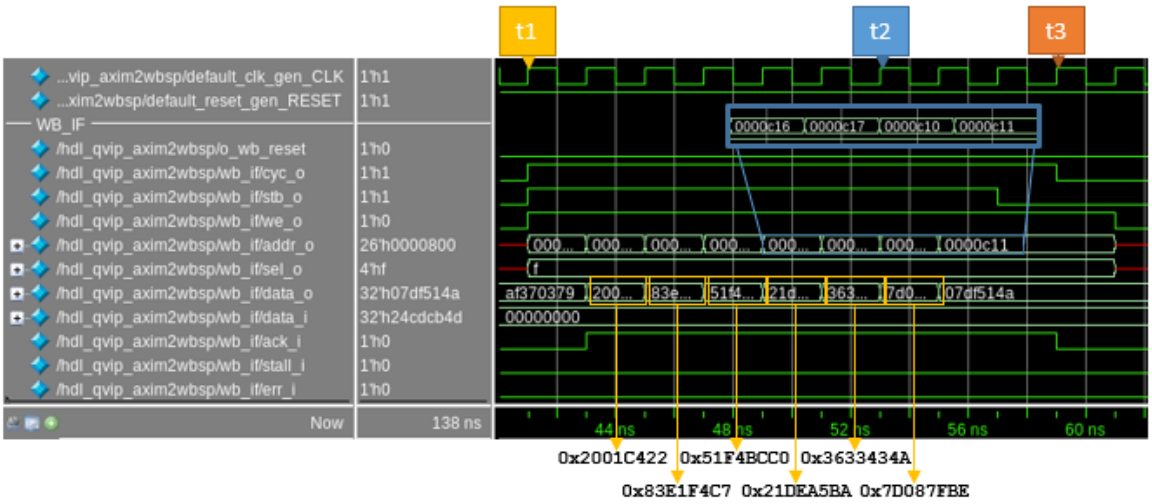

Figura 83 Simulación test qvip_axim2wbsp_simple_test: Parte 6

- **<u>Operación de lectura</u>**

El siguiente paso en esta simulación es el proceso de lectura. El proceso comienza con el canal `RADDR`, como se muestra en la Figura 84. Como se busca finalizar el proceso de escritura/lectura iniciado en la etapa anterior, la dirección utilizada en este caso es la misma que se envió anteriormente en el canal `AWADDR`. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

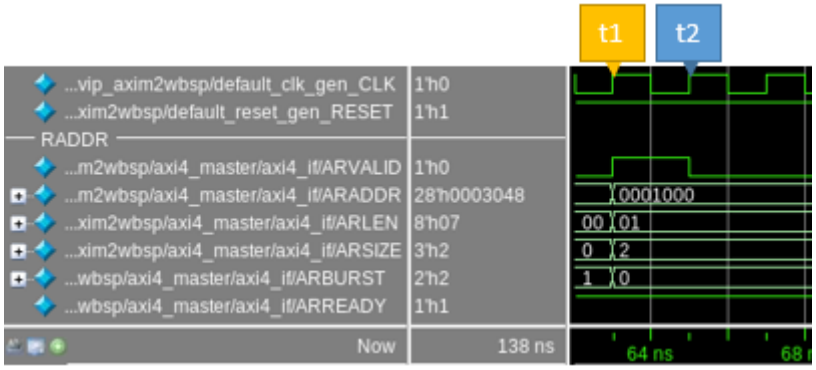

Figura 84 Simulación test qvip_axim2wbsp_simple_test: Parte 7

Al tratarse de una lectura, será la interfaz *Wishbone*, actuando como esclavo, la encargada de suministrar el dato pedido debiendo el DUV solicitar, a través de la interfaz *Wishbone*, el dato deseado. La Figura 85 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso `0x400`. A través del bus `data_i` el esclavo envía los datos al maestro. En el instante t2, se activa la señal de reconocimiento, `ack_i`, lo que provoca que en el instante t3 el maestro dé por finalizado el ciclo, desactivando la señal `cyc_o`. Es importante destacar que se han escrito dos datos distintos

en la dirección `0x400` durante la fase de escritura. Sin embargo, solo se lee dos veces el último dato que fue escrito. Esto se debe a que en el modo *fixed* se sobrescribe la dirección cuando llega un dato nuevo. Además, el agente *Wishbone* implementado no contempla el funcionamiento en modo FIFO usado en este tipo de accesos.

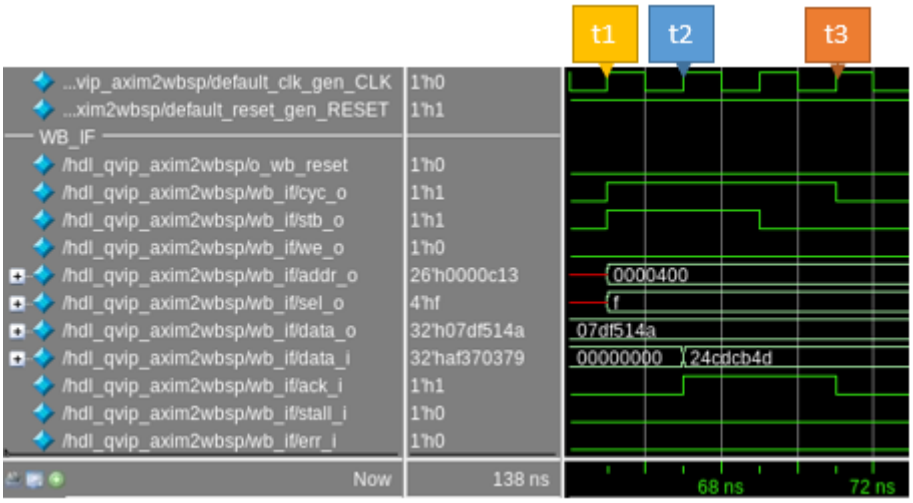

Figura 85 Simulación test qvip_axim2wbsp_simple_test: Parte 8

Para terminar con la lectura *fixed*, el DUV responde al maestro AXI como muestra la Figura 86. En este caso, el protocolo de *handshake* se realiza al revés, es el DUV quien inicia el protocolo y el maestro quien indicaría con la señal `RREADY` que está preparado para recibir los datos haciendo uso de la señal `RVALID`, instante t1. Hay que destacar que, el dato proporcionado es exactamente el dato escrito, `0x24CDCB4D`, que se ha almacenado usando el convenio *Big Endian*. En el instante t2 se indica que se envía por segunda vez el mismo dato al activar la señal `RLAST`. Hay que indicar que este funcionamiento es el funcionamiento esperado. En el instante t3, por último, se finaliza la lectura.

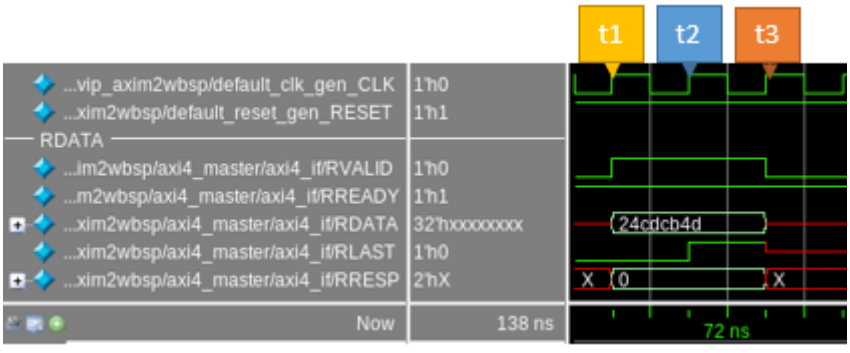

Figura 86 Simulación test qvip_axim2wbsp_simple_test: Parte 9

La Figura 87 muestra la segunda transacción de lectura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en el caso anterior con variaciones en los ajustes de la transacción. La dirección utilizada en este caso es la misma que se envió anteriormente en la

escritura *incremental*, `0x2000`. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura correspondiente. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

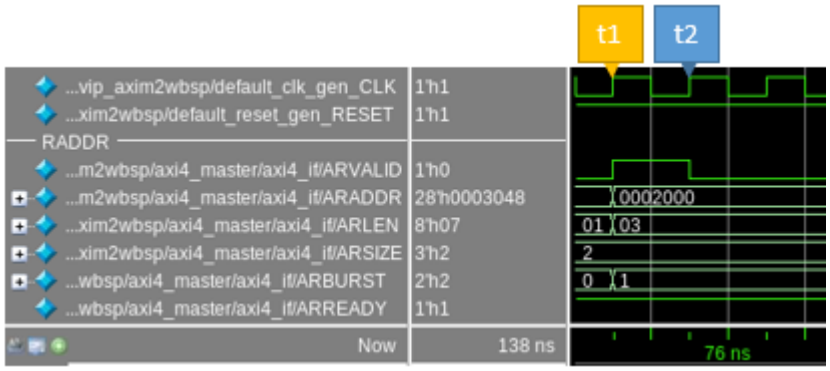

Figura 87 Simulación test qvip_axim2wbsp_simple_test: Parte 10

La Figura 88 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante T1, el DUV proporciona la dirección de lectura del dato, en este caso se parte de la dirección `0x800` y finaliza en la dirección `0x803`. En el instante t2 se envía el primer dato a través del bus `data_i` y se activa la señal de reconocimiento, `ack_i`, lo que provoca que en el instante t3 el maestro dé por finalizado el ciclo al reconocer el último dato, desactivando la señal `cyc_o`.

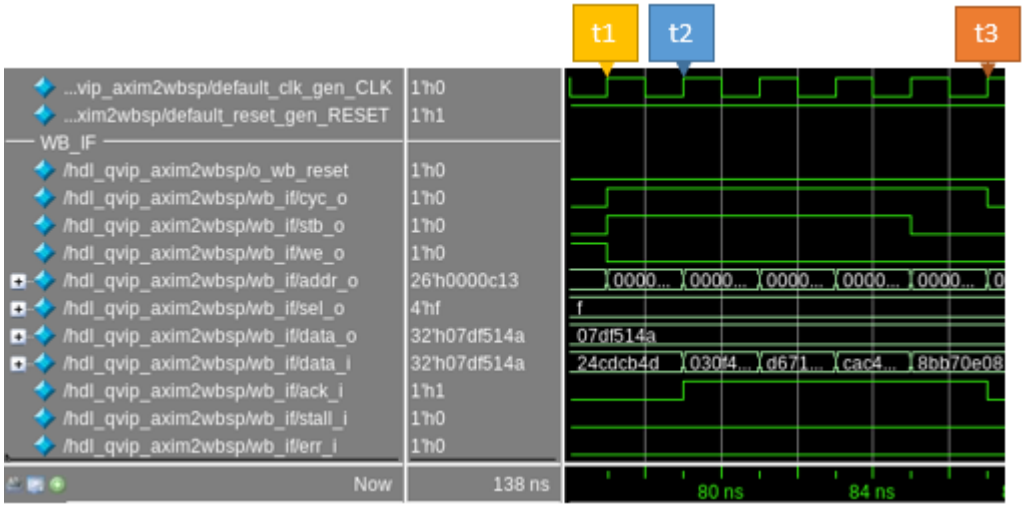

Figura 88 Simulación test qvip_axim2wbsp_simple_test: Parte 11

Para terminar con la lectura *incremental*, el DUV responde al maestro AXI como muestra la Figura 89. En el instante t1 comienza el protocolo de *handshake* y el envío de datos de lectura al maestro. En el instante t2 se finaliza la lectura. Hay que destacar que los datos proporcionados son exactamente los datos escritos anteriormente, pero usando el convenio *Big Endian*.

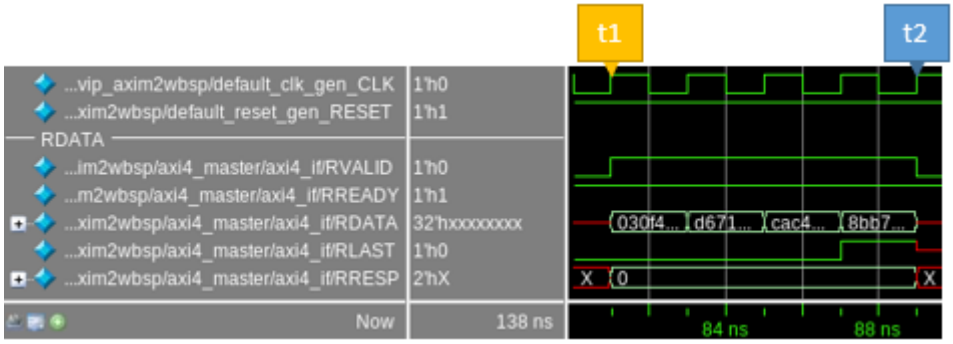

Figura 89 Simulación test qvip_axim2wbsp_simple_test: Parte 12

Para finalizar, la Figura 90 muestra la tercera y última transacción de lectura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en los casos anteriores con variaciones en la configuración de la transacción. La dirección utilizada en este caso es la misma que se envió anteriormente en la escritura *incremental*, `0x3048`. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura correspondiente. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

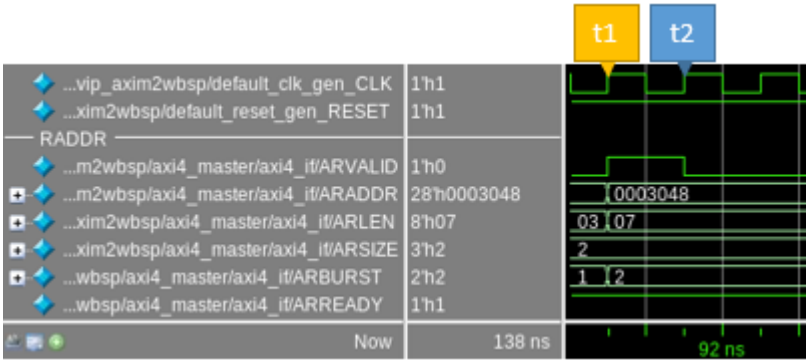

Figura 90 Simulación test qvip_axim2wbsp_simple_test: Parte 13

La Figura 91 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso se parte de `0xC12` y finaliza en la dirección `0xC11` debido al efecto de envoltura, que se produce en el instante t3, se pasa de la dirección `0xC17` a la dirección `0xC10`. En el instante t2 se envía el primer dato a través del bus `data_i` y se activa la señal de reconocimiento, `ack_i`, lo que provoca que en el instante t4 el maestro dé por finalizado el ciclo al reconocer el último dato, desactivando la señal `cyc_o`.

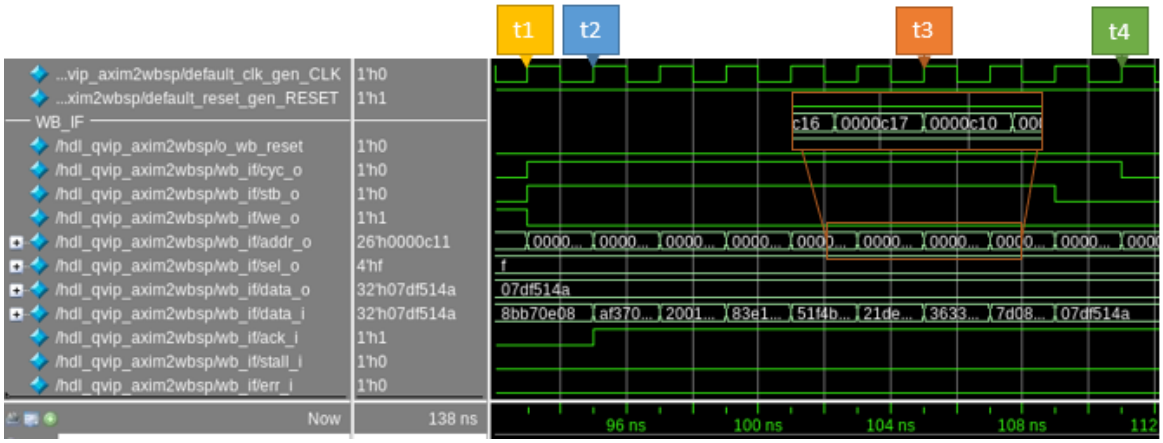

Figura 91 Simulación test qvip_axim2wbsp_simple_test: Parte 14

Para terminar con la lectura *wrapped*, el DUV responde al maestro AXI como muestra la Figura 92. En el instante t1 comienza el protocolo de *handshake* y el envío de datos de lectura al maestro. En el instante t2 se finaliza la lectura. Hay que destacar que los datos proporcionados son exactamente los datos escritos anteriormente, pero usando el convenio *Big Endian*.

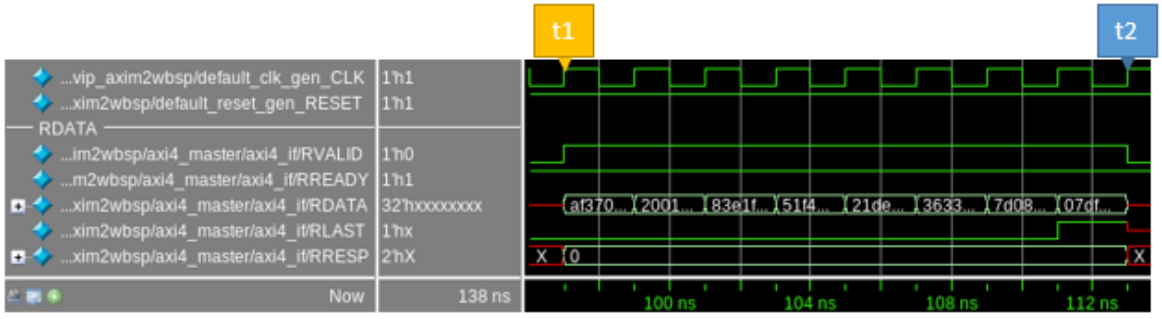

Figura 92 Simulación test qvip_axim2wbsp_simple_test: Parte 15

5.2.2. TEST `QVIP_AXIM2WBSP_FIXED_STALL_TEST`

En este caso se utilizará el test `qvip_axim2wbsp_fixed_stall_test` que inicia la secuencia `wb_stall_fixed_seq`. La principal diferencia que tiene esta secuencia con la explicada anteriormente es la estimulación de la señal de *stall* del protocolo *Wishbone*, por lo que el agente esclavo reactivo no estará disponible en cualquier instante de la simulación. La Figura 93 muestra la modificación del código respecto a la secuencia anterior.

- La primera modificación se encuentra en la línea 98 y corresponde al cambio de valor de la variable `stall_vec`, siendo una serie de bits que varían de manera controlada. Esta asignación se produce antes de comenzar el bucle de gestión del protocolo *handshake*. Esta variable está relacionada directamente con el valor de la señal `stall` en cada ciclo de reloj.
- El siguiente cambio se produce al gestionar esta variable y estimular la señal *stall* que se envía al componente *driver*. En la línea 122 se le asigna a la variable `stall` el bit 0 del vector de bits `stall_vec`. Acto seguido, línea 123, se rota el vector una posición

hacia la derecha, lo que permite preparar el valor de la señal de *stall* para la siguiente transferencia. Una vez se completa el desplazamiento de un total de 32 transferencias, se vuelve al valor inicial.

- Entre las líneas 124 y 127 se configura la variable `ack` para que se ajuste al valor que tenga la señal `stall_i` de la respuesta del *driver*.
- En la línea 129, al aleatorizar la transacción, se establecen los valores de las variables `ack` y `stall` en las señales correspondientes de la interfaz.
- Por último, en la línea 146 se modifica la condición para incrementar el número de secuencias en la variable `num_seq_rec`. Esta variable permite contabilizar el número de ráfagas enviadas y, por ende, permite finalizar el buvle *while* y, con ello, la ejecución de la tarea `body`.

```
95   virtual task body( );
96      // Creates the transaction
97      req = wb_item::type_id::create("req");
98      stall_vec = 32'b0100_0110_0010_1001_0100_0110_0010_1000;
99      dummy = 1'b1;
100
101     `uvm_info("SEQUENCE-Dbg", $sformatf("Initial rsp.adr_o=0x%0h", req.addr_o), UVM_MEDIUM)
102
103     while(num_seq_rec < num_seq) begin
121        // Step-2 A call to randomize
122        stall = stall_vec[0];
123        stall_vec = {stall_vec, stall_vec} >> 1;
124        if (!dummy && !rsp.stall_i)
125              ack = rsp.stb_o;
126        else
127              ack = 1'b0;
128
129        assert(req.randomize( ) with {data_i == datar; ack_i == local::this.ack; stall_i == stall;});
130        `uvm_info("SEQUENCE-Dbg", $sformatf("Tx randomize with data_i=0x%0h, ack_i=%0b, stall_i=%0b",
131                                            req.data_i, req.ack_i, req.stall_i), UVM_MEDIUM)
145        // Update num_seq_rec
146        if (rsp.cyc_o && !rsp.stb_o) begin
147           num_seq_rec++;
148           `uvm_info("SEQUENCE-Dbg", $sformatf("Increasing num_seq_rec"), UVM_MEDIUM)
149        end
```

Figura 93 Diferencias entre `simple_sequence` y `wb_stall_fixed_seq`

- **<u>Operación de escritura</u>**

La Figura 94 muestra la primera transacción de escritura desde el punto de vista de la interfaz AXI4. Al no haber modificado la secuencia correspondiente al agente AXI4, no hay diferencia con los valores mostrados en la Figura 78. En el instante t1 se indica en el bus `AWADDR` que se parte de la dirección `0x1000`. En esta ocasión, el parámetro `AWBURST` se establece a `0x0`, el parámetro `AWLEN` a `0x1` y el parámetro `AWSIZE` a `0x2`. Esto conlleva que la escritura sea de tipo *fixed* y que se van a enviar dos datos distintos de 32 bits. En el instante t2 se termina la transacción desde la interfaz AXI4 al finalizar el protocolo de *handshake* del canal `WDATA` con el último dato, comenzando el proceso en la interfaz *Wishbone*. Una vez finaliza este proceso, se lleva a cabo el envío de la respuesta por parte del DUV por el canal `WRESPONSE`, que ha sido OKAY como se muestra en el instante t3.

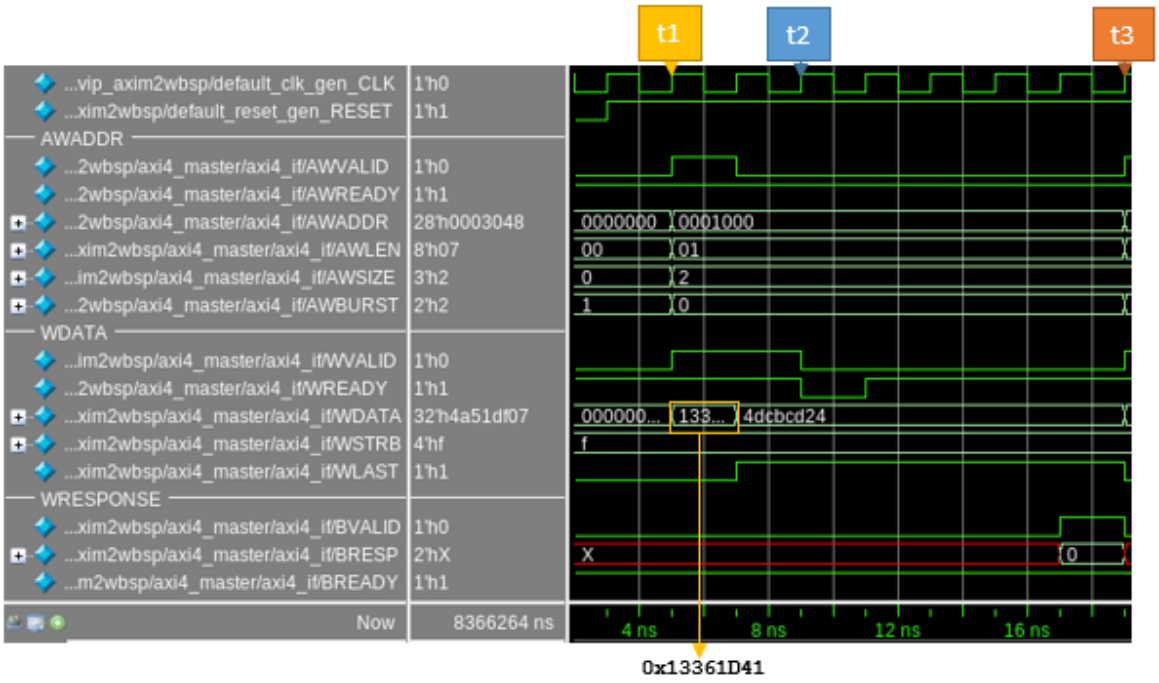

Figura 94 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 1

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 95. En el instante t1 comienza el ciclo de la transacción de escritura y se validan los datos enviados. En relación con las direcciones, se observa que la señal `addr_o` se mantiene fija en el valor `0x400`. En el instante t2 finaliza del ciclo, instante en el que se reconoce el último dato enviado. Esto conlleva a la respuesta generada en el canal `WRESPONSE`, como se mostró anteriormente. Aunque para esta ráfaga no haya sido relevante, cabe destacar que en este instante se activa la señal `stall_i`. Este hecho sí influirá en la siguiente ráfaga de datos que se gestione en esta interfaz.

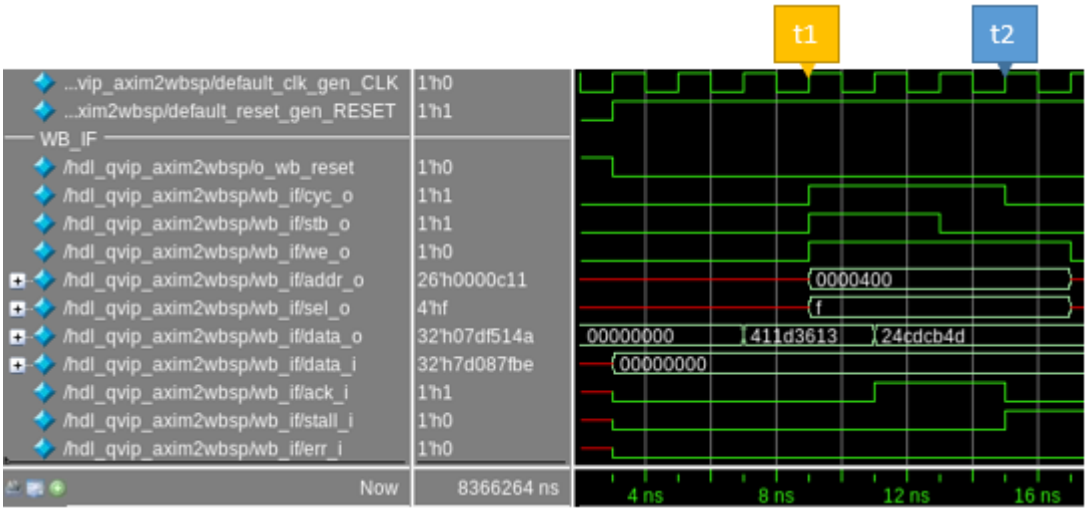

Figura 95 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 2

La Figura 96 muestra la segunda transacción de escritura en la interfaz AXI4. En el instante t1 se indica en el bus `AWADDR` que se parte de la dirección `0x2000`. En esta ocasión, el parámetro `AWBURST` se establece a `0x1` y el parámetro `AWLEN` a `0x3`. Esto conlleva que la escritura sea de tipo *incremental* y que se van a enviar cuatro datos distintos de 32 bits. En el transcurso de la transacción la señal `WREADY` se desactiva, instante t2, por lo que se mantienen los datos pausados esperando a que se reactive la señal, instante t3. Este efecto se produce también en el instante t4. En el instante t5 finaliza el envío de la transacción de escritura al DUV desde la interfaz AXI4, comenzando con el proceso en la interfaz *Wishbone*. Una vez finaliza este proceso, se lleva a cabo el envío de la respuesta por el canal `WRESPONSE`, que ha sido OKAY como se muestra en el instante t5.

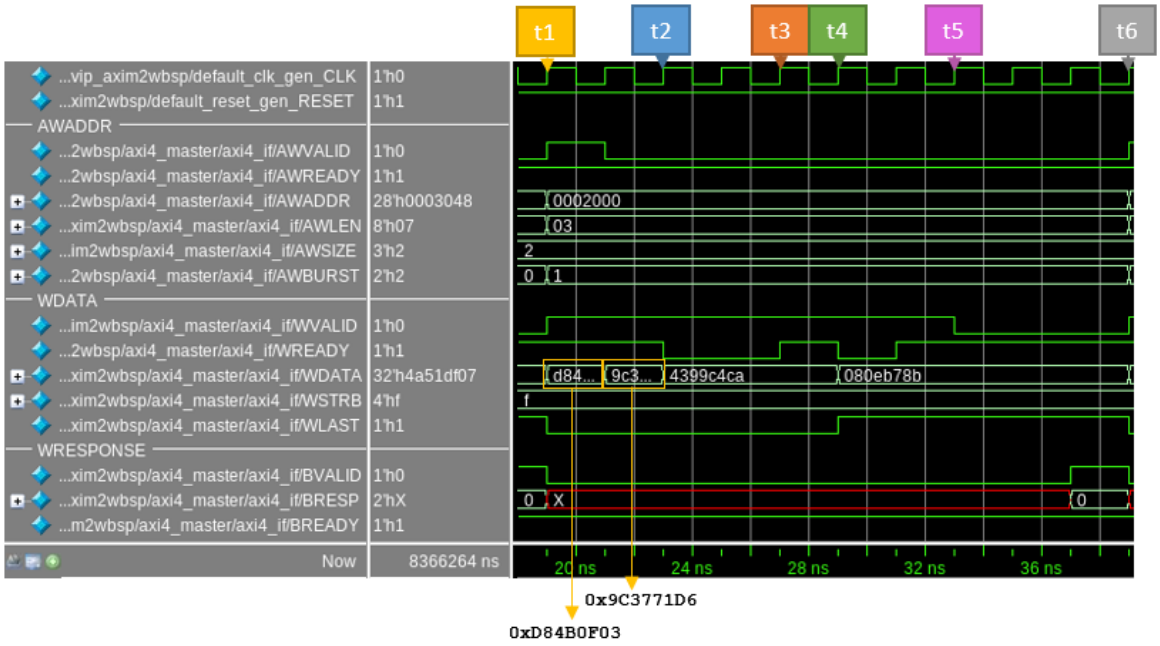

Figura 96 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 3

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 97. En el instante t1 comienza el ciclo de la transacción de escritura y se validan los datos enviados. En relación con las direcciones, se observa que la señal `addr_o` se incrementa consecutivamente, desde el valor `0x800` al valor `0x803`. Este incremento sucede cada vez que se completa una escritura de 32 bits de datos de cada dirección. Al validarse los 32 bits de cada dato, solo es necesaria una escritura para cada dirección. Hay que destacar que al comienzo del ciclo la señal `stall_i` está activada, lo que indica al DUV que debe mantener los datos. En el instante t2 se produce la desactivación de la señal `stall_i`, lo que permite que un ciclo de reloj más tarde, instante t3, se active la señal `ack_i`. En este instante se valida el primer dato. Sin embargo, en el instante t4 se desactiva la señal `ack_i` al volver a activarse la señal `stall_i`, no pudiendo validar el dato presente a la entrada. Esto produce que los datos de entrada se mantengan pausados a la espera de la validación por parte del agente esclavo. En el instante t5 finaliza el ciclo, instante en el que se reconoce el último dato enviado. Esto conlleva a la respuesta generada en el canal `WRESPONSE`, como se mostró anteriormente.

Como se ha comentado anteriormente, que se encuentre la señal de *stall* activa influye en la disposición del agente esclavo reactivo a gestionar los datos que recibe. Esto también se ve reflejado en la interfaz AXI, ya que provoca que se desactiva la señal `WREADY`. Este efecto, en este caso, es más común que en ejemplos anteriores, tal y como se puede observar al comparar la Figura 80 y la Figura 96.

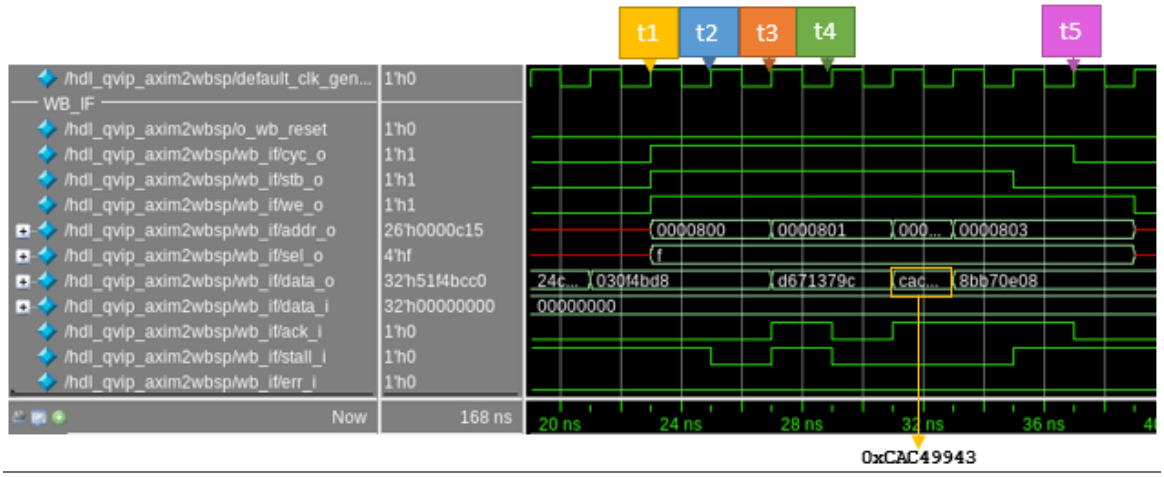

Figura 97 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 4

La Figura 98 muestra la tercera y última transacción de escritura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en los casos anterior, exceptuando las variaciones en la configuración de la transacción. En el instante t1 se indica en el bus `AWADDR` que se parte de la dirección `0x3048`. En esta ocasión, el parámetro `AWBURST` se establece a `0x2` y el parámetro `AWLEN` a `0x7`. Esto conlleva que la escritura sea de tipo *wrapped* y que se van a enviar ocho datos distintos de 32 bits. Como en el test anterior, la elección de la dirección y longitud se han elegido para satisfacer las necesidades de este tipo de ráfaga y de esta forma poder visualizarlo correctamente. En el transcurso de la transacción la señal `WREADY` se desactiva, instante t2, por lo que se mantienen los datos pausados esperando a que se reactive la señal, instante t3. Esto ocurre en otras ocasiones como en el instante t4. En el instante t5 finaliza el envío de la transacción de escritura al DUV desde la interfaz AXI4, comenzando con el proceso en la interfaz *Wishbone*. Una vez finaliza este proceso, se lleva a cabo el envío la respuesta por parte del DUV por el canal `WRESPONSE`, que ha sido OKAY como se muestra en el instante t6.

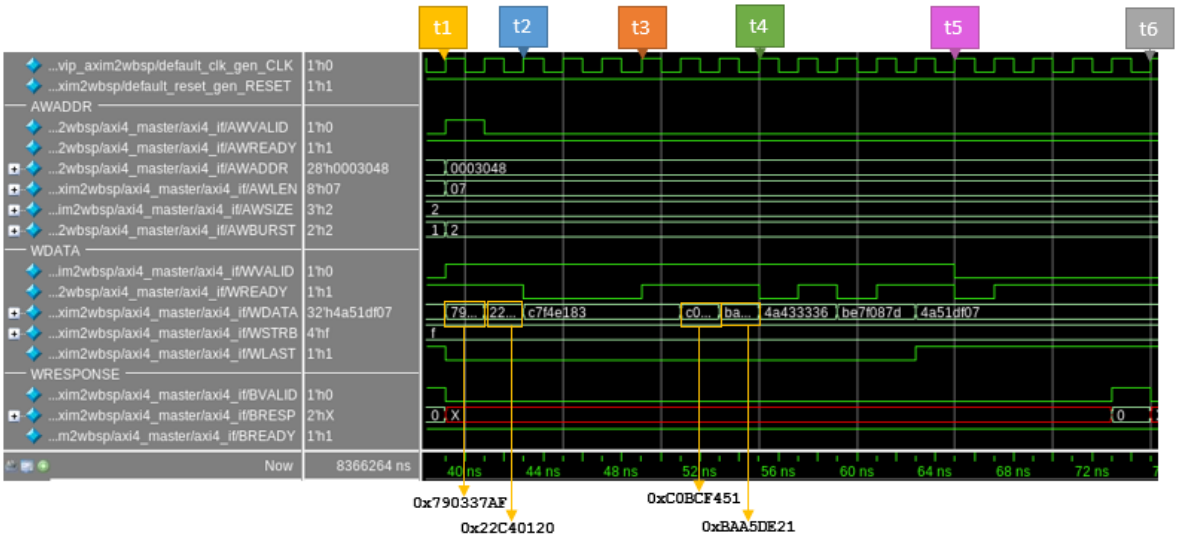

Figura 98 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 5

Tras recibir la dirección y el dato de escritura a través de la interfaz AXI4, el DUV gestiona esta información y la envía a través de la interfaz *Wishbone*, tal y como se muestra en la Figura 99. En el instante t1 comienza el ciclo de la transacción de escritura y se validan los datos enviados. En relación con las direcciones, se observa que la señal `addr_o` se incrementa consecutivamente, desde el valor `0xC12` al valor `0xC17` hasta el instante t3, en el que ocurre el efecto *wrapped* de este tipo de ráfaga. Cuando ocurre este efecto, se pasa de la dirección `0xC17` a la dirección `0xC10` y, por último, a la dirección `0xC11`, lo que confirma el efecto de envoltura. Este incremento sucede cada vez que se completa una escritura de 32 bits de datos de cada dirección. Al validarse los 32 bits de cada dato, solo es necesaria una escritura para cada dirección. El reconocimiento de los datos mediante la señal `ack_i` no comienza hasta el instante t2, que corresponde al siguiente ciclo de reloj desde que se desactiva la señal `stall_i`. En el instante t4 finaliza del ciclo, instante en el que se reconoce el último dato enviado. Esto conlleva a la respuesta generada en el canal `WRESPONSE`, como se mostró anteriormente. Al igual que en la transacción anterior, que haya ciclos en los que la señal `stall_i` está activa provocan que se produzcan más pausas en el envío de datos en la interfaz AXI4, tal y como se puede observar al comparar la Figura 98 con la Figura 82.

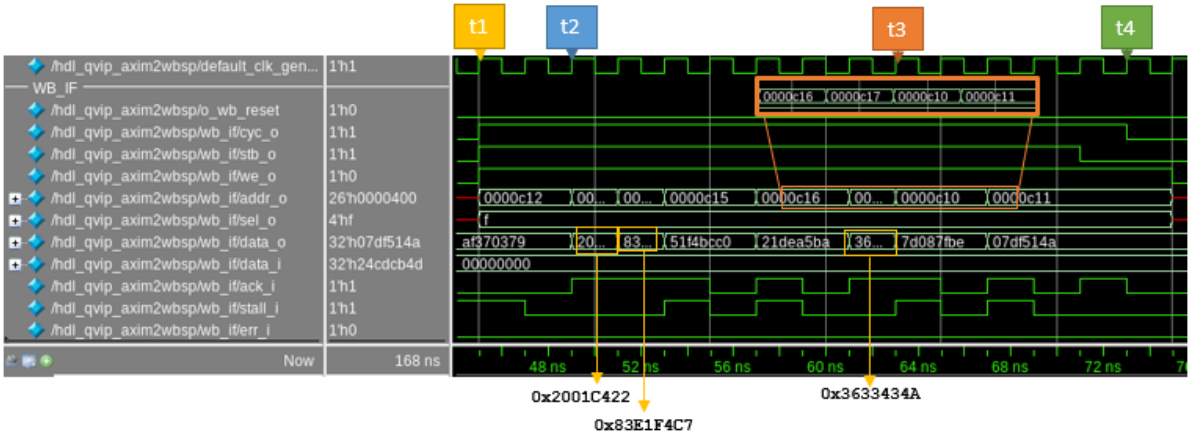

Figura 99 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 6

- **Operación de lectura**

El proceso de lectura se muestra en la Figura 100 y comienza con el canal `RADDR`. Como se busca finalizar el proceso de escritura/lectura iniciado en la etapa anterior, la dirección utilizada en este caso es la misma que se envió anteriormente en el canal *AWADDR*. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

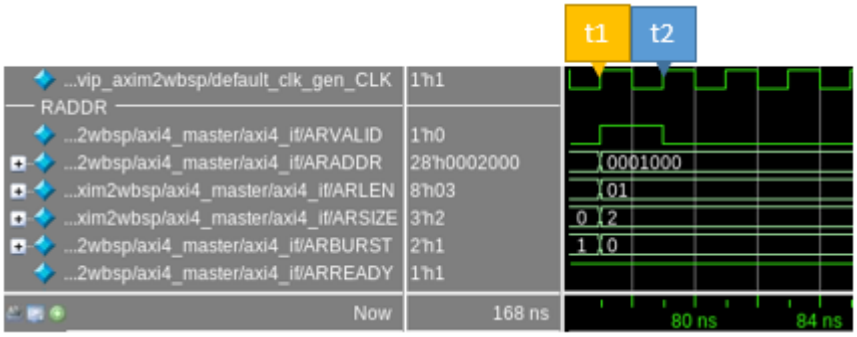

Figura 100 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 7

La Figura 101 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso `0x400`. A través del bus `data_i` el esclavo envía los datos al maestro. En el instante t2 se activa la señal de reconocimiento, `ack_i`. Además, en ese mismo instante se ha activado la señal `stall_i`. La activación de la señal `stall_i` provoca que se reconozca el último dato en el instante t3 y que el maestro dé por finalizado el ciclo en ese mismo instante, desactivando la señal `cyc_o`. Como se explicó en el test anterior, al ser una transacción *fixed* se lee dos veces el último dato que fue escrito.

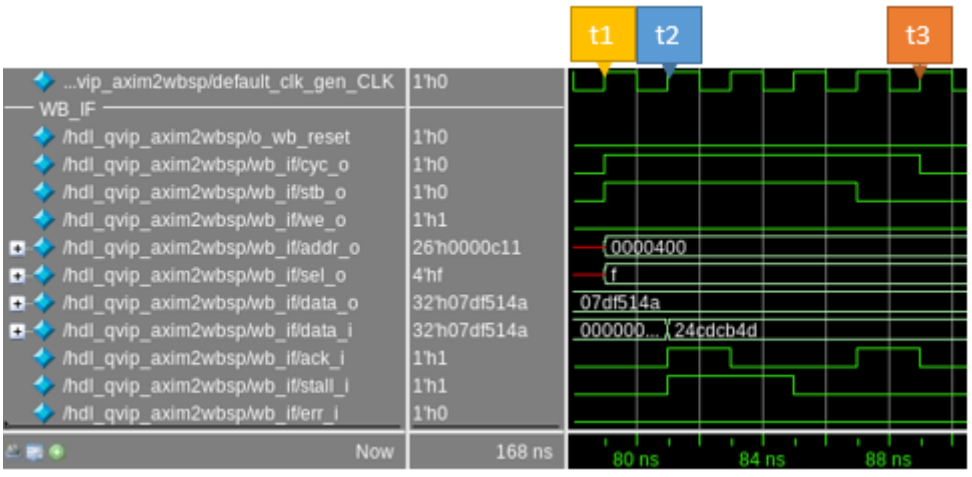

Figura 101 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 8

Para terminar con la lectura *fixed* finalizada la transacción en la interfaz *Wishbone*, el DUV responde al maestro AXI como muestra la Figura 102. En el instante t1 se inicia el protocolo. El protocolo lo inicia el DUV indicando al maestro con la señal `RVALID` que tiene un dato válido en el bus. En este caso, el maestro siempre tiene activa la señal `RREADY`, lo cual indica que está a disposición de recibir el dato. Hay que destacar que el dato proporcionado es exactamente el dato escrito, `0x24CDCB4D`, que se ha almacenado usando el convenio *Big Endian*. En el instante t2, el DUV desactiva la señal `RVALID` pausando la lectura de datos hasta que se vuelve a activar, instante t3. Esto se debe al retraso en el reconocimiento de la señal `ack_i` debido a la activación del *stall*, tal y como se mostró en la figura anterior. Hay que indicar que este funcionamiento es el funcionamiento esperado. En el instante t4, por último, se finaliza la lectura.

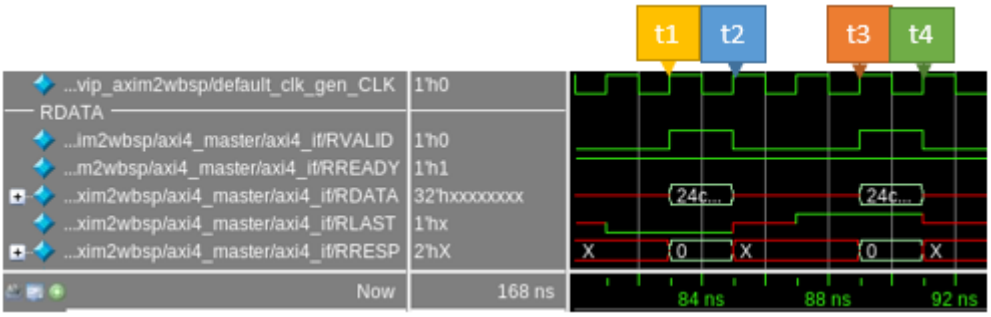

Figura 102 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 9

La Figura 103 muestra la segunda transacción de lectura en la interfaz AXI4. El funcionamiento es prácticamente el mismo que en el caso anterior con variaciones en los ajustes de la transacción. La dirección utilizada en este caso es la misma que se envió anteriormente en la escritura *incremental*, `0x2000`. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura correspondiente. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

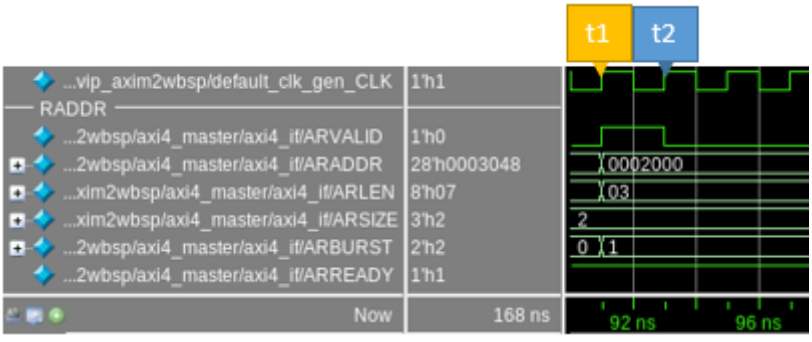

Figura 103 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 10

La Figura 104 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso se parte de la dirección `0x800` y finaliza en la dirección `0x803`. En el instante t2 se envía el primer dato a través del bus `data_i` y se activa la señal de reconocimiento, `ack_i`, y comienza la interacción

con la señal `stall_i`. En el instante t3 el maestro da por finalizado el ciclo al reconocer el último dato, desactivando la señal `cyc_o`.

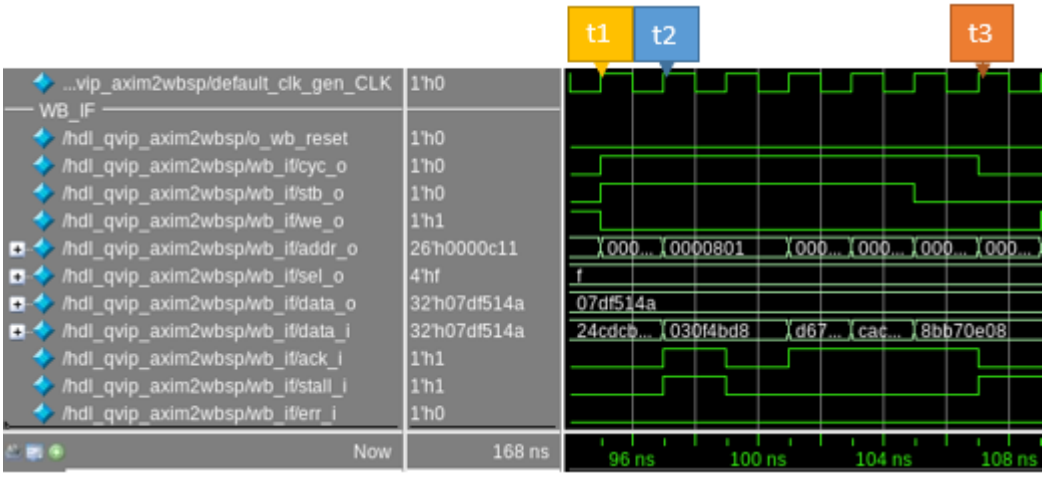

Figura 104 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 11

Para terminar con la lectura *incremental*, el DUV responde al maestro AXI como muestra la Figura 105. En el instante t1 comienza el protocolo de *handshake* y el envío de datos de lectura al maestro. En el instante t2, se desactiva la señal `RVALID` pausando la lectura de datos hasta que se vuelve a activar, instante t3, y continúa la recepción de datos. En el instante t4 se finaliza la lectura. Hay que destacar que los datos proporcionados son exactamente los datos escritos anteriormente, pero usando el convenio *Big Endian*.

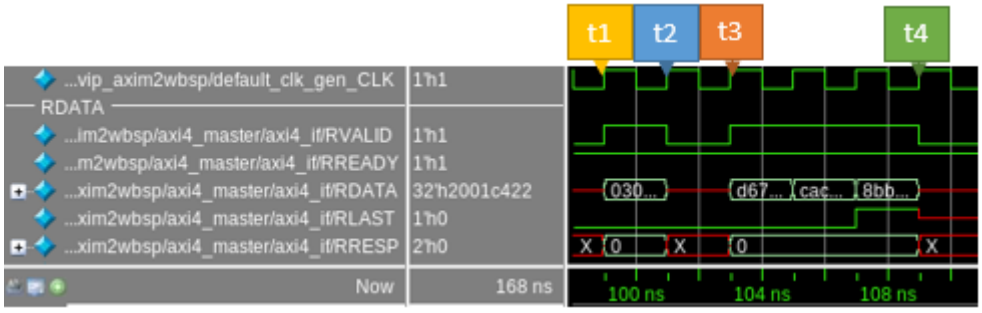

Figura 105 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 12

Para finalizar, la Figura 106 muestra la tercera y última transacción de lectura en la interfaz AXI4. La dirección utilizada en este caso es la misma que se envió anteriormente en la escritura *incremental*, `0x3048`. En el instante t1 se inicia el protocolo de *handshake* y se establecen los mismos valores de SIZE, LEN y BURST utilizados durante la escritura correspondiente. En el instante t2, y debido a que la señal de READY está activa, finaliza el ciclo de bus.

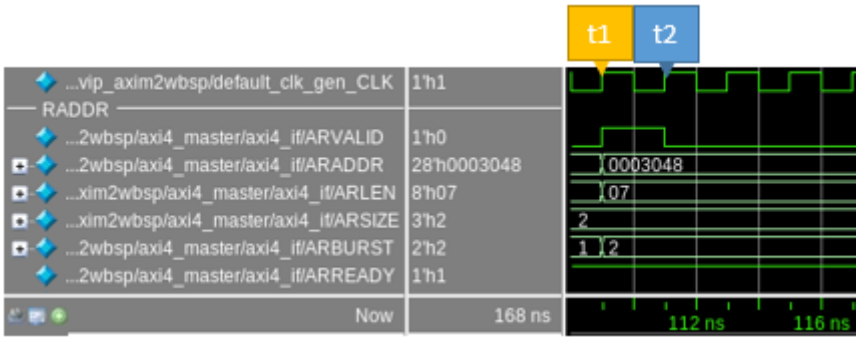

Figura 106 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 13

La Figura 107 muestra las señales de la interfaz Wishbone de este protocolo de lectura. Inicialmente, instante t1, el DUV proporciona la dirección de lectura del dato, en este caso se parte de `0xC12` y finaliza en la dirección `0xC11` debido al efecto de envoltura, que se produce en el instante t3. En este instante se pasa de la dirección `0xC17` a la dirección `0xC10`. En el instante t2 se envía el primer dato a través del bus `data_i` y se activa la señal de reconocimiento, `ack_i`, y comienza la interacción con la señal `stall_i`. En el instante t4 el maestro da por finalizado el ciclo al reconocer el último dato, desactivando la señal `cyc_o`.

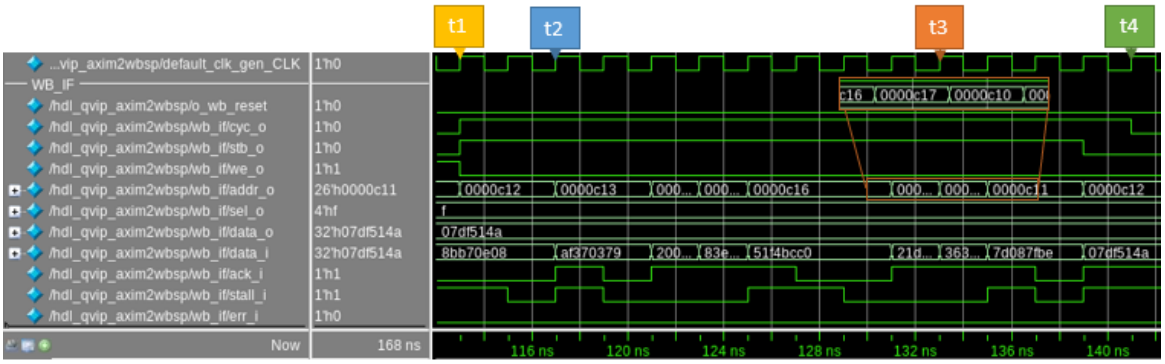

Figura 107 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 14

Para terminar con la lectura *wrapped*, el DUV responde al maestro AXI como muestra la Figura 108. En el instante t1 comienza el protocolo de *handshake* y el envío de datos de lectura al maestro. En los instantes t2 y t3 se continúa con la lectura de datos tras una pausa provocada por la desactivación por parte del DUV de la señal RVALID. En el instante t4 se finaliza la lectura al recibir el último dato. Hay que destacar que los datos proporcionados son exactamente los datos escritos anteriormente, pero usando el convenio *Big Endian*.

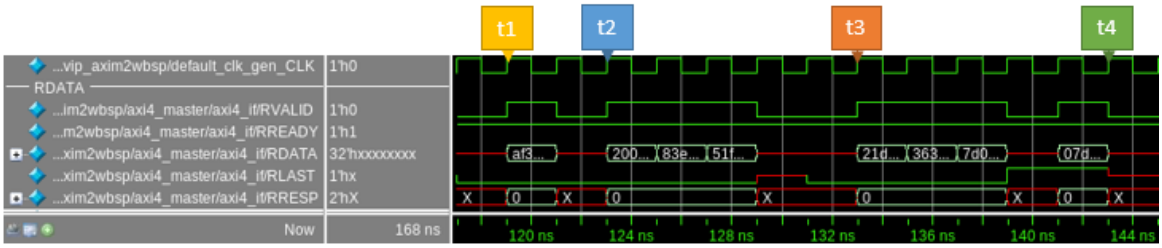

Figura 108 Simulación test qvip_axim2wbsp_fixed_stall_test: Parte 15

Capítulo 6: CONCLUSIONES Y LÍNEAS FUTURAS

6.1. CONCLUSIONES

Una vez finalizado el Trabajo Fin de Grado denominado "Diseño de un testbench UVM integrando IP de verificación *Mentor Graphics* y un IP propio, configurable y con soporte de cobertura", se comprueba que se han cumplido los objetivos propuestos inicialmente. El objetivo principal de este Trabajo Fin de Grado consiste en el diseño de un *testbench* UVM usando tanto un módulo QVIP de verificación proporcionado por *Mentor Graphics* como un IP de verificación propio, junto al desarrollo del mismo.

Respecto al DUV (*Device Under Verification*) que se ha verificado en este TFG, se ha escogido un módulo proporcionado por un tercero. El sistema escogido es un puente (*bridge*) entre protocolos, en este caso entre protocolos AXI4 y *Wishbone B4*. La función principal de este módulo es la conversión de comando de escritura y lectura entre sistemas con distintas interfaces, sin alterar la velocidad de transmisión de ambos buses. El procedimiento seguido para su verificación consistió en la estimulación de las señales de entrada del DUV y en el análisis de las señales de salida, comprobando así su comportamiento frente a estímulos controlados.

Para realizar la verificación se implementó un entorno de verificación en UVM en el que se desarrollaron los componentes que lo forman exceptuando el QVIP para AXI4, que actúa como componente UVM *Agent*. El *testbench* diseñado cuenta con una serie de secuencias que estimulan de manera eficaz las entradas del módulo. Estas secuencias no verifican todas las posibles situaciones en las que se puede ver comprometido el DUV. Esto no se ha podido completar debido al tiempo implicado en la realización de este Trabajo Fin de Grado y a la dedicación en la creación del entorno de verificación UVM. Además, se debe tener en cuenta tanto la dificultad propia de la metodología UVM como el aprendizaje del lenguaje *SystemVerilog* y de los protocolos de comunicación AXI4 y, en especial, *Wishbone B4*. Al contrario que para la interfaz *Wishbone,* las secuencias utilizadas para el protocolo AXI4 se programaron a partir de secuencias básicas proporcionadas por el QVIP, lo que facilitó el desarrollo de las mismas.

A la vista de los resultados obtenidos, todos los objetivos propuestos han sido cumplidos con éxito, por lo que el planteamiento propuesto desde un principio en este Trabajo Fin de Grado ha sido correcto. Esto se ha conseguido gracias a las indicaciones proporcionadas por los tutores de este TFG que, junto a los conocimientos adquiridos en el estudio de la metodología UVM y el módulo a verificar, han sido fundamentales en el desarrollo de este TFG.

6.2. LÍNEAS FUTURAS

En relación con las líneas futuras que se proponen para futuros Trabajos Fin de Grado o trabajos Fin de Máster, se consideran distintas posibilidades.

En primer lugar, se podría implementar un componente UVM *Agent* reactivo como el que se muestra en el apartado 3.4.5.3. , en el que se abandona la idea de utilizar una transacción *dummy* y se opta por modificar el componente UVM *Monitor* para que contribuya en el comportamiento reactivo del componente UVM *Agent*, acompañada de la implementación de una memoria FIFO que lo apoye.

Por otro lado, sería posible determinar y mejorar la *efectividad* de la verificación aplicada en el DUV al incluir métricas de cobertura (*coverage*).

Otra aportación al presente Trabajo Fin de Grado es la implementación de una función para la generación de la señal de `stall` que permita parametrizar el comportamiento de la misma. En este caso, además del comportamiento implementado en este TFG, se podría añadir una función aleatoria parametrizando el número máximo de ciclos consecutivos que podría estar activa dicha señal. Esto permitiría poder comprobar y forzar el protocolo *Wishbone* a situaciones no contempladas en este TFG.

Por último, se propone la implementación en el IP *Wishbone* del funcionamiento tipo FIFO, lo que permitiría verificar el caso de ráfagas tipo FIXED con este tipo de memoria. Esto posibilitaría adecuar el IP *Wishbone* a diferentes tipos de arquitecturas, muy comunes hoy en día en cualquier sistema de transferencia de datos, en los que las FIFO juegan un papel fundamental.

BIBLIOGRAFÍA

[1] "Part 8: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons." https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/ (accessed Feb. 08, 2022).

[2] *1800.2-2020 - IEEE Standard for Universal Verification Methodology Language Reference Manual*. IEEE, 2020.

[3] "Doulos." https://www.doulos.com/knowhow/systemverilog/what-is-systemverilog/ (accessed Feb. 08, 2022).

[4] "Part 10: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons." https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020-wilson-research-group-functional-verification-study/ (accessed Feb. 08, 2022).

[5] "Engineer - Consultant Jobs - ASIC Verification." https://www.chipright.com/consultant-jobs/asic-verification/ (accessed Feb. 08, 2022).

[6] "Design Verification Engineer | Empleos en Facebook." https://www.metacareers.com/v2/jobs/643699026189329/ (accessed Feb. 08, 2022).

[7] "AMBA AXI and ACE Protocol Specification Version E." https://developer.arm.com/documentation/ihi0022/e/ (accessed May 26, 2022).

[8] "Introduction to the Advanced Extensible Interface (AXI) - Technical Articles." https://www.allaboutcircuits.com/technical-articles/introduction-to-the-advanced-extensible-interface-axi/ (accessed May 26, 2022).

[9] "Wishbone B4 WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores Brought to You By OpenCores," 2010. [Online]. Available: www.opencores.org

[10] "Building Formal Assumptions to Describe Wishbone Behaviour." https://zipcpu.com/zipcpu/2017/11/07/wb-formal.html (accessed May 26, 2022).

[11] "Introduction To System Verilog | System Verilog Tutorial | System Verilog." http://www.asicguru.com/system-verilog/tutorial/introduction/1/ (accessed May 31, 2022).

[12] "Blog: An Introduction to Assertion-Based Verification - Part 1 - FirstEDA." https://firsteda.com/news/an-introduction-to-assertion-based-verification-part-1/ (accessed Jun. 11, 2022).

[13] "Universal Verification Methodology (UVM) 1.2 User's Guide," 2015, Accessed: Jun. 01, 2022. [Online]. Available: http://www.apache.org/licenses/.

[14] "UVM Phasing - The Art of Verification." https://www.theartofverification.com/uvm-phasing/ (accessed Jun. 07, 2022).

[15] "UVM Testbench Top." https://www.chipverify.com/uvm/uvm-testbench-top (accessed Jun. 08, 2022).

[16] “UVM Driver and Sequencer Communication | Universal Verification Methodology.” https://learnuvmverification.com/index.php/2015/07/07/uvm-driver-and-sequencer-communication/ (accessed Jun. 10, 2022).

[17] M. Litterick, J. Montesano, and T. Reddy, “MASTERING REACTIVE SLAVES IN UVM.” [Online]. Available: www.verilab.com

[18] Á. J. M. Florido, “Desarrollo de un testbench UVM integrando IP de verificación de Mentor Graphics (QVIP).” 2021.

[19] H. Foster - Siemens EDA, “IC/ASIC Functional Verification Trend Report,” 2020.

[20] “Simulation VIP | Cadence.” https://www.cadence.com/en_US/home/tools/system-design-and-verification/verification-ip/simulation-vip.html (accessed Jun. 13, 2022).

[21] “Verification IP.” https://www.synopsys.com/verification/verification-ip.html (accessed Jun. 13, 2022).

[22] “GitHub - ZipCPU/wb2axip: Bus bridges and other odds and ends.” https://github.com/ZipCPU/wb2axip (accessed May 23, 2022).

[23] “wb2axip/axim2wbsp.v at master · ZipCPU/wb2axip · GitHub.” https://github.com/ZipCPU/wb2axip/blob/master/rtl/axim2wbsp.v (accessed May 23, 2022).

PLIEGO DE CONDICIONES

PLIEGO DE CONDICIONES

En este apartado se detallarán las condiciones de los medios a partir de los cuales se ha desarrollado el presente Trabajo Fin de Grado. Se expone a continuación un despliegue de todos los medios con sus respectivas propiedades. Este apartado se ha dividido en Condiciones *hardware* y Condiciones *software*.

PC.1 CONDICIONES *HARDWARE*

En la Tabla 12 se muestran lo dispositivos *hardware* necesarios para el desarrollo de este Trabajo Fin de Grado.

Tabla 12 Condiciones *hardware*

Equipo/Dispositivo	Modelo	Fabricante
Ordenador personal	*Acer Aspire* F15: • CPU Intel Core i7-7500U a 2,7 GHz. • 16 GB RAM. • 512 GB SSD. • 2 puertos USB 3.0. • 1 puerto USB.	*Acer* Inc.
Estación de trabajo	*Acer Predator* G3: • CPU Intel Core i7-4790. • 8 GB RAM. • 1 TB HDD. • 4 puertos USB.	*Acer* Inc.

PC.1 CONDICIONES *SOFTWARE*

Una vez concretado los dispositivos *hardware* utilizados a lo largo del desarrollo de este Trabajo Fin de Grado, se muestran las herramientas software empleadas para la realización del mismo en la Tabla 13.

Tabla 13 Condiciones *software*

Equipo/Dispositivo	Modelo	Fabricante
Sistema operativo	*Microsoft windows* 10 *Home* 64 bits	*Microsoft Corporation*
Microsoft Office	2019	*Microsoft Corporation*
Microsoft Project	2018	*Microsoft Corporation*
Mozilla Firefox	102.0 – 64 *bits*	*Mozilla Corporation*
Sistema operativo	*Red Hat Enterprise Linux Server*	*Red Hat Inc.*
Simulador y compilador *QuestaSim*	Versión 2019.2	*Mentor Graphics, a Siemens Business*
Librería UVM	Versión 1.1d	*Mentor Graphics, a Siemens Business*
Adobe Acrobat Reader	*Versión* 2022.001.20142	*Adobe Systems Incorporated*

PRESUPUESTO

PRESUPUESTO

Para llevar a cabo el presente Trabajo Fin de Grado ha sido necesario el uso de distintos recursos. A partir de estos recursos se puede estimar la cuantía del valor de cada uno de ellos en función de su naturaleza y de la aportación que realiza al proyecto. Por ello, en este capítulo se detallarán los recursos humanos y materiales empleados. El presupuesto total para la realización de este Trabajo Fin de Grado se desglosará en los siguientes conceptos:

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

P.1. RECURSOS HUMANOS

Para el cálculo del coste asociado a las horas de trabajo empleadas durante el desarrollo de este Trabajo Fin de Grado se utiliza una ecuación recomendada por el Colegio Oficial de Ingenieros Técnicos de telecomunicación (COITT), correspondiente al cálculo de honorarios en función de las horas trabajadas dentro y fuera de la jornada laboral convencional. La ecuación para el cálculo de los costes de los Recursos Humanos (RRHH) se presenta a continuación:

$$Honorarios(€) = C_t * (74{,}88 * H_n + 96{,}72 * H_e)$$

Donde las incógnitas corresponden a lo siguiente:

- El término C_t representa el factor de corrección que se aplica en función de las horas invertidas en el proyecto.
- El término H_n hace referencia al número de horas trabajadas dentro de la jornada laboral convencional.
- El término H_e representa el número de horas especiales u horas extra realizadas fuera de la jornada laboral.

Los valores del factor de corrección C_t se muestran en la Tabla 14 de acuerdo con el procedimiento de cálculo marcado por el COITT:

Tabla 14 Factor de corrección según las horas trabajadas

Horas	Factor de corrección
Hasta 36	1,00
Desde 36 hasta 72	0,90
Desde 72 hasta 108	0,80
Desde 108 hasta 144	0,70
Desde 144 hasta 180	0,65
Desde 180 hasta 360	0,60
Desde 360 hasta 540	0,55
Desde 540 hasta 720	0,50
Desde 720 hasta 1080	0,45
Más de 1080	0,40

Según el Plan de Estudios del título Grado en Ingeniería en Tecnologías de la Telecomunicación, las horas destinadas a la asignatura Trabajo Fin de Grado son de 300 horas (12 ECTS). Debido a esto, en el caso particular de este TFG, el factor de corrección tendrá un valor de 0,6. Al utilizar la ecuación mostrada anteriormente el resultado es el siguiente:

$$Honorarios(€) = 0{,}6 * (74{,}88 * 300 + 96{,}72 * 0) = \ 13.478{,}4\ €$$

El coste asociado al tiempo invertido en términos de RRHH durante los últimos meses asciende a trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos (13.478,4 €). A este coste no se le han aplicado impuestos ni retenciones.

P.2. RECURSOS MATERIALES

Como se ha mencionado en el Pliego de Condiciones, en este TFG han sido utilizadas tanto herramientas *hardware* como *software*. Para realizar el cálculo del coste de cada uno de estos recursos se relaciona su coste total con el tiempo útil de los mismos. Para realizar este cálculo se utiliza la siguiente ecuación, teniendo en cuenta que se fija como referencia para el sistema de amortización lineal una vida útil de 3 años:

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

P.2.1. RECURSOS *HARDWARE*

En la Tabla 15 se muestran los resultados del coste total de los recursos *hardware* utilizados en el desarrollo de este TFG.

Tabla 15 Coste total de los recursos *hardware*

Equipo/dispositivo	Valor de adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Ordenador personal – *Acer Aspire*	650 €	36 meses	18,05 €	6 meses	108,3 €
Estación de trabajo – Acer *Predator*	1.000 €	36 meses	27,77 €	6 meses	166,62 €
Coste total					**274,92 €**

El coste total de los recursos *hardware* es de doscientos setenta y cuatro euros con noventa y dos céntimos (274,92 €).

P.2.2. RECURSOS *SOFTWARE*

En la Tabla 16 se muestra el coste total de los recursos *software* utilizados en el desarrollo del presente Trabajo Fin de Grado. Cabe destacar que la mayoría de estas herramientas han sido proporcionadas de forma gratuita por la Universidad de Las Palmas de Gran Canaria (ULPGC).

Tabla 16 Coste total de los recursos *software*

Equipo/dispositivo	Valor de adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Sistema operative *Microsoft Windows* 10 *Home* – 64 *bits*	145 €	36 meses	4,03 €	6 meses	24,18 €
Microsoft Office	0,00 € (licencia ULPGC)	36 meses	0,00 €	6 meses	0,00 €
Microsoft Project	0,00 € (licencia ULPGC)	36 meses	0,00 €	6 meses	0,00 €
Mozilla Firefox	0,00 € (software libre)	36 meses	0,00 €	6 meses	0,00 €
Sistema operativo	0,00 € (software libre)	36 meses	0,00 €	6 meses	0,00 €
Simulador y compilador *QuestaSim*	20.000,00 €	36 meses	555,55 €	6 meses	3.333,33 €

Librería UVM	0,00 € (librería de libre distribución)	36 meses	0,00 €	6 meses	0,00 €
Adobe Reader	0,00 € (software libre)	36 meses	0,00 €	6 meses	0,00 €
Coste total					**3.357,51 €**

El coste total de los recursos *software* es de tres mil trescientos cincuenta y siete euros con cincuenta y un céntimos (3.357,51 €).

P.3. MATERIAL FUNGIBLE

Los costes derivados del material fungible utilizado en este Trabajo Fin de Grado se muestran en la Tabla 17.

Tabla 17 Costes del material fungible

Concepto	Coste
Folios	10,00 €
Impresión de TFG	35,00 €
Encuadernado	5,00 €
Coste total	**50,00 €**

El coste relativo al material fungible es de cincuenta euros (50 €).

P.4. REDACCIÓN DEL TRABAJO

Partiendo del presupuesto del proyecto, se pueden calcular los costes asociados a la redacción de la memoria de este Trabajo Fin de Grado a partir de la siguiente fórmula:

$$R = 0{,}07 * P * C_n$$

Donde las incógnitas representan lo siguiente:

- El término R representa los honorarios derivados de la redacción del trabajo.
- El término P hace referencia al presupuesto del trabajo.
- El término C_n representa el coeficiente de corrección en función del presupuesto calculado.

El valor del presupuesto (P) se obtiene mediante la suma del coste relativo a los RRHH y los recursos materiales, tanto *hardware como* software, como muestra la siguiente ecuación:

$$P = 13.478{,}4 + 274{,}92 + 3.357{,}51 = 17.110{,}83 \text{ €}$$

Según el COITT, el coeficiente de corrección C_n tendrá por valor la unidad si el término P tiene un valor inferior a los 30.050,00 €. Por lo tanto, la ecuación queda de la siguiente forma:

$$R = 0{,}07 * 17.110{,}83 * 1 = 1.197{,}76\ €$$

El coste derivado de la redacción del presente Trabajo Fin de Grado es de mil ciento noventa y siete euros con setenta y seis céntimos (1.197,76 €).

P.5. DERECHOS DE VISADO DEL COITT

El Colegio de Ingenieros Técnicos de Telecomunicación proporciona anualmente valor a los costes asociados a los derechos de visado por la ejecución de proyectos técnicos de carácter general. De esta forma, para el año 2022 estos costes se calculan de la siguiente manera:

$$V = 0{,}006 * P_1 * C_1 + 0.003 * P_2 * C_2$$

Donde estas incógnitas representan lo siguiente:

- El término V es el coste final del visado.
- El término P_1 representa el presupuesto del proyecto.
- El término C_1 hace referencia al coeficiente reductor en función del presupuesto.
- El término P_2 representa el presupuesto de ejecución material correspondiente a la obra civil.
- El término C_2 hace referencia al coeficiente de corrección en función del presupuesto P_2.

En el caso de este TFG, se puede determinar que el coste asociado al presupuesto P_2es igual a 0,00 € por lo que no será necesario aplicar el coeficiente C_2. Respecto al coeficiente C_1, como el presupuesto P_1 no supera los 30.050, 00 €, tiene un valor igual a 1. En este caso se suma al presupuesto P_1 el coste asociado a la redacción del trabajo, por lo que se obtienen un valor de 18.308,59 €. La ecuación quedaría de la siguiente manera:

$$V = 0{,}006 * 18.308{,}59 * 1 = 109{,}85\ €$$

El coste asociado a los derechos de visado del presupuesto es de ciento nueve euros con ochenta y cinco céntimos (109,85 €).

P.6. GASTOS DE TRAMITACIÓN Y ENVÍO

Los gastos de tramitación y envío suponen seis euros (6 €) por cada documento visado, por lo que, en este caso, solo corresponde a la presente memoria.

P.7. COSTE TOTAL DEL PROYECTO

Una vez obtenidos todos los costes asociados a las diferentes actividades que forman parte del desarrollo de este Trabajo Fin de Grado, se puede obtener el coste total del proyecto. A este proyecto se le aplica el Impuesto General Indirecto Canario (IGIC) del siete por ciento (7%). El coste final del proyecto se muestra en la Tabla 18.

Tabla 18 Coste total del Trabajo Fin de Grado

Concepto	Coste
Honorarios por tiempo empleado	13.478,4 €
Recursos hardware	274,92 €
Recursos software	3.357,51 €
Material fungible	50,00 €
Costes de redacción	1.197,76 €
Derechos del visado del COITT	109,85 €
Costes de tramitación y envío	6,00 €
Subtotal	18.474,44 €
IGIC (7%)	1.293,21 €
Coste total	**19.767,65 €**

El importe final al que asciende el presupuesto del Trabajo Fin de Grado “Diseño de un *testbench* UVM integrando IP de verificación *Mentor Graphics* y un IP propio, configurable y con soporte de cobertura” es de diecinueve mil setecientos sesenta y siete euros con sesenta y cinco céntimos (19.767,65 €).

Fdo.: D. Daniel Baute Trujillo

En Las Palmas de Gran Canaria a 14 de julio de 2022