UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE
TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

"Desarrollo de una librería de secuencias UVM para la verificación de un módulo *crossbar* con interfaz AXI-4 usando IP de verificación de Mentor Graphics"

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas Electrónicos Autor:
D. Miguel Quevedo Rodríguez

Tutores: Dr. D. Valentín De Armas Sosa
Dr. D. Félix B. Tobajas Guerrero

Fecha: 10 de junio de 2022

RESUMEN

En este mundo de constante innovación, donde se producen cada vez más productos de sistemas hardware digitales, la verificación de estos está adquiriendo gran importancia, siendo en alguna medida la fase la que más tiempo se le dedica dentro del proceso de diseño.

Debido a que los sistemas cada vez son más complejos y con más número de componentes, este proceso de verificación es cada vez más dificultoso. Es por lo que cada vez se impone la necesidad de buscar estándares y métodos de verificación que simplifiquen estas funciones. Para solventar estos inconvenientes, surge la metodología Universal Verification Methodology (UVM), que se basa en el lenguaje de descripción y de verificación hardware SystemVerilog.

El objetivo principal de este Trabajo Fin de Grado (TFG), es realizar todas las secuencias de test, de un *testbench* UVM usando un IP de verificación de Mentor Graphics, de Siemens Business, que verifique las especificaciones descritas para un módulo *crossbar* con interfaz AXI4 que representará el diseño a verificar (DUV) en el presente TFG. El módulo IP está desarrollado con la metodología UVM y utiliza el protocolo de comunicación AMBA AXI4 y es totalmente configurable, pudiendo de esta forma verificar que el diseño cumple o no con sus especificaciones. Este TFG supone la continuación al Trabajo Fin de Grado realizado por Álvaro José Moreno Florido [1], con título “Desarrollo de un *testbench* UVM integrando IP de verificación de Mentor Graphics (QVIP)”. En este proyecto se definían todos los interfaces y unidades necesarias para la verificación de un diseño, utilizando las IP de verificación de Mentor Graphics (QVIP),

En primer lugar, se efectuará una etapa de estudio donde se profundizará en el funcionamiento del protocolo AXI4 y en los conceptos necesarios de la metodología UVM. En segundo lugar, se estudiará el IP de verificación de Mentor Graphics y la herramienta QVIP *configurator*, la cual creará el entorno de verificación UVM que se utilizará para la estimulación del módulo IP que se desea verificar. Tras esto, se estudiará en profundidad el dispositivo a verificar (DUV), para entender su funcionamiento y, reutilizando el *testbench* UVM creado en el TFG [1], se creará una librería de secuencias que verifiquen las especificaciones descritas para el módulo.

ABSTRACT

In this world of constant innovation, where more and more products of digital hardware systems are produced, the verification of these device is acquiring great importance, also this is the phase that is dedicated more time in the design process.

Because systems are becoming more complex and with more components, this verification process is increasingly difficult. That is why every time arises the need to look for standards and verification methods that simplify these functions. In this way is that arises the Universal verification Methodology to solve these drawbacks. This Methodology is based on the SystemVerilog hardware description and verification language.

The main objective of this Final Degree Project (TFG) is to perform all the test sequences, of a UVM *testbench* based an IP of verification of Mentor Graphics, from Siemens Business, just to verify the specifications described for the IP module. This IP module is developed with the UVM methodology and uses the AMBA AXI4 communication protocol and is fully configurable, thus being able to verify that the design complies or not with its specifications. This TFG is the continuation of the Final Degree Project carried out by Álvaro José Moreno Florido [1], entitled "Development of a UVM using a Mentor Graphics verification IP (QVIP)". In this project using the Mentor Graphics Verification IPs (QVIP), all the interfaces and units necessary for the verification of an IP were defined.

Firstly, a study stage will be carried out where the functioning of the AXI4 protocol and the necessary concepts of the UVM methodology will be deepened. Secondly, the verification IP of Mentor Graphics and the QVIP *configurator* tool will be studied, that will create the UVM verification environment that will be used for the stimulation of the IP module that you want to verify. After this, the device to be verified (DUV) will be studied in depth, to understand its operation, and reusing the UVM *testbench* created, we will create the library of sequences that verify the specifications described for the module.

ÍNDICE DE CONTENIDOS

Contenido

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ACRÓNIMOS

- ABV - *Assertion-Based Verification*
- AHB - Advanced High-performance Bus
- AMBA - Advanced Microcontroller Bus Architecture
- AP – Analysis Port
- API – Application programming interface
- ASIC - Application-Specific Integrated Circuit (ASIC)
- AVM - *Advanced Verification Methodology*
- BFM – Bus Functional Model, Modelo Funcional del bus
- COITT – Colegio Oficial de Ingenieros Técnicos de Telecomunicación
- DPI - *Direct Programming Interface*
- DUT – Device Under Test
- DUV - Device Under Verification
- ECTS – European Credit Transfer and Accumulation System
- EDA - *Electronic Design Automation*
- EITE – Escuela de Telecomunicación y Electrónica
- eRM - *e Reuse Methodology*
- 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*
- IGIC – Impuesto General Directo Canario
- IP – Intellectual Property
- MDV - *Metric-Driven Verification*
- OOP - *Object-Oriented Programming*
- OVM - *Open Verification Methodology*
- OVM - *Open Verification Methodology*
- RAL – Register Abstraction Layer
- RO – Read Only

- RRHH – Recursos Humanos
- RTL - *Register-Transfer Level*
- RVM - Reference Verification Methodology
- RW – Read/Write
- SiP – System in Package
- SoC – System on Chip
- SV – System Verilog
- TFG – Trabajo Fin de Grado
- TFM – Trabajo Fin de Máster
- TLM - *Transaction-Level Methodology*
- ULPGC – Universidad de Las Palmas de Gran Canaria
- UVC – Universal Verification Component
- UVM – Universal Verification Methodology
- VHDL - *Very High Speed Integrated Circuit Hardware Description Language*
- VMM - *Verification Methodology Manual*
- WO – Write Only

MEMORIA

INTRODUCCIÓN

En este primer apartado se expondrán las necesidades y motivaciones que justifican la elaboración de este Trabajo Fin de Grado (TFG). Además, se plantean los objetivos a alcanzar durante su elaboración y el objetivo final que se conseguirá con la finalización de este TFG. Finalmente, también se muestra la estructura del documento, proporcionando una estructura clara de los puntos que se van a describir en la memoria.

I.1. ANTECEDENTES

En este mundo de constante innovación, donde se producen cada vez más productos de sistemas hardware digitales, la verificación de estos está adquiriendo gran importancia, siendo en alguna medida la fase la que más tiempo se le dedica dentro del proceso de diseño.

Se debe tener en cuenta que las tareas relacionadas con la verificación funcional de un sistema llegan a consumir más de un 50% del tiempo designado al diseño [2] como se puede ver en la Figura 1.

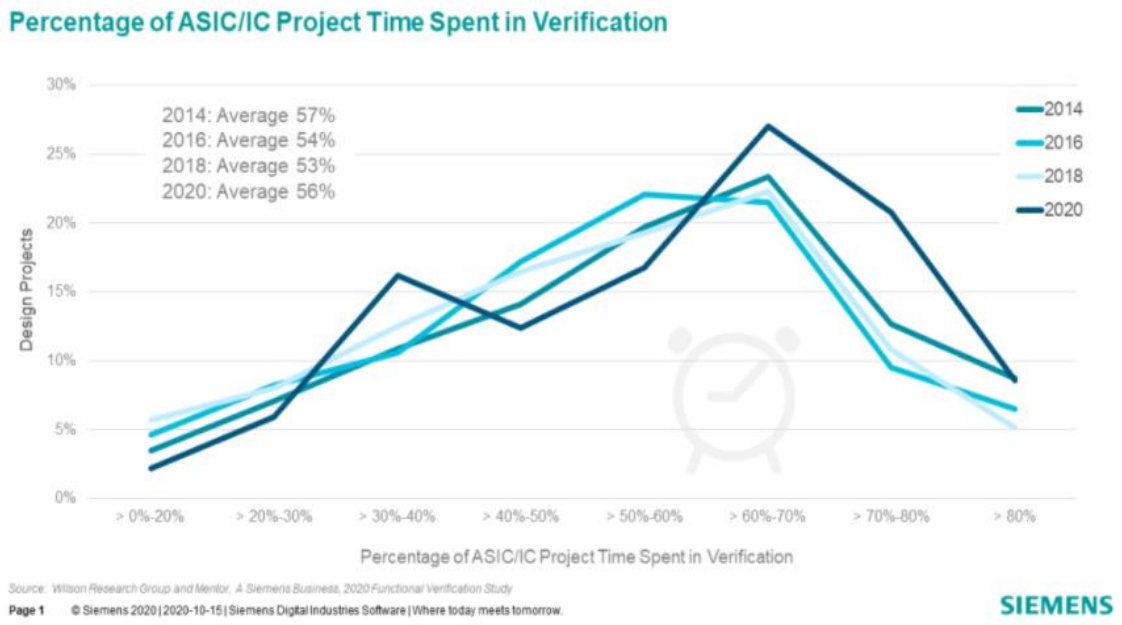


Figura 1 Porcentaje de tiempo del proyecto ASIC/IC dedicado a la verificación [2]

Debido a que los sistemas cada vez son más complejos y con más número de componentes, este proceso de verificación es cada vez más dificultoso. Es por lo que cada vez se impone la necesidad de buscar estándares y métodos de verificación que simplifiquen estas funciones. En la búsqueda de resolver los problemas añadidos por el

aumento de complejidad de los diseños actuales han sido muchas las propuestas de lenguajes de verificación. Estas propuestas han surgido desde los propios lenguajes de diseño, como VHDL y Verilog, hasta lenguajes propios y propietarios tales como Vera y Specman. Sin embargo, uno de los lenguajes que mejor aceptación ha tenido es SystemVerilog [3]. Gran parte de este éxito se debe a que, además de ser un lenguaje para verificación, también es válido para el diseño, lo que evita que el ingeniero de verificación tenga que lidiar con lenguajes distintos.

SystemVerilog se ha desarrollado gradualmente desde el anterior estándar el Accellera SystemVerilog 3.0 en 2002 hasta el estándar IEEE-1800 SystemVerilog en 2012. Después de más de diez años de actualizaciones y mejoras, SystemVerilog se ha convertido en el lenguaje dominante en el campo de verificación de sistemas integrados. Este punto se puede ver en el índice de uso y el gráfico de tendencias en la Figura 2.

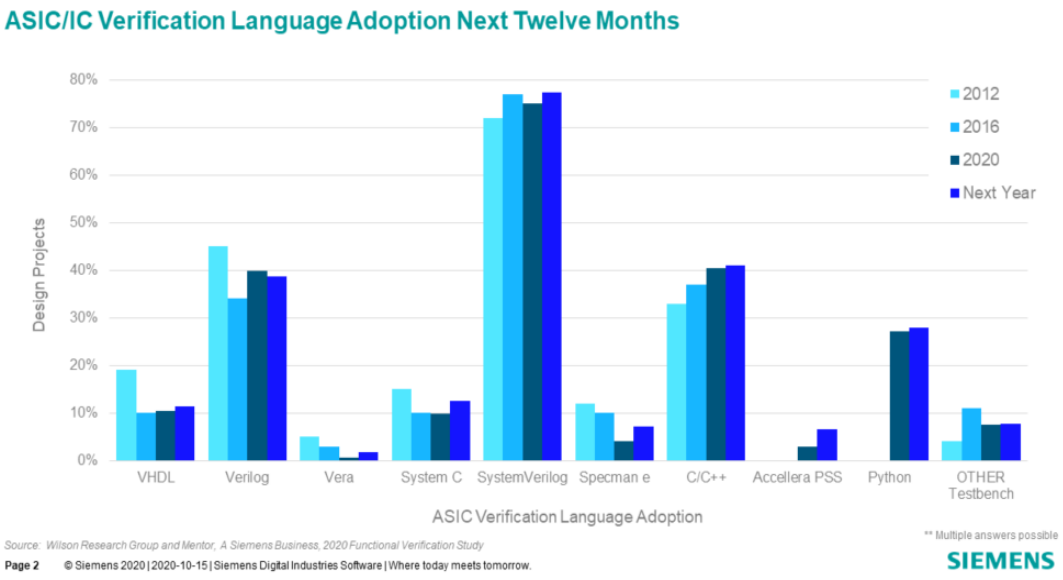


Figura 2 Lenguajes usados para verificación [2]

La evolución de los lenguajes de verificación da paso a la aparición de metodologías de verificación, que buscaban simplificar y dar la posibilidad de realizar entornos de verificación entre diseños. De todas las metodologías cabe destacar la *Universal Verification Methodology* que desde su aparición ha acaparado más de un 40% de cuota de uso tal y como se observa en la Figura *3*.

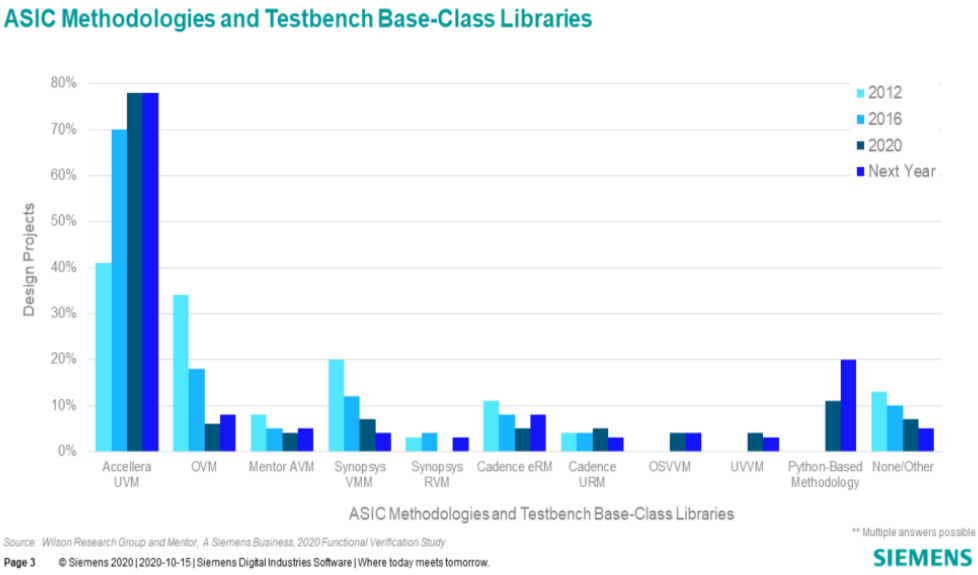


Figura 3 Metodologías usadas para verificación. [4]

Dentro del Instituto Universitario de Microelectrónica Aplicada (IUMA), la división de Diseño de Sistemas Integrados (DSI) se ha especializado, entre otras líneas, en la verificación de sistemas integrados. En sus comienzos, la verificación era una verificación *ad hoc*, ajustada únicamente a los requisitos del sistema o del IP (*Intellectual Property*) que se pretendía verificar. Con la salida y la estandarización de la metodología UVM, esta división se ha especializado en el uso de la metodología UVM.

La realización de un entorno de verificación usando la metodología UVM lleva asociado entre otras, el conocimiento del lenguaje SystemVerilog, concretamente, todos los aspectos relacionados con la programación orientada a objetos, modelado a nivel de (TLM) y toda la librería de clases propias de la metodología. Debido a la limitación de tiempo que un Trabajo de Fin de Grado tiene asignado en el plan de estudios, la realización de los Trabajos Fin de Grado de esta línea se ha planteado de forma escalonada. Es decir, cada Trabajo Fin de Grado sirve de escalón para que el siguiente pueda avanzar en la consecución de sus objetivos.

Para facilitar las tareas de verificación, y haciendo uso de las herramientas de las principales compañías de diseño electrónico, las compañías EDA (*Electronics Design Automation*), tales como Siemens Business, Cadence o, incluso, Xilinx, han optado por ofrecer unas librerías de módulos de verificación que, si bien no eximen del uso del lenguaje SystemVerilog para la realización de un *testbench*, sí que facilita enormemente la implementación de estos. Estos módulos de verificación, denominados VIP

(*Verification Intellectual Property*) [5] son módulos configurables que se encargan de traducir, a nivel de señales, las operaciones descritas a nivel de transacciones (unidades abstractas que describen las operaciones a realizar sobre un IP o sistema).

Una vez generado el entorno para poder realizar el, uno de los pasos más importantes en la realización del *testbench*, con esta metodología, es describir todas las especificaciones que se quieren verificar y, en base de ello, generar las secuencias del *testbench*, que comprueben que se cumplen estas especificaciones. Es en esta etapa en la que se desarrolla este Trabajo Fin de Grado.

Este TFG supone la continuación al Trabajo Fin de Grado realizado por Álvaro José Moreno Florido [1], con título “Desarrollo de un *testbench* UVM integrando IP de verificación de Mentor Graphics (QVIP)”. En este proyecto se definían todos los interfaces y unidades necesarias para la verificación de un IP, utilizando las IP de verificación de Mentor Graphics (QVIP), dejando el entorno de verificación completamente operativo y ejecutando únicamente, una secuencia básica para comprobar el funcionamiento del entorno.

En este Trabajo Fin de Grado, se realizarán todas las secuencias de test que verifiquen las especificaciones descritas para el módulo, pudiendo de esta forma, verificar que el diseño cumple o no con sus especificaciones.

I.2. OBJETIVOS

El objetivo principal de este Trabajo Fin de Grado consiste en el diseño de una librería de secuencias de test UVM para ejecutar sobre un *testbench* UVM construido sobre la base de IP de verificación de Mentor Graphics. El módulo a verificar, o DUV será un *crossbar* con interfaz AXI-4. Este módulo es configurable en cuanto al número de puertos de entrada (*máster*) y numero de puertos de salida (*slaves*). El desarrollo de las secuencias se ajustará a la metodología UVM.

Para la realización del *testbench* hay que tener en cuenta que cada una de las interfaces maestras o esclavas del DUV siguen un protocolo de comunicación AXI4 configurable. Para la realización de este TFG se parte de un entorno de referencia, desarrollado en el Trabajo Fin de Grado de Álvaro Moreno Florido [1], en el que se reutilizaran todas sus interfaces creadas y se procederá a generar las secuencias de test que permitan verificar todas y cada una de las características y propiedades detalladas en las especificaciones del módulo IP a verificar (*crossbar*).

Para poder realizar nuestro cometido, el objetivo final se puede dividir en los siguientes objetivos:

O1. Entender 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.

O2. Conocer el funcionamiento del protocolo AMBA AXI4, necesario para poder realizar las tareas de comprobación, a nivel de interfaz.

O3. Comprender la herramienta Mentor Graphics, su entorno y su funcionamiento.

O4. Conocer el funcionamiento del módulo IP que se va a verificar, lo que va a permitir describir e implementar las secuencias necesarias para comprobar su correcto funcionamiento.

O5. Obtener y acotar en un listado las características a comprobar del IP de estudio, que limitarán el alcance de este TFG.

O6. Ejecutar cada una de las secuencias sobre el *testbench* UVM de referencia para la verificación de todas las propiedades y características recogidas en el *objetivo **O5***.

Debido a la necesidad de ejecutar las secuencias, que se van a crear, sobre su *testbench* de referencia, no se descarta la necesidad, en caso de incompatibilidad, de

tener que modificar la clase test del citado *testbench*, encargada de ejecutar las secuencias.

I.3. PETICIONARIO

El peticionario del presente 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).

I.4. ESTRUCTURA DEL DOCUMENTO

El presente documento se ha dividido en cuatro secciones: Memoria, Pliego de Condiciones, Presupuesto y Anexo.

La Memoria se compone de una introducción y 7 capítulos, con el siguiente contenido y distribución,

Introducción: Este primer apartado en el cual estamos inmersos, inicia la Memoria y se exponen los antecedentes y las necesidades que justifican la realización de este Trabajo Fin de Grado. También incluye los objetivos que persigue este TFG, el peticionario y la estructura del presente documento.

Capítulo 1: Estructura de un entorno de verificación UVM. El objetivo de este capítulo es proporcionar un exhaustivo análisis de la metodología UVM y el entorno de verificación que se crea siguiendo dicha metodología. Se definirá el concepto de verificación de sistemas hardware digitales, profundizando en la definición de la verificación funcional. Además, se describirán los antecedentes y se detallarán las características propias de esta metodología, así como sus principios básicos de funcionamiento y los diferentes componentes que forman parte de un entorno de verificación UVM, con especial importancia en los diferentes tipos de interconexiones existentes entre los componentes de un entorno UVM, atendiendo a sus funcionalidades y modelos de programación.

Capítulo 2: **Módulo QVIP de *Mentor Graphics* con interfaz AXI4**, En este capítulo se profundiza en el funcionamiento de la interfaz *Advanced Extensible Interface* 4 (AXI4), siguiendo el protocolo *Advanced Microcontroller Bus Architecture* (AMBA). Se detallan los fundamentos teóricos en los que se basan las transacciones de datos del protocolo

AMBA AXI4, incluyendo las diferentes señales que se pueden encontrar. Se presenta su modelo funcional, se analiza el funcionamiento del protocolo haciendo especial hincapié, en lo referente a la transmisión de los datos. A continuación, este capítulo presenta el IP de verificación que se usará como base para la verificación del DUV. En este caso, será el *Questa Verification Intellectual Property* (QVIP), con interfax AXI4. Se analizan sus diferentes elementos que lo componen, incluyendo algunas bibliotecas de componentes desarrollado en *SystemVerilog* UVM. Finalmente se muestra un mapa de todos los archivos y carpetas que generan el entorno de verificación.

Capítulo 3: Diseño a verificar, *testbench* de referencia y plan de verificación. En este capítulo se muestra el funcionamiento del DUV elegido sobre el cual se van a ejecutar las secuencias de verificación durante este TFG. Este DUV se obtiene de una plataforma de desarrollo de IP con licencia GNU, desarrollado de forma independiente a los desarrolladores del IP de verificación. Se hará hincapié en el módulo de adaptación que se ha tenido que diseñar para adaptar este DUV a la metodología UVM compatible con el protocolo AXI-4.

Se analiza el entorno UVM desarrollado en el TFG de Álvaro Moreno Florido [1], que servirá como plataforma sobre la cual se van a ejecutar las secuencias a desarrollar en este TFG.

Se muestran las secuencias básicas ejecutadas sobre el *testbench* de referencia. Estas secuencias incluyen ciertas modificaciones en relación con el código del *testbench* original proporcionado por los desarrolladores, para adquirir el conocimiento sobre el funcionamiento de las mismas.

A continuación, se exponen las características a comprobar del DUV a verificar. Se muestra el listado de características a verificar del módulo DUV. **P**artiendo de la hoja de características del módulo a verificar DUV, se generará un listado con las características a verificar. Se analizan las condiciones necesarias que deben cumplirse (estado del DUV) para poder llevar a cabo la verificación de cada una de ellas. Por último, se realiza un análisis de todas las características funcionales del módulo a verificar anotando, además, qué requisitos previos se deben cumplir para realizar su verificación.

Capítulo 4: Secuencias de verificación UVM. Este capítulo está dedicado a la estructura de las secuencias de verificación, su configuración, sus elementos y el detalle de cómo se ejecutan. Además, se describe el uso de las API proporcionadas para el IP de verificación (QVIP), se describe el uso y la particularidad de cada una de estas API.

Capítulo 5: Ejecución del plan de verificación. Este capítulo se centra en el objetivo

principal del presente TFG, como es la generación de las secuencias para la comprobación de las características del módulo DUV. El número de secuencias, así como su tipología, debe estar en concordancia con el listado de características obtenidas en el capítulo anterior. Cada una de estas secuencias se ejecuta sobre el *testbench* UVM de referencia. Además, para cada secuencia se muestra el resultado de su ejecución, así como los cronogramas obtenidos

Capítulo 6: Desarrollo de una agente de configuración. Este capítulo se presenta como una ampliación al TFG planteado inicialmente. Esta ampliación está motivada porque durante la ejecución de las secuencias de validación, se ha visto que siempre que se modifica una característica de configuración inicial, se tiene que modificar los archivos *verilog* que definen el módulo top y volver a compilar y ejecutar todo el software. Esto implica un aumento considerable en el proceso de verificación para establecer las señales del interfaz.

El propósito de este TFG es generar un módulo agente para establecer las señales del interfaz de control del DUV. El principal inconveniente viene dado por el hecho de que las señales de la interfaz de control del DUV no tienen ningún protocolo asociado, por lo que no se puede usar su IP de verificación (QVIP). Por este motivo es necesario desarrollar su agente propio que permita realizar la configuración como parte de una secuencia extra más de configuración pudiéndose ejecutar y modificar en tiempo de ejecución, sin necesidad de volver a compilar todo el diseño.

Capítulo 7: Conclusiones y líneas futuras. Una vez completados todos los objetivos establecidos, se recopilarán una serie de conclusiones obtenidas a lo largo del desarrollo de este Trabajo Fin de Grado. Además, se analizarán las futuras ampliaciones que podrían surgir a partir de este TFG.

CAPÍTULO 1: ESTRUCTURA DE UN ENTORNO DE VERIFICACIÓN UVM

En este capítulo se detallan las principales características de la metodología UVM. Se explica el concepto de verificación de los sistemas digitales, mostrando la importancia en el flujo de diseño de sistemas hardware digitales. Se muestran los conceptos básicos de la creación del entorno de verificación UVM y todos los componentes que lo conforman, sus conexiones, funciones y cómo operan en la verificación. Además, se muestra como esta metodología está completamente relacionada en el lenguaje HVL (*Hardware Verification Language*) de *Systemverilog,* por lo que se hace una introducción a dicho lenguaje.

1.1. INTRODUCCIÓN A LA VERIFICACIÓN DE SISTEMAS DIGITALES

UVM es un estándar del consorcio Accellera [1] que fue creado mediante el trabajo cooperativo de los más importantes 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. Por lo tanto, es sencillo referenciar dichas librerías al código fuente durante su desarrollo.

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. Con una metodología de verificación común se incrementa la productividad de manera significativa, debido a que todo el equipo trabaja a partir de las mismas plantillas, propias de la metodología, lo que ayuda a mantener la coherencia del código de verificación

La metodología UVM se creó a partir de la sólida base de la metodología OVM (*Open Verification Methodology*), que fue una metodología desarrollada conjuntamente por Mentor Graphics, a Siemens Company y Cadence Design Systems. A su vez, los conceptos de reutilización propios de la metodología OVM derivaron de las metodologías eRM (*e Reuse Methodology*) y URM (*Universal Reuse Methodology*) de la empresa Verisity Design. La metodología UVM también tiene características relacionadas con la metodología AVM (*Advanced Verification Methodology*) de Mentor

Graphics.

Debido a estas evoluciones de las metodologías de verificación, UVM se ha convertido en una metodología potente y flexible con la que es posible implementar entornos de verificación reutilizables, escalables e interoperables. A continuación, se van a presentar una serie de ventajas que proporciona esta metodología:

- La reutilización y la 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.
- 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.
- Ofrece la posibilidad de jerarquizar el entorno de verificación UVM y dividirlo en un conjunto de ítem de datos (estímulos y respuestas) y componentes de verificación (UVC). De esta manera, se consigue estructurar y organizar con facilidad los objetos y las funcionalidades que provee dicho entorno. Además, proporciona ciertos mecanismos de configuración que simplifican la configuración de los diferentes componentes que conforman el entorno de verificación.
- Un mecanismo de fábrica de componentes y objetos que simplifica su 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.

Debido a todas estas características, UVM se ha consolidado como la metodología de verificación ASIC/IC (y también en la verificación FPGA) más utilizada en la actualidad.

La adopción de UVM se hace evidente visualizado la Figura 4.

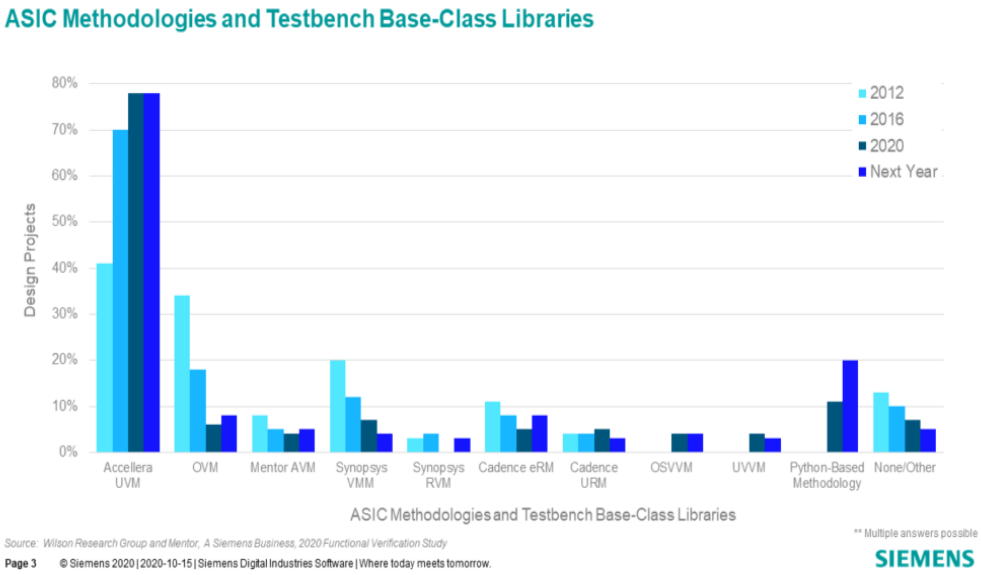


Figura 4 Metodologías usadas para verificación [2]

1.2. VERIFICACIÓN FUNCIONAL DE SISTEMAS HARDWARE DIGITALES

Uno de los problemas que han surgido en el diseño de sistemas digitales, es poder comprobar que los dispositivos diseñados funcionan correctamente y son capaces de cumplir con las especificaciones de diseño que se necesitan y se especifican.

Cada vez aumenta más la complejidad y la cantidad de funciones de los diseños electrónicos actuales. Esto supone un incremento de la dificultad del proceso de verificación. Debido a este aumento de la complejidad, junto al deseo de lograr un menor tiempo de acceso al mercado, el proceso de verificación ha obtenido especial relevancia en el diseño electrónico y ha provocado una rápida evolución de las herramientas destinadas a este fin.

El proceso de verificación es un proceso que se puede abordar desde distintos enfoques. Se puede llevar a cabo mediante técnicas de verificación estáticas (verificación formal) y dinámicas (verificación funcional). En el caso de este TFG se desarrollará a partir de un entorno de verificación funcional, basándose en la simulación o en la emulación de los diseños que se pretenden verificar.

Como se muestra en la Figura 5, en relación con la información [1] que se conoce del DUV, las implementaciones de la verificación funcional se pueden plantear siguiendo tres filosofías: verificación *Black-Box*, verificación *White-Box* y verificación *Grey-Box.*

Figura 5 Modelos de verificación

Black-box: En este modelo, no se conoce cómo está diseñado el sistema, los componentes que lo conforman, y cómo se comunican entre ellos. Por lo tanto, se enfatiza en qué es lo que hace el dispositivo, sin importar el cómo lo hace. El procedimiento para seguir se basa en el análisis de las interfaces de entrada y salida, por lo que se estimularán las entradas del sistema para comprobar su comportamiento a la salida del diseño. Con este enfoque se evalúa la especificación a cumplir de manera íntegra, como si de un "usuario final" se tratara. De esta manera, se consideraría una verificación exitosa si se consiguen los valores esperados a la salida en función de los estímulos generados en la entrada.

White-box: En este modelo, el ingeniero de verificación conoce en su totalidad el DUV. Por lo tanto, esta filosofía es contraria a la verificación *Black-Box*. Como se tiene acceso a la estructura del diseño, cualquier modificación afecta a la verificación funcional, pero la gran ventaja se produce durante el proceso de depuración *(debug),* ya que resulta más sencillo de elaborar. Esto se debe a que se pueden evaluar y gestionar todas las señales de flujo interno del dispositivo, que se dan durante la ejecución del *testbench.*

Grey-box: Finalmente en este modelo, el ingeniero de verificación conocerá parcialmente la estructura interna del dispositivo. Este enfoque resulta especialmente interesante porque se consigue un compromiso entre el nivel de abstracción de la implementación de la verificación *Black-Box* y el nivel de dependencia propio de la verificación *White-Box*.

Con este enfoque es posible controlar y observar el diseño mediante el análisis de sus interfaces a alto nivel y, además, se pueden gestionar las señales internas del dispositivo que se producen durante el tiempo de ejecución.

1.3. LENGUAJE DE DESCRIPCION Y REFERENCIA HARDWARE SYSTEMVERILOG

SystemVerilog [1] es un estándar universal para la descripción y verificación hardware versátil estandarizado como IEEE 1800 que es utilizado en el diseño y verificación de sistemas electrónicos. Este lenguaje es una extensión de Verilog-2001, pero combina conceptos de otros lenguajes como VHDL, C y C++. En la

Figura 6 se muestran las diferentes funcionalidades que van asociadas a este lenguaje de descripción y verificación hardware.

La metodología UVM va de la mano del lenguaje SystemVerilog, que abarca, al mismo tiempo, tanto la funcionalidad de lenguaje de descripción hardware (*Hardware Description Language - HDL*) como la de lenguaje de verificación hardware (*Hardware Verification Language - HVL*).

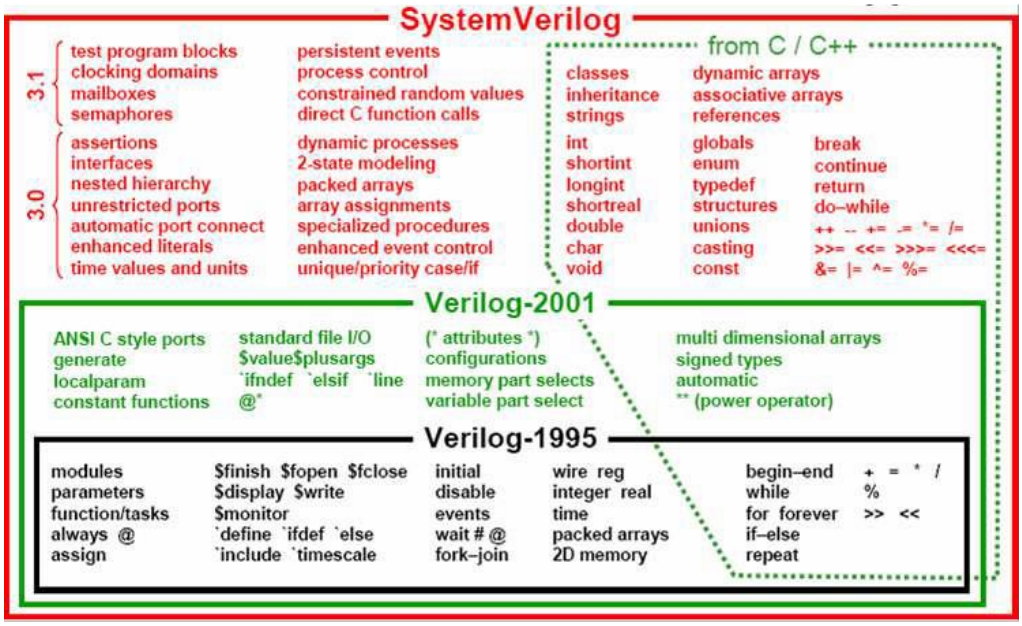


Figura 6 Funcionalidades de SystemVerilog [6]

Como ya se ha comentado, SystemVerilog ha evolucionado a partir del lenguaje HDL Verilog y ha añadido funciones de otros lenguajes. Por lo tanto, SystemVerilog concibe las características básicas para la implementación de sistemas electrónicos, pero el principal propósito de SystemVerilog está ligado a la verificación del correcto funcionamiento de estos sistemas electrónicos. Esto se debe a múltiples razones, de

las que se destacan las siguientes [7].

- Utiliza 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. Esta metodología hace uso de clases, las cuales son un conjunto de variables y rutinas que definen el comportamiento del objeto que se va a referenciar. En SystemVerilog, son elementos dinámicos que permiten el modelo de herencia simple y que pueden ser parametrizados (función adquirida 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 sólo tendrá que implementar algunos métodos adicionales.
- Se utilizan paquetes *(packages)* para compartir código entre distintos módulos, ya que incluye declaraciones y definiciones. Con estas declaraciones se pueden definir diferentes constantes, tipos, funciones, clases, tareas, etc.
- Permite el uso de *assertions* 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* con la finalidad de comprobar una determinada propiedad. Esta propiedad se comprueba durante la fase de simulación. Además, soporta la verificación ABV (*Assertion-Based Verification*) y define un mecanismo de cobertura y de adquisición de datos de manera flexible.
- El lenguaje cuenta con varios tipos de datos, tanto dinámicos como estáticos.
- Una característica clave que posee SystemVerilog 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 siguiendo algunas restricciones.
- Para la comunicación de distintos elementos permite el uso de interfaces. Otro factor importante para la metodología UVM es que SystemVerilog soporta las comunicaciones TLM, lo que también permite la reutilización de los entornos de verificación en distintos niveles de abstracción.
- El uso de la interfaz DPI (*Direct Programming Interface*) permite referenciar, de manera directa, funciones descritas en C en el código SystemVerilog.

SystemVerilog se ha consolidado en el mercado debido a todas las características y las ventajas antes analizadas. En la Figura 7 se muestra la evolución desde 2012 del uso de los diferentes lenguajes de verificación para la comprobación del comportamiento de

diseños ASIC (esta tendencia es también válida para los diseños con FPGA).

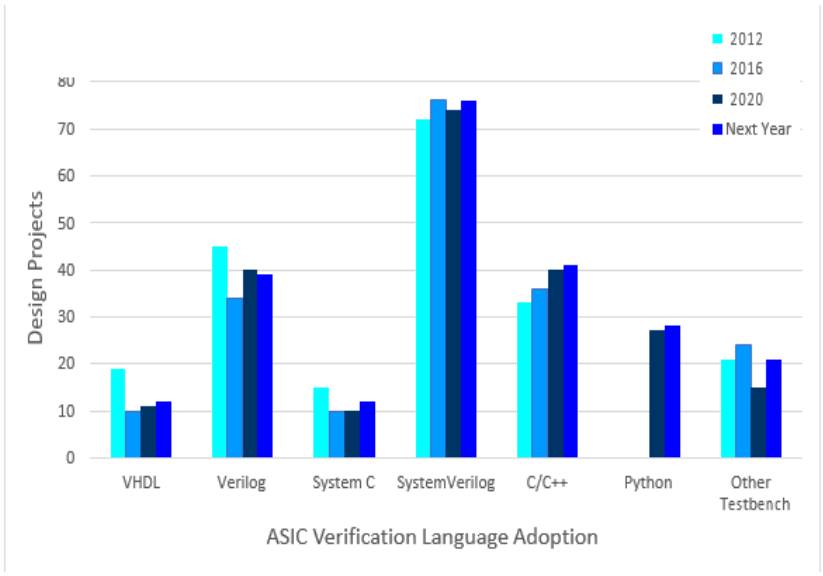


Figura 7 Evolución temporal de los lenguajes utilizados en la etapa de verificación [8]

1.4. BIBLIOTECA DE CLASES BÁSICAS

Con el propósito de [1] construir cualquier entorno de verificación basado en la metodología UVM son necesarios una serie de elementos, algunos de ellos 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. A continuación, se van a explicar las principales clases de la biblioteca, las cuales se pueden ver en la Figura 8.

- *uvm_object:* esta clase deriva de la clase base *uvm_void* y se comporta como la clase padre en la jerarquía de las clases de UVM, es decir, el resto de las clases heredan directa o indirectamente todos los métodos y propiedades de la clase *uvm_object*. La principal función de esta clase reside en proporcionar al resto de clases un conjunto de métodos para operaciones comunes como pueden ser: crear, copiar, comparar, imprimir, registrar o descomprimir.
- *uvm_component:* es la clase padre de todos los componentes de verificación que integran el entorno. Además de las características y métodos heredados de la clase incorpora la clase *Factory*, con la que se podrán crear nuevos componentes y otros objetos basados en una configuración específica del objeto o del componente a crear. También es la clase encargada de definir las fases que crean y conectan los componentes y les proporciona funcionalidad a los diferentes componentes que conforman el entorno de verificación.
- *uvm_sequence* y *uvm_sequence_item*: estas clases son las encargadas de la generación de estímulos y de la agrupación de las respuestas del DUV que se desea verificar.

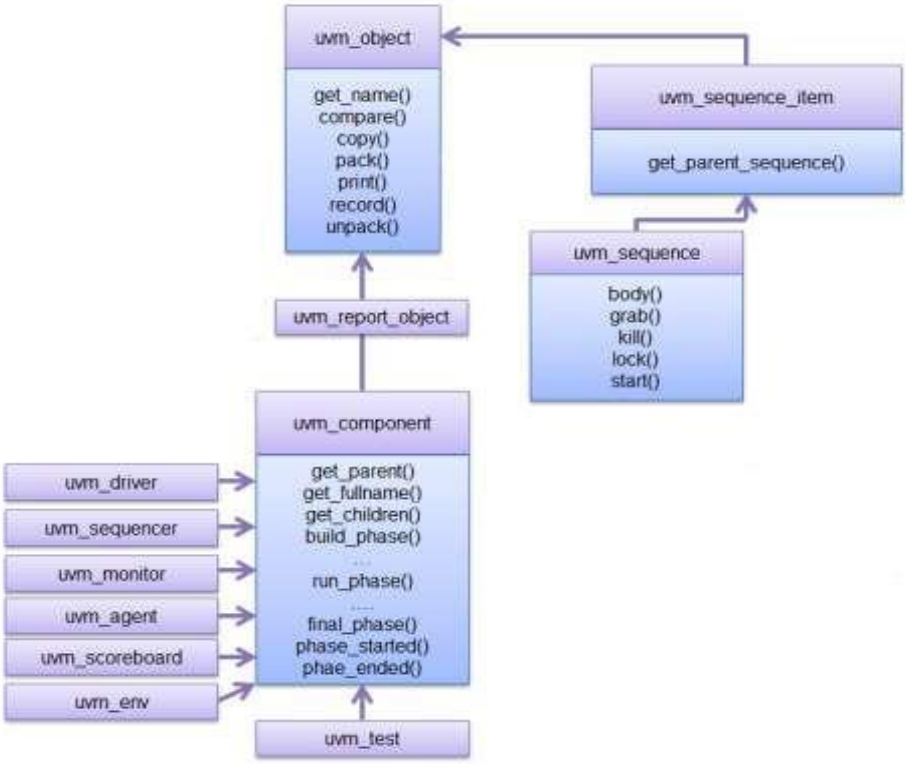


Figura 8 Jerarquía parcial de la biblioteca de clases de UVM [8]

Los objetos de las subclases que se derivan de la clase *uvm_component* son estáticos y forman parte de la jerarquía del *testbench* durante toda la simulación, ya que se crean durante las fases iniciales. En cambio, los objetos que se derivan de las clases *uvm_sequence* y *uvm_sequence_item* son objetos dinámicos que se crean y se destruyen en tiempo de ejecución.

1.5. MECANISMO DE FASES DE UVM

Se puede considerar que [1] una simulación que sigue las directrices de esta metodología se basa en la ejecución en secuencia de una serie de fases. Previo a la ejecución del *testbench*, debe crearse el entorno de verificación UVM en su totalidad. Además, se deben crear todos los elementos y componentes del sistema antes de realizar el conexionado entre dichos componentes. Para ello, UVM cuenta con un conjunto de fases que actúan como mecanismo de sincronización a la hora de ejecutar un *testbench*. El orden de ejecución de las fases [9] en UVM se detalla en la Figura 9, donde las fases más importantes son las siguientes:

- `build_phase()`. El propósito de esta fase es la creación de los componentes UVM. Esta es la primera fase en el flujo de ejecución, y es la

única fase que sigue una filosofía *top-down*, es decir, la jerarquía se construirá desde lo más alto a lo más bajo.

- `connect_phase()`. Esta fase se ejecuta inmediatamente después de la fase `build_phase()`. La función de esta fase es la de realizar el conexionado de los componentes creados anteriormente, formando la topología completa del entorno. De esta manera, gracias al conexionado, los diferentes componentes podrán comunicarse entre ellos. A diferencia de la fase anterior, la fase `connect_phase()` tiene una filosofía *bottom-up*, por lo que, en primer lugar, se conectarán los componentes de más bajo nivel hasta llegar al nivel más alto en la jerarquía.
- `end_of_elaboration_phase()/start_of_simulation_phase()`. Con el uso de estas fases se habilita al ingeniero de verificación una ventana de tiempo exacto en la que poder mostrar en pantalla información de relevancia, como puede ser la topología final de la jerarquía del entorno.
- `run_phase()`. En esta fase se describe el comportamiento de los diferentes componentes que constituyen el entorno de verificación. Dichos componentes implementan esta fase de manera paralela en el tiempo mediante el uso de una estructura *fork-join* [6]. Como se puede observar en la Figura 8, esta fase está compuesta por múltiples subfases. Durante el desarrollo de un test de complejidad baja o media, muchas de estas subfases carecen de utilidad práctica, por lo que el código se escribe directamente en la fase principal `run_phase()`. Es la única fase que se implementa como una tarea, lo que implica que consume tiempo de ejecución, el ingeniero de verificación deberá incluir todos aquellos eventos o hilos que quiera que se ejecuten durante el tiempo de simulación para realizar la verificación funcional del diseño. 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 instante en el que la fase `run_phase()` finaliza en un componente. Cuando se invoca este método en todos aquellos componentes que habían utilizado el método `raise_objection()` anteriormente explicado, la fase `run_phase()` se da por terminada y se prepara la ejecución de las fases de procesamiento.
    - `set_drain_time()`. Este método declara un intervalo de tiempo

a esperar una vez que finaliza la simulación.

- `extract_phase()/check_phase()/report_phase()`.Estas fases se pueden usar para extraer y recopilar los resultados de la simulación o información de cobertura. De esta manera, con la información recopilada se puede determinar un estado aprobado *(pass)* o suspenso *(fail)* del sistema y los resultados obtenidos se pueden recopilar para su posterior depuración.

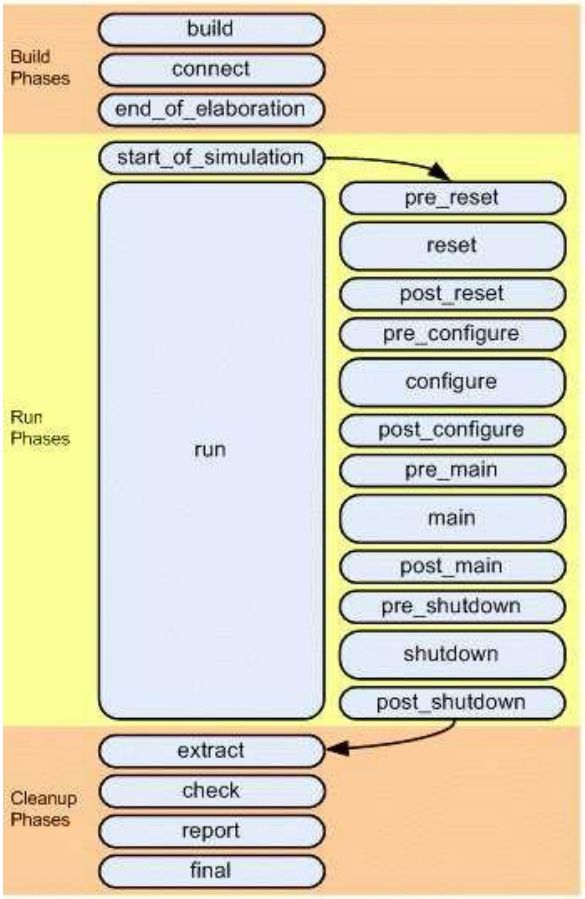


Figura 9 Fases UVM y orden de ejecución [10]

1.6. EL MECANISMO *FACTORY* EN UVM

El *Factory* es la forma de registrar los objetos en la metodología UVM. Las clases que definen el mecanismo *Factory* permiten registrar los objetos antes de utilizarlo y, una vez registrados, estos objetos se pueden crear para poder utilizarlos. Esto, además, permite la creación de subclases que deriven de las clases registradas en el *Factory*. Este procedimiento es similar al patrón de diseño '*Factory*' de la programación orientada a objetos. La metodología UVM utiliza un patrón parecido, registra los objetos y componentes por tipos. Todos los componentes y objetos de un entorno de verificación UVM deben estar registrados en la *Factory* y se crearán a partir de ahí, en función del tipo de objeto o componente.

Con este método, los objetos se construyen dinámicamente según el tipo de especificación del objeto, esto es importante porque se puede modificar el comportamiento del código preconstruido sin modificar el código.

Dentro de la metodología UVM las clases derivadas desde `uvm_object` y `uvm_component` son compatibles con el *Factory* y han de cumplir tres pasos básicos, registro, construcción y anulación.

Para hacer el registro en el *Factory*, UVM tiene macros predefinidas.

- `uvm_component_utils ( class_type_name )`
- `uvm_component_param_utils ( class_type_name #( params ))`
- `uvm_object_utils ( class_type_name )`
- `uvm_object_param_utils ( class_type_name #(params ))`

Los macros `uvm_*_param_utils` se utilizan para registrar clases parametrizadas y las otras dos macros para registrar clases no parametrizadas.

1.7. MODELADO A NIVEL DE TRANSACCIÓN TLM

La metodología UVM proporciona [1] una serie de interfaces y canales de comunicación TLM (*Transaction-Level Modeling*) [11] los cuales permiten la conexión a nivel de transacciones de diferentes componentes presentes en el entorno de verificación. Esto supone un ahorro de tiempo para el ingeniero de verificación debido a que, por ejemplo, no necesita gestionar e implementar la comunicación entre los componentes que forman el *testbench*, ya que dicha comunicación se realiza a través de los puertos TLM.

Esta comunicación a nivel de transacciones es necesaria en diseños complejos que requieran un entorno de verificación 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 diferentes componentes del entorno que se encuentran a distintos niveles de abstracción y que implementen una misma interfaz.

Con el fin de enviar y recibir transacciones de datos, la metodología UVM dispone de los puertos de comunicación *TLM ports* y *TLM exports*. Por un lado, los puertos *TLM ports* son los encargados de especificar todos los métodos que se pueden utilizar en la comunicación y de iniciar las peticiones de transacción. Por otro lado, los puertos *TLM exports* implementan los métodos definidos por puertos *TLM ports*. Es común que durante la fase `connect_phase()` estos puertos se conecten a un único puerto mientras se realiza la construcción del entorno de verificación. Para ello, durante la citada fase, los puertos *TLM ports* invocan al método `connect()`. Lógicamente, esta conexión se realizaría después de la creación de los componentes UVM y antes de que se realice la simulación.

1.8. SISTEMA DE MENSAJES E INFORMES

Cuando se ejecuta la simulación de un entorno de verificación, se reportan una serie de mensajes informativos, los cuales son de gran utilidad durante la fase de depuración del sistema y que permite saber en cada momento cada una de las tareas realizadas durante la verificación. En UVM existen un conjunto de macros que permiten la notificación de estos mensajes. Estas macros son las siguientes:

- `uvm_info (ID, MSG, VERBOSITY)`
- `uvm_warning (ID, MSG)`
- `uvm_error (ID, MSG)`
- `uvm_fatal (ID, MSG)`

El campo `ID` es una etiqueta para identificar el mensaje en el registro. El segundo argumento que se debe pasar en los cuatro casos es una variable tipo *string* con el mensaje que se desea imprimir en pantalla. El argumento `VERBOSITY` hace referencia al nivel de filtro de información que se desea imprimir. Según este nivel, se mostrará más o menos información.

1.9. ENTORNO DE VERIFICACIÓN UVM

Un *testbench* UVM [9] utiliza la verificación funcional como método de verificación, pero ofrece la posibilidad de tener un mayor control sobre los siguientes procesos:

- La generación de los estímulos de entrada.
- La recepción de las salidas.
- La comparación de los datos de salida con los esperados.

Un entorno de verificación UVM se crea a partir de la referencia de componentes de todas y cada una de las clases que heredan de `uvm_component`. La jerarquía se determina a partir de la relación entre estos componentes, de forma que unos son referenciados dentro de otros. Esta jerarquía puede observarse en la Figura 10, como un modelo básico de un entorno UVM.

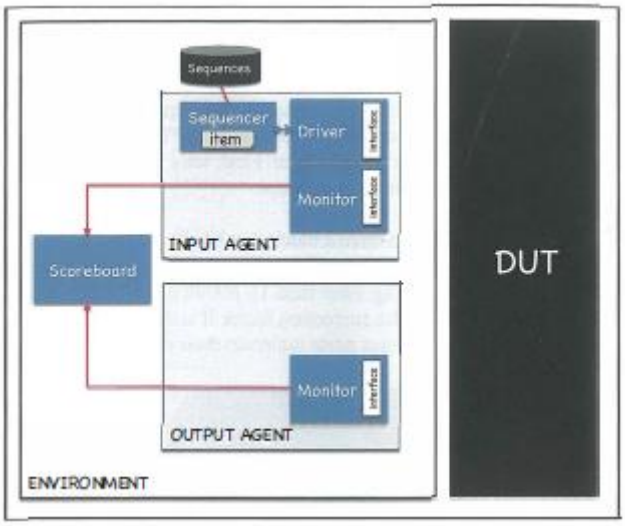


Figura 10 Jerarquía de un testbench UVM [9]

Como se puede observar existen distintos componentes dentro de este diagrama, siendo el más fundamental el componente *Agent* (agente). Un componente *Agent* es un contenedor que contiene todos los componentes necesarios para un protocolo en particular. Un agente UVM típico puede contener un componente *Driver, Monitor y Sequencer.*

El componente *Sequencer* controla el flujo de secuencias que se crean con datos aleatorios o específicos con restricciones. Se puede pensar en una secuencia como un paquete de datos que representa una transacción o protocolo. Esencialmente, un

componente *Sequencer* pasa una transacción al componente *Driver*. A continuación, el componente *Driver* la convierte y la comunica a la interfaz en función del protocolo de bus que se esté utilizando con esa transacción a nivel de señales. El componente *Driver* también puede enviar una respuesta al componente *Sequencer*, si es necesario. El componente *Monitor* captura las transacciones en función del protocolo y puede implementar comprobadores para verificar el protocolo. Todas estas piezas, el componente *Driver*, el componente *Monitor*, etc. son los componentes básicos de un UVM *testbench* y son piezas de código reutilizables que juntas crean el protocolo.

Como se observa en la Figura 10 el componente *Agent* de salida solo contiene un componente *Monitor*. El componente *Agent* de entrada se denomina "activo" porque está conduciendo activamente transacciones en la interfaz. Sin embargo, el componente *Agent* de salida es "pasivo", es solo para capturar las transacciones de salida y pasar la información al componente *Scoreboard* para su comprobación.

Todos los componentes Agent creados, y puede haber más de uno como veremos en este TFG, se configuran dentro de un paquete de superior jerarquía llamado componente *Environment*.

Siguiendo con el modelo de jerarquías de UVM, se tienen los niveles superiores que constituyen el modelo de referencia básico de lo que sería un entorno de verificación de un IP mediante la metodología UVM, Figura 11. En esta figura se muestra el componente *Test* y el módulo *Top*. En el módulo principal *(tb_top*) se referencia el sistema que se pretende verificar y, además, se ejecuta una tarea que permite la creación del entorno de verificación UVM al completo.

El nivel más alto de la jerarquía es el componente *UVM Test* que contiene todos los componentes *UVM Environment*, pudiendo ser más de uno, que contiene el *testbench.*

La Figura 11 muestra la arquitectura básica de un entorno de verificación. A continuación se explica con mayor detalle las características propias de cada uno de los componentes que conforman el entorno de verificación diseñado con la metodología UVM.

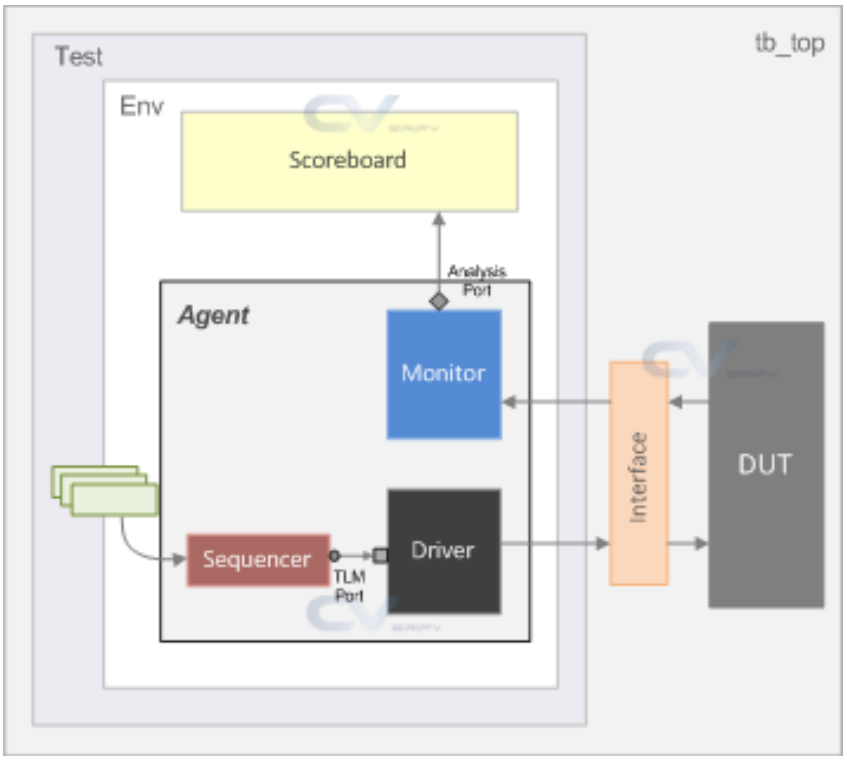


Figura 11 Arquitectura básica de un entorno de verificación [12]

1.9.1. COMPONENTE *UVM AGENT*

Un componente *UVM Agent* [9] es un componente que se deriva a partir de la clase `uvm_agent` y encapsula un componente *UVM Sequencer*, un componente *UVM Driver* y un componente *UVM Monitor* en una sola entidad, dentro de la cual se referencia a cada uno de ellos, creando una capa de abstracción extra en el sistema. Los UVC que contiene un componente *UVM Agent* se conectan a través de interfaces TLM con el propósito de crear una comunicación entre ellos.

A grandes rasgos, el componente *UVM Sequencer* se encarga de controlar el flujo de estímulos que le llegan en forma de secuencia. El componente *UVM Driver* conduce dichos estímulos hacia la interfaz física del DUV, transformando dichos estímulos en señales. Por último, el componente *UVM Monitor* tiene la función de muestrear las señales que capte en la interfaz del dispositivo que se desea verificar.

1.9.2.COMPONENTE *UVM SEQUENCER*

Un componente *UVM Sequencer* [9] es un componente que se deriva de la clase `uvm_Sequencer`. Esto se debe a que la clase base `uvm_Sequencer` posee toda la funcionalidad requerida para permitir la comunicación con el componente *UVM Driver*.

El componente *UVM Sequencer* genera transacciones de datos como objetos de clase y las envía al componente *UVM Driver* para su ejecución. Esta comunicación existente entre los componentes *UVM Sequencer* y *UVM Driver* se realiza a través de un protocolo de *handshake.* Este protocolo es sumamente importante ya que libera de la gestión temporal de eventos entre ambos componentes.

La generación de objetos dentro del componente *UVM Sequencer* es muy limitado en términos de flexibilidad y reutilización de código. La manera de superar esta limitación es mediante la utilización de secuencias.

1.9.3. COMPONENTE *UVM DRIVER*

Un componente *UVM Driver* [9] es un componente cuyo cometido es conducir las transacciones enviadas desde el componente *UVM Sequencer* hasta la interfaz que conecta con el DUV. Por lo tanto, el componente *UVM Driver* solicita las transacciones de datos al componente *UVM Sequencer* a través de un puerto TLM, para luego transformarlas en señales a nivel RTL y aplicarlas sobre dicha interfaz. Todos los componentes *UVM Driver* deben derivar de la clase `uvm_Driver`, propia de UVM.

De los componentes explicados hasta este momento, el componente *UVM Driver* es el único que tiene comunicación directa con el DUV a través de una interfaz virtual. No se trata de una interfaz física (estática) porque no puede ser referenciada dentro de una clase que opera en un dominio dinámico como es la clase `uvm_Driver`. Esta interfaz se implementa mediante un objeto tipo `interface` en SystemVerilog.

1.9.4. COMPONENTE *UVM MONITOR*

Un componente *UVM Monitor* [9] es un componente que deriva de la clase `uvm_monitor` y es responsable de la captura de la actividad de las señales de la interfaz que conecta con el DUV. De esta manera, los datos recopilados se exportan a través de un puerto TLM, más en concreto de su puerto tipo *Analysis Port* para ponerlos a disposición del resto del entorno de verificación.

Aunque en ocasiones puede resultar útil, por lo general el componente *UVM Monitor* no realiza ningún procesamiento interno sobre las transacciones generadas. La metodología recomienda delegar en otros componentes del entorno diversas tareas como, por ejemplo, tareas de comprobación y cobertura que se utilizan para garantizar que se cumple el plan de verificación establecido, acciones de mensajería en consola, etc.

1.9.5. COMPONENTE UVM *SCOREBOARD*

Un componente *UVM Scoreboard* [9] es un componente de verificación que se deriva de la clase `uvm_scoreboard` y se encarga de la validación del comportamiento del DUV mediante la comparación de los objetos de transacción captados en la interfaz de un dispositivo bajo prueba con los valores esperados.

Por ejemplo, durante la ejecución del *testbench* se realiza una operación de escritura en un registro, el componente *UVM Scoreboard* recibe el valor esperado y, después de leer el correspondiente registro, se le envía también el valor guardado en dicho registro. A continuación, compara ambos datos y, en caso de no coincidir, el componente *UVM Scoreboard* notifica un error en la operación de escritura.

Para que el componente *UVM Scoreboard* obtenga los valores esperados se puede realizar de dos maneras: mediante la utilización de un modelo de referencia conectado a la entrada del DUV o mediante la lectura de un fichero en el que se encuentren almacenados estos valores de referencia.

Para obtener los valores reales, se deberán conectar al componente *UVM Scoreboard* los puertos TLM *analysis ports* de los componentes *UVM Monitor*. De esta manera, el componente *UVM Scoreboard* recibirá todas las transacciones que lleguen al dispositivo que se desea verificar.

1.9.6. COMPONENTE *UVM ENVIRONMENT*

Un componente *UVM Environment* [9] es un contenedor que se deriva de la clase `uvm_env`. El componente *UVM Environment* posee propiedades de configuración que permiten personalizar la topología del entorno de verificación y el comportamiento de los componentes que integra, pudiendo enfocarse a diversas tareas de verificación. El componente *UVM Environment* se encuentra en el segundo nivel en la jerarquía del entorno UVM y es el encargado de albergar otros componentes como los explicados anteriormente, e incluso otros componentes *UVM Environment*, de inferior jerarquía. Teóricamente es posible crear referencias a todos estos componentes sin necesidad de crear el componente *UVM Environment* y hacerlo directamente en el componente *UVM Test*, pero esto no se recomienda en ningún caso. Esto se debe a tres razones principalmente:

- El *testbench* dejaría de ser reutilizable porque se basan en una estructura de entorno específica para cada caso.

- Los cambios en la topología del sistema obligarían a la actualización de varios módulos del *testbench*.

- El ingeniero de verificación necesitaría saber cómo configurar el entorno.

Por lo tanto, esto se traduce en que el componente *UVM Environment* es absolutamente necesario para garantizar la reusabilidad del código y para disminuir el tiempo de verificación de un DUV.

1.9.7. COMPONENTE UVM TEST

El componente *UVM Test* [9] engloba todo el entorno de verificación UVM y a todos los componentes que lo integran, siendo el nivel más alto de la jerarquía de dicho entorno. Por lo tanto, cuando comienza el proceso de creación del entorno, y una vez se ejecuta la fase `build_phase()`, lo primero que se creará es el componente *UVM Test*, para luego descender por toda la jerarquía.

A grandes rasgos, una librería de test es una colección de pruebas que se utiliza para estimular un DUV. Cuando un ingeniero de verificación construye una librería de test con la metodología UVM es recomendable empezar con una clase base que, a su vez, se deriva de la clase *`uvm_test`*, que incorpore una referencia al componente *UVM Environment* principal y otros elementos comunes. A partir de esta clase base se derivan otros tests individuales, los cuales pueden tener características diferentes como pueden ser la configuración del entorno de verificación o la selección de distintas secuencias a ejecutar en los componentes *UVM Sequencer*.

1.9.8. COMPONENTE TOP (UVM *TESTBENCH*)

Todos los componentes de verificación, interfaces y el propio DUV son referenciados en el módulo Top [9] (también conocido como *UVM Testbench*), que es del tipo *module* en SystemVerilog. Por lo tanto, este componente es un contenedor estático que contiene tras la compilación las interfaces y el módulo DUV. Una vez inicializada la simulación, este módulo activa la generación de todos los componentes que conforman el entorno de verificación UVM.

La principal función de este componente es referenciar y configurar la conexión entre el entorno de verificación UVM y el DUV. Para ello, en primer lugar, se crearán y se referenciarán una o varias interfaces virtuales que permitirán la comunicación entre el DUV y el entorno UVM. Estas interfaces definirán todas las señales del DUV a las que se quiera tener acceso durante la verificación de este. En segundo lugar, se conectarán

físicamente las señales de los puertos del DUV con sus homólogos situados en las interfaces virtuales anteriormente creadas. Otra función de este componente es la generación de las señales de reloj y reset.

Este componente debe incluir las interfaces virtuales creadas en la base de datos. De esta manera, los diferentes componentes que forman parte del entorno, como el componente *UVM Driver* y el componente *UVM Monitor*, podrán acceder a las interfaces para interactuar con el DUV, con el fin de estimular y monitorizar dicho dispositivo.

Es importante INDICAR QUE en este módulo se realiza la llamada del método `run_test()`, el cual inicia la ejecución de las distintas fases de verificación, lo que provoca el comienzo de la simulación UVM.

CAPÍTULO 2: MÓDULO QVIP DE MENTOR GRAPHICS CON INTERFAZ AXI4

Este capítulo está dividido en dos apartados importantes, por un lado, el protocolo de comunicación AXI4 de nuestro modulo DUV, y en el otro, se describe el módulo VIP de Mentor Graphics utilizado por la herramienta QVIP *configurator*, que facilita la labor de verificación mediante la metodología UVM.

Para la realización de este TFG se ha partido del trabajo iniciado por Álvaro Moreno Florido [1], en su TFG, siendo su final el punto de partida de este trabajo. En el TFG de partida, con el uso de la herramienta de Mentor Graphics `QVIP configurator` se realizó de forma dinámica el entorno de verificación de UVM. El principal objetivo de aquel Trabajo Fin de Grado fue el desarrollo completo de un *testbench* UVM para verificar el comportamiento de un DUV proporcionado por un tercero. Inicialmente, este objetivo de una gran complejidad no podría ser realizado sin el uso de esta herramienta por dos motivos: la dificultad inherente a la metodología UVM y el limitado tiempo asociado a la realización del TFG. El factor clave para acometer este objetivo fue la utilización de una herramienta que permitiera reducir el tiempo empleado en la generación de los componentes UVM del *testbench*. La herramienta que se utilizo fue `QVIP configurator` de la empresa Mentor Graphics, la cual creó y configuró de manera dinámica el entorno de verificación UVM. En este capítulo se va a describir a grandes rasgos la herramienta `QVIP configurator` utilizada.

El protocolo comunicación en el que trabaja el DUV seleccionado, un *crossbar* configurable, es el interfaz AXI-4. Este es un protocolo de comunicación en ráfagas compatible con interfaz AMBA, muy útil a la hora de realizar transacciones de datos en las que estos se transmitan en paralelo. Este tipo de protocolo requiere una interfaz específica para poder ser llevado a cabo, además de seguir unos pasos muy concretos a la hora de llevar a cabo la comunicación.

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, cuya principal funcionalidad es la interconexión de bloques funcionales en un *System on a Chip (SoC*).

Esta es una interfaz paralela de alto rendimiento [13], 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, además 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*).

2.1. INTERFAX AXI4

2.1.1. CANALES DE COMUNICACIÓN

La comunicación mediante el protocolo AXI se realiza a través de cinco canales independientes. Cada uno de estos canales dispone, además, de tres señales especiales, `VALID`, `READY` y `LAST`, una de cada por canal, que permiten gestionar las comunicaciones, mediante un protocolo de *handshake*.

En el protocolo AXI, uno de los extremos actúa como maestro (el que va a enviar los datos) y el otro como esclavo (el que los va a recibir).

- **Canal de escritura *(Write data*).** El canal de escritura se encarga de enviar los datos desde el maestro hasta el esclavo.
- **Canal de lectura (*Read data*).** Este canal envía información desde el esclavo al maestro, siempre que este solicite una lectura de datos.
- **Canal de dirección de escritura (*Write address*)**. Este canal envía la información sobre la dirección en la que se van a escribir los datos durante una transacción de escritura de datos.
- **Canal de dirección de lectura (*Read address*).** Este canal envía la información sobre la dirección de la que se van a leer los datos durante una transacción de lectura de datos.
- **Canal de respuesta a escritura (*Write Response*).** Este canal envía el reconocimiento *(acknowledgement*) del esclavo al maestro una vez se ha completado la escritura. Su uso no es obligatorio.

La arquitectura de comunicación de estos canales se muestra en las Figura 12 y Figura 13, en ellas podemos ver los canales que intervienen en cada comunicación.

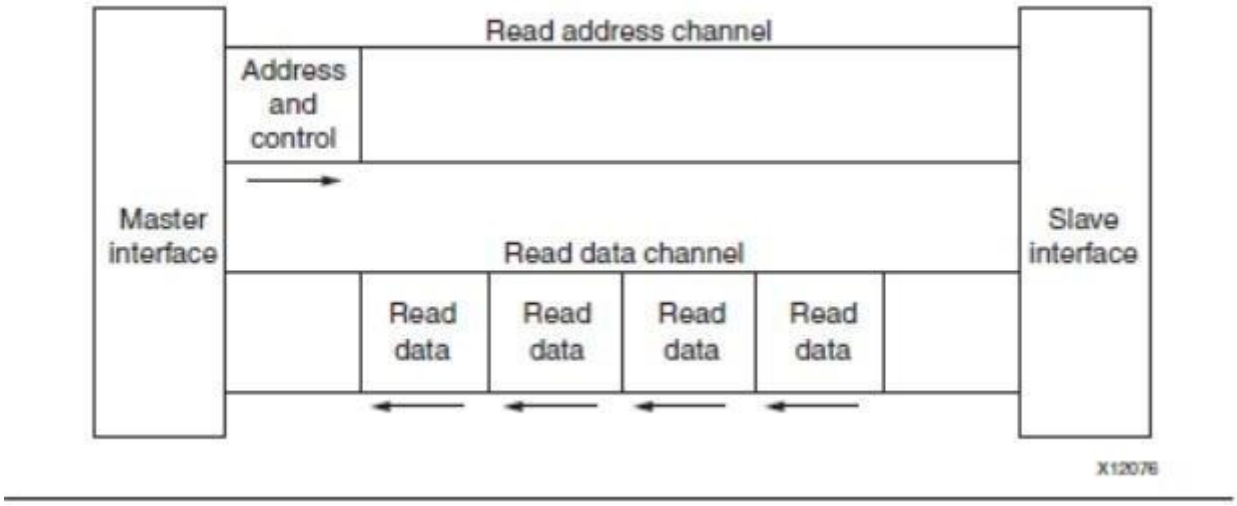


Figura 12 Arquitectura del canal de lectura de AXI [13]

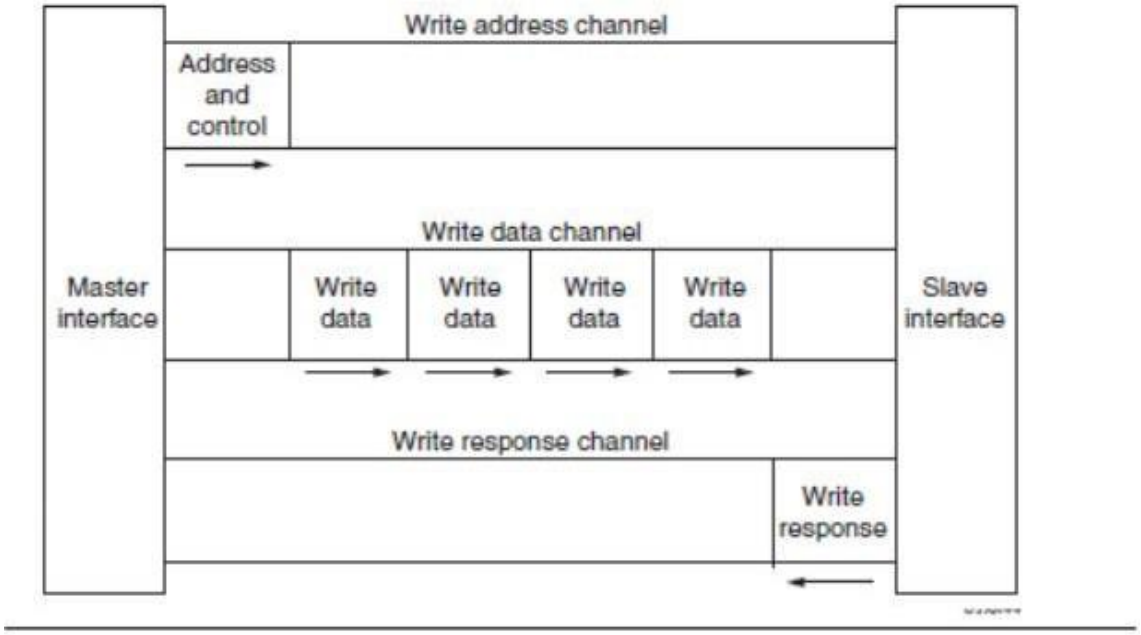


Figura 13 Arquitectura del canal de escritura de AXI [13]

2.1.2. PROTOCOLO DE HANDSHAKE EN AXI4

El protocolo de comunicación por *handshake* en AXI4, como ya se mencionó anteriormente, hace uso de tres señales especiales, denominadas `VALID, READY y LAST`. Cada canal dispone de sus propias señales de este tipo. Por ejemplo, las señales de control del canal de escritura serán `WVALID, WREADY y WLAST`.

El procedimiento a seguir en los canales es el siguiente, el extremo que va a enviar los datos depositará los mismos en el bus adecuado y, una vez hecho esto, pondrá la señal VALID a 1, indicando así al otro extremo que los datos están listos.El extremo que recibe los datos activa la señal `READY` para indicar que está listo para aceptar la transacción.

El envío de datos a través de cualquiera de los canales se produce cuando ambas señales están a nivel alto en un mismo ciclo de reloj. Según la secuencia en la que se activan las señales `VALID y READY`, se pueden dar tres tipos de comunicación. Estas tres formas son mostradas en la Figura 14, Figura 15, y Figura 16.

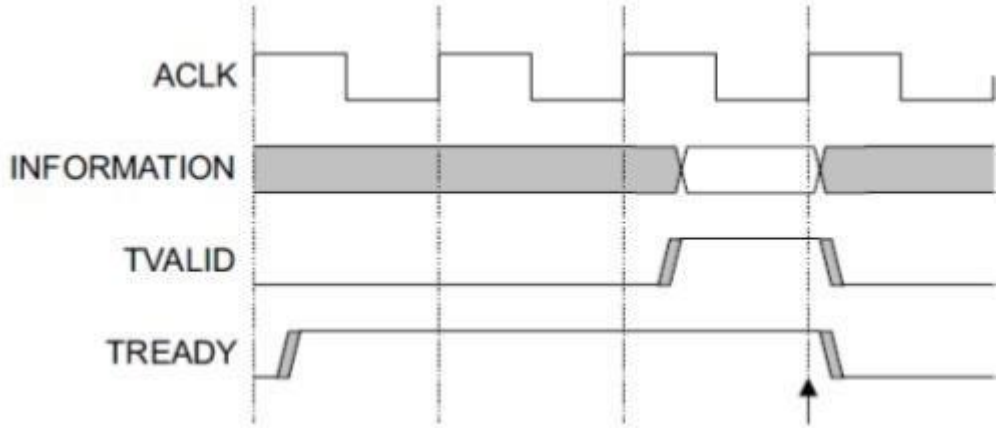


Figura 14 Handshake AXI con establecimiento de READY previo al de VALID.[14]

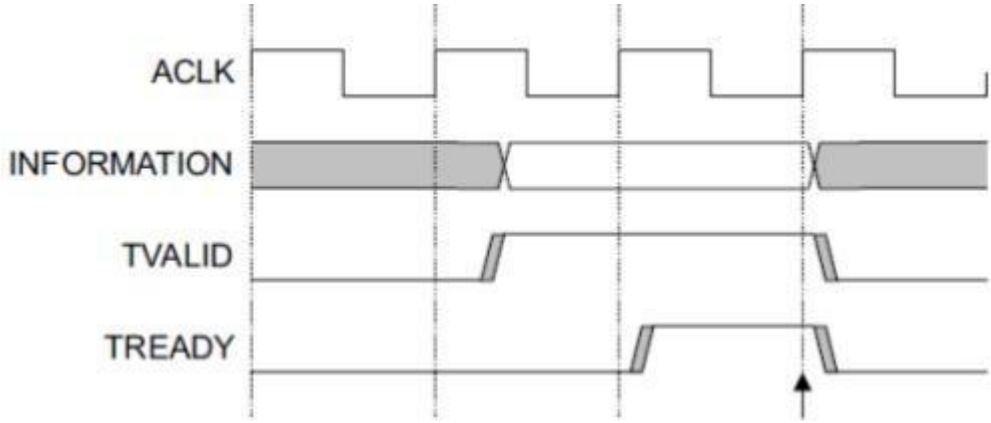


Figura 15 Handshake AXI con establecimiento de VALID previo al de READY.[14]0

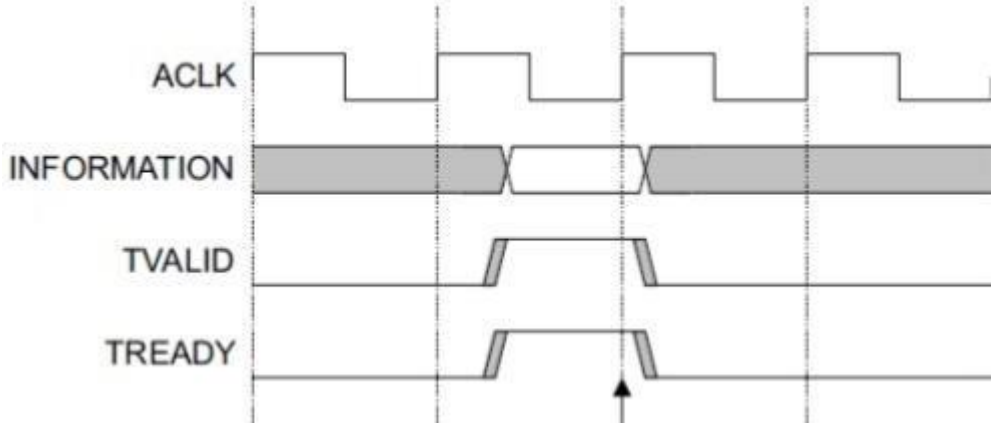


Figura 16 Handshake AXI con establecimiento de READY y VALID en el mismo instante.[14]

Como última señal en este protocolo está el uso de la señal LAST. Esta señal es necesaria en el protocolo por su particularidad en las transferencias de ráfagas. Esta señal, se usa para indicar el final de una comunicación. Permanecerá a cero en todo momento durante la transacción, activándose únicamente cuando el último bloque de datos esté depositado en el bus. La señal se activará a la vez que se activa el último VALID, de modo que el receptor sabrá, al leer ese paquete de datos, que se termina la ráfaga.

Para una mayor profundidad en el intercambio de información que utilizan estos canales y sus particularidades en las secunecias se puede hacer uso de la bibibliografía citada en este documento [13] .

2.1.3. CONFIGURACIÓN DE LAS TRANSACCIONES

Desde el punto de vista de la verificación, uno de los aspectos más importantes del protocolo AXI-4, es la caracterización de las transacciones mediante los parámetros SIZE, LEN, BURST, CACHE, PROT, REGION, USER y ID. Estos parámetros se establecen durante la fase de direcciones (*address*), 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 y todas las señales que intervienen en la comunicación han sido definidos de forma detallada en las tablas que se muestran a continuación por canales.

Canal *Write Data*. Canal de escritura de datos para realizar una transferencia de datos de un maestro a un esclavo. La Tabla 1 muestra las señales que definen dicho canal.

Tabla 1 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).*
WUSER	Maestro	Bus de datos de usuario. Es opcional y su contenido viene definido por el usuario.
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.

Canal *Write Address*. Canal de dirección de escritura. El desglose de las señales que están relacionadas con este canal se encuentra recogidas en la Tabla 2, donde se incluyen las señales necesarias que indican la dirección de escritura de datos y las señales de control relacionadas.

Tabla 2 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 *burst.*
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.
AWCACHE	Maestro	Este bus indica los atributos de la transacción en términos de gestión de memoria y de los niveles de caché definidos en el sistema.
AWPROT	Maestro	Este bus indica el nivel de privilegio y seguridad de la transacción, y si ésta es un acceso a datos o instrucciones.
AWREGION	Maestro	Identificador de región que permite que una única interfaz física en un esclavo se utilice para múltiples interfaces lógicas.
AWUSER	Maestro	Bus opcional de datos y su uso queda definido por el usuario.
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.

Canal *Read Data*. Canal de lectura de datos para transferir datos de un esclavo a un maestro. Este canal incluye los datos a transferir y una señal que indica el estado de la transferencia, todas estas señales están descritas en la Tabla 3.

Tabla 3 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 *burst.*
RUSER	Esclavo	Este bus es opcional y su uso queda definido por el usuario.
RVALID	Esclavo	Esta señal valida los datos de lectura presentes en l RDATA, así como la información asociada a la misma.
RREADY	Maestro	Esta señal indica que el maestro puede aceptar nuevos datos de lectura.

Canal *Read Address*. Canal de dirección de lectura. Este canal incluye las señales necesarias para indicar la dirección de lectura de datos y la información de control asociada a dicha dirección como se ve en la Tabla 4.

Tabla 4 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.*
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.
ARCACHE	Maestro	Tipo de memoria. Este bus indica cómo se requieren las transacciones para progresar a través de un sistema.
ARPROT	Maestro	Tipo de protección. Esta señal indica el nivel de privilegio y seguridad de la transacción, y si ésta es un acceso a datos o instrucciones.
ARREGION	Maestro	Identificador de región que permite que una única interfaz física en un esclavo se utilice para múltiples interfaces lógicas.
ARUSER	Maestro	Señal de usuario. Este es un bus de datos opcional definida por el usuario.
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.

Canal *Write Response*. Canal de respuesta a las escrituras que indica el estado final de la escritura realizada, es decir, si la transacción se ha realizado correctamente o ha habido algún error. La Tabla 5 enumera las señales que contiene este canal.

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.
BUSER	Esclavo	Señal de usuario. Este bus de datos es opcional y está definido por el usuario.
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.

2.2. MÓDULO QUESTA VERIFICATION IP

Durante el primer apartado de este capítulo se profundizará en los conceptos básicos que definen un IP de verificación de Mentor Graphics, denominado QVIP, y se conocerá 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 las ventajas y desventajas asociadas a su uso en la verificación de sistemas hardware digitales y las etapas que se deben seguir durante la integración de un QVIP en un entorno de verificación. Por último, se hará especial énfasis en la herramienta de configuración. Esta herramienta creará el entorno de verificación UVM de una manera automática cuando se haya completado el siguiente flujo de diseño: importar el DUV, añadir los módulos QVIP, configurar el conexionado y, por último, añadir el mapa de direcciones [15].

2.2.1. PRINCIPIOS BÁSICOS DE LA HERRAMIENTA QVIP *CONFIGURATOR*

La herramienta [1] QVIP *configurator* únicamente funciona con diseños elaborados en SystemVerilog o VHDL, y se integra con cierta facilidad en entornos de verificación que siguen la metodología UVM. La familia de módulos QVIP para el protocolo AMBA proporciona una serie de API, denominados modelos funcionales de bus (BFM) con funcionalidad completa para todos los modelos de uso de este protocolo. Estos modelos vienen con soporte para todo tipo de estímulos sobre la interfaz, así como una biblioteca de secuencias que se pueden utilizar para la estimulación del DUV, asegurando una amplia cobertura de escenarios de verificación. Los módulos QVIP permiten verificar las interfaces de un bus con rapidez, por lo que se reduce el esfuerzo de los ingenieros durante la etapa de verificación del DUV, proceso clave durante el desarrollo de cualquier sistema electrónico. Este hecho supone un ahorro de tiempo en el proceso de verificación, tiempo que se puede emplear en tareas de mayor importancia para el proyecto, manteniendo la calidad y fiabilidad del diseño y reduciendo el tiempo de producción.

Los módulos Questa Verification IP se encuentran disponibles para una amplia gama de protocolos de comunicación, como se visualiza en la Figura 17, entre los que destacan, por ejemplo, protocolos como AXI, AHB, PCIe, Ethernet, USB y bus seriales como SPI, UART o I2C. La disponibilidad que presenta el QVIP, en cuanto a protocolos de comunicación, es un factor clave para la implementación y normalización de esta herramienta, ya que se adapta con mayor facilidad a las necesidades existentes en

el mercado.

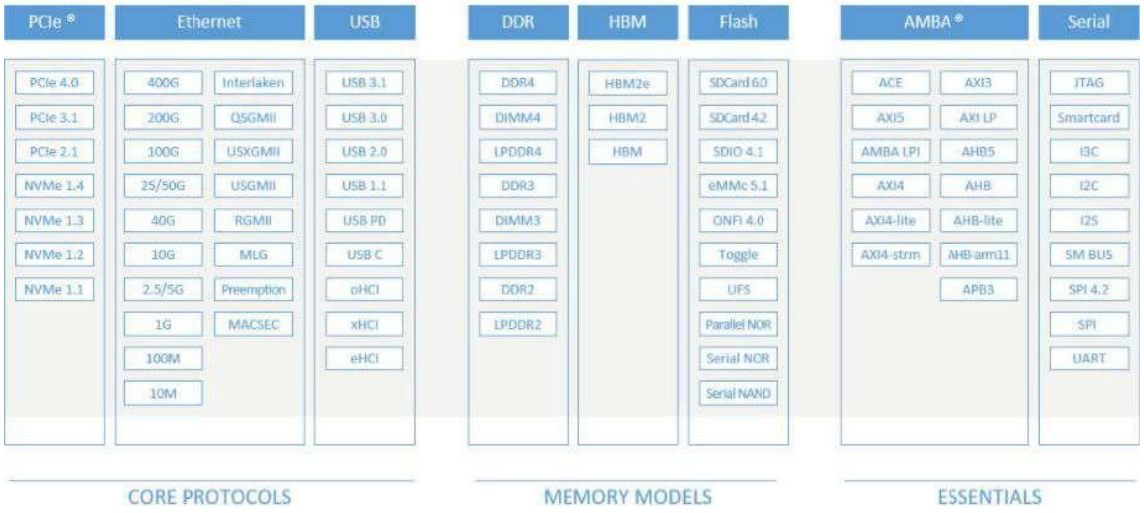


Figura 17 Protocolos de comunicación compatibles con el Questa Verification IP[15]

Los conceptos básicos y que explican la creciente adopción de la herramienta *configurator* QVIP, se aclaran en los siguientes puntos:

- Es compatible con diseños elaborados en SystemVerilog o VHDL, lenguajes cuyo uso está muy extendido en el diseño y verificación de sistemas hardware digitales.
- Cuenta con el soporte de los principales simuladores existentes en el mercado.
- La herramienta de configuración se puede arrancar en dos de los sistemas operativos más utilizados en la actualidad: Linux y Windows. Para el uso legal de la herramienta de configuración se debe tener una licencia QVIP.
- QVIP *configurator* acelera el montaje del entorno de verificación porque genera el *testbench* casi de manear automática.
- Como ya se ha comentado, el ingeniero de verificación puede elegir diferentes protocolos de comunicación para incluir los correspondientes módulos QVIP, con el fin de que se pueda comunicar de manera exitosa con el DUV en cuestión.
- A lo largo del diseño de un sistema digital, en múltiples ocasiones surgen inconvenientes que pueden derivar en un cambio de la topología del sistema o del protocolo de comunicación utilizado. Si la empresa decide hacer un cambio en el protocolo de comunicación propio del sistema diseñado, este hecho puede afectar gravemente a los plazos de entrega y al tiempo de puesta en el mercado, lo que afectaría al volumen de ventas. En este sentido, los módulos QVIP cuentan con una gran ventaja, ya que, si el proceso de verificación del DUV

ha sido realizado antes del cambio del protocolo de comunicación, no afectará al *testbench* UVM diseñado. La herramienta `QVIP configurator`, junto a la metodología UVM, permiten la reusabilidad del código, en especial de las secuencias diseñadas y ejecutadas en el componente *UVM Test*. Por lo tanto, sólo haría falta cambiar el componente *UVM Environment* y los componentes *UVM Agent* para modificar el protocolo de comunicación utilizado por estos módulos. Esto se traduce en que los efectos negativos de este tipo de contratiempos se pueden paliar fácilmente usando esta herramienta.

- QVIP proporciona una depuración intuitiva mediante la visualización de transacciones y archivos de seguimiento en varios niveles de abstracción.

2.2.2. DIAGRAMA DE FLUJO PARA EL USO DE QVIP *CONFIGURATOR*

Mentor Graphics [1] ha diseñado los módulos IP de verificación con el fin de facilitar las labores a la hora de comprobar el correcto funcionamiento de un sistema. Para ello, los módulos QVIP se deben adaptar casi a la perfección al entorno de verificación sin perjudicar el flujo de trabajo del equipo de verificación [16]. Por estas razones, el ingeniero de verificación debe seguir cuatro pasos para realizar la integración de los módulos QVIP dentro del *testbench* UVM. Se aconseja que los tres primeros pasos se implementen en la herramienta de configuración que, además, generará el código correspondiente para la realización del último paso. En el caso particular de este Trabajo Fin de Grado, los procesos han sido realizados por Álvaro Moreno Florido [1] y el tercer paso no se realizará en la herramienta de configuración, pues la generación de estímulos controlados se elaborará manualmente que es el objetivo de este TFG. Para dar un mayor detalle del flujo, a continuación, se especificarán cada uno de los pasos a seguir durante las diferentes etapas:

- En primer lugar, el ingeniero de verificación debe importar el diseño DUV en el entorno QVIP *configurator*. Además, se deben integrar tantos módulos QVIP como interfaces tenga el DUV, con el fin de estimular las entradas del DUV y de *monitor*izar las salidas de este. Por lo tanto, es importante la configuración inicial de los módulos QVIP. En primera instancia, se debe elegir si se quiere añadir un módulo maestro o esclavo. Además, debe especificarse el protocolo de comunicación con el que trabajará. Con esta información, la herramienta incluirá el modelo funcional del bus para el protocolo de comunicación seleccionado y las reglas de verificación formal en SystemVerilog para dicho protocolo. Por defecto, la herramienta de configuración crea un módulo para la generación de la señal de reloj y la señal de reset. Una vez que todos los módulos estén creados

en el entorno, se procederá a realizar el conexionado entre el DUV y los módulos QVIP integrados.

- En segundo lugar, se deben configurar los parámetros más específicos de los módulos QVIP anteriormente integrados, desde la anchura de los buses de escritura y lectura a nivel RTL hasta el mapa de direcciones.
- En la tercera etapa se deben especificar los estímulos a enviar al DUV. En este punto, la herramienta ofrece una biblioteca de secuencias aleatorias, así como unas API genéricas de lectura y escritura que favorecen la generación de estímulos de manera eficaz. Se debe tener en cuenta que las secuencias proporcionadas son genéricas, por lo que no se adaptan a las necesidades de una verificación más exhaustiva de un DUV de mayor complejidad. Al finalizar este paso, se debe generar todo el código referente al entorno de verificación, incluyendo los módulos QVIP. La herramienta, además, genera una serie de *scripts* y archivos de soporte para los principales simuladores del mercado.
- Durante la última etapa en el flujo de integración de los módulos QVIP en el entorno de verificación se debe compilar todo el código generado. Aunque la herramienta `QVIP configurator` resulta de gran utilidad, por lo general, se deberán adaptar algunos fragmentos del código para que todo el sistema encaje a la perfección.

Finalizada la generación del *testbench*, se debe ejecutar el mismo y comenzar la fase de depuración de errores del DUV. Esta fase de depuración se basa en la simulación. Para facilitar esta etapa, la herramienta QVIP genera unos archivos de registro que recogen las transacciones de alto nivel enviadas y también las señales específicas del protocolo de comunicación de bajo nivel para poder analizar las formas de ondas en caso de error.

CAPÍTULO 3: DISEÑO A VERIFICAR, TESTBENCH DE REFERENCIA Y PLAN DE VERIFICACIÓN

Durante el presente capítulo se describirán las características y especificaciones del sistema que será objeto de verificación y que se conectará al entorno de verificación UVM. El código en cuestión ha sido diseñado por un investigador del Laboratorio de Sistemas Integrados de la Escuela Politécnica Federal de Zúrich (ETHZ) en Suiza [17]. El proyecto elaborado por Andreas Kurth está protegido con una licencia Apache License Version 2.0, la cual permitirá el uso y modificación de dicho código, siempre referenciando a su autor. En el caso particular de este TFG, el código original no ha sido modificado, usando este diseño simplemente como un sistema que se desea verificar. El módulo de mayor jerarquía en este proyecto se denomina `axi_xbar`.

La integración del módulo `axi_xbar` en el entorno generado en el TFG de Álvaro Moreno [1] no es directa. El módulo `axi_xbar` presenta un inconveniente ya que dicho módulo utiliza puertos de entrada y salida basados en arrays multidimensionales, característica que no soporta la herramienta QVIP *configurator*. De ahí surge la necesidad de la creación de un módulo, denominado `axi_xbar_wrap`, el cual transformará los puertos multidimensionales en puertos de una sola dimensión, adaptándose así a las exigencias de la herramienta de configuración, encargado de la creación del entorno de verificación UVM.

3.1. MÓDULO *AXI_XBAR*

Para la realización del presente TFG se ha tomado como diseño a verificar un componente de interconexión, cuya utilización está muy generalizada en los sistemas de comunicación y conmutación de datos. Este componente se llama *crossbar* y su principal función es la conexión simultánea de todos los puertos de entrada y salida, siempre que no se intente acceder a la misma salida al mismo tiempo. El conexionado interno de este DUV se basa en una estructura de multiplexores y demultiplexores, pero este detalle no afectará al proceso de verificación del sistema. El procedimiento a seguir se basará en la estimulación de las entradas del DUV y en el exhaustivo análisis de las interfaces de salida. De esta manera se comprobará el comportamiento del dispositivo ante unos estímulos controlados.

La verificación del comportamiento de este DUV se llevará a cabo siguiendo un enfoque *black-box*, aunque no se seguirá la filosofía *black-box* de manera estricta, debido a que es necesario conocer los parámetros configurables del módulo `axi_xbar`, como se

explicará en los sucesivos apartados.

3.1.1. CARACTERÍSTICAS BÁSICAS

En GitHub se encuentra un repositorio [18] que proporciona varios módulos para construir redes de comunicación en un SoC, los cuales se podrían adherir a los estándares AXI4 y AXI4-Lite. Entre ellos se encuentra el módulo cuyo comportamiento se pretende verificar. Todos los módulos proporcionados cuentan con una serie de características comunes que se exponen a continuación:

- El usuario puede implementar cualquier topología de red que desee, ya que se proporcionan bloques de construcción elementales y componentes de interconexión comunes en AXI4 como, por ejemplo, multiplexores, demultiplexores de protocolo y *crossbar*.
- Se prioriza el diseño por composición en detrimento de un diseño por configuración, lo que favorece la modularidad del diseño.
- Estos módulos son parametrizables en relación a la anchura de datos y concurrencia de las transacciones, lo que supone una mejora de la versatilidad del diseño. Por lo tanto, se pueden adaptar a múltiples redes de manera optimizada ya que cubre una amplia gama de requisitos de rendimiento, potencia y área.
- Como se ha comentado, todos los módulos poseen una adaptación total al estándar AXI4.
- Estos módulos están desarrollados en SystemVerilog y son compatibles con múltiples herramientas EDA. Por esta razón, se han utilizado construcciones de lenguaje lo más simples posible, persiguiendo siempre la simplificación del código para extender la compatibilidad con el mayor número posible de herramientas EDA.

3.1.2. VISIÓN GENERAL DEL DISEÑO

De todos los módulos que contiene el repositorio en GitHub, el que concierne al presente Trabajo Fin de Grado es el módulo `axi_xbar`, el cual implementa un componente de interconexión llamado *crossbar* [19]. La implementación de este dispositivo resulta muy útil en algunos protocolos de comunicación y su uso está muy extendido. En la Figura 18 se presenta el diagrama de bloques del sistema, donde cada módulo IP maestro se conecta a un puerto esclavo y, de esta manera, tiene conexión directa a todos los módulos IP esclavos que forman parte de dicho sistema y que, a su vez, están conectados a los puertos maestros. Para una mayor modularidad y versatilidad del componente de

interconexión *crossbar*, el usuario puede configurar el número de puertos maestros y esclavos, entre otros muchos parámetros.

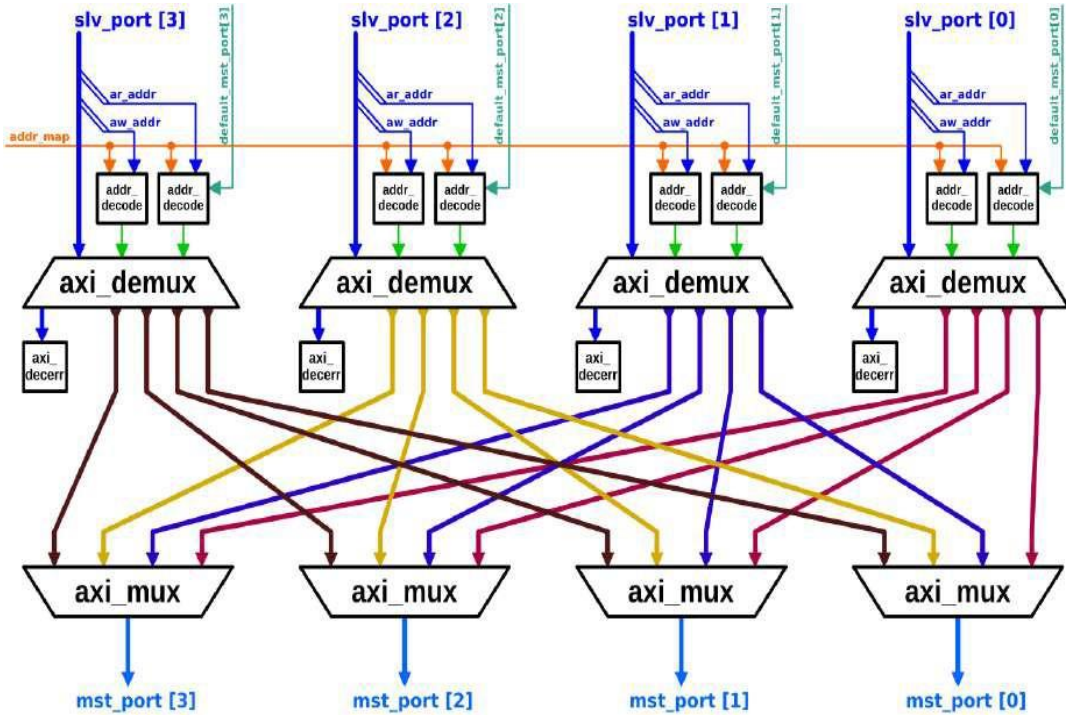


Figura 18 Diagrama de bloques del crossbar (Fuente: [20])

Otra característica a destacar es que todos los puertos maestros comparten un *address map*, en el que se guardarán un conjunto de reglas cuyo propósito es asignar un rango de direcciones a un puerto esclavo. Se deben tener en cuenta las siguientes consideraciones:

- Un *address map* debe contener al menos una regla.
- Se pueden asociar varias reglas a un mismo puerto maestro.
- Los rangos de direcciones de dos reglas diferentes se pueden superponer, es decir, una misma dirección puede estar registrada en dos o varios rangos de direcciones asociados a un mismo puerto maestro.
- El *address map* es un objeto que se puede definir y modificar en tiempo de ejecución, pero no se debe cambiar durante una operación de escritura o lectura. Debido a esta circunstancia, mientras el canal *Address Write* o *Address Read* de cualquier puerto esclavo sea válido, el ingeniero de verificación debe respetar la estructura del *address map* definida con anterioridad.
- En caso de que la dirección de una transacción entrante no esté registrada en ningún rango de direcciones y, por lo tanto, no coincida con ninguna regla del *address map*, la transacción se redirige al módulo esclavo de decodificación que

está presente en cada puerto esclavo. Cuando este módulo absorbe la transacción responde, sin embargo, con un error de decodificación.

- Cada puerto esclavo puede tener asignado un puerto maestro por defecto. En estos casos, ante la llegada de una transacción cuya dirección no coincida con ninguna dirección del *address map*, la transacción se dirige al puerto maestro por defecto y no al módulo de decodificación interno del puerto esclavo. Para que esto suceda, es imperativo que el puerto maestro por defecto se encuentre habilitado. Al igual que el *address map*, el puerto maestro por defecto asociado a un puerto esclavo puede ser modificado en tiempo de ejecución mientras no esté en curso ninguna operación de escritura o lectura.
- Cuando se produce una colisión de transacciones, algo muy común en este tipo de diseños, este conflicto se puede resolver de diversas maneras, ya sea implementando un árbitro, una estructura *Round-robin*, un sistema de prioridades, etc. En este caso, cuando un puerto esclavo recibe dos transacciones de igual ID y misma dirección (por lo que ambas son de lectura o de escritura) pero con distinto destino, no se enviará la segunda transacción hasta que la primera se haya completado. Para ello, el módulo *crossbar* detiene el canal *Address Read* o *Address Write* de ese puerto esclavo en concreto. Esta restricción de ordenación se debe a que, por razones de eficiencia, el *crossbar* no implementa ningún buffer de ordenación.

El módulo *crossbar* es totalmente compatible con el protocolo de comunicación AXI4. Sin embargo, sus señales de interfaz no se han asignado por canales, sino por tipos, resultando un total de cuatro tipos, dos de entrada y dos de salida. Cada uno de ellos agrupa por un lado, las señales de tipo *request* y por otro, las señales tipo *response*. En la Tabla 6 se han listado los distintos puertos de entrada y salida que forman parte del DUV. Los puertos esclavos y maestros se han implementado como arrays bidimensionales a través de los cuales se deben pasar todas las señales pertenecientes al estándar AXI4, concatenadas en un orden específico. Para ello, es necesario conocer dicho orden, que estará marcado por el módulo *crossbar*. Por lo tanto, como se comentó anteriormente, no se seguirá un enfoque *black-box* estricto en este caso.

Tabla 6 Puertos de entrada y salida del DUV

Nombre del puerto	Descripción
`clk_i`	Señal de reloj que sincroniza el resto de las señales.
`rst_ni`	Señal *reset* asíncrona y activa a nivel bajo.
`test_i`	Señal asociada a la habilitación del modo test y activa a nivel alto.
`slv_ports_*`	Matriz de los puertos esclavos del *crossbar*. El índice del array hace referencia a un puerto esclavo en concreto. Este índice se debe anteponer a todas las solicitudes en uno de los puertos maestros.
`mst_ports_*`	Matriz de los puertos maestros del *crossbar*. El índice del array hace referencia a su respectivo puerto maestro.
`addr_map_i`	*Address map* o mapa de direcciones del *crossbar*.
`en_default_mst_port_i`	Señal que define si el maestro por defecto para cada esclavo está activo.
`default_mst_port_i`	Índice del puerto maestro por defecto para el correspondiente puerto esclavo, en caso de estar activo.

3.1.3. *TESTBENCH* de referencia original

En el repositorio de GitHub, el módulo *`axi_xbar`* incorpora su propio *testbench* y éste destaca por ser bastante completo tanto en contenido como en forma. Es un *testbench ad hoc*, es decir, es una verificación que se ajusta únicamente a los requisitos del módulo *`axi_xbar`*, por lo que limita la reutilización de código de verificación desarrollado. Es aquí donde cobra especial interés el desarrollo de un *testbench* UVM, ya que la metodología UVM proporciona herramientas que permiten la creación de un entorno de verificación donde se garantiza la reusabilidad del código.

Durante la preparación del presente TFG, se hizo un estudio completo del *testbench* original. Este *testbench* se ejecutó y se analizaron las formas de ondas generadas por el simulador QuestaSim. Esto ha permitido la realización de diferentes simulaciones usando las mismas condiciones que en el *testbench* original.

Una de las principales ventajas [1] que tiene el uso de los módulos IP de verificación frente a la topología de verificación original, es la fácil adaptación a la metodología UVM. La topología original permite la verificación del sistema en el dominio estático, por lo que el test a ejecutar será siempre el mismo, siendo imposible ejecutar otro test sin antes volver a compilar todo el diseño. En el caso del *testbench* usando IP de verificación con

metodología UVM, el *testbench* se divide en secuencias. Una vez compiladas estas secuencias, se puede ejecutar cualquiera de ellas, o grupos de ellas, sin necesidad de volver a compilar el sistema.

Por último, cabe destacar que la realización de un *testbench* mediante el uso de secuencias reduce la complejidad de este de forma considerable, ya que el ingeniero de verificación no tiene por qué poner atención al protocolo de comunicación utilizado por el DUV, además de obviar todo lo relacionado con las restricciones temporales que implica el protocolo utilizado.

3.1.4. MÓDULO *WRAPPER axi_xbar_wrap*

El módulo `axi_xbar` [1] presenta un inconveniente para el correcto desarrollo de este Trabajo Fin de Grado, ya que dicho módulo no se adapta de manera adecuada a un entorno de verificación UVM, concretamente a la herramienta QVIP *configurator* debido a los motivos ya expuestos y que se analizarán más adelante. De ahí surge la necesidad de la creación de un módulo denominado `axi_xbar_wrap`, el cual se encargará de adaptar el interfaz de aquel a los tipos soportados por la herramienta de configuración, encargada de la creación del entorno de verificación UVM.

La necesidad de implementar el módulo `axi_xbar_wrap` es prioritaria por varios motivos, los cuales se comentarán a continuación. Además, se analizarán las diferentes funcionalidades de este módulo.

- El módulo `axi_xbar` es completamente incompatible con la herramienta QVIP *configurator*, debido a que esta herramienta no acepta módulos con tipos de datos parametrizables y el DUV declara algunos parámetros de esa manera (parameter type). Una solución aceptable sería la adaptación de dicho código modificando esos parámetros para que la herramienta QVIP *configurator* aceptase el DUV, pero se ha optado por no realizar modificaciones al módulo *crossbar* original. Por las razones expuestas, se creará un módulo envolvente *wrapper* que referencie el módulo `axi_xbar` y que adapte los tipos utilizados por este a aquellos compatibles con la herramienta de Mentor Graphics.
- Como ya se ha comentado, el módulo `axi_xbar` tiene además parámetros que se configuran a través de un objeto de configuración externo, el cual está definido en un módulo diferente llamado `axi_pkg`. En este TFG se ha preferido no mantener esa dependencia a un módulo externo. Por lo tanto, el módulo *wrapper* se encargará de crear el objeto de configuración que contenga los parámetros propios de configuración del DUV.

- Por último se adaptarán las entradas y salidas del módulo *axi_xbar*, ya que no posee una interfaz AXI4 diferenciada por puertos, aunque la asociación de las señales de interfaz se realizará en el componente *UVM Top* creado por el QVIP *configurator*.

A la hora de implementar el código del módulo *wrapper* se ha seguido la línea de trabajo empleada en el módulo *crossbar*. Por esta razón, el *wrapper* está elaborado con las estructuras definidas en el *testbench* original. En esta implementación se han obviado cuatro señales respecto al diseño *axi_xbar* original: la señal de entrada correspondiente al *address map*, porque este mapa directamente se implementa en el módulo *wrapper*; la señal de activación del modo test y las dos señales que asignan y habilitan un maestro por defecto, ya que estas son irrelevantes para el diseño planteado.

El diseño contará, por un lado, con 6 maestros que estarán conectados a los puertos esclavos habilitados por el DUV y, por otro lado, con 8 esclavos que irán asociados a sus respectivos puertos maestros. Debido a esta configuración de los puertos de entrada y salida, estos puertos se han implementado como matrices de dos dimensiones para cada puerto maestro y esclavo, cuyos campos estarán compuesto por las señales concatenadas propias del protocolo AXI4 y por el número de esclavos y maestros que tenga el diseño. En la Figura 19 vemos la asignación que hacemos de estos puertos, y sus correspondientes puertos asociados que se generan.

```
1  module axi_xbar_wrap#(
2      parameter int NoMasters = 6,
3      parameter int NoSlaves = 8
4  ) (
5    input  logic                                clk_i,
6    input  logic                                rst_ni,
7    input  var logic [NoMasters-1:0][228:0]     slv_ports_req_i,
8    output logic     [NoMasters-1:0][91:0]      slv_ports_resp_o,
9    output logic     [NoSlaves-1:0][234:0]      mst_ports_req_o,
10   input  var logic [NoSlaves-1:0][97:0]       mst_ports_resp_i
11 );
```

Figura 19 Definición del módulo axi_xbar_wrap

A partir de las señales concatenadas creadas (`mst_req_t`, `mst_resp_t`, `slv_req_t` y `slv_resp_t`), se añade cuatro arrays, denominados *masters*`_req`, *masters*`_resp`, *slaves*`_req` y *slaves*`_resp`, cuyos tamaños vienen determinados por el número de maestros, los dos primeros, y por el número de esclavos,

los dos últimos. Cada uno de estos arrays contendrá la siguiente información:

- *Masters*_req: agrupa todos los campos del protocolo AXI4 generados por los maestros hacia las interfaces maestras del DUV. Un caso típico corresponde a las señales VALID de los canales de dirección y dato de escritura que van de los dispositivos maestros al DUV.
- *Masters*_resp: agrupa todos los campos del protocolo AXI4 generados por los esclavos a través de las interfaces maestras del DUV. Un caso típico corresponde a las señales READY de los canales de dirección y dato de escritura que van del DUV hacia los dispositivos maestros.
- *Slaves*_req: agrupa todos los campos del protocolo AXI4 generados por los maestros a través de las interfaces esclavas del DUV. Un caso típico corresponde a las señales VALID de los canales de dirección y dato de escritura que van del DUV a los dispositivos esclavos.
- *Slaves*_resp: agrupa todos los campos del protocolo AXI4 generados por los esclavos hacia las interfaces esclavas del DUV. Un caso típico corresponde a las señales VALID de los canales de dirección y dato de escritura que van de los dispositivos esclavos al DUV.

Para una mayor comprensión, se adjunta el esquema de la Figura 20, donde se visualiza el módulo *crossbar* conectado al módulo *wrapper* y, a su vez, al entorno de verificación. Es un esquema parcial que sólo muestra el componente *UVM Top Module* o *Testbench* del entorno de verificación. Este componente UVM es el encargado de realizar la concatenación de todas las señales AXI4 en los correspondientes puertos de entrada y salida y conectar el módulo *wrapper* con el entorno mediante las interfaces virtuales definidas.

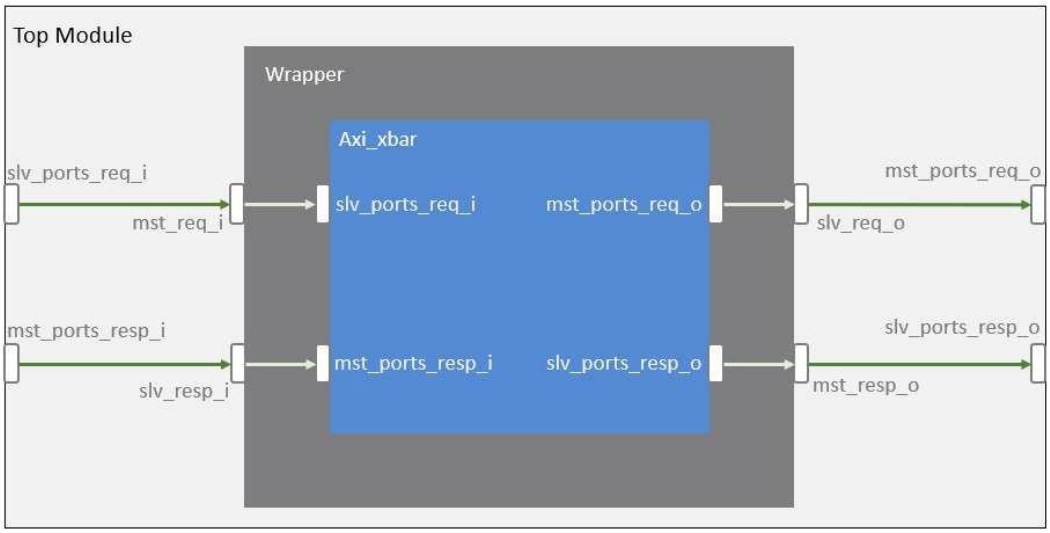


Figura 20 Esquema parcial de los módulos crossbar y wrapper conectados al componente UVM Top Module

3.2. *TESTBENCH* UVM

Los componentes que van a formar el *testbench* se generan de manera automática desde la herramienta de *QVIP configurator*. Esta herramienta aporta agilidad al proceso de verificación, ya que a partir de un DUV se crea todo el entorno, elaborando todo el conexionado que se requiere para su funcionamiento.

En el caso particular del presente Trabajo Fin de Grado se ha reutilizado el código generado por el TFG de Álvaro Moreno Florido [1] utilizando la herramienta de configuración mencionada con el propósito de generar el código base del entorno de verificación UVM. Sin embargo, debido a que las interfaces del DUV seleccionado están agrupadas, el conexionado no se ha efectuado de manera automática.

3.2.1. ESQUEMA GENERAL DEL *TESTBENCH* UVM GENERADO

El entorno de verificación UVM tiene una jerarquía bien definida, consiguiendo un mayor nivel de abstracción y una notable mejora de la modularidad y la reusabilidad del código. En la Figura 21 se presenta la jerarquía del sistema diseñado durante este TFG, que cuenta con diferentes componentes UVM que conforman todo el entorno de verificación UVM. A nivel práctico, se ha optado por representar dicha jerarquía de manera parcial, sin dibujar el número total de interfaces, debido a una cuestión estética. El sistema real diseñado incorpora seis módulos QVIP maestros y ocho módulos QVIP esclavos. Como se estudiará más adelante, estos módulos se comportarán como componentes *UVM Agent*, pudiendo estar configurados tanto de manera activa como pasiva.

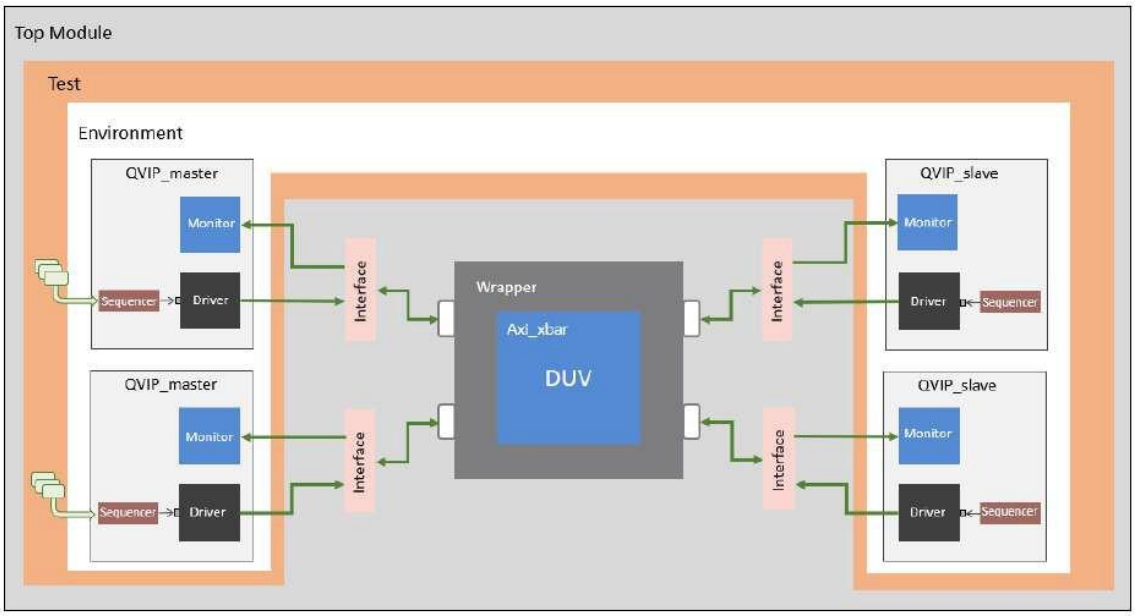


Figura 21 Esquema parcial del entorno de verificación UVM y el módulo DUV

3.2.2. ARCHIVO DE SIMULACIÓN

La herramienta de configuración QVIP también genera un archivo de simulación que se puede ejecutar en un terminal Linux. Es un archivo sencillo, cuyo contenido se presenta en la Figura 22 y que contiene una referencia a todos los comandos y paquetes que se utilizan. Este archivo se llama `Questa_run` y no forma parte de la jerarquía UVM. Su función consiste en simplificar la compilación y optimización del código, así como la posterior simulación del entorno.

```
rm -r work

echo "Setting up work directory"
vlib work
vlog -sv ${QUESTA_MVC_HOME}/include/questa_mvc_svapi.svh                    \
    +define+MAP_PROT_ATTR ${QUESTA_MVC_HOME}/questa_mvc_src/sv/mvc_pkg.sv  \
    +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv                           \
    +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4                      \
    +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules              \
    +incdir+../config_policies  \
    +incdir+../../../../common_cells-master/include  \
    +incdir+../../../../axi-master/include           \
    +incdir+../sequence_lib     \
    +incdir+../uvm_tb           \
    ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/mgc_axi4_v1_0_pkg.sv         \
    ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules/axi4_master.sv       \
    ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules/axi4_slave.sv        \
    ../config_policies/qvip_axi_xbar_params_pkg.sv                         \
    ../../../../common_cells-master/src/cf_math_pkg.sv    \
    ../../../../common_cells-master/src/lzc.sv            \
    ../../../../common_cells-master/src/deprecated/fifo_v2.sv   \
    ../../../../common_cells-master/src/fifo_v3.sv              \
    ../../../../common_cells-master/src/stream_register.sv      \
    ../../../../common_cells-master/src/addr_decode.sv          \
    ../../../../common_cells-master/src/rr_arb_tree.sv          \
    ../../../../common_cells-master/src/spill_register.sv       \
    ../../../../common_cells-master/src/delta_counter.sv        \
    ../../../../common_cells-master/src/counter.sv              \
    ../../../../common_verification-master/src/rand_id_queue.sv \
    ../../../../common_verification-master/src/clk_rst_gen.sv   \
    ../../../../axi-master/src/axi_pkg.sv                  \
    ../../../../axi-master/src/axi_intf.sv                 \
    ../../../../axi-master/src/axi_atop_filter.sv          \
    ../../../../axi-master/src/axi_demux.sv                \
    ../../../../axi-master/src/axi_id_prepend.sv           \
    ../../../../axi-master/src/axi_mux.sv                  \
    ../../../../axi-master/src/axi_err_slv.sv              \
    ../../../../axi-master/src/axi_xbar.sv                 \
    ../uvm_tb/qvip_axi_xbar_conf_if.sv         \
    ../uvm_tb/test_params_pkg.sv               \
    ../uvm_tb/qvip_axi_xbar_pkg.sv             \
    ../sequence_lib/qvip_axi_xbar_seq_pkg.sv   \
    ../top/axi_xbar_wrap.sv                    \
    ../common/default_clk_gen.sv               \
    ../common/default_reset_gen.sv             \
    ../top/hdl_qvip_axi_xbar.sv                \
    ../top/hvl_qvip_axi_xbar.sv

echo "Optimisation step"
vopt -o top_opt hdl_qvip_axi_xbar hvl_qvip_axi_xbar
echo "Running test"
vsim   -mvchome ${QUESTA_MVC_HOME} top_opt +nowarnTSCALE -t 1ns -do "do
./dofiles/wavemio.do; run -a;q" +UVM_TESTNAME=qvip_axi_xbar_test_base
+SEQ=qvip_axi_xbar_vseq_t10_p
```

Figura 22 Código del archivo de simulación questa_run

Dentro del archivo que se muestra en la Figura 22, se invocan los comandos `vlib`, `vlog`, `vopt` y `vsim`, propios del entorno QuestaSim de Mentor Graphics, a Siemens Business. Estos comandos, básicamente, crean la librería de trabajo; compilan el código RTL; optimizan el código resultante; y, por último, simulan el código compilado.

Si el lector está interesado en profundizar en las diferentes opciones con las que se invocan estos comandos, puede consultarlos en [20].

3.2.3. ESQUEMAS DE DIRECTORIOS

A continuación, se muestra la Figura 23 en la que se detalla la estructura de directorios que crea la herramienta `QVIP` ***`configurator`*** como herramienta al generar el *testbench* que nos ocupa.

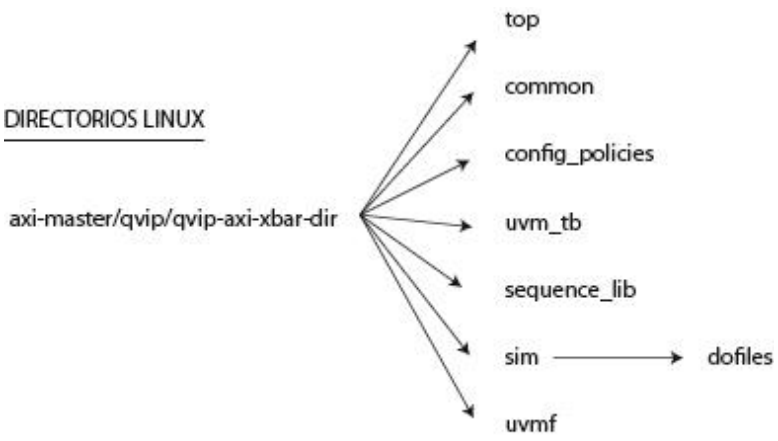


Figura 23 Esquema de directorios generados por QVIP configurator

A continuación, se resume el contenido y propósito de cada uno de los directorios.

Top: En este directorio se incluyen los archivos pertenecientes la capa de mayor abstracción del entorno de verificación. Se tienen:

- ***axi_xbar*_wrap.sv:** Archivo que contiene el módulo de mayor jerarquía del DUV que, en este caso corresponde al módulo envolvente.
- **hdl_qvip_*axi_xbar*.sv:** Módulo top que referencia al módulo anterior y referencia a los diferentes QVIP maestros y esclavos conectados al mismo.
- **hvl_qvip_*axi_xbar*.sv:** Este módulo únicamente comienza la ejecución de fases de UVM.

Common: En este directorio se encuentran los ficheros que definen el reloj general y el reset.

Config_policies: Este directorio contiene las configuraciones de los QVIP *masters* y los QVIP *slaves* conectados al DUV. Se define un fichero por cada QVIP *master* y por cada QVIP *slave*. Además, se define un fichero general que engloba todas las configuraciones. En el presente TFG, este fichero se denomina `qvip_axi_xbar-params-pkg.sv`. A modo de ejemplo, la Figura 24 muestra un extracto del mismo, donde se aprecia la definición, en primer lugar, de la configuración del número de maestros y el número de esclavos. Posteriormente se muestra los parámetros de configuración para el maestro #0. En este diseño, todos los *masters* y *slaves* están configurados con los mismos parámetros.

```
7   package qvip_axi_xbar_params_pkg;
8       import addr_map_pkg::*;
9       import rw_delay_db_pkg::*;
10      //
11      // Import the necessary QVIP packages:
12      //
13      import mgc_axi4_v1_0_pkg::*;
14      class axi_xbar_wrap_params;
15          localparam int NoMasters = 6;
16          localparam int NoSlaves  = 8;
17      endclass: axi_xbar_wrap_params
18
19      class axi4_master0_params;
20          localparam int AXI4_ADDRESS_WIDTH   = 32;
21          localparam int AXI4_RDATA_WIDTH     = 64;
22          localparam int AXI4_WDATA_WIDTH     = 64;
23          localparam int AXI4_ID_WIDTH        = 4;
24          localparam int AXI4_USER_WIDTH      = 5;
25          localparam int AXI4_REGION_MAP_SIZE = 16;
26      endclass: axi4_master0_params
27
```

Figura 24 Parámetros de configuración master y slaves

uvm_tb: Es en este directorio donde se encuentran todos los ficheros que componen el *testbench*, tal como se han definido anteriormente*, env, test, seq_lib*, etc …

sequence_lib: En este directorio es donde están colocadas todas las secuencias con las que se va a trabajar. Además, es donde se ubicarán las secuencias *ad hoc* generadas automáticamente por la herramienta de configuración.

sim: Es la carpeta donde se colocan los ficheros para ejecutar la compilación y la simulación. El fichero que se usará en este TFG será el Questa_run que corresponde al entorno de verificación de Mentor Graphics. En esta carpeta existe una carpeta *`dofiles`* que es donde se almacenaran los ficheros que contienen las configuraciones para las representaciones de las formas de ondas durante la simulación. Se almacenará un fichero por cada test o secuencia a ejecutar.

uvmf: Este directorio contiene todos los ficheros del *testbench* para su ejecución con el entorno *UVM Framework (UVMF)*, que está fuera del alcance de este TFG.

3.2.4. DIRECTORIOS ESPECIALES DE COMPILACION

En el proceso de instalación de Mentor se crean unas carpetas que son importantes para comprender este TFG, estas carpetas contienen las librerías y las clases de las cuales se heredaran todas las clases que posteriormente se utilizaran. Además, se hace referencia también a una carpeta que ha servido de estudio, donde mentor ha creado un *testbench* de referencia *ad hoc*, en donde se presenta un *testbench* de un sistema con un único *master* conectado mediante AXI-4 con un único *slave*.

El primer directorio es:

- `Soft/eucad/mentor/2019/Questa_vip_2019_1_1_Linux/Questa_mvc_src/sv/axi4`

El *testbench* de referencia está ubicado en:

- `qvip/axi4`

3.3. CARACTERÍSTICAS A COMPROBAR DEL DUV A VERIFICAR

En este apartado se analizará la hoja de datos del diseño que se va a verificar y se extraerá un listado de las secuencias que se van a crear para verificar sus especificaciones.

3.3.1. DESCRIPCION DE LAS CARACTERISTICAS

A modo aclaratorio en el módulo descrito, y para facilitar su entendimiento se hará un cambio de nomenclatura a la hora de nombrar los puertos en la traducción. El puerto *slave* es el de entrada y el *master* el de salida, lo cual no tiene sentido para un mejor entendimiento, es por eso por lo que en la traducción se ha optado por cambiar *slave* por *master*, y *master* por *slave* respectivamente, de forma que se usara un único criterio a la hora de nombrarlos. Además, el módulo *`axi_xbar_wrap`* que se generó para la verificación de este DUV, y que anteriormente se nombró en el apartado 3.1.4, ya hace referencia de la forma correcta a los puertos *masters* y *slaves*, tal y como se va a nombrar en este apartado.

Nº	Sección	Descripción	Traducción
1 Y 2	*Address map*	"Rule. Each rule maps one address range to one *master* port. Multiple rules can map to the same *master* port. The address ranges of two rules may overlap: in case two address ranges overlap, the rule at the higher (more significant) position in the *address map* prevails. "	Cada módulo *slave* tiene una dirección asignada en el mapa de reglas de direcciones *rule map*. Si embargo, un módulo *slave* se puede ver afectado por varias reglas definidas dentro del *rule map*. Este hecho puede provocar que los rangos de direcciones se solapen. Si esto ocurriera es decir que se solapen los rangos de direcciones, la regla que tenga la posición más alta en *el adrress map* prevalece.
3	*Address map*	Each address range includes the start address but does not include the end address. That is, an address matches. an address range if and only if addr >= start_addr && addr < end_addr	Cada rango de direcciones *(range address)* de los módulos *slaves* incluye una dirección de comienzo, pero no incluye una dirección de fin. Por lo que cuando se direcciona sobre una dirección en modo ráfagas, esta dirección se dará por buena si y solo si la dirección es mayor que la dirección de comienzo y menor que la dirección de fin.

4	*Decode Errors and Default Slave Port*	Each *slave* port has its own internal decode error *slave* module. If the address of a transaction does not match any rule, the transaction is routed to that decode error *slave* module. That module absorbs each transaction and responds with a decode error (with the proper number of beats). The data of each read response beat is 32'hBADCAB1E (zero-extended or truncated to match the data width).	Cada módulo *master* tiene su propio módulo de error interno. Si la dirección de respuesta asignada de la transacción no pertenece a ninguna regla de asignación de direcciones de los módulos *slaves*, entonces se le asigna la dirección del módulo de error interno asignada al módulo *master*. El módulo *master* recoge la transacción y responde con un `decode error` (con el número adecuado de ciclos). Los datos de cada ciclo de respuesta de lectura son 32'hBADCAB1E (cero extendido o truncado para que coincida con el ancho de datos).
5	Decode Errors and Default *Slave* Por	Each *slave* port can have a default *master* port. If the default *master* port is enabled for a *slave* port, any address on that *slave* port that does not match any rule is routed to the default *master* port instead of the decode error*slave*. The default *master* port can be enabled and changed at run time (it is an input signal to the *crossbar*), and the same restrictions as for the *address map* apply.	Cada módulo *master* puede tener un puerto modulo *slave* por defecto. Si la señal `default_master_port` está habilitada, cada acceso a una dirección que se envíe y no tenga una dirección de módulo *slave* correspondiente dentro del mapa de direcciones, entonces será enviada a este módulo *slave* de defecto, y no al de la dirección real, no incluida

			dentro del mapa de direcciones. Este módulo *default slave* es configurable.
6	*Ordering and Stalls*	When one *slave* port receives two transactions with the same ID and direction (i.e., both read or both write) but targeting two different *master* ports, it will not accept the second transaction until the first has completed.	Cuando un módulo *master* recibe dos transacciones con el mismo identificador *ID* y misma dirección, pero apuntando desde dos diferentes módulos *slaves,* no aceptará la segunda transacción hasta que la primera se haya completado.
7	*Design Rationale for No Pipelining Inside Crossbar*	all W beats have to arrive in the same order as the AW beats regardless of the ID at the *slave* module. Thus, the different *master* ports of the multiplexer exclude each other because the order is given by the arbitration tree of the AW channel.	Todas las órdenes *write data* tienen que llegar en el mismo orden que todas las ordenes *write addr*, independientemente del id del módulo *master*. El multiplexor de diferentes puertos módulos *slave*, excluye cada orden de *write data*, porque el orden es dado por el arbitraje del canal de escritura de dirección de la orden *write addr*.
8	*Design Rationale for No Pipelining Inside Crossbar*	AXI does not allow interleaving of W beats and requires W bursts to be in the same order as AW beats.	AXI4 no permite entrelazar las transacciones de ciclos de escritura de datos (W) y requiere que las transacciones de ciclos de escritura en ráfagas estén en

			el mismo orden que las transacciones de escritura de ciclos de direcciones (AW).
9	*Design Rationale for No Pipelining Inside Crossbar*	It is constructed in a way by giving each of its *slave* ports an increasing priority and then comparing pairwise down till a winner is chosen. When the winner gets transferred, the priority state is advanced by one position, preventing starvation.	El DUV está basado en dar a cada uno de sus puertos *master* una prioridad y entonces compararla por parejas hasta que elige el de mayor prioridad. Entonces cuando se transfiera, el estado de prioridad es avanzado una posición, evitando el fenómeno de *starvation*. Es decir, asigna prioridades al *master*, transfiere el de más prioridad, y entonces vuelve a asignar prioridades subiendo el nivel y el transferido pasa a la posición final de prioridad.
10	*No in data sheet (Exclusive access)*	Demostración del acceso exclusive con escritura backdoor	Demostración del acceso *exclusive* con escritura *backdoor*

Tabla 7 Características a verificar del DUV

3.3.2. NOMBRES DE LAS SECUENCIAS Y ACCIONES A EJECUTAR

A continuación, se describen las acciones a seguir y las respuestas esperadas en cada una de ellas.

Nº	Secuencia	Acción a ejecutar	Salidas esperadas
1 **Y** **2**	qvip_*axi_xbar*_vseq_t1.svh qvip_*axi_xbar*_vseq_t2.svh	Cambiar las reglas de rango de direcciones para que haya solapamiento en dos módulos *slaves*. Escribir en una o varias de las direcciones solapadas.	Todas las escrituras deben unirse al mismo modulo *slave*. El módulo *slave* debe corresponder con el módulo que ocupa la posición más alta en el mapa de reglas de direcciones.
3	qvip_*axi_xbar*_vseq_t3.svh	Escribir más de un byte en el modo incremental en la posición del módulo *slave* que se desea asignar, en el límite de su rango de direcciones.	Como la condición a cumplir es que esté dentro de un rango de direcciones, y se intenta apuntar a un rango que se sale del permitido, la respuesta esperada es que responda con un mensaje de error.
4	qvip_*axi_xbar*_vseq_t4.svh	Se procede a modificar los campos de direcciones de las reglas de asignación de los módulos *slaves* para dejar una dirección fuera de rango de direcciones, y escribir en esa dirección para observar la respuesta.	La respuesta esperada es un `decode error`.

5	qvip_*axi_xbar*_vseq_t5.svh	Establecer la señal del *enable* activo para el módulo correspondiente y asignar un módulo *slave* por defecto para los módulos *master*s. Posteriormente escribir en una de esas direcciones fuera del rango de direcciones, asignadas en las reglas de direccionamiento, '*rules'*, y observar la salida.	La respuesta que se espera es que no devuelva un mensaje de error, y se direccione el módulo *slave* correspondiente establecido en la señal *master_port.*
6	qvip_*axi_xbar*_vseq_t6.svh	Escribir dos transacciones desde el mismo módulo *master* con el mismo identificador y dirección, hacia dos módulos *slave* diferentes.	La respuesta esperada es que se ejecutarán de forma consecutiva, no al mismo tiempo.
7	qvip_*axi_xbar*_vseq_t7.svh	Se envian varias órdenes de direcciones desde distintos módulos *masters*, al mismo módulo *slave*, y se hará en modo concurrente con el comando `fork-join`, de forma que no se determina una ejecución secuencial y ordenada, incluso con	Que se escriban en orden de escritura de dirección, y sin comunicar una respuesta de error. Es decir, que el propio DUV sea capaz de administrar las llegadas de ciclos de direcciones y datos y asignadas en orden correcto.

		dos escrituras desde el mismo módulo *master* con dos identificadores diferentes. Las órdenes iniciarán dos ciclos distintos de escrituras de direcciones y de datos independientes.	
8	qvip_*axi_xbar*_vseq_t8.svh	Se generan dos transacciones de escritura de ráfagas de ciclo de escritura de datos (W), y una sola transacción de ciclo de escritura de dirección (AW) y observar el resultado.	Que devuelva un mensaje de error.
9	qvip_*axi_xbar*_vseq_t9.svh	Se generan ráfagas de transferencia de datos desde el mismo módulo *master* y también ráfagas de datos desde otros módulos *master* para comprobar que las transferencias de los módulos se intercalan, y un módulo no bloquea a los demás módulos, este término es lo que se conoce como *starvation*.	Las transacciones desde el mismo módulo *master* no tienen que transferirse consecutivas, sino alternadas con otros módulos *masters*.

10	qvip_*axi_xbar*_vseq_t10.svh	Se generan distintas transacciones de lectura y escritura y se intercala un acceso por modo *backdoor* para modificar los datos.	Respuesta con `EXOKAY`, o bien `OKAY` en los casos correctos

Tabla 8 Secuencias y acciones a ejecutar sobre el DUV

Estas especificaciones han sido extraídas de las secciones de la hoja de datos que se aportan en el Anexo I.

CAPÍTULO 4: SECUENCIAS DE VERIFICACIÓN UVM

En este capítulo, por ser el objetivo principal de este TFG, se describe la realización de un plan de verificación completo del DUV selecionado. En este capítulo se describe y analizan las partes, las fases y la forma de ejecución de las secuencias de verificación siguiendo la metodología UVM. Este plan de verificación se sustenta en las secuencias de verificación, es por eso por lo que se dedica un capítulo completo en analizar su estructura y su uso en la metodología UVM. En este capítulo se presenta la distribución de una secuencia UVM, el protocolo de *handshake* utilizado para mandar su secuencia UVM al componente *UVM Driver* a través del componente *UVM Sequencer*; y el análisis y estudio de las secuencias proporcionadas por el *testbech* generado por la herramienta *QVIP configurator.*

4.1 SECUENCIAS Y TRANSACCIONES EN UVM

En un entorno de verificación UVM [9], las secuencias son objetos dinámicos que no forman parte de la jerarquía de componentes. Es una clase que hereda, a su vez, de la clase `uvm_sequence`, por lo que se podrán referenciar las diferentes funciones propias de la clase padre.

La secuencia UVM, *UVM Sequence,* es un contenedor que se compone de varios elementos de datos que se agrupan de diferentes formas para crear distintos escenarios mediante la estimulación del DUV. Una secuencia es ejecutada por un componente *UVM Sequencer*, asignado a un componente que envía estos elementos de datos a su componente *UVM Driver*. De esta manera, las secuencias juegan un papel principal en el proceso de verificación, ya que constituyen los estímulos centrales durante el tiempo de simulación de la fase `run_phase`.

UVM proporciona dos macros que permiten enviar sus secuencias al componente *UVM Sequencer*:

- `uvm_do (SEQ_OR_ITEM)`
- `uvm_do_with (SEQ_OR_ITEM,CONSTRAINTS)`

Estas macros realizan la creación, aleatorización y el envío de las transacciones que componen una secuencia. Por lo tanto, el primer argumento de la macro corresponde con el nombre de la secuencia a enviar que ha tenido que ser definida previamente. Ambas macros realizan exactamente la misma labor, pero existe una pequeña diferencia entre ellas. La macro `uvm_do_with()` ofrece la posibilidad de añadir restricciones

en la aleatorización de los campos de una transacción.

Una transacción UVM es un ítem [9] de datos que hereda de la clase `uvm_sequence_item`. A grandes rasgos, una transacción UVM es un objeto que se puede definir como un conjunto de campos de datos cuyo principal y único propósito es la estimulación de las entradas del DUV en cuestión. Por lo general, se suelen aleatorizar los campos de datos que componen una transacción UVM, dando la posibilidad de que el ingeniero de verificación cree ciertas restricciones a aplicar durante dicho proceso de aleatorización.

A diferencia de los componentes del *testbench*, la creación de las transacciones UVM es dinámica y se produce en tiempo de ejecución, por lo tanto, ocurre durante la ejecución de la fase `run_phase`.

4.2. PROTOCOLO HANDSHAKE *SEQUENCER-DRIVER*

Aunque escapa al propósito de este TFG, ya que este TFG se centra en la depuración de las secuencias y todo este procedimiento se realiza en un nivel superior de abstracción, se considera muy importante analizar cómo se realiza la comunicación entre el componente *UVM Sequencer* y el componente *UVM Driver.* Este mecanismo de comunicación se detalla en la Figura 25.

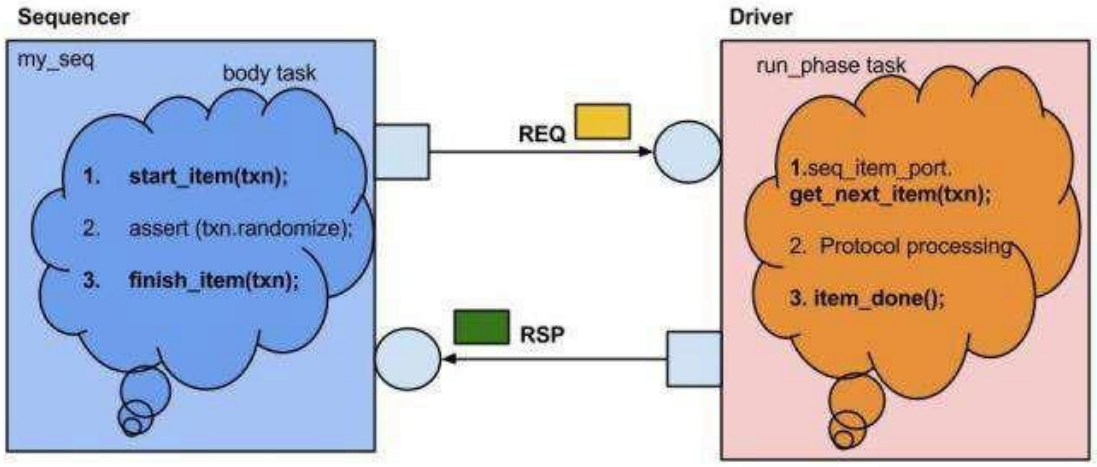


Figura 25 Esquemático del protocolo handshake Sequencer-Driver [21]

Este mecanismo [21] de comunicación sigue su protocolo *handshake* entre estos dos componentes. El protocolo para enviar una transacción desde el componente *UVM Sequencer* hasta el componente *UVM Driver* consta de cuatro pasos desde el punto de vista del *UVM Sequencer* que se enumeran a continuación:

Cuando una secuencia se dirige al componente *UVM Sequencer*, se descompone en una serie de ítems, cada ítem es una transacción que, a su vez, serán enviados al componente *UVM Driver*. Este último convierte cada uno de esos ítems en señales que son aplicadas a los diferentes pines del DUV, repitiendo este proceso en cada ciclo de reloj.

- En primer lugar, mediante la función `start_item(txn)` se crea el ítem de la transacción con su respectivo identificador usando el mecanismo *Factory* anteriormente explicado.
- En segundo lugar, se bloquea el componente *UVM Sequencer* hasta que el componente *UVM Driver* solicita una nueva transacción mediante la función `get_next_item(txn)`. Esta función desbloquea el componente *UVM Sequencer.*
- En tercer lugar, se aleatoriza la transacción respetando las restricciones programadas por el ingeniero de verificación. Este proceso se realiza invocando la función `randomize` de la clase padre de la transacción creada.
- Por último, se llama a la función *finish_item (transaction_item_handle)*, la cual es de naturaleza bloqueante y cuyo propósito es bloquear el componente *UVM Sequencer* a la espera de que el componente *UVM Driver* transfiera los datos de transacción relacionados con el protocolo. Una vez el componente *UVM Driver* ha enviado la transacción, ejecuta la función `item_done`, que desbloque el componente *UVM Sequencer*.

4.3. ANÁLISIS DE LA ESTRUCTURA DE UNA SECUENCIA

4.3.1. ANÁLISIS DEL CÓDIGO DE LA SECUENCIA BASE

En este apartado se va a profundizar en el análisis del código de la secuencia base, para posteriormente ver su aplicación en una secuencia propia, y cómo se ejecuta esta última.

Esta secuencia está definida como la clase base que los usuarios utilizan para definir transacciones AXI4 básicas de lectura y/o escritura. El usuario siempre deriva de esta clase que, a su vez, deriva de otras clases como se muestra en la Figura 26. Esta figura muestra la jerarquía de la clase, donde se observa las clases heredadas, hasta llegar a la clase *`mvc_sequence`* que, a su vez, deriva de la clase `uvm_sequence`. Los usuarios usan esta secuencia genérica de lectura y escritura, o bien pueden usar

secuencias específicas del protocolo AXI4, para definir secuencias propias, personalizándolas con el uso de las API que se definen en las diferentes clases.

Figura 26 Jerarquía secuencia base

En la Figura 27 se muestra el código de la secuencia `axi4_master_deparam_seq` que deriva de la clase `axi4_rw_api_base_seq`. En la primera línea de código se define la clase.

El primer paso es registrar la secuencia en la *Factory*, mediante la macro `uvm_object_utils`, tal y como se muestra en la segunda línea. A continuación se definen una serie de variables que van a ser usadas posteriormente para la verificación de la configuración de la secuencia.

Posteriormente, se define una función y una tarea. La función, `function new`, corresponde al constructor de la clase. La tarea `task body`, contiene la secuencia de operaciones a realizar y, en su caso, se ha declarado tipo `virtual`. Esto último implica que dicha tarea debe ser definida en las clases que heredan de esta clase.

La funcionalidad del método virtual es que se establece en tiempo de ejecución, lo que permite asignar identificadores de clase extendidos a identificadores de clase base. Como consecuencia las llamadas al método en tiempo de ejecución usando el identificador de clase base, ejecutarán la funcionalidad del método de la clase derivada. Esta es una característica poderosa utilizada en Systemverilog que se llama polimorfismo, muy práctica en las técnicas de Programación Orientada a Objetos.

```
class axi4_master_deparam_seq extends axi4_rw_api_base_seq;

  `uvm_object_utils(axi4_master_deparam_seq)

  // For internal purpose
  axi4_rw_api_base_seq target_seq;
  local mvc_agent host_agent;
  local uvm_sequencer_base host_sequencer;

  // Function:- new
  //
  // Create a new instance of <axi4_master_deparam_seq>.
  //
  function new(string name="axi4_master_deparam_seq");
    super.new(name);
  endfunction

  // Task:- body
  //
  // Starts target sequence
  //
  virtual task body();
    get_target_seq();
  endtask
```

Figura 27 Código de secuencia base axi4_master_deparam_seq

Una de las principales funciones en una secuencia base es la de obtener el *target* sobre el que se está ejecutando la secuencia. Si el usuario ejecuta una secuencia y no indica el componente *UVM Sequencer*, el sistema debe detectarlo y abortar la ejecución. Esta función se denomina `get_target_seq()`.

En la Figura 28 se observa la implementación de la función `get_target_seq()`, que como se comentó hace uso de las variables internas definidas anteriormente. El propósito de esta función es depurar errores de configuración de la secuencia en su forma más básica, evitando de esta manera errores inesperados por mal uso de los ingenieros de verificación de configuraciones o asignaciones de parámetros de las secuencias a utilizar.

UVM es una metodología de verificación que se caracteriza en todo momento en ser muy estricta y que contempla en todo momento los posibles errores por el uso incorrecto de la misma, este control le lleva a contar en todo su código de múltiples métodos de verificación, que avisan al ingeniero si los componentes han sido inicializados de forma correcta, o no son definidas del tipo correcto para evitar errores inesperados y que en el proceso de verificación podrían dar resultados inciertos. Esta función está dirigida exclusivamente a depurar errores de configuración. Para ello esta función lee la variable `target_seq` indicando si devuelve el valor `null`, indicando que no ha sido configurado

y notifica un error. Posteriormente se realiza una llamada a la función `get_Sequencer`() y se comprueba si se le ha asignado un componente *UVM Agent* padre para el componente *UVM Sequencer*, en caso de que no exista vuelve a notificar un error.

```
// Function:- get_target_seq
//
// Gets target sequence handle
//
function void get_target_seq();
  if(target_seq == null) begin
    if(m_sequencer == null) begin
      `uvm_error("AXI4_MASTER_DEPARAM_SEQ", "Sequencer for this sequence should be set using set_sequencer before randomizing it")
    end
    else begin
      host_sequencer = get_sequencer();
      $cast(host_agent, host_sequencer.get_parent());
      $cast(target_seq, host_agent.get_master_seq());

      if(target_seq == null) begin
        `uvm_error("AXI4_MASTER_DEPARAM_SEQ", "The target sequence has not been set up")
        return;
      end
    end
  end
endfunction
```

Figura 28 Código de secuencia base, función get_targer_seq.

A continuación, siguiendo con el código que se muestra en la Figura 29 se crean todas las funciones que tiene la clase para la configuración y lectura de los atributos de la secuencia. Estas funciones se han declarado tipo `extern`, lo que indica que su definición está fuera de la clase. Entre otros, se pueden ver parámetros asociados al protocolo AXI y que posteriormente se usaran, tales como son los parámetros `BURST`, `LOCK`, `SIZE`, `BURST_LENGHT`, etc., y en los cuales se profundizará posteriormente, en el apartado 4.5.

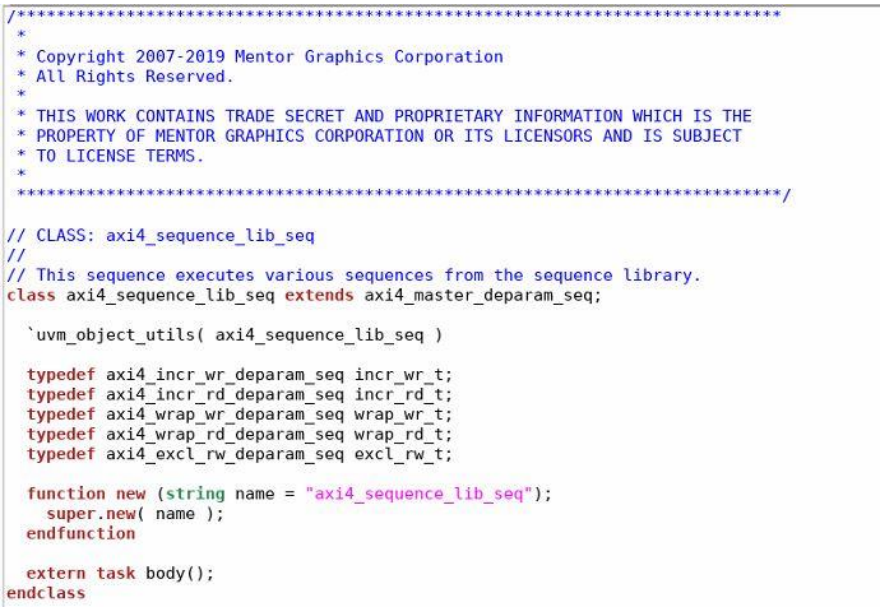
```
// Function: set_attr
//
// Sets default transaction attributes.
extern function void      set_attr        (axi4_def_attr_s attr, int id=-1);

// Function: set_cache_type
//
// Sets cache type of transaction.
extern function void      set_cache_type  (axi4_cache_e cache_type, int id=-1);

// Function: set_lock
//
// Sets lock type of transaction.
extern function void      set_lock        (axi4_lock_e lock, int id=-1);

// Function: set_burst_type
//
// Sets burst type of transaction.
extern function void      set_burst_type  (axi4_burst_e burst, int id=-1);

// Function: set_qos
//
// Sets QoS of transaction.
extern function void      set_qos         (bit[3:0] qos, int id=-1);

// Function: set_size
//
// Sets size of transaction.
extern function void      set_size        (axi4_size_e size, int id=-1);

// Function: set_burst_length
//
// Sets the maximum allowed burst_length of transaction.
extern function void      set_burst_length        (bit [7:0] burst_length, int id=-1);
```

Figura 29 Código de secuencia base cont. 2

4.3.2. ANÁLISIS DEL CÓDIGO DE UNA SECUENCIA PROPIA

En este apartado se muestra una secuencia *ad hoc* creada mediante la herramienta *QVIP configurator*. Además, se detallan cada uno de los pasos antes descritos para la ejecución de la secuencia. Esta forma de proceder es la que posteriormente se utilizará en todas las secuencias que se van a crear, haciendo hincapié en la particularidad de cada una de ellas.

Una secuencia se puede crear derivándola a partir de una secuencia base, `mvc_sequence`, `uvm_sequence` o `axi4_master_deparam_seq`. Estas secuencias propias se componen, a su vez, de secuencias integradas o API de lectura y escritura, que están configuradas de antemano con el tipo de transacción que se quieren realizar, simplificando de esta manera su ejecución. La Figura 30 muestra la secuencia `axi4_sequence_lib_seq` que se deriva de la base `axi4_master_deparam_seq`. Esta secuencia se compone de 5 secuencias básicas, denominadas `incr_wr_t`, `incr_rd_t`, `wrap_wr_t`, `wrap_rd_t` y `excl._rw_t`. Cada una de estas secuencias bases las proporciona la plataforma de configuración.



```
/*****************************************************************************
 *
 * Copyright 2007-2019 Mentor Graphics Corporation
 * All Rights Reserved.
 *
 * THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION WHICH IS THE
 * PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS LICENSORS AND IS SUBJECT
 * TO LICENSE TERMS.
 *
 *****************************************************************************/

// CLASS: axi4_sequence_lib_seq
//
// This sequence executes various sequences from the sequence library.
class axi4_sequence_lib_seq extends axi4_master_deparam_seq;

  `uvm_object_utils( axi4_sequence_lib_seq )

  typedef axi4_incr_wr_deparam_seq incr_wr_t;
  typedef axi4_incr_rd_deparam_seq incr_rd_t;
  typedef axi4_wrap_wr_deparam_seq wrap_wr_t;
  typedef axi4_wrap_rd_deparam_seq wrap_rd_t;
  typedef axi4_excl_rw_deparam_seq excl_rw_t;

  function new (string name = "axi4_sequence_lib_seq");
    super.new( name );
  endfunction

  extern task body();
endclass
```

Figura 30 Código UVM de una secuencia propia, denominada axi4_sequence_lib_seg

El primer paso coincide con lo anteriormente descrito, que es registrar la secuencia en la *Factory*, mediante la macro `uvm_object_utils`. Posteriormente se definen los tipos de las secuencias básicas a utilizar. Finalmente se define una función y una tarea. La función, `function new`, corresponde al constructor de la clase. La tarea `task body`, contiene la secuencia de operaciones a realizar y, en su caso, se ha declarado tipo `extern`, lo que indica que su definición está fuera de la clase.

La Figura 31 muestra el código del cuerpo `task body`, que es donde se describe el comportamiento de la secuencia. En primer lugar, se crea un objeto por cada una de las transacciones básicas que se vayan a utilizar. Posteriormente, y antes de ejecutar nada, se ejecuta la tarea `body` de la clase padre por si hubiera alguna acción en ella. A partir de este instante se ejecutan los diferentes pasos para completar la secuencia que se está definiendo.

```
// Group: Methods
//
// Task: body
//
// This is a standard UVM body task including the main sequence code where
// the sequences from sequene lib are executed.

// (begin inline source)
task axi4_sequence_lib_seq::body();

  incr_wr_t                   incr_wr                   = incr_wr_t::type_id::create("incr_wr");
  incr_rd_t                   incr_rd                   = incr_rd_t::type_id::create("incr_rd");
  wrap_rd_t                   wrap_rd                   = wrap_rd_t::type_id::create("wrap_rd");
  wrap_wr_t                   wrap_wr                   = wrap_wr_t::type_id::create("wrap_wr");
  excl_rw_t                   excl_rw                   = excl_rw_t::type_id::create("excl_rw");

  super.body();

  incr_rd.set_sequencer(m_sequencer);
  incr_rd.id = 2;
  if(!incr_rd.randomize() with {addr == 32'h2500_0000; rd_bytes == 16;})
    `uvm_error(this.get_full_name(),"Randomisation failure");

  incr_wr.set_sequencer(m_sequencer);
  incr_wr.id = 2;
  if(!incr_wr.randomize() with {addr == 32'h2500_0000; wr_data.size == 16;})
    `uvm_error(this.get_full_name(),"Randomisation failure");

  incr_wr.start(m_sequencer);
  incr_rd.start(m_sequencer);
```

Figura 31 Código UVM de la tarea body de una secuencia propia..

En el código que muestra la Figura 31 se detalla cómo enviar dos secuencias básicas, denominadas `incr_rd` e `incr_wr`. Estas secuencias corresponden a dos secuencias básicas de lectura y escritura incremental en el protocolo AXI4. En primer lugar, se asocia el *Sequencer* sobre el que se ejecutara la secuencia `incr_rd`, mediante la función `set_sequencer`. Posteriormente, se asigna el valor 2 como identificador, parámetro `id` del protocolo, de la secuencia. Finalmente se aleatoriza el resto de los campos, manteniendo las restricciones de los campos `addr` y `rd_bytes`. Estos mismos pasos se realizan para la secuencia básica `incr_wr`.

Por último, se debe utilizar la función `start`, que inicia la transacción. En este ejemplo en particular, este tipo heredado son tipos de transacciones de un nivel de abstracción superior, y que por sí solas implementan el `start_item` y `finish_item` del que se hizo mención anteriormente, liberando al ingeniero de hacerlo.

4.4. SECUENCIA BASE CON MÚLTIPLES QVIP

En todas las secuencias mostradas hasta ahora, solo se ha tenido un único agente en la configuración del *testbench*. En el caso del diseño que nos ocupa, existen múltiples agentes, por lo que la secuencia base debe incluir la definición de todos los manejadores asociados con cada una de los QVIP implicados en el *testbench*.

En el proceso de creación de todos los componentes UVM del *testbench*, la herramienta QVIP *configurator* crea una secuencia base para poder ejecutar sobre los agentes maestros. Esta secuencia base se deriva de la clase `mvc_sequence` y esta, a su vez,

de la clase `uvm_sequence`, que corresponde al nivel más bajo en la jerarquía UVM.

La Figura 32 muestra el código correspondiente a la secuencia base. La secuencia se ha definido con el nombre de `qvip_axi_xbar_vseg_base`. En primer lugar, se registra la secuencia en la *Factory* mediante el uso de la macro `uvm_object_utils()`. Esta secuencia base no ejecuta ninguna acción. Es decir, su tarea `body` no invoca ninguna de las funciones vistas en el apartado anterior. Sin embargo, esta secuencia base contiene un manejador por cada QVIP presente en el diseño. En este caso particular contiene 6 manejadores de QVIP maestros y 8 manejadores de QVIP esclavos. Eso permitirá a cualquier secuencia que derive de esta, poder acceder a cualquiera de los componentes *UVM Sequencer* para enviar datos a los componentes *UVM Driver* correspondientes, tal y como se verá en el siguiente capítulo.

```
1  class qvip_axi_xbar_vseq_base extends mvc_sequence;
2    `uvm_object_utils(qvip_axi_xbar_vseq_base)
3
4    mvc_sequencer axi4_master0;
5    mvc_sequencer axi4_master1;
6    mvc_sequencer axi4_master2;
7    mvc_sequencer axi4_master3;
8    mvc_sequencer axi4_master4;
9    mvc_sequencer axi4_master5;
10   mvc_sequencer axi4_slave0;
11   mvc_sequencer axi4_slave1;
12   mvc_sequencer axi4_slave2;
13   mvc_sequencer axi4_slave3;
14   mvc_sequencer axi4_slave4;
15   mvc_sequencer axi4_slave5;
16   mvc_sequencer axi4_slave6;
17   mvc_sequencer axi4_slave7;
18
19   function new ( string name = "qvip_axi_xbar_vseq_base" );
20     super.new(name);
21   endfunction
22
23   task body;
24
25   endtask: body
26
27 endclass: qvip_axi_xbar_vseq_base
```

Figura 32 Código secuencia base

4.5. INTERFAZ DE PROGRAMACION PARA LAS OPERACIONES DE LECTURA Y ESCRITURA

Una API o interfaz de programación de aplicaciones es un conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de las aplicaciones.

Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo. Las API otorgan flexibilidad y simplifican el diseño.

En el entorno de verificación UVM las secuencias se simplifican enormemente,

abstrayendo al ingeniero de complejos protocolos de comunicación, y simplificándolo en secuencias de lectura y escritura. Para este propósito el lenguaje dispone de distintas API que realizan estas tareas, y que se describen a continuación, mostrando los distintos niveles de abstracción de las mismas.

4.5.1. BASE READ Y BASE WRITE API

La categoría de secuencia base es una clase de secuencia (`axi4_master_deparam_seq`) que proporcionan un conjunto de API. Se pueden derivar secuencias maestras a partir de esta clase para ejecutar operaciones de lectura y escritura mediante secuencias específicas del protocolo AXI4. Las API de lectura y escritura se utilizan para realizar transacciones básicas de lectura y escritura y proporcionan una interfaz básica para seleccionar la transacción más adecuada y los atributos correspondientes.

Las API de lectura y escritura no dependen del protocolo y se pueden usar con cualquier protocolo. La clase `rw_api_base_seq` es la clase de secuencia base para la secuencia `axi4_master_deparam_seq` que implementa las API de lectura y escritura.

En la Figura 33 se muestran las API pertenecientes a esta clase.

API	Description
Abstract Write APIs	
write8	Writes a single byte data at a specified address
write16	Writes a single 16-bit half-word data at a specified address
write32	Writes a single 32-bit word data at a specified address
write64	Writes a single 64-bit long word data at a specified address
write	Writes a variable length array of data at a specified address

API	Description
Abstract Read APIs	
read8	Reads a single byte data from a specified address
read16	Reads a single 16-bit half-word data from a specified address
read32	Reads a single 32-bit word data from a specified address
read64	Reads a single 64-bit long word data from a specified address
read	Reads a specified number of bytes of data from the specified address

Figura 33 API de lectura y escritura definidas en la secuencia base [15].

Estas API están definidas en la clase `rw_api_base_seq` que se deriva de la clase `mvc_sequence`.

Las API de lectura/escritura implementan operaciones de lectura y escritura básicas como si fuera de un programa que se ejecuta en un procesador. Proporcionan una interfaz básica, lo que permite a la implementación de la secuencia de API elegir la transacción de bus más adecuada y los atributos correspondientes. Las API solo tienen un conjunto de argumentos simples: comando, dirección y datos, sin dependencia directa de atributos específicos del bus.

Los argumentos de entrada comunes en estas API son:

- `Cmd`: La operación a realizar, por ejemplo. *READ* o *WRITE*
- `Addr`: La dirección de inicio desde la que se realiza la lectura o escritura.
- `Data`: Bytes para leer o escribir

En la Figura 34 y la Figura 35 se muestran las interfaces de las APIs y sus atributos.

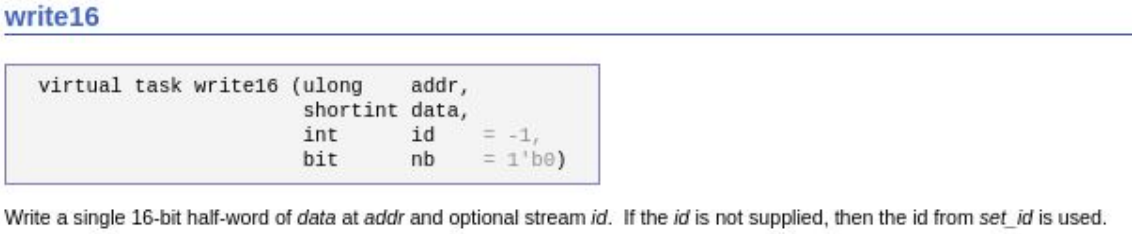


Figura 34 Interfaz API write16

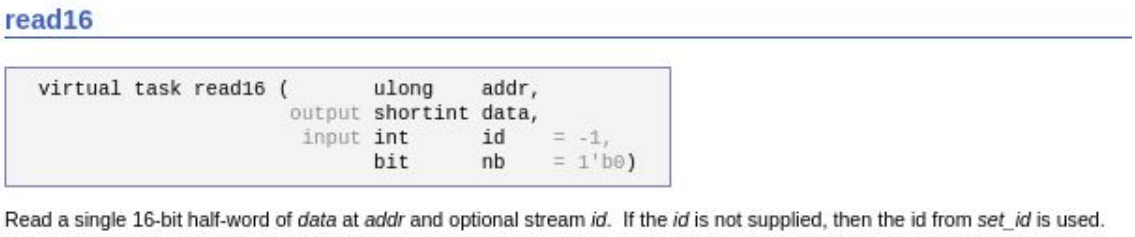


Figura 35 Interfaz de API read16

Los atributos utilizados en estas API de lectura y escritura se pueden fijar mediante el uso de las funciones `set`. Estos atributos de las API se listan en la Figura 36.

API	Description
Set APIs	
set_attr	Sets the collective values for the following attributes: • cache type • size • lock • burst type • burst length
set_cache_type	Sets the cache attribute of a read or write transaction
set_lock	Sets the lock attribute of a read or write transaction
set_burst_type	Sets the burst attribute of a read or write transaction
set_size	Sets the size attribute of a read or write transaction
set_burst_length	Sets the value of the length attribute to the maximum value allowed for read or write transactions
set_conc_txn	Allows the read/write APIs to run transactions concurrently to utilize the maximum bus bandwidth
set_addr_user_data	Sets the address user signal attribute of a read or write transaction
set_wdata_user_data	Sets the write data user signal attribute of a write transaction

Figura 36 Funciones set para la inicialización de atributos

Los diferentes atributos en que se centra este TFG son:

- `Set_lock (tipo, id)`
  - AXI4_NORMAL: Acceso normal del protocolo.
  - AXI4_EXCLUSIVE: Acceso en modo exclusivo.
- `Set_burst_type (tipo,id)`
  - AXI4_INCR: Modo de lectura o escritura en ráfagas, donde se incrementa una posición de memoria en cada acción.
  - AXI4_WRAP: Modo de lectura o escritura donde se incrementa una posición de memoria, pero dentro de un intervalo, volviendo a la posición inicial cuando se alcanza la última posición. Este comportamiento es propio de una cola circular de datos.
  - AXI4_FIXED: Modo de lectura o escritura en ráfagas siempre sobre la misma posición de memoria.

A continuación se muestra en la Figura 37 un ejemplo de configuración de una transacción, utilizando las API mencionadas.

```
// Exclusive transaction read/write 1 byte
// Set lock type as exclusive for id 2.
set_lock(AXI4_EXCLUSIVE, 2);
read8('h1000, rd_data_8, 2);
write8('h1000, wr_data_8, 2);

// Wrap transactions write/read 64-bit longin
// Set burst type as wrap for id 3.
set_burst_type(AXI4_WRAP, 3);
write64('h4000, wr_data_64, 3);
read64('h4000, rd_data_64, 3);

// Wrap transactions write/read data array
// Set burst type as wrap for id 3.
set_burst_type(AXI4_WRAP, 3);
write('h5000, wr_data, 3);
read('h5000, rd_data, 16, 3);

// Reset lock type as normal for id 2.
set_lock(AXI4_NORMAL, 2);
set_burst_type(AXI4_INCR, 3);
ndtask
/ (end inline source)
```

Figura 37 Ejemplo de configuración de una transacción mediante el uso de API [15]

En esta figura se muestra el código de distintos tipos de transacciones y la fijación de los atributos que se hacen en cada una de ellas. Este tipo de API contiene los campos de dirección de escritura, el dato a escribir y el número del identificador definido dentro del paréntesis. El dato a escribir ha sido definido como una variable que en el primer bloque tiene el nombre de `rd_data_8` y `wr_data_8`. El primer bloque son dos transacciones de lectura y escritura de 8 bits `read8` y `write8`, donde se ha fijado el atributo `LOCK` a un acceso exclusivo, `AXI_EXCLUSIVE` y con el identificador `ID=2`. Como se observa, primero se hace una escritura en la dirección ‘h1000 de 8 bits, y luego una escritura en la misma dirección. El segundo bloque son API que realizan la transacción de escritura y lectura de 64 bits. En este caso lo que se fija es el atributo `BURST_TYPE` a `AXI4_WRAP` y el identificador ID es fijado a 3. El tercer bloque son transacciones de escritura y lectura sin limitar la longitud de datos mediante una variable tipo array. Esta API realiza un acceso en modo `WRAP` hasta terminar con el *arrray* completo. Para finalizar, el último bloque ejecuta un reset de los atributos definidos `LOCK` y `BURST_TYPE`, fijándolos en los modos `AXI4_NORMAL` y `AXI4_INCR`.

4.5.2. STIMULI READ Y WRITE API

La categoría de transacciones STIMULI es una subcategoría más de las API del protocolo AXI4. Dentro de esta categoría se definen diferentes tipos de secuencias propias del protocolo AXI4. En la Figura 38 se describen las secuencias AXI4 disponibles en cada subcategoría dentro de la categoría de secuencia de `STIMULI`.

Sequence	Description
Atomic	Contains sequences that execute various exclusive read and write transactions Example: To execute two exclusive read transactions in parallel and then start a exclusive write transaction, use the *axi4_excl_parallel_rd_wr_deparam_seq* sequence.
Burst	Contains sequences that executes various burst read and write transactions Example: To execute fixed burst write transaction, use the *axi4_fixed_wr_deparam_seq* sequence.
Concurrent	Contains sequences that executes outstanding read and write transaction with same or different IDs Example: To execute outstanding read transactions with read data valid to ready delay with same ID, use the *axi4_rd_os_rvalid2rdy_dly_deparam_seq* sequence.
Delay	Contains sequences that execute read and write transaction with different delays. Example: To execute write transaction with addr2data delay and all other write delays, use the *axi4_wr_addr2data_all_dly_deparam_seq* sequence.

Figura 38 Tipos de secuencias en la categoría STIMULI

Entre los cuatro tipos definidos, este TFG se centra en las dos primeras.

Las API `Atomic` son las que se muestran en la Figura 39. Estas secuencias son aquellas que se ejecutan en modo `EXCLUSIVE` alternando operaciones de lectura y escritura. El acceso `EXCLUSIVE` pretende verificar la integridad de los datos leídos y escritos. Para ello se realiza una operación de lectura siempre antes de cada operación de escritura. Con este tipo de acceso se garantiza que el acceso a esta posición de datos no ha sido modificado por otro componente entre el proceso de lectura y escritura realizado, garantizando de esta manera la integridad de los datos.

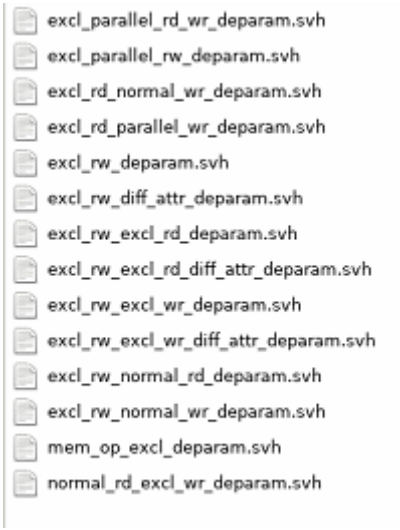


Figura 39 API STIMULI ATOMIC proporcionadas

Las API *burst* son las que se muestran en la Figura 40. Estas API están dirigidas a la realización de operaciones de lecturas y escrituras en ráfagas. Las ráfagas se generan proporcionando únicamente la dirección inicial y especificando la longitud de la misma. Dentro de esta subcategoría se recogen tres tipos de ráfagas generales, `INCR, FIZED` y `WRAP,` cuyos modos se definieron en el apartado anterior.

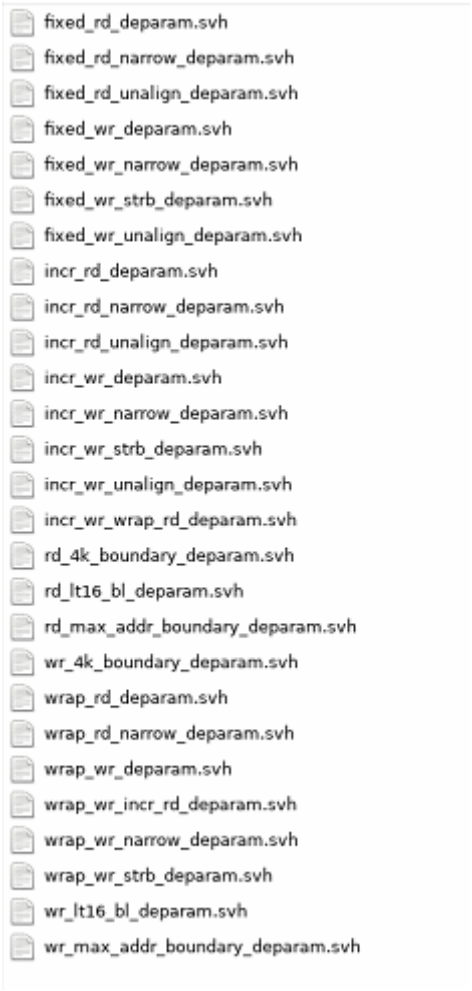


Figura 40 API STIMULI BURST proporcionadas

Todas están API están contenidas en el directorio de instalación y se importan en el momento de compilación, desde el directorio de instalación

```
$QUESTA_MVC_HOME/questa_mvc_src/sv/axi4/sequence_lib/
```

4.5.3. BUILT-IN READ Y WRITE API

Estas transacciones son el estímulo de nivel más bajo que una secuencia genera y envía al componente *UVM Driver*. El AXI4 QVIP proporciona varias transacciones básicas que se pueden enviar directamente para su ejecución. Como ya se comentó anteriormente, estas transacciones en el protocolo AXI4 se producen como transferencias de lectura y escritura. En la Figura 41 se enumeran algunas de las transacciones básicas utilizadas con mayor frecuencia

Sequence Item	Description
Master Sequence Items	
axi4_master_rw_transaction	Initiates a read or write transaction from an AXI4 master
axi4_master_read	Initiates a read transaction on the interface
axi4_master_write	Initiates a write transaction on the AXI4 interface
Slave Sequence Items	
axi4_slave_read_addr_channel_phase	Receives a phase level read When initiated, it receives an address phase from the read address channel
axi4_slave_write_data_burst	Receives the data burst of a write transaction
axi4_slave_write_resp_channel_phase	Receives a write response phase from the slave

Figura 41 API Built-in más utilizadas en AXI4

A diferencia de las API anteriores que se pueden iniciar directamente con la llamada a la API o con la función `START`, función que llevaba implícito las llamadas a los métodos `start_item` y `finish_item`, estas API al ser el nivel más bajo tienen que ser iniciadas llamando directamente a los métodos `start_item` y terminadas imperativamente con la llamada al método `finish_item`.

4.6. MODO DE EJECUCION EN PARALELO (`FORK-JOIN`)

La ejecución de SystemVerilog es una ejecución secuencial, pero de cara a la verificación puede interesar el carácter arbitrario que tiene una ejecución en paralelo para poder verificar DUV con interfaces independientes, pudiéndose dar situaciones inesperadas que aparecían en su ejecución secuenciada. De esta forma se podrá obtener resultados que no dependan del carácter secuencial en el que se ejecutan.

Para ello SystemVerilog proporciona el comando `fork-join`. Esta estructura permite

que todos los procesos declarados entre los tokens `fork-join`, se lancen en paralelo y se ejecuten en apariencia al mismo tiempo. No se continúa con la ejecución secuencial hasta que terminen todos los procesos invocados dentro de esta estructura. En la Figura 42 se muestra un ejemplo en el que se invocan una serie de procesos, *Thread1, Thread 2, … Thread 3* dentro de la estructura. Todos estos procesos se ejecutarán en paralelo.

```
fork
    // Thread 1
    // Thread 2
    // ...
    // Thread 3
join
```

Figura 42 Estructura fork-join de System Verilog

Si dentro de una estructura `fork-join` se incluye un conjunto de sentencias agrupadas dentro de un bloque `begin-end`, los comandos incluidos dentro de este bloque se ejecutarán en modo secuencial, y se consideran un proceso separado por sí mismo.

En el ejemplo de la Figura 43 se muestra la ejecución de 3 hilos simultáneos, donde el hilo 2 contiene dos comandos que se ejecutan de forma secuencial. La respuesta en cuanto al orden de ejecución es totalmente inesperada y puede darnos resultados distintos en cada ejecución dependiendo únicamente del manejo que haga el sistema operativo

```
module tb;
    initial begin
      $display ("[%0t] Main Thread: Fork join going to start", $time);
        fork
            // Thread 1
            #30 $display ("[%0t] Thread1 finished", $time);

            // Thread 2
            begin
                #5 $display ("[%0t] Thread2 ...", $time);
                #10 $display ("[%0t] Thread2 finished", $time);
            end

            // Thread 3
            #20 $display ("[%0t] Thread3 finished", $time);
        join
      $display ("[%0t] Main Thread: Fork join has finished", $time);
    end
endmodule
```

Figura 43 Comando fork-join con un bloque Begin-end

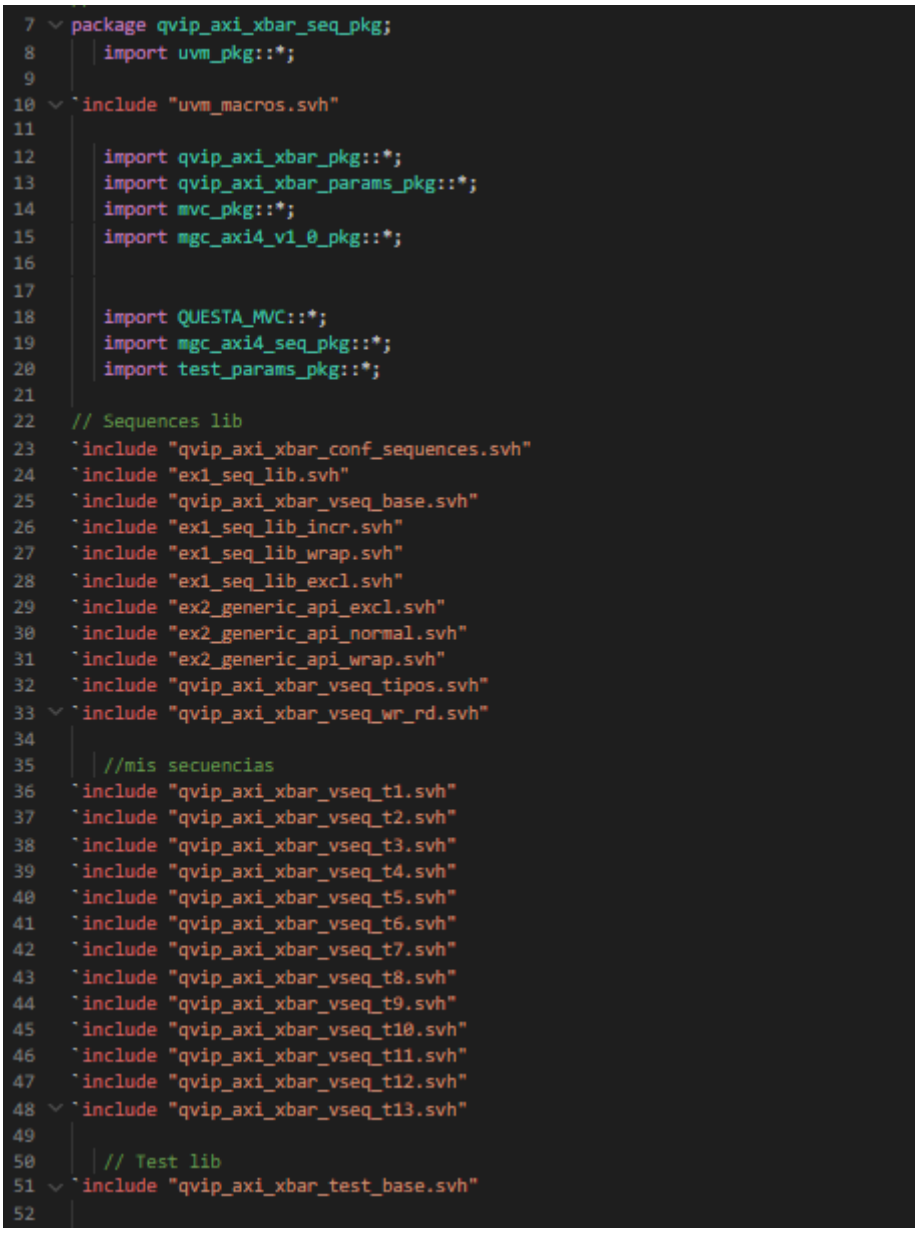
.

CAPÍTULO 5: EJECUCIÓN DEL PLAN DE VERIFICACIÓN

En este capítulo se ejecuta el plan de verificación y se comprueba que las características especificadas en el capítulo 3 se cumplen. Para poder comprobar cada una de las características se exponen las condiciones iniciales necesarias para poder generar el estado deseado, y a continuación se presentan las secuencias que estimulan el sistema para obtener las respuestas esperadas. La comprobación de los estímulos generados se realiza a través del software *Questa Sim de* Mentor Graphics*,* mediante la interfaz gráfica *Wave*, la cual se mostrará en cada uno de los apartados.

Hay que tener en cuenta las siguientes consideraciones para el funcionamiento de las secuencias que a continuación se exponen de forma genérica:

- El nombre de cada una de las secuencias se define en un archivo con extensión '*svh'*, denominado con el mismo nombre que la clase definida para esa secuencia.
- Todas las secuencias derivan de la secuencia base `qvip_axi_xbar_vseq_base.`
- Todas las secuencias tienen que ser registrada en el *Factory*.
- Antes de iniciar una secuencia, esta tiene que ser asociada al manejador del componente *UVM Agent* sobre el que se quiere estimular.
- Se recomienda crear las secuencias propias a partir de las secuencias base que usan las API para, de esa forma, poder crear muchos estímulos dentro de una misma secuencia.
- Todas las secuencias creadas se han referenciado dentro del archivo `qvip_axi_xbar_seq_pkg.sv`, tal y como se muestra en la Figura 44 a partir de la línea 35 en adelante.



```
package qvip_axi_xbar_seq_pkg;
  import uvm_pkg::*;

`include "uvm_macros.svh"

  import qvip_axi_xbar_pkg::*;
  import qvip_axi_xbar_params_pkg::*;
  import mvc_pkg::*;
  import mgc_axi4_v1_0_pkg::*;


  import QUESTA_MVC::*;
  import mgc_axi4_seq_pkg::*;
  import test_params_pkg::*;

// Sequences lib
`include "qvip_axi_xbar_conf_sequences.svh"
`include "ex1_seq_lib.svh"
`include "qvip_axi_xbar_vseq_base.svh"
`include "ex1_seq_lib_incr.svh"
`include "ex1_seq_lib_wrap.svh"
`include "ex1_seq_lib_excl.svh"
`include "ex2_generic_api_excl.svh"
`include "ex2_generic_api_normal.svh"
`include "ex2_generic_api_wrap.svh"
`include "qvip_axi_xbar_vseq_tipos.svh"
`include "qvip_axi_xbar_vseq_wr_rd.svh"

  //mis secuencias
`include "qvip_axi_xbar_vseq_t1.svh"
`include "qvip_axi_xbar_vseq_t2.svh"
`include "qvip_axi_xbar_vseq_t3.svh"
`include "qvip_axi_xbar_vseq_t4.svh"
`include "qvip_axi_xbar_vseq_t5.svh"
`include "qvip_axi_xbar_vseq_t6.svh"
`include "qvip_axi_xbar_vseq_t7.svh"
`include "qvip_axi_xbar_vseq_t8.svh"
`include "qvip_axi_xbar_vseq_t9.svh"
`include "qvip_axi_xbar_vseq_t10.svh"
`include "qvip_axi_xbar_vseq_t11.svh"
`include "qvip_axi_xbar_vseq_t12.svh"
`include "qvip_axi_xbar_vseq_t13.svh"

  // Test lib
`include "qvip_axi_xbar_test_base.svh"

```

Figura 44 Paquete qvip_axi_xbar_seq_pkg.sv: listado de secuencias

- Cuando una secuencia es parametrizable hay que definir los parámetros en el fichero `qvip_axi_xbar_seq_pkg.sv` y registrarla en la *Factory* de UVM, como un tipo parametrizable. Un ejemplo de cómo registrar una secuencia se muestra en la Figura 45.

```
52
53
54      typedef uvm_object_registry #( qvip_axi_xbar_vseq_t7
55                                     #(AXI4_ADDRESS_WIDTH,
56                                       AXI4_RDATA_WIDTH,
57                                       AXI4_WDATA_WIDTH,
58                                       AXI4_ID_WIDTH,
59                                       AXI4_USER_WIDTH,
60                                       AXI4_REGION_MAP_SIZE
61                                       ) ,
62                                     "qvip_axi_xbar_vseq_t7_p")
63      qvip_axi_xbar_vseq_t7_t;
64
```

Figura 45 Registro en la Factory de UVM de una secuencia parametrizable

La Figura 46 muestra la forma de declarar una secuencia parametrizable en el fichero donde se define la misma.

```
20  class qvip_axi_xbar_vseq_t7 #(int AXI4_ADDRESS_WIDTH   = 32,
21                                 int AXI4_RDATA_WIDTH     = 64,
22                                 int AXI4_WDATA_WIDTH     = 64,
23                                 int AXI4_ID_WIDTH        = 4,
24                                 int AXI4_USER_WIDTH      = 5,
25                                 int AXI4_REGION_MAP_SIZE = 16
26                                 ) extends qvip_axi_xbar_vseq_base;
27
28    typedef qvip_axi_xbar_vseq_t7                  #(AXI4_ADDRESS_WIDTH,
29                                                     AXI4_RDATA_WIDTH,
30                                                     AXI4_WDATA_WIDTH,
31                                                     AXI4_ID_WIDTH,
32                                                     AXI4_USER_WIDTH,
33                                                     AXI4_REGION_MAP_SIZE
34                                                     ) this_t;
35
```

Figura 46 Declaración de una secuencia parametrizable

- Las secuencias a ejecutar se especifican a nivel de línea de comando, desde el *script* `questa_run`. En este *script* se invoca el comando `vsim`, en cuya línea de parámetros se pueden especificar una o varias secuencias, tal y como se muestra en la Figura 47 y en la Figura 48.

```
65  vsim   -mvchome ${QUESTA_MVC_HOME} top_opt +nowarnTSCALE -t 1ns -do "do ./dofiles/wavemio.do; run -a;q"
66  +UVM_TESTNAME=qvip_axi_xbar_test_base +SEQ=qvip_axi_xbar_vseq_t13_p
67
```

Figura 47 Comando vsim de Questa *ejecutando una sola secuencia*

```
#vsim -c top_opt   -mvchome $QUESTA_MVC_HOME -do "run -a;q"
+SEQ=axi4_sequence_lib_seq +SEQ=ex2_master_phase +SEQ=ex3_backdoor_rd_wr +SEQ=axi4_generic_api_seq
+SEQ=axi4_generic_api_attr_seq +SEQ=axi4_generic_nb_api_seq +SEQ=ex7_simple_rd_wr    +UVM_TESTNAME=test

```

Figura 48 Comando vsim de Questa *ejecutando varias secuencias*

5.1. SECUENCIA 1

5.1.1. DEFINICIÓN

Cada módulo *slave* tiene una dirección asignada en el mapa de reglas de direcciones, denominado *rule map*. Si embargo, un módulo *slave* se puede ver afectado por varias reglas definidas dentro del *rule map*. Este hecho puede provocar que los rangos de direcciones se solapen. Si esto ocurriera, es decir que se solapen los rangos de direcciones, la regla que tenga la posición más alta en *el address map* prevalece.

5.1.2. MÉTODO

Cambiar las reglas de rango de direcciones para que haya solapamiento en dos módulos *slaves*. Escribir en una o varias de las direcciones solapadas.

5.1.3. RESPUESTA ESPERADA

Todas las escrituras deben unirse al mismo modulo *slave*. El módulo *slave* debe corresponder con el módulo que ocupa la posición más alta en el mapa de reglas de direcciones.

5.1.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t1.svh`

5.1.5. PROCEDIMIENTO

Lo primero que se va realiza es establecer las condiciones iniciales para poder comprobar que el DUV cumple con esta especificación.

En el módulo `axi_xbar_wrap.sv` es donde se definen las reglas de direccionamiento de los puertos *slaves*. La Figura 49 muestra el código donde se definen las 8 reglas correspondientes a los 8 módulos *slaves*, en orden decreciente desde el módulo `slave0` hasta el módulo `slave7`, con sus rangos de direcciones iniciales. En esta configuración inicial, no existe solapamientos entre ellos.

```
97      //Mapeado original
98      '{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0FFF_FFFF},
99      '{idx: 32'd1, start_addr: 32'h1000_0000, end_addr: 32'h1FFF_FFFF},
100     '{idx: 32'd2, start_addr: 32'h2000_0000, end_addr: 32'h2FFF_FFFF},
101     '{idx: 32'd3, start_addr: 32'h3000_0000, end_addr: 32'h3FFF_FFFF},
102     '{idx: 32'd4, start_addr: 32'h4000_0000, end_addr: 32'h4FFF_FFFF},
103     '{idx: 32'd5, start_addr: 32'h5000_0000, end_addr: 32'h5FFF_FFFF},
104     '{idx: 32'd6, start_addr: 32'h6000_0000, end_addr: 32'h6FFF_FFFF},
105     '{idx: 32'd7, start_addr: 32'h7000_0000, end_addr: 32'h7FFF_FFFF}
106   };
107
```

Figura 49 Definición original de las reglas del mapa de direcciones para la secuencia 1

Se realiza la modificación de la regla para solapar rangos, Figura 50, modificando las líneas 98 y 99.

```
//simulacion para secuencia t1
'{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0000_8000},
'{idx: 32'd1, start_addr: 32'h0000_4000, end_addr: 32'h1FFF_FFFF},  //direccion cambiada
```

Figura 50 Modificación de las reglas del mapa de direcciones para la secuencia 1

Lo que se ha hecho es configurar el módulo *slave0* para que responda desde la dirección `h'00000000` hasta la `h'00008000`, y que el módulo *slave1* responda desde la `h'00004000` hasta la `h'1FFFFFFF`. De esta forma se produce un solapamiento de los módulos *slave0* y *slave1*, siendo el más significativo el módulo *slave0*. Por este motivo la respuesta esperada es que la solicitud sea direccionada hacia el módulo *slave0*, como especifica en sus características.

5.1.6. SECUENCIA

La Figura 51 muestra la definición de la secuencia y los tipos que se van a utilizar para crear los estímulos. Para la realización de esta secuencia se va a utilizar las API a un nivel de abstracción alto, donde muchos de los parámetros ya vienen definidos en el apartado 4.5. En concreto se van a usar API de escritura de forma incremental.

```
//
// File: qvip_axi_xbar_vseq_wr_rd.svh
//
// Generated from Mentor VIP Configurator (20190214)
// Generated using Mentor VIP Library ( 2019_1.1 : 02/24/2019:07:17 )
//
class qvip_axi_xbar_vseq_t1 extends qvip_axi_xbar_vseq_base;
    `uvm_object_utils(qvip_axi_xbar_vseq_t1)

     typedef axi4_incr_wr_deparam_seq    incr_wr_t;
     typedef axi4_incr_rd_deparam_seq    incr_rd_t;
     typedef axi4_wrap_wr_deparam_seq    wrap_wr_t;
     typedef axi4_wrap_rd_deparam_seq    wrap_rd_t;
     typedef axi4_excl_rw_deparam_seq    excl_rw_t;

    function new
    (
        string name = "qvip_axi_xbar_vseq_t1"
    );
        super.new(name);
    endfunction

    extern task body;

endclass: qvip_axi_xbar_vseq_t1

```

Figura 51 Código UVM de la Secuencia 1

A continuación, se detalla el cuerpo de la secuencia, tarea `body`. En primer lugar, se crean las diferentes transacciones a utilizar en esta secuencia. En este caso se crean dos transacciones del tipo incremental por cada esclavo, una de lectura y otra de escritura, tal y como se muestra en la Figura 52 .

```
task qvip_axi_xbar_vseq_t1::body;

        incr_wr_t                incr_wr0                = incr_wr_t::type_id::create("incr_wr0");
        incr_rd_t                incr_rd0                = incr_rd_t::type_id::create("incr_rd0");

        incr_wr_t                incr_wr1                = incr_wr_t::type_id::create("incr_wr1");
        incr_rd_t                incr_rd1                = incr_rd_t::type_id::create("incr_rd1");

```

Figura 52 Creación de transacciones para la secuencia 1

A continuación, se describen las acciones de la tarea `body` para esta secuencia, Figura 53. Inicialmente se asigna el *secuencer* correspondiente a la transacción de escritura a enviar por el *master0*. Seguidamente, se asocia esta transacción con el identificador 0. Finalmente se aleatoriza los campos de la transacción, fijando el valor de la dirección `h'00004500` y el tamaño del dato (16 bytes en este caso). Este proceso se repite para la transacción de escritura correspondiente al *master1*. En este caso se utiliza la misma dirección y tamaño de dato, pero asociado al identificador 1.

Hay que destacar, que la dirección `h'00004500`, tal y como se muestra en la Figura

50, corresponde a una dirección solapada con los esclavos *slave0* y *slave1*, siendo el más significativo el *slave0*. Por último, y mediante el uso del mecanismo `fork-join`, se ejecutan ambas transacciones en paralelo.

```
begin
    //master0 write
            incr_wr0.set_sequencer(axi4_master0);
            incr_wr0.id = 0;
            if(!incr_wr0.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");
    //master1 write
            incr_wr1.set_sequencer(axi4_master1);
            incr_wr1.id = 1;
            if(!incr_wr1.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");

     fork
        incr_wr0.start(axi4_master0);
        incr_wr1.start(axi4_master1);


     join
end
```

Figura 53 Cuerpo de la tarea body correspondiente a la secuencia 1

5.1.7. RESPUESTA OBTENIDA

La Figura 54 muestra las transacciones enviadas por el *master0*, `incr_wr0` y por el *master1*, `incr_wr1`. Ambas transacciones se envían de forma concurrente.

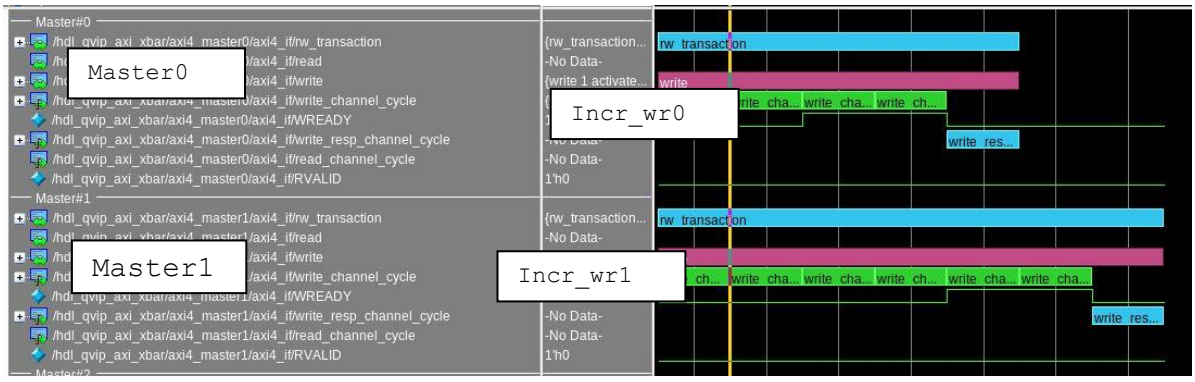


Figura 54 Transacciones realizadas secuencia 1

La Figura 55 muestra el interfaz del componente *slave0*. En esta figura se observa cómo ambas transacciones han sido enviadas al interfaz del módulo *slave0*, lo que coincide con el comportamiento esperado.

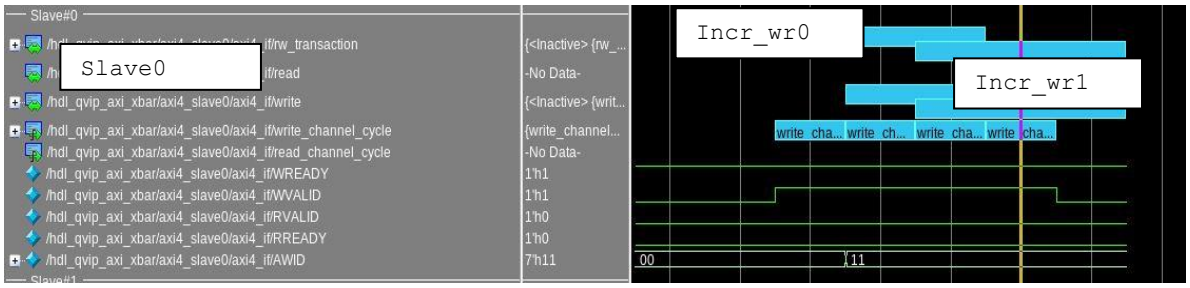


Figura 55 Respuesta obtenida secuencia 1

Con lo representado en este apartado se puede considerar verificada esta característica.

5.2. SECUENCIA 2

5.2.1. DEFINICIÓN

Esta secuencia es continuación de la secuencia 1 donde se ha complicado la asignación de direcciones interviniendo tres módulos *master*s, por lo que la definición es la misma que en el caso anterior.

Cada módulo slave tiene una dirección asignada en el mapa de reglas de direcciones rule map. Si embargo, un módulo slave se puede ver afectado por varias reglas definidas dentro del rule map. Este hecho puede provocar que los rangos de direcciones se solapen. Si esto ocurriera es decir que se solapen los rangos de direcciones, la regla que tenga la posición más alta en el *address map* prevalece.

5.2.2. MÉTODO

Cambiar el rango de direcciones para que coincidan tres módulos *slaves*. Posteriormente escribir en la misma dirección, desde tres módulos *masters* diferentes.

5.2.3. RESPUESTA ESPERADA

Cuando coincidan las direcciones, todas las escrituras en el mismo modulo *slave*, siendo este el módulo con el rango más significativo.

5.2.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t2.svh`

5.2.5. PROCEDIMIENTO

Lo primero que se realiza es establecer las condiciones iniciales para poder comprobar que el DUV cumple con esta especificación. Para ello se realiza la modificación de la regla para solapar las direcciones asignadas. En la Figura 56, se puede ver las modificaciones a realizar en las líneas 98, 99 y 100 del direccionamiento original.

```
//simulacion para secuencia t2
'{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0000_8000},
'{idx: 32'd1, start_addr: 32'h0000_4000, end_addr: 32'h1FFF_FFFF},  //direccion cambiada
'{idx: 32'd2, start_addr: 32'h0000_0000, end_addr: 32'h2FFF_FFFF},
```

Figura 56 Modificación del mapeado secuencia 2

Lo que se ha hecho es configurar el módulo *slave0* para que responda desde la dirección `h'00000000` hasta la `h'00008000`, que el módulo *slave1* responda desde la `h'00004000` hasta la `h'1FFFFFFF`, y que el módulo *slave2* responda desde la `h'00000000` hasta la `h'2FFFFFFF`. De esta forma se consigue que las direcciones de memoria asignadas se solapen entre los módulos *slave0, slave1 y el slave2*. El más significativo es el módulo *slave0* entre los tres asignados con lo que se espera que la respuesta sea dirigida hacia él. Por otro lado, entre el módulo *slave1* y el módulo *slave2*, el más significativo es el módulo *slave1*, luego el conflicto entre ambos se resolverá con la asignación del módulo *slave1*.

5.2.6. SECUENCIA

El procedimiento de declaración de las transacciones y los tipos definidos utilizados son los mismos que en el caso anterior, por lo que no se muestran de nuevo y se puede comprobar en los archivos en el Anexo II. En la Figura 57 se puede observar cómo en esta secuencia se han creado 5 estímulos, se ha escrito en las mismas direcciones de los módulos *slave*, y se ejecutan en dos bloques de secuencia temporal. Primero se va a actuar sobre las transacciones `incr_wro`, `incr_wr1` e `incr_w2` en modo concurrente. Se ha establecido que todas las transacciones apunten a la dirección `h'0000_4500`, y a cada una de ellas se le ha asignado un identificador distinto, identificador 0,1 y 2. Posteriormente, y una vez terminada estas transacciones, se actúa sobre las `transacciones incr_wr3` y `incr_wr4` también en modo concurrente, esta vez se le ha asignado la dirección `h'0000_8500` y los identificadores 3 y 4.

```
begin
    //master0 write
            incr_wr0.set_sequencer(axi4_master0);
            incr_wr0.id = 0;
            if(!incr_wr0.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");
    //master1 write
            incr_wr1.set_sequencer(axi4_master1);
            incr_wr1.id = 1;
            if(!incr_wr1.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");

    //master2 write
            incr_wr2.set_sequencer(axi4_master2);
            incr_wr2.id = 2;
            if(!incr_wr2.randomize() with {addr == 32'h0000_4500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");

    //master1 write
            incr_wr3.set_sequencer(axi4_master1);
            incr_wr3.id = 3;
            if(!incr_wr3.randomize() with {addr == 32'h0000_8500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");
    //master2 write
            incr_wr4.set_sequencer(axi4_master2);
            incr_wr4.id = 4;
            if(!incr_wr4.randomize() with {addr == 32'h0000_8500; wr_data.size == 16;})
              `uvm_error(this.get_full_name(),"Randomisation failure");
```

Figura 57 Código de la tarea body de la secuencia 2

Se muestra a continuación la secuencia de estímulos enviado en la Figura 58.

```
        fork
            incr_wr0.start(axi4_master0);
            incr_wr1.start(axi4_master1);
            incr_wr2.start(axi4_master2);
        join

        fork
            incr_wr3.start(axi4_master1);
            incr_wr4.start(axi4_master2);
        join
```

Figura 58 Código de la ejecución de las transacciones en la secuencia 2

5.2.7. RESPUESTA OBTENIDA

En la Figura 59 se puede observar las transacciones realizadas sobre cada uno de los módulos, y en el orden en el que fueron ejecutadas.

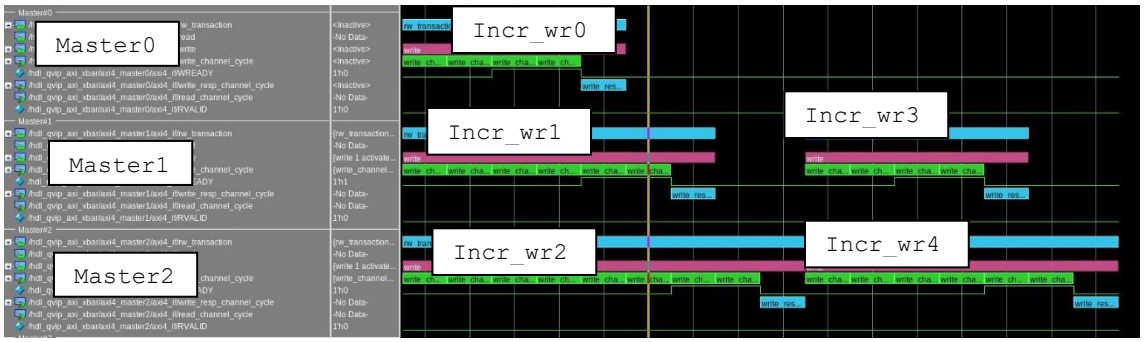


Figura 59 Transacciones realizadas secuencia 2

En la Figura 60 se puede observar cómo han sido escritas en las tres ocasiones, en el módulo *slave0*, y las dos siguientes transacciones en el módulo *slave1,* que es lo que se esperaba como respuesta.

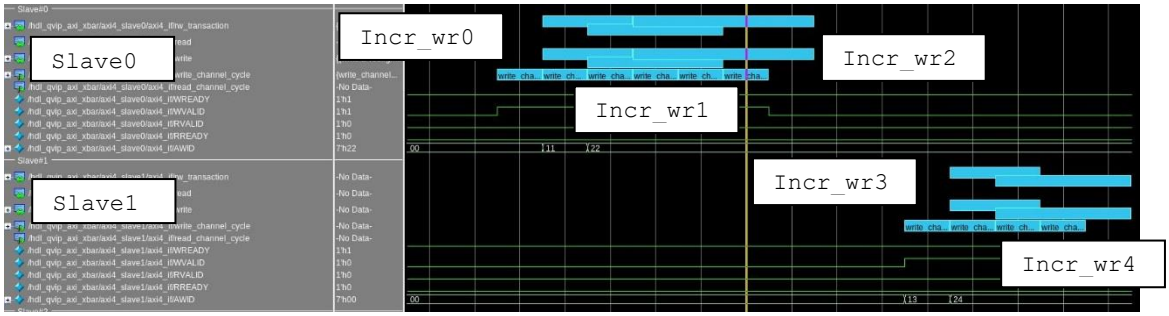


Figura 60 Respuesta obtenida secuencia 2

De esta forma se puede concluir que esta la característica queda verificada de forma satisfactoria.

5.3. SECUENCIA 3

5.3.1. DEFINICIÓN

Cada rango de direcciones *(range address)* de los módulos *slaves* incluye una dirección de comienzo, pero no incluye una dirección de fin. Por lo que cuando se direcciona sobre un módulo *slave* en modo ráfagas, esta dirección se dará por buena si y solo si la dirección es mayor que la dirección de comienzo y menor que la dirección de fin.

5.3.2. MÉTODO

Escribir más de un byte en el modo incremental en la posición del módulo *slave* que se desea asignar, en el límite de su rango de direcciones.

5.3.3. RESPUESTA ESPERADA

Como la condición a cumplir es que la dirección esté dentro de un rango de direcciones, y se intenta apuntar a un rango que se sale del permitido, la respuesta esperada es que responda con un mensaje de error.

5.3.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t3.svh`

5.3.5. PROCEDIMIENTO

Lo primero que se realiza es establecer las condiciones iniciales para poder comprobar que el DUV cumple con esta especificación, para ello se modifica la regla para solapar rangos, Figura 61, cambiando las líneas 98 y 99 de la asignación original.

```
//simulacion para secuencia t3
'{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0000_8000},
'{idx: 32'd1, start_addr: 32'h0000_8001, end_addr: 32'h1FFF_FFFF},  //direccion cambiada
```

Figura 61 Modificación del mapeado, secuencia 3

Lo que se ha hecho es configurar el módulo *slave0* para que responda desde la dirección `h'00000000` hasta la `h'00008000`, y que el módulo *slave1* responda desde la `h'00008001` hasta la `h'1FFFFFFF`.

5.3.6. SECUENCIA

El procedimiento de declaración de las transacciones y los tipos definidos utilizados son los mismos que en el caso anterior, por lo que no se muestran de nuevo y se puede comprobar en los archivos en el Anexo II.

En la Figura 62 se muestra el código de la tarea *body* de la secuencia a verificar. Se ha procedido a configurar una transacción en modo incremental, con el nombre `incr_wr0`. Lo primero que se ha realizado es asignar el manejador coincidente con el *master0*, a la secuencia definida. Posteriormente se le ha asignado el identificador 3 y, como último paso, se ha inicializado en modo aleatorio con la restricción de la dirección de memoria h'`0000_8000` y una cantidad de bytes de escritura de 16 bytes.

Se puede observar, como en la ejecución se ha hecho una escritura en el límite del rango de direcciones asignadas al módulo *slave0*, con lo que, al escribir en el límite de direcciones, y en modo incremental, se saldrá de su rango de direcciones asignado y debería responder con una respuesta de error.

```
27  task qvip_axi_xbar_vseq_t3::body;
28
29      incr_wr_t                   incr_wr0                = incr_wr_t::type_id::create("incr_wr0");
30      incr_rd_t                   incr_rd0                = incr_rd_t::type_id::create("incr_rd0");
31
32
33
34    super.body();
35
36
37      begin
38        //master0 write
39              incr_wr0.set_sequencer(axi4_master0);
40              incr_wr0.id = 3;
41              if(!incr_wr0.randomize() with {addr == 32'h0000_8000; wr_data.size == 16;})
42                `uvm_error(this.get_full_name(),"Randomisation failure");
43
44
45          fork
46              incr_wr0.start(axi4_master0);
47
48          join
49
50      end
51  endtask: body
```

Figura 62 Código de la tarea body de la secuencia 3

5.3.7. RESPUESTA OBTENIDA

En la Figura 63 y Figura 64 se puede observar las transacciones realizadas y la respuesta obtenida en el módulo *master0*.

Figura 63 Transacción realizada secuencia 3.

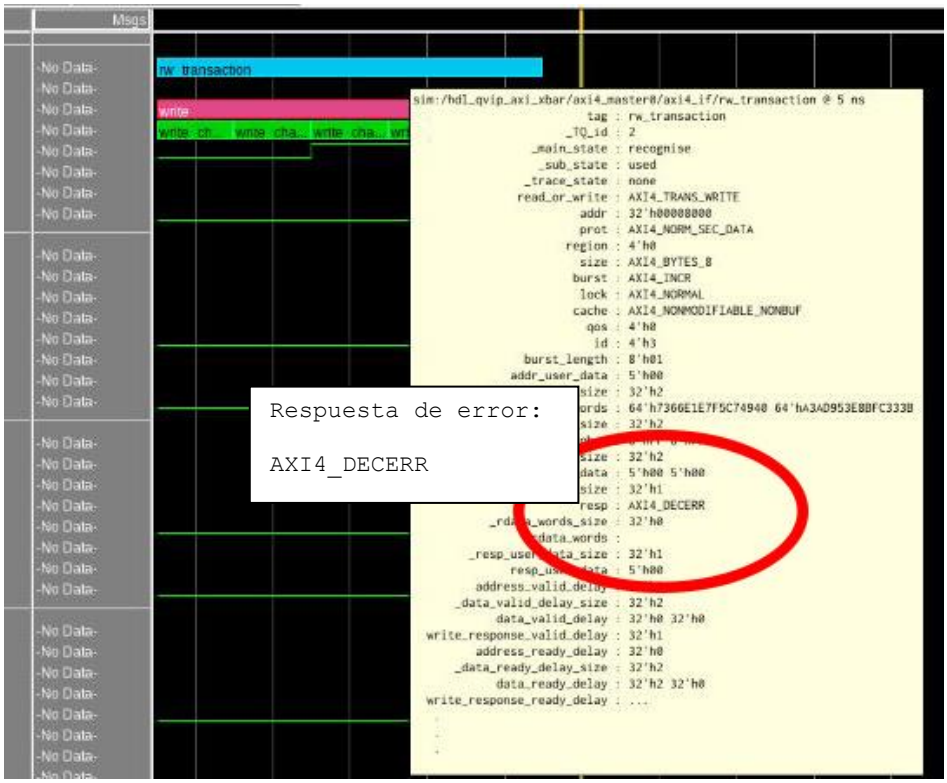


Figura 64 Código de error de respuesta secuencia 3

Podemos concluir que esta característica se ha cumplido, y con lo presentado se puede considerar verificada esta característica.

5.4. SECUENCIA 4

5.4.1. DEFINICIÓN

Cada módulo *master* tiene su propio módulo de error interno. Si la dirección de respuesta asignada de la transacción no pertenece a ninguna regla de asignación de direcciones de los módulos *slaves*, entonces se le asigna la dirección del módulo de error interno asignada al módulo *master*. El módulo *master* recoge la transacción y responde con un `decode error`.

5.4.2. MÉTODO

Se ha procedido a modificar los campos de direcciones de las reglas de asignación de los módulos *slaves* para dejar una dirección fuera de rango de direcciones, y escribir en esa dirección para observar la respuesta.

5.4.3. RESPUESTA ESPERADA

La respuesta esperada es un `decode error`.

5.4.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t4.svh`

5.4.5. PROCEDIMIENTO

Lo primero que se va a realizar es establecer las condiciones iniciales para poder comprobar que el DUV cumple con esta especificación.

En el archivo denominado `axi_xbar_wrap.sv` se definen las reglas de direcciones de los puertos *slaves* como se muestra en la Figura 65. En esta figura se pueden observar los 8 módulos *slaves,* ordenados en orden decreciente desde el módulo *slave0* hasta el módulo *slave7*, con sus rangos de direcciones iniciales sin solapamientos entre ellos.

```
97      //Mapeado original
98      '{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0FFF_FFFF},
99      '{idx: 32'd1, start_addr: 32'h1000_0000, end_addr: 32'h1FFF_FFFF},
100     '{idx: 32'd2, start_addr: 32'h2000_0000, end_addr: 32'h2FFF_FFFF},
101     '{idx: 32'd3, start_addr: 32'h3000_0000, end_addr: 32'h3FFF_FFFF},
102     '{idx: 32'd4, start_addr: 32'h4000_0000, end_addr: 32'h4FFF_FFFF},
103     '{idx: 32'd5, start_addr: 32'h5000_0000, end_addr: 32'h5FFF_FFFF},
104     '{idx: 32'd6, start_addr: 32'h6000_0000, end_addr: 32'h6FFF_FFFF},
105     '{idx: 32'd7, start_addr: 32'h7000_0000, end_addr: 32'h7FFF_FFFF}
106   };
107
```

Figura 65 Rango de direcciones original

En la Figura 66 se puede observar la modificación de la regla de direcciones para solapar los rangos, modificando las líneas 98 y 99 de la Figura 65.

```
//simulacion para secuencia t4 y t5
'{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0000_7000},
'{idx: 32'd1, start_addr: 32'h0000_8001, end_addr: 32'h1FFF_FFFF},  //direccion cambiada
```

Figura 66 Modificación de las direcciones en la secuencia 4

Lo que se ha configurado es el módulo *slave0* para que responda desde la dirección `h'00000000` hasta la `h'00007000`. Posteriormente se configura el módulo *slave1* para que responda desde la dirección `h'00008001` hasta la `h'1FFFFFFF`. De esta forma las direcciones incluidas en el rango `h'00007001` hasta `h'00008000` están fuera de cualquier intervalo, es decir no corresponde con ninguna dirección de salida de módulos *slave*.

En las líneas 149 y 150 de la Figura 67 se muestran las señales que definen el uso del módulo interno, `en_default_mst_port_i` y `default_mst_port_i`. Estas señales son las que permiten habilitar esta característica. En concreto, la señal `en_default_`*`master`*`_port_i` es la que habilita el modo *enable* de estos módulos internos, configurando su uso. Cada bit de esta señal corresponde a un módulo *master*, empezando por la derecha con el módulo *master*0. Un bit a 0, deshabilita el módulo interno de error y un 1 lo habilita. La señal `default_mst_port_i` es la que define hacia qué módulo *slave* se va a direccionar el módulo *master* por defecto. Los valores que puede tomar se representan en bloques de tres bits, correspondientes a la codificación de cada uno de los módulos *master* de derecha a izquierda, siendo en este caso los tres bits de la derecha los correspondientes al módulo *master0* y los siguientes al módulo *master1*, etc. Estos bits representan el número del módulo *slave* al que apuntan por defecto. Como se puede observar la señal en `default_mst_port_i` está a 0 en todos sus bits por lo que el modo *enable* de los módulos internos no está habilitado en ningún módulo *master*.

```
144     //parametros originales
145     //.en_default_mst_port_i ( en_default_mst_port ),
146     // .default_mst_port_i    ( default_mst_port )
147
148     //parametros para t4
149     .en_default_mst_port_i ( 6'b000000 ),
150     .default_mst_port_i    ( 18'b000_000_000_000_000_010)
151   );
152
```

Figura 67 Parámetros de configuración de las señales de módulo interno de error en el fichero `axi_xbar_wrap.sv`

5.4.6. SECUENCIA

En la Figura 68 se muestra el código de la definición de la secuencia. Se definen los tipos de transacciones a utilizar y la tarea `body` que se va a utilizar para crear los estímulos.

```
7  class qvip_axi_xbar_vseq_t4 extends qvip_axi_xbar_vseq_base;
8      `uvm_object_utils(qvip_axi_xbar_vseq_t4)
9
10     typedef axi4_incr_wr_deparam_seq    incr_wr_t;
11     typedef axi4_incr_rd_deparam_seq    incr_rd_t;
12     typedef axi4_wrap_wr_deparam_seq    wrap_wr_t;
13     typedef axi4_wrap_rd_deparam_seq    wrap_rd_t;
14     typedef axi4_excl_rw_deparam_seq    excl_rw_t;
15
16     function new
17     (
18         string name = "qvip_axi_xbar_vseq_t4"
19     );
20         super.new(name);
21     endfunction
22
23     extern task body;
24
25 endclass: qvip_axi_xbar_vseq_t4
26
27 task qvip_axi_xbar_vseq_t4::body;
28
29     incr_wr_t                   incr_wr0                    = incr_wr_t::type_id::create("incr_wr0");
30     incr_rd_t                   incr_rd0                    = incr_rd_t::type_id::create("incr_rd0");
31
32
33   super.body();
34
35
36     begin
37       //master0 write
38             incr_wr0.set_sequencer(axi4_master0);
39             incr_wr0.id = 3;
40             if(!incr_wr0.randomize() with {addr == 32'h0000_7050; wr_data.size == 16;})
41               `uvm_error(this.get_full_name(),"Randomisation failure");
42
43
44        fork
45           incr_wr0.start(axi4_master0);
46
47     end
48 endtask: body
```

Figura 68 Código de la secuencia 4

En la tarea `body` se define la transacción denominada `incr_wr0`. En primer lugar, y como es habitual en todas las transacciones, se le asigna el manejador del *master* que

va a ejecutarla, `axi4_master0` en este caso. Posteriormente se le asigna el identificador 3 y se inicializa en modo `randomize`, restringiendo la dirección al valor `h'00007500`, que no se corresponde con ninguna dirección asignada en el rango de direcciones. En este caso en concreto, como no se ha habilitado la respuesta por defecto del módulo interno de error del módulo *master0,* lo que se tiene que obtener es un código de error.

5.4.7. RESPUESTA OBTENIDA

En la Figura 69 se puede observar la transacción realizada, y la respuesta obtenida.

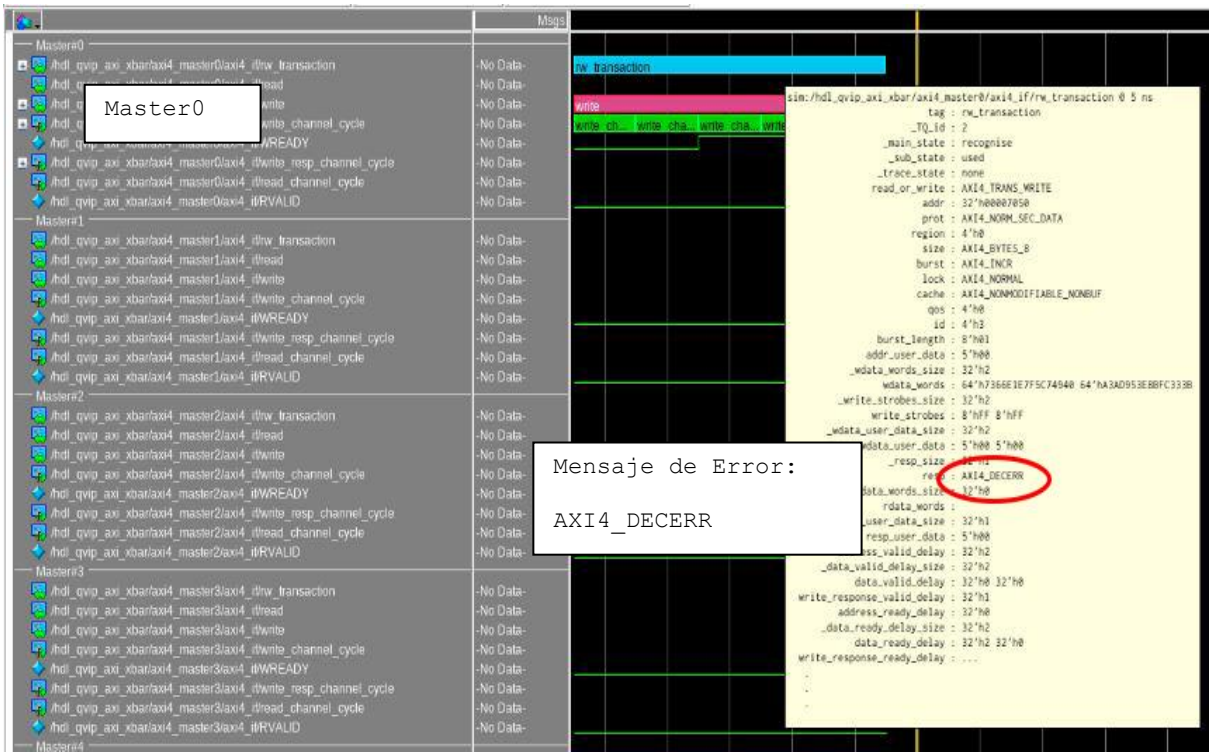


Figura 69 Respuesta obtenida en la secuencia 4

Como se observa, esta respuesta contiene un código de error, que es lo que se esperaba.

5.5. SECUENCIA 5

5.5.1. DEFINICIÓN

Cada módulo *master* puede tener un puerto modulo *slave* por defecto. Si la señal `default_master_port` está habilitada, cada acceso a una dirección que se envíe y no tenga una dirección de módulo *slave* correspondiente dentro del mapa de direcciones, entonces será enviada a este módulo *slave* de defecto, y no al de la dirección real, no incluida dentro del mapa de direcciones. Este módulo *default slave* es configurable.

5.5.2. MÉTODO

Establecer la señal del *enable* activo para el módulo correspondiente y asignar un módulo *slave* por defecto para los módulos *master*s. Posteriormente escribir en una de esas direcciones fuera del rango de direcciones, asignadas en las reglas de direccionamiento, '*rules'*, y observar la salida.

5.5.3. RESPUESTA ESPERADA

La respuesta que se espera es que no devuelva un mensaje de error y que se direccione el módulo *slave* correspondiente codificado en la señal *master_port.*

5.5.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t5.svh`

5.5.5. PROCEDIMIENTO

Lo primero que se va a realizar es establecer las condiciones iniciales necesarias para poder comprobar que el DUV cumple con esta especificación.

En el archivo denominado `axi_xbar_wrap.sv` se definen las reglas de direcciones de los puertos *slaves* como se muestra en la Figura 70. Se pueden observar los 8 módulos *slaves,* ordenados en orden decreciente desde el módulo *slave0* hasta el módulo *slave7*, con sus rangos de direcciones iniciales sin solapamientos entre ellos.

```
97      //Mapeado original
98      '{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0FFF_FFFF},
99      '{idx: 32'd1, start_addr: 32'h1000_0000, end_addr: 32'h1FFF_FFFF},
100     '{idx: 32'd2, start_addr: 32'h2000_0000, end_addr: 32'h2FFF_FFFF},
101     '{idx: 32'd3, start_addr: 32'h3000_0000, end_addr: 32'h3FFF_FFFF},
102     '{idx: 32'd4, start_addr: 32'h4000_0000, end_addr: 32'h4FFF_FFFF},
103     '{idx: 32'd5, start_addr: 32'h5000_0000, end_addr: 32'h5FFF_FFFF},
104     '{idx: 32'd6, start_addr: 32'h6000_0000, end_addr: 32'h6FFF_FFFF},
105     '{idx: 32'd7, start_addr: 32'h7000_0000, end_addr: 32'h7FFF_FFFF}
106   };
107
```

Figura 70 Código del fichero `axi_xbar_`wrap.sv del rango de direccionamiento original

En la Figura 71, se muestra la modificación de la regla para solapar los rangos de direcciones, modificando las líneas 98 y 99 de la figura anterior.

```
//simulacion para secuencia t4 y t5
'{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0000_7000},
'{idx: 32'd1, start_addr: 32'h0000_8001, end_addr: 32'h1FFF_FFFF},  //direccion cambiada
```

Figura 71 Modificación del rango de direccionamiento de la secuencia 5

Se ha configurado el módulo *slave0* para que responda desde la dirección `h'00000000` hasta la `h'00007000`, y que el módulo *`slave`*`1` responda desde la dirección `h'00008001` hasta la `h'1FFFFFFF`, de esta forma se configura el rango de direcciones entre `h'00007001` hasta `h'00008000` fuera de rango de direcciones de los módulos *slaves,* es decir no corresponde con ninguna dirección de salida de los módulos *slave.*

En la Figura 72, se muestran los valores de las señales que permiten definir la dirección automática a un módulo *slave* predeterminado, como ya se explicó en la anterior secuencia. En este caso en concreto, en la línea 149 se puede ver la señal `en_default_mst_port` con el bit del módulo *master0* en modo *enable*. La línea 150 muestra la señal `default_msat_port_i` con la codificación 'b010 (*slave2*) para el módulo *master0.* Estas señales están definidas dentro del archivo denominado `axi_xbar_wrap.sv`.

```
144     //parametros originales
145     //.en_default_mst_port_i ( en_default_mst_port ),
146     // .default_mst_port_i    ( default_mst_port )
147
148     //parametros para t5
149     .en_default_mst_port_i ( 6'b000001 ),
150     .default_mst_port_i    ( 18'b000_000_000_000_000_010)
151   );
152
```

Figura 72 Definición de señales en el archivo denominado `axi_xbar_wrap.sv`

5.5.6. SECUENCIA

En la Figura 73 se muestra el código de la definición de la secuencia. Inicialmente se declaran los tipos básicos definidos de las transacciones que se van a utilizar para crear los estímulos. Seguidamente se define la tarea *body* de la secuencia.

```
7   class qvip_axi_xbar_vseq_t5 extends qvip_axi_xbar_vseq_base;
8     `uvm_object_utils(qvip_axi_xbar_vseq_t5)
9
10    typedef axi4_incr_wr_deparam_seq    incr_wr_t;
11    typedef axi4_incr_rd_deparam_seq    incr_rd_t;
12    typedef axi4_wrap_wr_deparam_seq    wrap_wr_t;
13    typedef axi4_wrap_rd_deparam_seq    wrap_rd_t;
14    typedef axi4_excl_rw_deparam_seq    excl_rw_t;
15
16    function new
17    (
18      string name = "qvip_axi_xbar_vseq_t5"
19    );
20      super.new(name);
21    endfunction
22
23    extern task body;
24
25  endclass: qvip_axi_xbar_vseq_t5
26
27  task qvip_axi_xbar_vseq_t5::body;
28
29      incr_wr_t                 incr_wr0                  = incr_wr_t::type_id::create("incr_wr0");
30      incr_rd_t                 incr_rd0                  = incr_rd_t::type_id::create("incr_rd0");
31
32
33   super.body();
34
35
36     begin
37       //master0 write
38           incr_wr0.set_sequencer(axi4_master0);
39           incr_wr0.id = 3;
40           if(!incr_wr0.randomize() with {addr == 32'h0000_7050; wr_data.size == 16;})
41             `uvm_error(this.get_full_name(),"Randomisation failure");
42
43
44       fork
45         incr_wr0.start(axi4_master0);
46
47       join
48
49
50     end
51  endtask: body
```

Figura 73 Código de la secuencia 5

Como se puede observar, el proceso de configuración de las transacciones es igual a

las anteriores secuencias definidas en los apartados anteriores. En este caso en concreto solo se particulariza en escribir en la dirección `h'00007500`, la cual no se corresponde con ninguna dirección asignada en el mapa de direcciones. En este caso particular, como se ha habilitado la respuesta del módulo *master0*, con la señal *enable* a 1 y se ha configurado la salida por defecto hacia el módulo *slave2*, se tendrá que obtener una salida hacia ese puerto por defecto.

5.5.7. RESPUESTA OBTENIDA

En la Figura 74 se observa la transacción realizada, y cómo en este caso la respuesta obtenida durante la operación de escritura no es un mensaje de error.

Figura 74 Transacción realizada en la secuencia 5

La Figura 75 muestra el interfaz de entrada correspondiente al módulo salve2. Su análisis muestra que la transacción se ha escrito correctamente en el módulo *slave2,* tal y como se esperaba. Además, este interfaz genera una respuesta con un código de respuesta correcta, y no en un código de error. Con esto se concluye que el sistema de direccionamiento por defecto funciona correctamente.

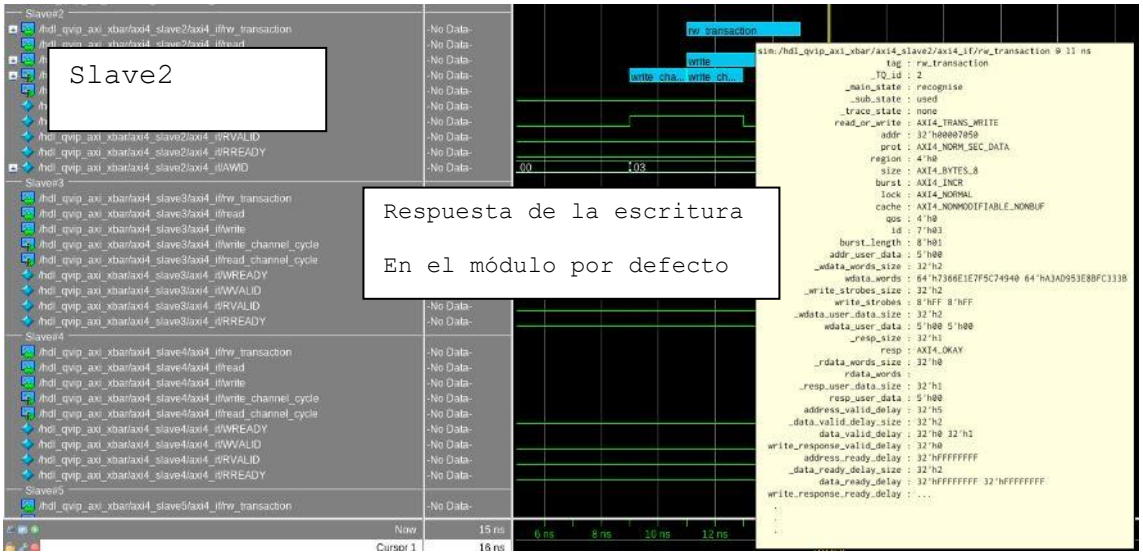


Figura 75 Respuesta de la secuencia 5

5.6. SECUENCIA 6

5.6.1. DEFINICIÓN

Cuando un módulo *master* recibe dos transacciones con el mismo identificador *ID* y misma dirección, pero apuntando desde dos módulos *slaves* diferentes, no aceptará la segunda transacción hasta que la primera se haya completado.

5.6.2. MÉTODO

Escribir dos transacciones desde el mismo módulo *master* con el mismo identificador y dirección, hacia dos módulos *slave* diferentes.

5.6.3. RESPUESTA ESPERADA

La respuesta esperada es que las transacciones se ejecutarán de forma consecutiva, no al mismo tiempo.

5.6.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t6.svh`

5.6.5. PROCEDIMIENTO

En primer lugar, se establecerán las condiciones iniciales para poder comprobar que el DUV responde de la forma esperada.

En el archivo denominado `axi_xbar_wrap.sv` se definen las reglas de direcciones de los puertos *slaves* como se muestra en la Figura 76. Se pueden observar los 8 módulos *slaves,* ordenados en orden decreciente desde el módulo *slave0* hasta el módulo *slave7*, con sus rangos de direcciones iniciales sin solapamientos entre ellos. Como se muestra el módulo *slave0* está definido en la línea 98, y el módulo *slave1* en la línea 99.

```
97      //Mapeado original
98      '{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0FFF_FFFF},
99      '{idx: 32'd1, start_addr: 32'h1000_0000, end_addr: 32'h1FFF_FFFF},
100     '{idx: 32'd2, start_addr: 32'h2000_0000, end_addr: 32'h2FFF_FFFF},
101     '{idx: 32'd3, start_addr: 32'h3000_0000, end_addr: 32'h3FFF_FFFF},
102     '{idx: 32'd4, start_addr: 32'h4000_0000, end_addr: 32'h4FFF_FFFF},
103     '{idx: 32'd5, start_addr: 32'h5000_0000, end_addr: 32'h5FFF_FFFF},
104     '{idx: 32'd6, start_addr: 32'h6000_0000, end_addr: 32'h6FFF_FFFF},
105     '{idx: 32'd7, start_addr: 32'h7000_0000, end_addr: 32'h7FFF_FFFF}
106   };
107
```

Figura 76 Direccionamiento original

5.6.6. SECUENCIA

En la Figura 77 se muestra el código de la definición de la secuencia y los tipos de transacciones que se van a utilizar para crear los estímulos.

```
class qvip_axi_xbar_vseq_t6 extends qvip_axi_xbar_vseq_base;
    `uvm_object_utils(qvip_axi_xbar_vseq_t6)

     typedef axi4_incr_wr_deparam_seq    incr_wr_t;
     typedef axi4_incr_rd_deparam_seq    incr_rd_t;
     typedef axi4_wrap_wr_deparam_seq    wrap_wr_t;
     typedef axi4_wrap_rd_deparam_seq    wrap_rd_t;
     typedef axi4_excl_rw_deparam_seq    excl_rw_t;

    function new
    (
        string name = "qvip_axi_xbar_vseq_t6"
    );
        super.new(name);
    endfunction

    extern task body;

endclass: qvip_axi_xbar_vseq_t6

task qvip_axi_xbar_vseq_t6::body;

      incr_wr_t                 incr_wr0                = incr_wr_t::type_id::create("incr_wr0");
      incr_rd_t                 incr_rd0                = incr_rd_t::type_id::create("incr_rd0");

      incr_wr_t                 incr_wr1                = incr_wr_t::type_id::create("incr_wr1");
      incr_rd_t                 incr_rd1                = incr_rd_t::type_id::create("incr_rd1");

  super.body();

    begin
      //master0 write
            incr_wr0.set_sequencer(axi4_master0);
            incr_wr0.id = 1;
            if(!incr_wr0.randomize() with {addr == 32'h0000_3500;})
              `uvm_error(this.get_full_name(),"Randomisation failure");
      //master1 write
            incr_wr1.set_sequencer(axi4_master0);
            incr_wr1.id = 1;
            if(!incr_wr1.randomize() with {addr == 32'h1000_3500;})
              `uvm_error(this.get_full_name(),"Randomisation failure");

        fork
          incr_wr0.start(axi4_master0);
          incr_wr1.start(axi4_master0);
        join
    end
endtask: body
```

Figura 77 Código de la secuencia 6

Para la realización de esta secuencia se van a definir dos transacciones `inc_wr0` e `incr_wr1`. Las dos transacciones son asignadas al mismo manejador `axi4_master0`, se les asigna el mismo identificador, `incr_wr.id=1` y distinta dirección de escritura en el puerto de salida, dirección `h'0000_3500` y la dirección `h'1000_3500`, respectivamente.

Inicialmente se escribe en la dirección `h'00003500` con el ID=1 desde el módulo *master0*, y en la dirección `h'1000_3500` con el ID=1 también desde el módulo *master0*.

5.6.7. RESPUESTA OBTENIDA

En las Figura 78 y Figura 79 se observa cómo se inician las dos transacciones en orden. En la Figura 79 se muestra cómo primero se ejecuta la transacción `inc_wr1` y, una vez finalizada, se ejecuta la transacción `incr_wr0`. Es decir, no se interrumpe una transacción para mandar la otra, sino que se acaba la transacción en curso antes de empezar con la siguiente.

En la Figura 79 se puede observar el final de la primera transacción, determinado por la respuesta a través del canal de respuesta del protocolo AXI4.

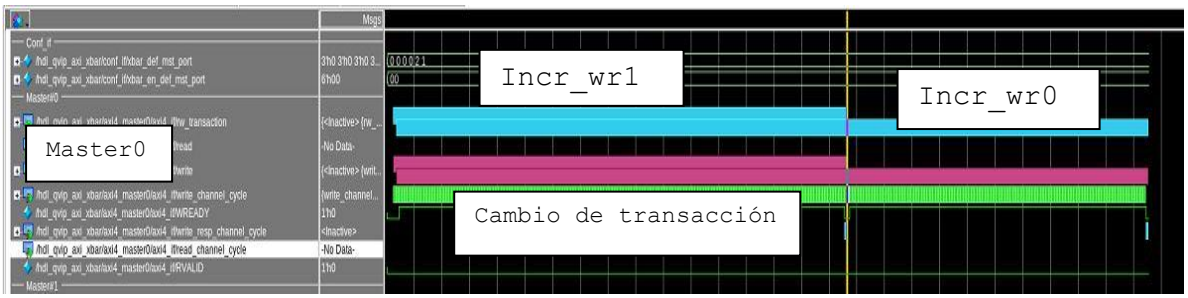


Figura 78 Transacción generada en la secuencia 6

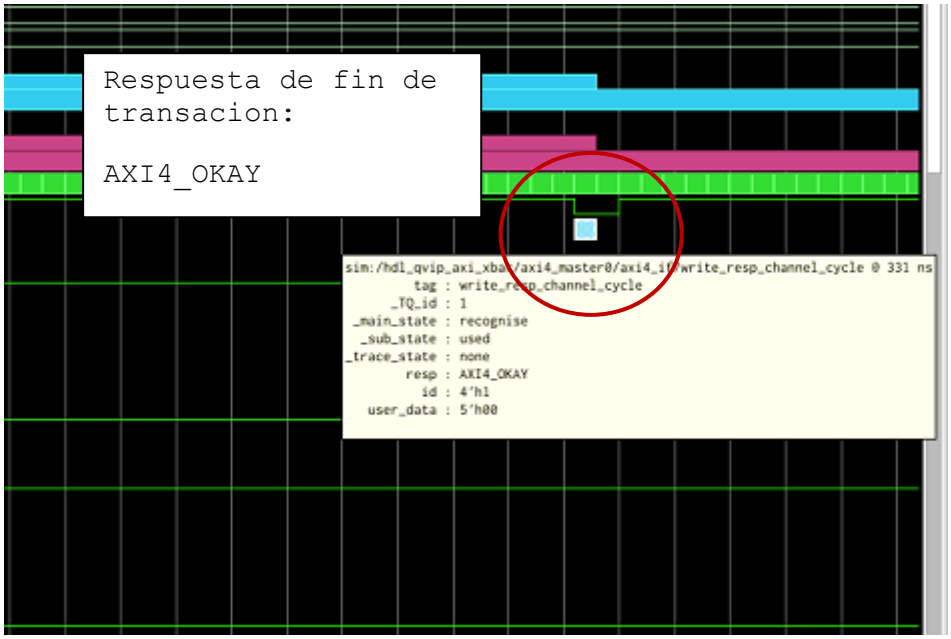


Figura 79 Respuesta al acabar la escritura de la primera transacción

A continuación, en la Figura 80 se muestra la respuesta obtenida. Como se observa, al ser dos módulos *slaves* distintos aunque vengan del mismo módulo *master* y con el mismo ID, este escribe primero una transacción completa en el módulo *slave1*, y

posteriormente hace la escritura en el módulo *slave0*, sin intercalar dichas escrituras. Con lo que la característica 6 se ha cumplido correctamente.

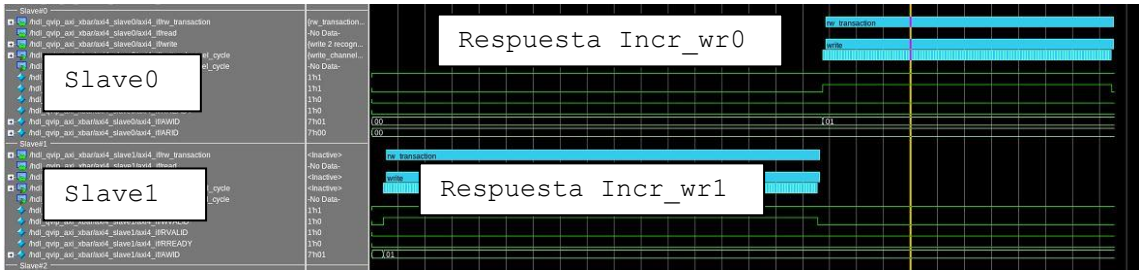


Figura 80 Respuesta a la secuencia 6

5.7. SECUENCIA 7

En este tipo de secuencia a diferencia de las anteriores secuencias, se va a disminuir un nivel de abstracción en la forma de generar las secuencias. Hasta ahora las API utilizadas y definidas permitían generar transacciones completas sin poder manejar directamente los protocolos de comunicación correspondientes a los diferentes canales de lectura o de escritura utilizados en el protocolo AXI4. En este caso, se van a utilizar funciones que permiten controlar por separado las escrituras por canales del protocolo AXI4. De esta forma se permite la generación de situaciones imposibles de establecer cuando se utilizan las API genéricas del protocolo AXI4.

5.7.1. DEFINICIÓN

Todas las órdenes de escritura de datos tienen que llegar en el mismo orden que todas las órdenes de escritura de direcciones, independientemente del identificador del módulo *master*. El multiplexor de diferentes puertos módulos *slave*, excluye cada orden de escritura de datos *write data*, porque el orden viene dado por el arbitraje del canal de escritura de dirección de la orden *write addr*.

5.7.2. MÉTODO

Se mandan varias órdenes de direcciones desde distintos módulos *masters* al mismo módulo *slave*. El envío de estas órdenes se hará en modo concurrente con el comando `fork-join`. De esta forma, no se provocará una ejecución secuencial y ordenada, incluso con dos escrituras desde el mismo módulo *master* con dos identificadores diferentes. Las órdenes iniciarán dos ciclos distintos de escrituras de direcciones y de datos independientes.

5.7.3. RESPUESTA ESPERADA

La respuesta esperada es que se escriban en orden de llegada de las escrituras de dirección, y sin comunicar una respuesta de error. Es decir que el propio DUV sea capaz de administrar las llegadas de ciclos de direcciones y datos y asignadas en orden correcto.

5.7.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t7.svh`

5.7.5. PROCEDIMIENTO

En la Figura 81 se muestra cómo en este tipo de secuencia, que es parametrizable, se tiene que definir como se describió al comienzo de este capítulo. El nombre asignado a esta secuencia es `qvip_axi_xbar_vseq_t7_p`. En este caso en particular, durante la declaración de la secuencia se especifican los parámetros: ancho del campo de direcciones, ancho de los buses de datos, ancho del campo identificador, ancho del campo usuario y, finalmente, tamaño del *region_map*.

```
typedef uvm_object_registry #( qvip_axi_xbar_vseq_t7
                               #(AXI4_ADDRESS_WIDTH,
                                 AXI4_RDATA_WIDTH,
                                 AXI4_WDATA_WIDTH,
                                 AXI4_ID_WIDTH,
                                 AXI4_USER_WIDTH,
                                 AXI4_REGION_MAP_SIZE
                                 ) ,
                               "qvip_axi_xbar_vseq_t7_p")
qvip_axi_xbar_vseq_t7_t;
```

Figura 81 Definición de la secuencia 7 parametrizable

La Figura 82, muestra la secuencia que se ejecutará desde el script `Questa_run` que, tal y como se muestra será la secuencia parametrizable `qvip_axi_xbar_vseq_t7_p`.

```
vsim   -mvchome ${QUESTA_MVC_HOME} top_opt +nowarnTSCALE -t 1ns -do "do ./dofiles/wavemio.do; run -a;q"
+UVM_TESTNAME=qvip_axi_xbar_test_base +SEQ=qvip_axi_xbar_vseq_t7_p
```

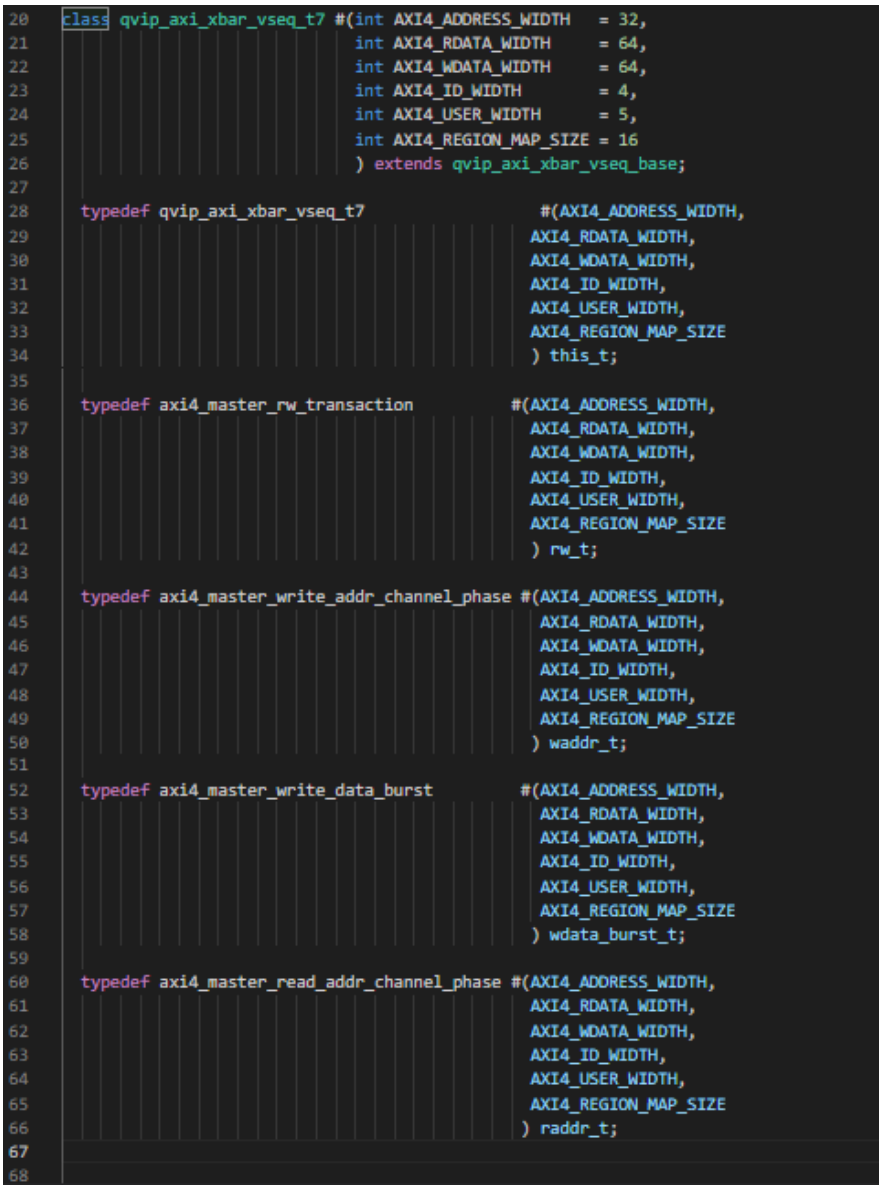
Figura 82 Ejecución del script Questa_run *para la secuencia 7*

Lo que se ha creado dentro de esta secuencia son transacciones de escritura de direcciones, y transacciones de escritura de datos de forma independiente. De esta forma se separan los canales que existen en el protocolo AXI4. Se quiere comprobar qué ocurre si se manda primero una escritura en el canal de datos sin existir una escritura en un canal de direcciones o, incluso, varias escrituras de direcciones sin que se hagan escrituras de datos. Al utilizar la estructura `fork-join`, estas operaciones se hacen de forma concurrente, lo que permite la ejecución en cualquier orden. El DUV ha de ser capaz de administrar las llegadas, ya sean de escritura de direcciones o de datos, y direccionarlas correctamente.

5.7.6. SECUENCIA

La Figura 83 muestra la primera parte del código de definición de la secuencia que ha

creado. Este fragmento del código muestra los tipos de transacciones que intervienen en esta secuencia. A diferencia de las anteriores, se observa como todos los tipos son parametrizables. Los tipos tratados en esta secuencia particular son `axi4_master_write_channel_phase (waddr_t)`, que se corresponde con el canal de la escritura de una dirección, y `axi4_master_write_data_burst (wdata_burst_t)`, que se corresponde con el canal de escritura de datos en modo ráfagas.



```
20  class qvip_axi_xbar_vseq_t7 #(int AXI4_ADDRESS_WIDTH   = 32,
21                                 int AXI4_RDATA_WIDTH     = 64,
22                                 int AXI4_WDATA_WIDTH     = 64,
23                                 int AXI4_ID_WIDTH        = 4,
24                                 int AXI4_USER_WIDTH      = 5,
25                                 int AXI4_REGION_MAP_SIZE = 16
26                                 ) extends qvip_axi_xbar_vseq_base;
27
28    typedef qvip_axi_xbar_vseq_t7                  #(AXI4_ADDRESS_WIDTH,
29                                                     AXI4_RDATA_WIDTH,
30                                                     AXI4_WDATA_WIDTH,
31                                                     AXI4_ID_WIDTH,
32                                                     AXI4_USER_WIDTH,
33                                                     AXI4_REGION_MAP_SIZE
34                                                     ) this_t;
35
36    typedef axi4_master_rw_transaction            #(AXI4_ADDRESS_WIDTH,
37                                                     AXI4_RDATA_WIDTH,
38                                                     AXI4_WDATA_WIDTH,
39                                                     AXI4_ID_WIDTH,
40                                                     AXI4_USER_WIDTH,
41                                                     AXI4_REGION_MAP_SIZE
42                                                     ) rw_t;
43
44    typedef axi4_master_write_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
45                                                     AXI4_RDATA_WIDTH,
46                                                     AXI4_WDATA_WIDTH,
47                                                     AXI4_ID_WIDTH,
48                                                     AXI4_USER_WIDTH,
49                                                     AXI4_REGION_MAP_SIZE
50                                                     ) waddr_t;
51
52    typedef axi4_master_write_data_burst          #(AXI4_ADDRESS_WIDTH,
53                                                     AXI4_RDATA_WIDTH,
54                                                     AXI4_WDATA_WIDTH,
55                                                     AXI4_ID_WIDTH,
56                                                     AXI4_USER_WIDTH,
57                                                     AXI4_REGION_MAP_SIZE
58                                                     ) wdata_burst_t;
59
60    typedef axi4_master_read_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
61                                                     AXI4_RDATA_WIDTH,
62                                                     AXI4_WDATA_WIDTH,
63                                                     AXI4_ID_WIDTH,
64                                                     AXI4_USER_WIDTH,
65                                                     AXI4_REGION_MAP_SIZE
66                                                     ) raddr_t;
67
68
```

Figura 83 Secuencia 7 tipos definidos

La Figura 84 muestra el cuerpo principal de la tarea *body*. Lo primero que se ejecuta es esperar que se realice la función *reset* y la función de configuración del reloj *clock* inicial,

antes de ejecutar el código de la transacción en sí. Estas funciones se obtienen de la configuración del entorno donde se ejecuta la secuencia. En este caso, y al ser un nivel de abstracción inferior a las anteriores secuencias, es labor del ingeniero el control de estos eventos para poder garantizar su éxito. Entre las líneas 106 a 114 se envían las diferentes transacciones programadas. De igual modo, una vez enviadas las transacciones, a partir de la línea 116 en adelante, lo que se realiza es esperar que se termine las transacciones para poder observar correctamente los resultados en la interfaz gráfica `Wave`. Este es un mecanismo muy usado para poder observar correctamente las respuestas obtenidas.

Para enviar las transacciones dentro de la estructura `fork-join`, se ha utilizado un bucle `for`, línea 108 hasta la línea 112, que contiene dos funciones, `execute_wr_addr_data()` y `execute_rd_addr()`. El número de repeticiones del bucle se declara en la línea 98 en la variable `num_txn`, lo que permite diseñar un código reutilizable y configurable a múltiples transacciones. En este caso, donde lo que se pretende es enviar una sola repetición, se configura esta variable a 1, con lo que se ejecuta una única operación de escritura y otra de lectura. La función de lectura se tiene que hacer para dar tiempo al DUV de enviar a la salida los datos recibidos en las operaciones de escritura y, de esta forma, poder visualizarlos en la interfaz gráfica `Wave`.

```
96  task qvip_axi_xbar_vseq_t7::body();
97
98    int num_txn = 1;
99    // Get the config object so that you can wait for a clock after reset
100   // to issue the first sequence_item.
101   cfg = axi4_master0_cfg_t::get_config(axi4_master0);
102
103   cfg.wait_for_reset();
104   cfg.wait_for_clock();
105
106   fork
107     begin
108       for(int i=0;i<num_txn;++i)
109       begin
110         execute_wr_addr_data();
111         execute_rd_addr();
112       end
113     end
114   join
115
116    // Just to ensure that all transactions complete
117    // before the sequence ends
118       repeat(num_txn*2)
119         begin
120           rw_t rw_txn = rw_t::type_id::create("rw_txn");
121           rw_txn.m_receive_id = get_stream_id(this);
122           rw_txn.receive(cfg);
123         end
124
125 endtask
```

Figura 84 Tarea body de la secuencia 7

A continuación, Figura 85, se muestra las declaraciones de los tipos de transacciones y las asignaciones de cada módulo *master* de las transacciones que se van a usar. Las líneas 171, 172, 173 y 174, muestran las transacciones de los ciclos de escrituras de direcciones, y las líneas 176, 177, 178 y 179, las transacciones para los ciclos de escrituras de datos. Las asignaciones de los componentes *UVM Agent utilizados* son: `axi4_master0, axi4_master1 y axi4_master2.` El agente `axi4_master0` se asigna a dos transacciones diferentes, tanto para las direcciones como para los datos.

```
159  task qvip_axi_xbar_vseq_t7::execute_wr_addr_data();
160
161    waddr_t       waddr0 = waddr_t::type_id::create("waddr0");
162    waddr_t       waddr01 = waddr_t::type_id::create("waddr01");
163    waddr_t       waddr1 = waddr_t::type_id::create("waddr1");
164    waddr_t       waddr2 = waddr_t::type_id::create("waddr2");
165
166    wdata_burst_t wdata0 = wdata_burst_t::type_id::create("wdata0");
167    wdata_burst_t wdata01 = wdata_burst_t::type_id::create("wdata01");
168    wdata_burst_t wdata1 = wdata_burst_t::type_id::create("wdata1");
169    wdata_burst_t wdata2 = wdata_burst_t::type_id::create("wdata2");
170
171    waddr0.set_sequencer(axi4_master0);
172    waddr01.set_sequencer(axi4_master0);
173    waddr1.set_sequencer(axi4_master1);
174    waddr2.set_sequencer(axi4_master2);
175
176    wdata0.set_sequencer(axi4_master0);
177    wdata01.set_sequencer(axi4_master0);
178    wdata1.set_sequencer(axi4_master1);
179    wdata2.set_sequencer(axi4_master2);
```

Figura 85 Inicio de transacciones en la secuencia 7

La Figura 86 muestra el código de ejecución de las transacciones. En primer lugar se tiene que invocar la función `start_item()` de cada secuencia a enviar. Posteriormente, se inicializan los parámetros de las transacciones. Esta inicialización se hace de forma aleatoria usando la función `randomize`, con las restricciones que interesan para esta secuencia. En concreto, en este caso se ha restringido el tipo `LOCK (AXI4_NORMAL);` la dirección de escritura, `addr`, es decir el módulo *slave* al que direccionamos; la dimensión de la ráfaga, `burst_length`, para no generar una secuencia muy larga y poderla observar mejor; y además, se ha configurado el identificador de los módulos *master.*

Lo que se ha hecho es generar cuatro escrituras, dos desde el módulo *master0*, una desde el módulo *master1*, y otra desde el módulo *master2,* todas apuntando hacia el

módulo *slave0*, pero apuntando a distintas posiciones, `h'1000`, `h'1500` y `h'2000` dos veces. Una vez inicializada la transacción se termina con la función `finish_item()`.

Cada proceso de configuración de una transacción se inicia y se termina dentro de su bloque `begin` y `end` localizado dentro del bloque `fork-join` con lo que no siguen un orden secuencial en su ejecución, sino que se realizan de forma concurrente.

```
fork
 begin
// ****************************************
// secuencia del master 0 escritura de direccion
  start_item( waddr0 );
  `uvm_info("wr_addr0", "start_item", UVM_LOW)
    if(!waddr0.randomize() with {waddr0.lock == AXI4_NORMAL;
                                 waddr0.addr == 32'h0000_1000;
                                 waddr0.burst_length <= 4;
                                 waddr0.id == 3;
                                 }) `uvm_error("axi4_master_phase_seq","Write address randomization failure")
  finish_item(waddr0);
  `uvm_info("wr_addr0", "finish_item", UVM_LOW)
// ****************************************
 end

 begin
// ****************************************
// secuencia del master 01 escritura de direccion
  start_item( waddr01 );
  `uvm_info("wr_addr01", "start_item", UVM_LOW)
    if(!waddr01.randomize() with {waddr01.lock == AXI4_NORMAL;
                                 waddr01.addr == 32'h0000_1500;
                                 waddr01.burst_length <= 20;
                                 waddr01.id == 2;
                                 }) `uvm_error("axi4_master_phase_seq","Write address randomization failure")
  finish_item(waddr01);
  `uvm_info("wr_addr01", "finish_item", UVM_LOW)
// ****************************************
 end
```

Figura 86 Código de ejecución de las transacciones de dirección para la secuencia 7

La Figura 87 muestra las transacciones de datos y los parámetros que se configuran en ellas. Estos han sido configurados utilizando el mismo método que en las configuraciones de transacciones de escritura de direcciones. Los parámetros restringidos son: el parámetro `burst`, que se asigna al mismo valor utilizado en su correspondiente transacción de escritura de dirección y el parámetro `strobe` de validación de los datos. En este último parámetro se han puesto todos los bits a 1 para validar todos los bytes presentes en el bus de datos.

Al igual que con las transacciones de dirección, las transacciones de datos se ejecutan dentro de un bloque `fork-join`, con lo que la transacción del ciclo de escritura de datos puede ocurrir antes que la de la escritura de su correspondiente ciclo de dirección. Si eso ocurriera y no se hubiera generado un ciclo de canal de escritura de dirección, el DUV no tendría una dirección donde escribir y por lo tanto tendría que responder con un error, que es lo que se trata de verificar. En el caso que esto ocurra, el DUV es capaz

de ponerlo en espera y gestionar la llegada de los canales correspondientes para que tenga el orden necesario dentro del protocolo de transmisión.

```
257   begin
258   // ****************************************
259   // escritura de datos del master 01
260
261     start_item( wdata01 );
262     `uvm_info("wr_data01", "start_item", UVM_LOW)
263       if(!wdata01.randomize() with {wdata01.burst_length  == waddr01.burst_length;
264                                     foreach(wdata01.write_strobes[i]) wdata01.write_strobes[i] == '1;
265
266                                     }) `uvm_error("axi4_master_phase_seq", "Write data randomization failure")
267     finish_item( wdata01 );
268     `uvm_info("wr_data01", "finish_item", UVM_LOW)
269   // ****************************************
270   end
271
272   begin
273   // ****************************************
274   // escritura de datos del master 1
275
276     start_item( wdata1 );
277     `uvm_info("wr_data1", "start_item", UVM_LOW)
278       if(!wdata1.randomize() with {wdata1.burst_length  == waddr1.burst_length;
279                                     foreach(wdata1.write_strobes[i]) wdata1.write_strobes[i] == '1;
280
281                                     //foreach ( wdata1.write_data_beats_delay[i] ){wdata1.write_data_beats_delay[i] == 5;}
282                                   }) `uvm_error("axi4_master_phase_seq", "Write data randomization failure")
283     finish_item( wdata1 );
284     `uvm_info("wr_data1", "finish_item", UVM_LOW)
285   // ****************************************
286   end
```

Figura 87 Código de las transacciones de datos para la secuencia 7.

5.7.7. RESPUESTA OBTENIDA

La Figura 88 muestra la secuencia generada y las respuestas obtenidas en los ciclos de escritura. Como se puede observar todas las transacciones se han realizado. Este hecho se ha resaltado dibujando un círculo en el ciclo de respuesta de cada transacción enviada. No se ha perdido ninguna transacción. En la figura se observa tanto las transacciones realizadas desde distintos módulos *masters* e, incluso, las dos transacciones enviadas desde el mismo módulo *master0*.

Figura 88 Secuencia 7 generada y respuestas obtenidas.

La Figura 89 muestra el detalle de la información para una de las transacciones. En este caso, como se puede ver, corresponde a una de las transacciones enviadas por el módulo *master0*, donde se muestra la dirección, el `ID`, y la respuesta `AXI4_OKAY`, indicando que no ha habido error.

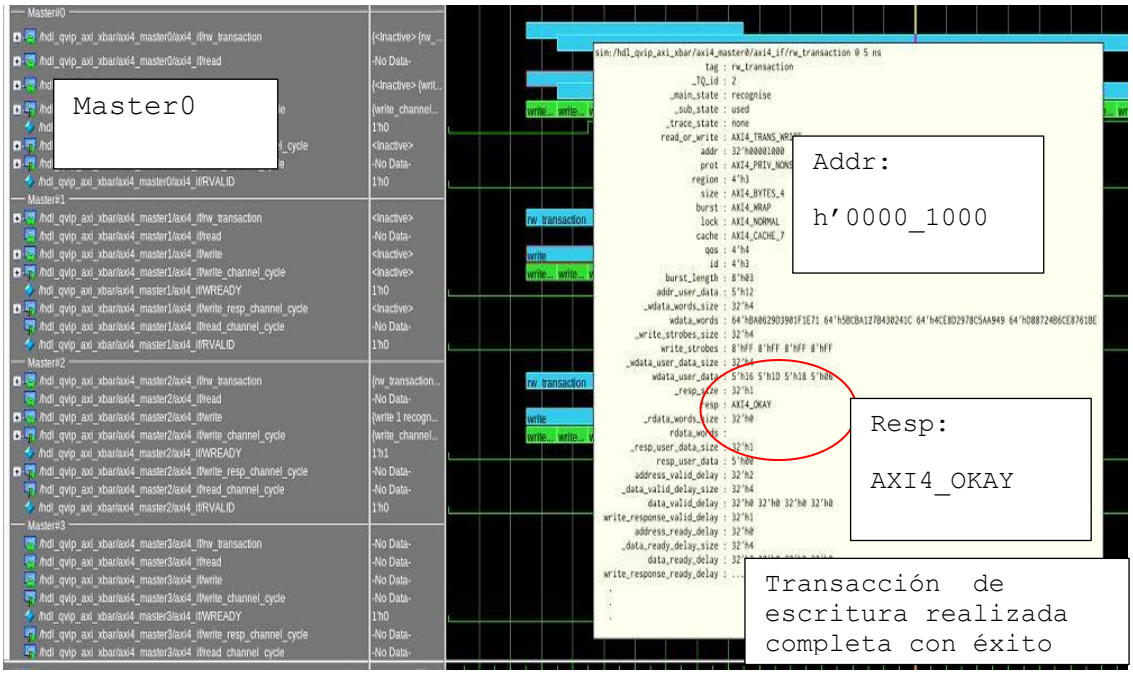


Figura 89 Detalle de una transacción de escritura dentro de la Secuencia 7.

La Figura 90 muestra las respuestas obtenidas en el módulo *Slave0*. En esta figura se detalla la respuesta del canal `AXI4_OKAY`, para la última transacción recibida. En la misma figura y, señalado con el rotulo `rw_transantion` también se puede observar las cuatro transacciones realizadas y completadas, representadas por los cuatro intervalos de color azul representados.

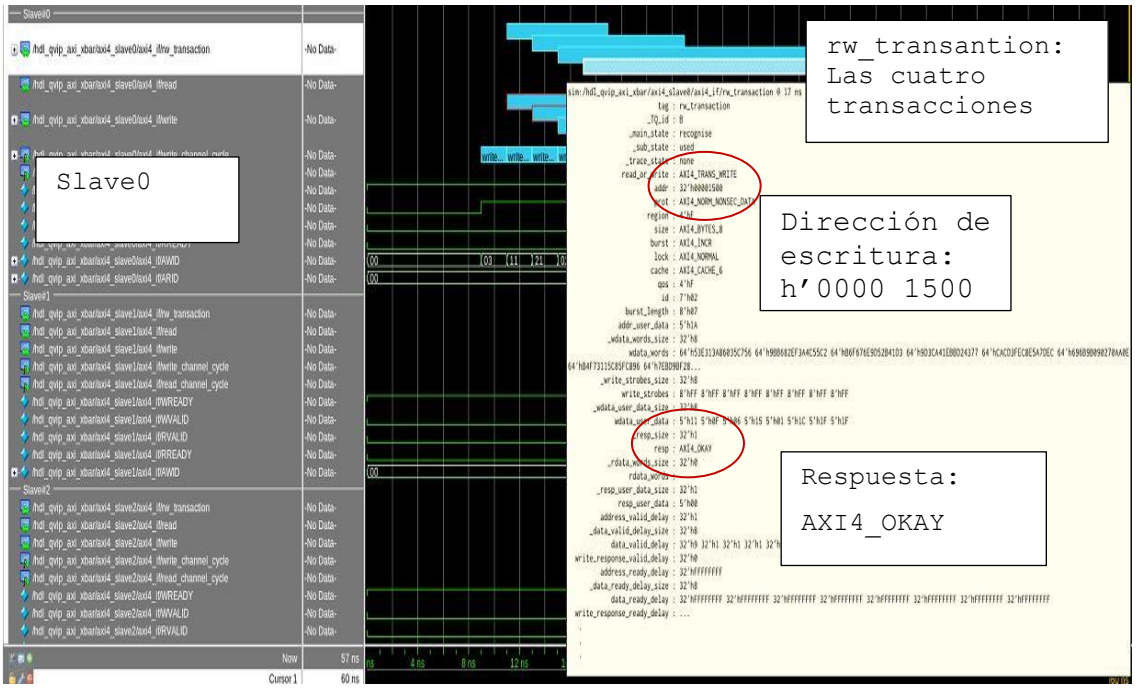


Figura 90 Respuesta del módulo Slave0 a la secuencia 7

Con lo analizado se puede concluir que el DUV ha respondido correctamente, pudiendo gestionar las transacciones y cumpliendo con la característica que se quería demostrar.

5.8. SECUENCIA 8

Esta secuencia se asemeja a la secuencia 7. Se utilizarán funciones que permiten controlar de forma independiente los canales de lectura y de escritura utilizados según el protocolo AXI4 descrito. Si bien el uso de estas funciones complica la definición de la secuencia, es la única forma de poder controlar todos y cada uno de los canales del protocolo de forma independiente. Esto posibilita la descripción de estaciones que permitan la verificación de determinadas características.

5.8.1. DEFINICIÓN

AXI4 no permite entrelazar las transacciones de ciclos de escritura de datos (*write data*) y requiere que las transacciones de ciclos de escritura en ráfagas se sucedan en el mismo orden que las transacciones de escritura de ciclos de direcciones (*write addr*).

5.8.2. MÉTODO

El procedimiento a seguir será generar dos transacciones de escritura de ráfagas de ciclo de escritura de datos (*write data*), y una sola transacción de ciclo de escritura de dirección (write addr) y observar el resultado.

5.8.3. RESPUESTA ESPERADA

La respuesta esperada es que el diseño devuelva un mensaje error.

5.8.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t8.svh`

5.8.5. PROCEDIMIENTO

Como ya se mostró en la anterior secuencia, este tipo de secuencia es parametrizable, y sigue el mismo concepto de configuración de atributos, inicialización y nombre que la secuencia 7, con lo que no se va a describir este proceso de nuevo, pudiéndose dirigir a los apartados correspondiente de la secuencia 7 en caso de duda. La secuencia que se ejecutará desde el *script* `Questa_run` será la secuencia parametrizable `qvip_axi_xbar_vseq_t8_p`.

En esta secuencia se ha ejecutado una transacción de un único ciclo de escritura de dirección, y dos transacciones, tipo `ráfaga 'BURST'`, de ciclo de escritura de datos, y se ha observado la respuesta del sistema.

Hay que destacar que, en este caso, es importante el orden de ejecución por lo que se ejecutan de forma secuencial.

5.8.6. SECUENCIA

Todo el proceso del código de la tarea *body,* la definición de las transacciones, parametrización, definición de los tipos, y forma de ejecutar la secuencia es igual que en la anterior, por lo que no se explicara aquí de nuevo. Se puede referir a la secuencia 7 si se necesita más detalle.

Lo que sí es relevante en esta secuencia es que se tiene que ejecutar de forma secuencial, por lo que no se utilizará el bloque `fork-join`. En este caso, se ejecuta una transacción de ciclo de escritura de dirección, `wadrr0`, y dos transacciones de ciclo de escritura de datos `wdata0, wdata01`. Ambas transacciones han sido limitadas con una longitud, `burst-lenght,` de 4 palabras para visualizarlas más cómodamente.

La Figura 91 muestra la tarea `body` de la secuencia 8, que como se puede observar no varía de la anterior secuencia 7.

```
95   task qvip_axi_xbar_vseq_t8::body();
96
97     int num_txn = 1;
98     // Get the config object so that you can wait for a clock after reset
99     // to issue the first sequence_item.
100    cfg = axi4_master0_cfg_t::get_config(axi4_master0);
101
102    cfg.wait_for_reset();
103    cfg.wait_for_clock();
104
105    fork
106      begin
107        for(int i=0;i<num_txn;++i)
108        begin
109          execute_wr_addr_data();
110          execute_rd_addr();
111        end
112      end
113    join
114
115     // Just to ensure that all transactions complete
116     // before the sequence ends
117        repeat(num_txn*2)
118          begin
119            rw_t rw_txn = rw_t::type_id::create("rw_txn");
120            rw_txn.m_receive_id = get_stream_id(this);
121            rw_txn.receive(cfg);
122          end
123
124  endtask
125
```

Figura 91 Código de la tarea body de la secuencia 8

La Figura 92 muestra la definición de las transacciones a utilizar en esta secuencia que, como se puede observar, es la misma que la descrita en la secuencia 7.

```
158  task qvip_axi_xbar_vseq_t7::execute_wr_addr_data();
159
160    waddr_t       waddr0 = waddr_t::type_id::create("waddr0");
161    waddr_t       waddr01 = waddr_t::type_id::create("waddr01");
162    waddr_t       waddr1 = waddr_t::type_id::create("waddr1");
163    waddr_t       waddr2 = waddr_t::type_id::create("waddr2");
164
165    wdata_burst_t wdata0 = wdata_burst_t::type_id::create("wdata0");
166    wdata_burst_t wdata01 = wdata_burst_t::type_id::create("wdata01");
167    wdata_burst_t wdata1 = wdata_burst_t::type_id::create("wdata1");
168    wdata_burst_t wdata2 = wdata_burst_t::type_id::create("wdata2");
169
170    waddr0.set_sequencer(axi4_master0);
171    waddr01.set_sequencer(axi4_master0);
172    waddr1.set_sequencer(axi4_master1);
173    waddr2.set_sequencer(axi4_master2);
174
175    wdata0.set_sequencer(axi4_master0);
176    wdata01.set_sequencer(axi4_master0);
177    wdata1.set_sequencer(axi4_master1);
178    wdata2.set_sequencer(axi4_master2);
179
```

Figura 92 Asignación de módulos master en la secuencia 8

La Figura 93 muestra la ejecución de las transacciones en modo secuencial para la secuencia 8. Para su ejecución, se ha modificado la función `execute_wr_addr_data`, eliminando el bloque `fork_join`. En primer lugar, se realiza una transacción de escritura de direcciones sobre el módulo *master0*. El nombre de esta transacción es `waddr0`. Los atributos de esta transacción se han configurado de igual forma que en la secuencia 7. Se ha configurado con los siguientes parámetros, `lock=AXI4_NORMAL, addr=h'0000_1000, burst_lenght=4` y un `id=3`. Posteriormente se realiza una transacción de ciclo de escritura de datos, denominada `wdata0`, donde se mantiene el mismo `burst_lenght` que el utilizado en la escritura de dirección, y se ponen todas las señales `strobe=1`, tal y como se hizo en la secuencia 7. A continuación, se realiza lo mismo pero esta vez con una nueva transacción denominada `wdata1`.

```
begin
// ******************************************
// secuencia del master 0 escritura de direccion
 start_item( waddr0 );
 `uvm_info("wr_addr0", "start_item", UVM_LOW)
  if(!waddr0.randomize() with {waddr0.lock == AXI4_NORMAL;
                               waddr0.addr == 32'h0000_1000;
                               waddr0.burst_length <= 4;
                               waddr0.id == 3;
                               }) `uvm_error("axi4_master_phase_seq","Write address randomization failure")
 finish_item(waddr0);
 `uvm_info("wr_addr0", "finish_item", UVM_LOW)
// ******************************************
end

begin
// ******************************************
// escritura de datos del master 0

 start_item( wdata0 );
 `uvm_info("wr_data0", "start_item", UVM_LOW)
  if(!wdata0.randomize() with {wdata0.burst_length  == waddr0.burst_length;
                               foreach(wdata0.write_strobes[i]) wdata0.write_strobes[i] == '1;

                               }) `uvm_error("axi4_master_phase_seq", "Write data randomization failure")
 finish_item( wdata0 );
 `uvm_info("wr_data0", "finish_item", UVM_LOW)
// ******************************************
end


begin
// ******************************************
// escritura de datos del master 01

 start_item( wdata01 );
 `uvm_info("wr_data01", "start_item", UVM_LOW)
  if(!wdata01.randomize() with {wdata01.burst_length  == waddr0.burst_length;
                               foreach(wdata01.write_strobes[i]) wdata01.write_strobes[i] == '1;

                               }) `uvm_error("axi4_master_phase_seq", "Write data randomization failure")
 finish_item( wdata01 );
 `uvm_info("wr_data01", "finish_item", UVM_LOW)
// ******************************************
end
```

Figura 93 Código de las transacciones de la secuencia 8

5.8.7. RESPUESTA OBTENIDA

La Figura 94 muestra la secuencia generada. Como se puede observar se ha ejecutado un ciclo de escritura completo, escritura de dirección y de datos, cuya finalización se ha remarcado mediante un círculo rojo en la gráfica mostrada.

A continuación, y tras enviar la siguiente transacción de escritura de datos, el sistema no ha generado ninguna transacción en el canal de respuesta al ciclo de escritura de datos.

Además, en la Figura 94 se puede observar que, durante este segundo ciclo de escritura de datos, la interfaz gráfica muestra todo este ciclo en color rojo y no en color verde como era habitual en las escrituras anteriores. Este hecho, puede estar motivado al no

haberse generado una respuesta a la escritura.

Figura 94 Grafica de la generación de transacciones de la secuencia 8.

La Figura 95 muestra que no se genera respuesta en el canal de respuesta de escritura, por lo que no cumple con el protocolo AXI4. Un analisis mas detallado de la transacion muestra que, aunque no responde con un mensaje de error en el canal de respuesta, hay dos campos internos que sí muestran error, el campo `sub_state` y el campo `trace_state` que aparecen con un mensaje de error. Hay que indicar que esos campos internos son propios de la herramienta utilizada por lo que no es posible entrar a analizar su significado.

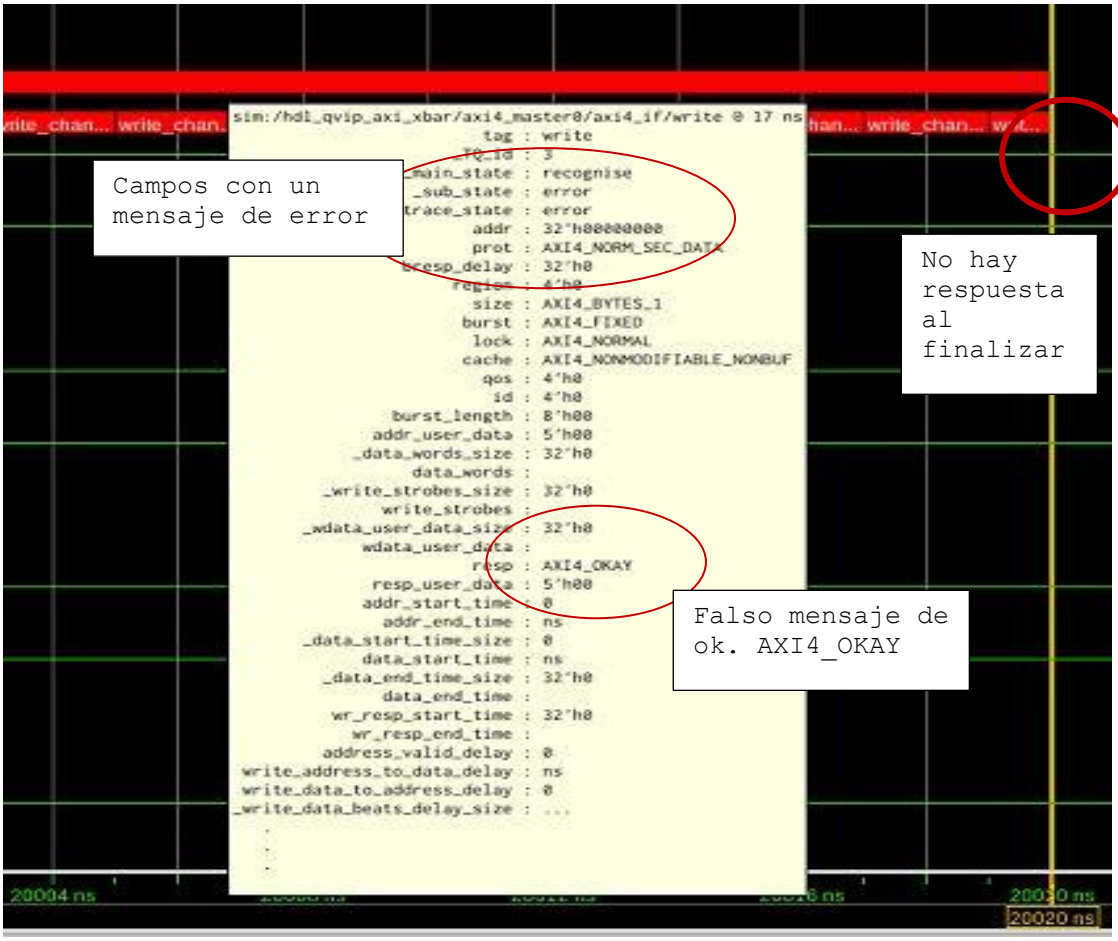


Figura 95 Mensajes de error de la secuencia 8

La Figura 96 muestra un extracto del fichero *transcript* generado, donde el error aparece recogido en la cuarta línea del código mostrado.

Figura 96 Información del script Questa Sim de error de la secuencia 8

La Figura 97 muestra la escritura recibida en el módulo *slave0*. Se pueden observar las transacciones recibidas, y cómo solo se ha producido un ciclo de escritura completo, con lo que la segunda escritura de datos no se ha realizado. De su análisis, se concluye que en todo el proceso correspondiente a la segunda escritura de datos desde el módulo

master0, estos datos se han perdido, hecho que coincide con la visualización en rojo que se observó en la figura anterior.

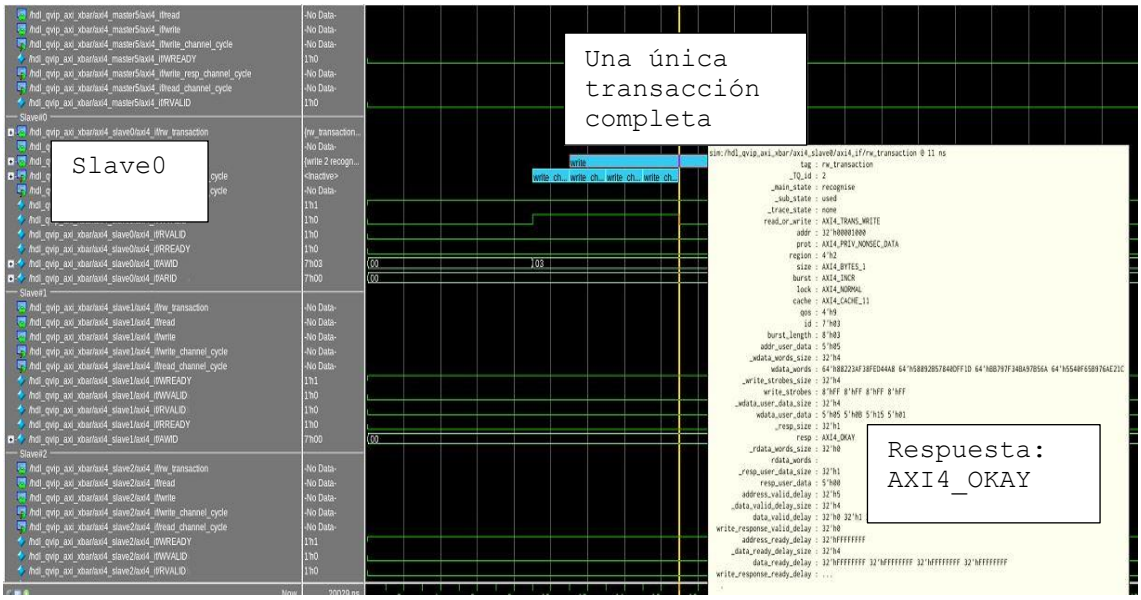


Figura 97 Respuesta a la escritura en el módulo Slave0 de la secuencia 8

Con todo lo observado se puede concluir, que a cada ciclo de escritura de datos le tiene que preceder una escritura de dirección para cumplir con lo establecido en el protocolo AXI4. En caso contrario, el DUV verificado no es capaz de gestionarlo. La única forma de controlar que el proceso no se ha hecho correctamente, es el hecho de que no se va a generar una repuesta en el canal `response`, obligatoria siempre que se ejecuta una escritura en el protocolo AXI4. Ante esta falta de respuesta, no se puede dar por correcta la escritura según el protocolo.

5.9. SECUENCIA 9

Esta secuencia vuelve a hacer uso de las API de alto nivel facilitadas por la herramienta QVIP de *Mentor Graphics*. Este tipo de secuencia se caracterizan por subir a un nivel de abstracción mayor, y permiten un mayor aislamiento de la configuración de los protocolos de comunicación y configuración de atributos automáticos para ejecutar la secuencia.

5.9.1. DEFINICIÓN

Este apartado está destinado para demostrar la característica de evitar el *'Starvation'* de los puertos de los módulos *master*. El método que implementa para realizar esta tarea es que a cada uno de sus puertos en los módulos *master* se le otorga una prioridad cuando se inicia una transacción. Posteriormente el protocolo del DUV compara estas prioridades por pares hasta que elige el puerto de mayor prioridad para, entonces, iniciar la transacción de mayor prioridad. Una vez finalizada la transacción, el estado de prioridad es avanzado un nivel en todos los módulos *masters* restantes. De esta forma, un módulo *master* no puede acaparar siempre la mayor prioridad en sus transferencias, bloqueando a los demás módulos. Es decir, inicialmente se asignan prioridades establecidas a los módulos *master*. En segundo lugar, se transfiere el control al módulo de mayor prioridad. Al terminar la transacción, se reasignan prioridades, subiendo el nivel de prioridades de los módulos pendientes. Al módulo transferido, en caso de tener más transacciones pendientes se le asigna la prioridad de menor valor, siendo colocado en la posición final.

5.9.2. MÉTODO

Se va a proceder a generar ráfagas de transferencia de datos desde el mismo módulo *master* y también ráfagas de datos desde otros módulos *master* para comprobar que las transferencias de los módulos se intercalan, y que un módulo no bloquea a los demás módulos. Este hecho es lo que se conoce como *Starvation*.

5.9.3. RESPUESTA ESPERADA

Las transacciones desde el mismo módulo *master* no tienen que transferirse de forma consecutiva, sino alternadas con otros módulos *masters*.

5.9.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t9.svh`

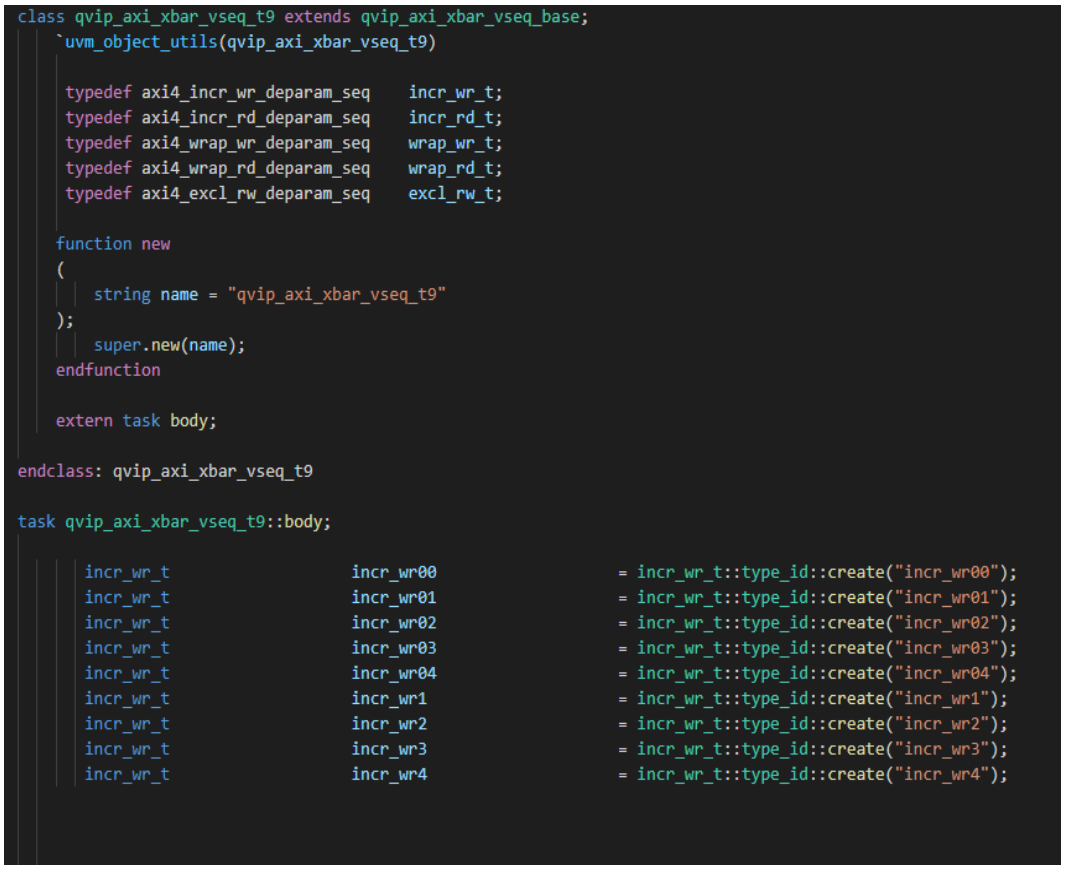
5.9.5. PROCEDIMIENTO

El procedimiento seguido en esta secuencia es la creación de varias transacciones desde distintos módulos *masters* y observar si se produce alternancia entre ellas. De esa forma se podrá verificar que un módulo *master* no acapara los servicios, y que sus transacciones se intercalan con las transacciones de otros módulos. Se van a generar 5 transacciones desde el módulo *master0*, donde el destino de las transferencias no es relevante, 2 transacciones desde el módulo *master1* y otras 2 transacciones de otros módulos, *master2 y master3*.

La secuencia que se ejecutará desde el *script* `Questa_run` es `qvip_axi_xbar_vseq_t9`.

5.9.6. SECUENCIA

La Figura 98 muestra el código de la secuencia. La definición de la secuencia sigue el mismo patrón descrito en las secuencias anteriores. En primer lugar, se registra la secuencia en el *Factory* y posteriormente se declaran los tipos a utilizar. Posteriormente, y dentro de la tarea *body,* se crean las transacciones a utilizar. En este caso en particular, todas las transacciones utilizadas son de tipo incremental. Las API utilizadas van a permitir al ingeniero de verificación la abstracción del protocolo de transferencia.



```
class qvip_axi_xbar_vseq_t9 extends qvip_axi_xbar_vseq_base;
    `uvm_object_utils(qvip_axi_xbar_vseq_t9)

    typedef axi4_incr_wr_deparam_seq    incr_wr_t;
    typedef axi4_incr_rd_deparam_seq    incr_rd_t;
    typedef axi4_wrap_wr_deparam_seq    wrap_wr_t;
    typedef axi4_wrap_rd_deparam_seq    wrap_rd_t;
    typedef axi4_excl_rw_deparam_seq    excl_rw_t;

    function new
    (
        string name = "qvip_axi_xbar_vseq_t9"
    );
        super.new(name);
    endfunction

    extern task body;

endclass: qvip_axi_xbar_vseq_t9

task qvip_axi_xbar_vseq_t9::body;

        incr_wr_t                   incr_wr00                   = incr_wr_t::type_id::create("incr_wr00");
        incr_wr_t                   incr_wr01                   = incr_wr_t::type_id::create("incr_wr01");
        incr_wr_t                   incr_wr02                   = incr_wr_t::type_id::create("incr_wr02");
        incr_wr_t                   incr_wr03                   = incr_wr_t::type_id::create("incr_wr03");
        incr_wr_t                   incr_wr04                   = incr_wr_t::type_id::create("incr_wr04");
        incr_wr_t                   incr_wr1                    = incr_wr_t::type_id::create("incr_wr1");
        incr_wr_t                   incr_wr2                    = incr_wr_t::type_id::create("incr_wr2");
        incr_wr_t                   incr_wr3                    = incr_wr_t::type_id::create("incr_wr3");
        incr_wr_t                   incr_wr4                    = incr_wr_t::type_id::create("incr_wr4");
```

Figura 98 Código de la declaración de la secuencia 9

La Figura 99, muestra cómo se configuran las transacciones. Como las transacciones siguen el mismo método ya descrito en anteriores secuencias, se muestran, a modo de ejemplo, cuatro de las transacciones utilizadas. En concreto se muestran las transacciones asignadas a los módulos *master0*, *master1*, *master2* y *master3*, denominadas `incr_wr04`, `incr_wr1`, `incr_wr2` y `incr_wr3` respectivamente. Si se quiere tener un detalle completo del código de todas las transacciones, se puede consultar el apartado de la secuencia 9 en el Anexo II de este TFG.

La Figura 99 muestra entre las líneas 68 y 71, la configuración de la transacción `incr_wr04`. Se describen las configuraciones de las transacciones con el código como ejemplo de la primera transacción mostrada, `incr_wr04`, siendo las demás transacciones definidas de igual modo. En primer lugar, línea 68 se le asocia el componente *UVM Agente* correspondiente, mediante su manejador, `axi4_master0` sobre el que se va a ejecutar la transacción. Posteriormente, en la línea 69 se configura su identificador, `ID=4`. Para terminar, la línea 70 y 71, se inicializa el resto de parámetros invocando a la función *randomize* con las restricciones de su dirección de memoria y el

tamaño del dato a escribir, `addr=h'0000_2000` y `wr_data.size=16`.

```
67      //master0 write
68              incr_wr04.set_sequencer(axi4_master0);
69              incr_wr04.id = 4;
70              if(!incr_wr04.randomize() with {addr == 32'h0000_2000; wr_data.size == 16;})
71                `uvm_error(this.get_full_name(),"Randomisation failure");
72
73      //master1 write
74              incr_wr1.set_sequencer(axi4_master1);
75              incr_wr1.id = 5;
76              if(!incr_wr1.randomize() with {addr == 32'h0000_4000; wr_data.size == 16;})
77                `uvm_error(this.get_full_name(),"Randomisation failure");
78      //master2 write
79              incr_wr2.set_sequencer(axi4_master2);
80              incr_wr2.id = 6;
81              if(!incr_wr2.randomize() with {addr == 32'h1000_4000; wr_data.size == 16;})
82                `uvm_error(this.get_full_name(),"Randomisation failure");
83       //master3 write
84              incr_wr3.set_sequencer(axi4_master3);
85              incr_wr3.id = 7;
86              if(!incr_wr3.randomize() with {addr == 32'h2000_4000; wr_data.size == 16;})
87                `uvm_error(this.get_full_name(),"Randomisation failure");
```

Figura 99 Código de la configuración de transacciones de la secuencia 9

La Figura 100 muestra cómo una vez configuradas todas las transacciones estas se ejecutan en modo concurrente dentro del bloque `fork_join`. La ejecución de las transacciones se realiza mediante el método `start`.

```
94              fork
95                  incr_wr00.start(axi4_master0);
96                  incr_wr01.start(axi4_master0);
97                  incr_wr02.start(axi4_master0);
98                  incr_wr03.start(axi4_master0);
99                  incr_wr04.start(axi4_master0);
100                 incr_wr1.start(axi4_master1);
101                 incr_wr2.start(axi4_master2);
102                 incr_wr3.start(axi4_master3);
103                 incr_wr4.start(axi4_master1);
104             join
105     end
```

Figura 100 Código de ejecución de las transacciones en la secuencia 10

5.9.7. RESPUESTA OBTENIDA

La Figura 101 muestra la secuencia generada. Se puede observar cómo el orden de las transacciones ha sido alternado, y no ha sido acaparado por el módulo *master0*, con lo que no se ha producido la situación de *starvation*. Las transacciones de distintos módulos *masters* se han intercalado en el tiempo. En primer lugar, se han enviado las transacciones `incr_wr04`, `inc_wr2` y `incr_wr3`, siendo estas de distintos módulos *master*. Posteriormente, y siguiendo el orden de prioridades que asigna automáticamente el DUV, se han ejecutado las transacciones `incr_wr03` e `incr_wr1`, una vez más enviada de distintos módulos. Por último, se ha transferido la transacción `incr_wr4`, seguida de `incr_wr02`, `incr_wr01` e `incr_wr00`. Con lo que se verifica que la característica descrita está cumpliéndose.

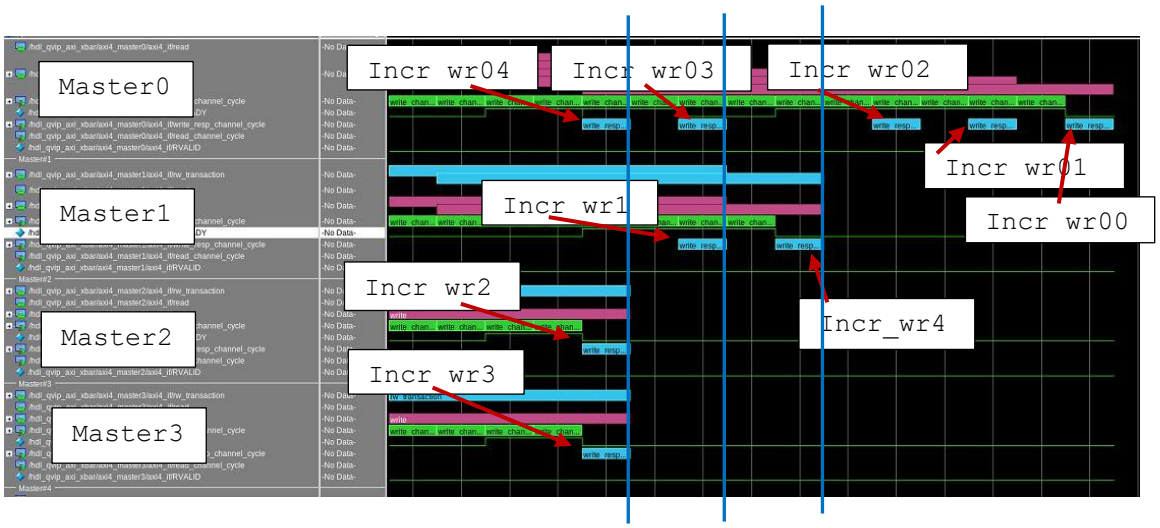


Figura 101 Secuencia 9 ejecutada

Para poder identificar con mayor detalle las diferentes transacciones, en la Tabla 9 se muestra un resumen de las direcciones, identificadores y módulos *slaves* que señalan

cada una de las transacciones.

Tabla 9 Resumen de transacciones generadas en la secuencia 9

Transacción	Dirección	ID	Módulo *Slave*
incr_wr00	h'0000_4500	0	*slave0*
incr_wr01	h'1000_4000	1	*slave1*
incr_wr02	h'0000_1000	2	*slave0*
incr_wr03	h'1000_1000	3	*slave1*
incr_wr01	h'0000_2000	4	*slave0*
incr_wr1	h'0000_4000	5	*slave0*
incr_wr2	h'1000_4000	6	*slave1*
incr_wr03	h'2000_4000	7	*slave2*
incr_wr04	h'0000_3500	8	*slave0*

La Figura 102 muestra las respuestas en los módulos *slaves* de las transacciones enviadas. En esta figura se puede observar cómo todas las transacciones llegan de forma correcta y ordenada al módulo *slave* asignado. El orden de llegada coincide con el orden temporal de generación de las mismas.

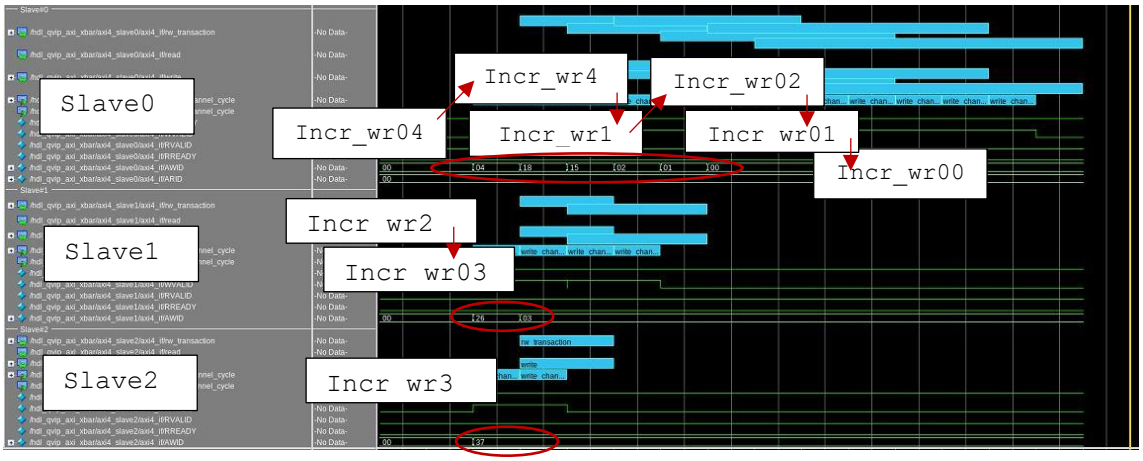


Figura 102 Respuesta de los módulos slaves para la secuencia 9

Con lo visto anteriormente se puede concluir que el DUV verificado cumple con esta característica.

5.10. SECUENCIA 10

Se ha considerado importante su demostración para verificar que los modos de ejecución descritos en el atributo `LOCK` se cumplen en el DUV del presente TFG, aunque esta característica no se recoge en la hoja de propiedades. Esta secuencia está relacionada con el acceso en modo exclusivo `AXI4_EXCLUSIVE`. Este modo de transferencia es una de las formas de transferencias definidas en el protocolo de transferencia de AXI4. Este acceso genera una transacción de lectura seguida de una transacción de escritura y garantiza que la posición de memoria leída no ha sido modificada cuando se escribe posteriormente en ella. Este modo es una forma de verificar quién está accediendo a esa posición de memoria, y si es fiable el resultado obtenido. Como se ha descrito, se tendrá que hacer un acceso de lectura y posteriormente otro de escritura, con el modo `lock=AXI4_EXCLUSIVE`. Cuando el acceso en modo exclusivo ha sido correcto, el sistema debe responder a la transacción de escritura con `AXI4_EXOKAY`. En caso contrario se obtendrá la respuesta de `AXI4_OK`.

Para la modificación de los datos escritos, se va a hacer uso de un método de acceso a memoria sin usar el protocolo de comunicación AXI4. Mediante este procedimiento se pueden modificar los datos escritos, como si esta modificación fuera realizada sin control por el protocolo de comunicación. De esta manera se puede comprobar si el protocolo es capaz de detectar estas irregularidades. Este método de acceso se denomina acceso por *backdoor*.

5.10.1. DEFINICIÓN

Demostración del acceso en modo exclusivo con modificación de los datos en modo escritura por *backdoor*

5.10.2. MÉTODO

El método empleado consiste en generar distintas transacciones de lectura y escritura intercalando un acceso por *backdoor* para modificar los datos escritos previamente en memoria. En concreto la secuencia seguirá el siguiente orden de transacciones, que se secuencian de la siguiente manera:

- En primer lugar, se realizan operaciones de lectura y escritura; lectura, modificación por *backdoo*r y escritura; y, finalmente una operación de lectura.
- Posteriormente se realizan operaciones de escritura, escritura, lectura y

finalmente, una operación de escritura.

5.10.3. RESPUESTA ESPERADA

Siguiendo el orden de las transacciones que se ha descrito, se tendrá que obtener las siguientes respuestas que se muestran intercaladas.

- Primera operación de lectura y escritura: respuesta `AXI4_EXOKAY`.
- Segunda operación de lectura, modificación por *backdoor* y escritura: respuesta `AXI4_OKAY`.
- Tercera operación de lectura: respuesta, el dato leído ha de ser el correspondiente a la última escritura, posterior a la modificación por *backdoor*.
- Siguiente operación de escritura: respuesta `AXI4_EXOKAY`; operación de escritura: respuesta `AXI4_OKAY`; operaciones de lectura y escritura: respuesta `AXI4_EXOKAY`.

5.10.4. NOMBRE DE LAS SECUENCIAS

- `qvip_axi_xbar_vseq_t10.svh`

5.10.5. PROCEDIMIENTO

El procedimiento que se va a realizar en este tipo de secuencia es parametrizable, al igual que el utilizado en la secuencia 7. Su descripción y procedimiento de configuración asemeja a la secuencia antes mencionada, por lo que no se va a describir de nuevo estos conceptos. Este apartado se centra directamente en el código de la tarea *body* relativo a la ejecución de las transacciones mencionadas. La secuencia que se ejecuta desde el *script* `Questa_run` será la secuencia parametrizable `qvip_axi_xbar_vseq_t0_p`.

5.10.6. SECUENCIA

La Figura 103 muestra la configuración inicial de los tipos de transacciones que se van a ejecutar, en este caso particular únicamente se utiliza el tipo `write_item`.

```
87  task qvip_axi_xbar_vseq_t10::body();
88
89    bit [AXI4_ADDRESS_WIDTH-1:0] addr    [32];
90    bit [7:0]                    wr_data [32];
91    bit [7:0]                    rd_data [32];
92
93    read_item_t       read_item         = read_item_t::type_id::create("read_item");
94    write_item_t      write_item        = write_item_t::type_id::create("write_item");
95
96    cfg = axi4_vip_config_t::get_config(axi4_master0);
97    slv_cfg = axi4_slave0_cfg_t::get_config(axi4_slave0);
98
99    cfg.wait_for_reset();
100   cfg.wait_for_clock();
```

Figura 103 Cabecera de la tarea body de la secuencia 10

A continuación, se describe, paso a paso la ejecución de la secuencia:

- **Paso 1**(Figura 104): En este primer paso se ejecuta una transacción de lectura y escritura en modo exclusivo. La línea 106 realiza el primer paso previo al envío de la transacción que es la asignación de su manejador, y que determina el componente *UVM Agent* al que apunta, en este caso `axi4_master0`. Posteriormente, en la línea 107, se llamará la función `start_item`. Finalmente, y antes de enviarla desde la línea 108 hasta la línea 121, se inicializan los diferentes campos de la transacción usando la función *randomize* con restricciones en algunos de sus atributos. Una vez inicializada, se envía la transacción mediante la función `finish_item`.

  A lo largo de la ejecución de estas transacciones se ha detectado que es importante que los atributos entre las transacciones de lectura y escritura sean iguales para cumplir con los requisitos en modo *exclusivo*. Para poder garantizar que se cumple con los mismos atributos en la transacción de escritura, estos se asignan con los valores de la transacción de lectura anterior, tal y como se muestra en las líneas 129 y 130.

```
102  //      **********************PASO 1 ****************************
103
104  // Reading the slave memory via axi4 protocol
105
106   read_item.set_sequencer(axi4_master0);
107   start_item( read_item );
108     if(!read_item.randomize() with
109       {
110
111   read_item.burst_length == 0;
112         read_item.id           == 1;
113         read_item.burst        == AXI4_INCR;
114         read_item.addr         == 32'h00001000;
115         read_item.lock         == AXI4_EXCLUSIVE;
116         read_item.size         == AXI4_BYTES_8;
117
118
119       }
120     ) `uvm_error(this.get_full_name(),"Randomisation failure");
121     finish_item( read_item );
122
123  // writing the slave memory via axi4 protocol
124
125   write_item.set_sequencer(axi4_master0);
126   start_item( write_item );
127     if(!write_item.randomize() with
128       {
129   write_item.prot   == read_item.prot;
130         write_item.cache  == read_item.cache;
131   write_item.addr         == 32'h00001000;
132         write_item.burst_length == 0 ;
133         write_item.id           == 1;
134         write_item.write_strobes[0]  == 8'hFF;
135
136         write_item.burst        == AXI4_INCR;
137         write_item.lock         == AXI4_EXCLUSIVE;
138         write_item.size         == AXI4_BYTES_8;
139
140       }
141
142     ) `uvm_error(this.get_full_name(),"Randomisation failure");
143
144     finish_item( write_item );
145
```

Figura 104 Secuencia 10: Paso 1

- **Paso 2** (Figura 105): En este paso se han realizado las mismas operaciones ejecutadas en el paso 1, intercalando un acceso a los datos sin usar el protocolo de comunicación AXI4, para modificar el dato de memoria. Las líneas 167 hasta la 173, muestran el código de acceso en modo *backdoo*r. Este modo de escritura facilita el acceso a las posiciones de memoria sin hacer uso de la interfaz de comunicación, lo que permite modificar el contenido de la memoria y poder simular un acceso no registrado, o un error por modificación de los datos por cualquier otro motivo.

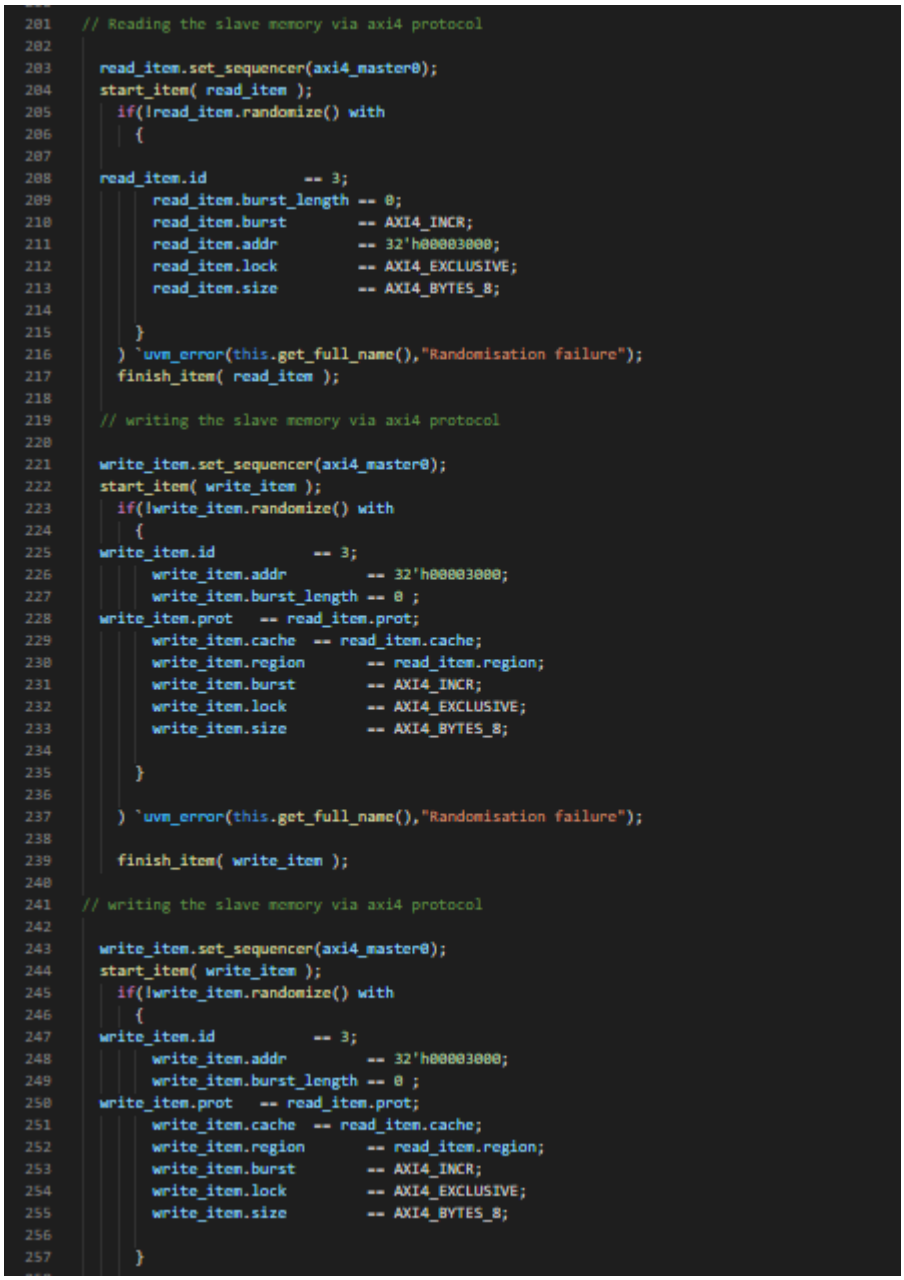
  En la línea 170 se muestra el código utilizado. Este procedimiento mediante el uso un bucle *for* realiza la escritura de 16 bytes a partir de la dirección `h'1000`. La línea 171 muestra la llamada a la función `backdoor_write`, a la cual se le

pasan dos parámetros. El primero, la dirección `h'0000_1000`, y el segundo el dato que se escribe, en este caso particular, el valor `h'ff`.

```
146   //     **********************PASO 2 ******************************
147
148   // Reading the slave memory via axi4 protocol
149
150   read_item.set_sequencer(axi4_master0);
151   start_item( read_item );
152     if(!read_item.randomize() with
153       {
154
155   read_item.id           == 2;
156         read_item.burst_length == 0;
157         read_item.burst        == AXI4_INCR;
158         read_item.addr         == 32'h00001000;
159         read_item.lock         == AXI4_EXCLUSIVE;
160         read_item.size         == AXI4_BYTES_8;
161
162
163       }
164     ) `uvm_error(this.get_full_name(),"Randomisation failure");
165     finish_item( read_item );
166
167 // Programming the slave memory through backdoor write
168
169   `uvm_info("SEQ/BACKDOOR/WR","Starting backdoor write access" , UVM_MEDIUM)
170   for(int i = 0; i < 16; i++)
171     slv_cfg.slv_mem.backdoor_write(32'h00001000 + i, 8'hff);
172
173   `uvm_info("SEQ/BACKDOOR/WR","Backdoor write access done" , UVM_MEDIUM)
174
175
176
177   // writing the slave memory via axi4 protocol
178
179   write_item.set_sequencer(axi4_master0);
180   start_item( write_item );
181     if(!write_item.randomize() with
182       {
183   write_item.id           == 2;
184         write_item.addr         == 32'h00001000;
185         write_item.burst_length == 0 ;
186   write_item.prot    == read_item.prot;
187         write_item.cache  == read_item.cache;
188         write_item.region       == read_item.region;
189         write_item.burst        == AXI4_INCR;
190         write_item.lock         == AXI4_EXCLUSIVE;
191         write_item.size         == AXI4_BYTES_8;
192
193       }
194
```

Figura 105 Secuencia 10 Paso 2

- **Paso 3**(Figura 106): Se genera una transacción de lectura y, posteriormente dos transacciones seguidas de escritura. Tal y como se muestra en el código de esta figura, ambas transacciones de escritura mantienen los mismos atributos. Estos atributos se corresponden con los atributos de la transacción de lectura.



```
201  // Reading the slave memory via axi4 protocol
202
203    read_item.set_sequencer(axi4_master0);
204    start_item( read_item );
205      if(!read_item.randomize() with
206        {
207
208    read_item.id           == 3;
209          read_item.burst_length == 0;
210          read_item.burst        == AXI4_INCR;
211          read_item.addr         == 32'h00003000;
212          read_item.lock         == AXI4_EXCLUSIVE;
213          read_item.size         == AXI4_BYTES_8;
214
215        }
216      ) `uvm_error(this.get_full_name(),"Randomisation failure");
217      finish_item( read_item );
218
219    // writing the slave memory via axi4 protocol
220
221    write_item.set_sequencer(axi4_master0);
222    start_item( write_item );
223      if(!write_item.randomize() with
224        {
225    write_item.id           == 3;
226          write_item.addr         == 32'h00003000;
227          write_item.burst_length == 0 ;
228    write_item.prot    == read_item.prot;
229          write_item.cache  == read_item.cache;
230          write_item.region       == read_item.region;
231          write_item.burst        == AXI4_INCR;
232          write_item.lock         == AXI4_EXCLUSIVE;
233          write_item.size         == AXI4_BYTES_8;
234
235        }
236
237      ) `uvm_error(this.get_full_name(),"Randomisation failure");
238
239      finish_item( write_item );
240
241  // writing the slave memory via axi4 protocol
242
243    write_item.set_sequencer(axi4_master0);
244    start_item( write_item );
245      if(!write_item.randomize() with
246        {
247    write_item.id           == 3;
248          write_item.addr         == 32'h00003000;
249          write_item.burst_length == 0 ;
250    write_item.prot    == read_item.prot;
251          write_item.cache  == read_item.cache;
252          write_item.region       == read_item.region;
253          write_item.burst        == AXI4_INCR;
254          write_item.lock         == AXI4_EXCLUSIVE;
255          write_item.size         == AXI4_BYTES_8;
256
257        }
```

Figura 106 Secuencia 10 Paso 3

- **Paso 4**(Figura 107): Se ejecuta una transacción de lectura y otra de escritura, pero sin restringir los atributos de inicialización al invocar la función *randomize*, de la transacción. Tal y como se observa en la figura, los atributos especificados en las líneas 292, 293 y 294 han sido comentados, por lo que sus valores se asignaran de forma aleatoria siendo, prácticamente, imposible que dichos valores coincidan con los atributos de la transacción de lectura.

```
263   //      ********************** PASO 4 ******************************
264
265   // Reading the slave memory via axi4 protocol
266
267     read_item.set_sequencer(axi4_master0);
268     start_item( read_item );
269       if(!read_item.randomize() with
270         {
271
272     read_item.id           == 4;
273           read_item.burst_length == 0;
274           read_item.burst        == AXI4_INCR;
275           read_item.addr         == 32'h00004000;
276           read_item.lock         == AXI4_EXCLUSIVE;
277           read_item.size         == AXI4_BYTES_8;
278
279         }
280       ) `uvm_error(this.get_full_name(),"Randomisation failure");
281       finish_item( read_item );
282
283     // writing the slave memory via axi4 protocol
284
285     write_item.set_sequencer(axi4_master0);
286     start_item( write_item );
287       if(!write_item.randomize() with
288         {
289     write_item.id           == 4;
290           write_item.addr         == 32'h00004000;
291           write_item.burst_length == 0 ;
292     //write_item.prot    == read_item.prot;
293           //write_item.cache  == read_item.cache;
294           //write_item.region       == read_item.region;
295           write_item.burst        == AXI4_INCR;
296           write_item.lock         == AXI4_EXCLUSIVE;
297           write_item.size         == AXI4_BYTES_8;
298
299         }
300
301     ) `uvm_error(this.get_full_name(),"Randomisation failure");
302
303     finish_item( write_item );
304
305   endtask
306
```

Figura 107 Secuencia 10 Paso 4

5.10.7. RESPUESTA OBTENIDA

La Figura 108 muestra la secuencia generada. Se puede observar cómo la respuesta obtenida en el paso 1 es la respuesta esperada `AXI4_EXOKAY`, lo que coincide con la finalización normal en una transacción de escritura, lectura en modo exclusivo.

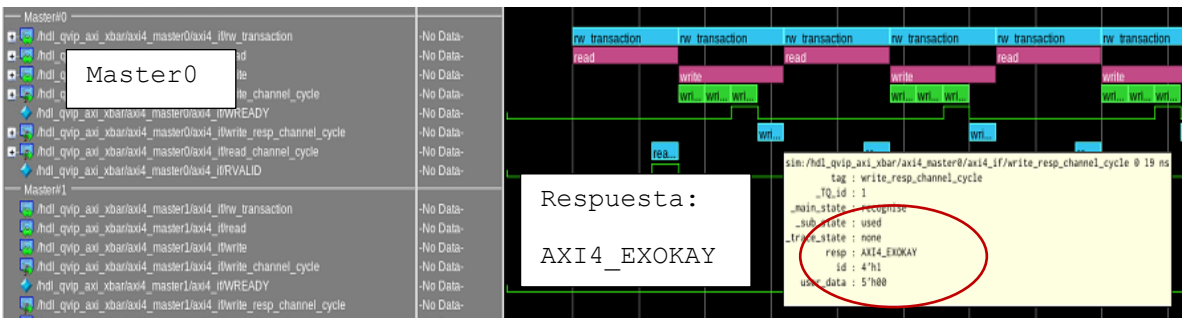


Figura 108 Respuesta al paso 1 de la secuencia 10

A continuación, en la Figura 109 se observa la respuesta correspondiente al paso 2. En este paso también se ha devuelto una repuesta `AXI4_EXOKAY`, si bien en medio de las operaciones escritura y lectura se ha modificado el dato con un acceso directo, *backdoor,* al no usar el protocolo de comunicación AXI4 la respuesta es correcta. Este supuesto podría ocurrir si en algún momento se modificaran los datos al mismo tiempo por dos puertos distintos, y uno de ellos no mantuviera el acceso por protocolo AXI4. En consecuencia, con el resultado obtenido se puede afirmar que el DUV verificado no tiene la capacidad de gestionar este supuesto. Esto no significa que su funcionamiento sea erróneo, ya que la modificación no se ha realizado siguiendo el protocolo AXI4.

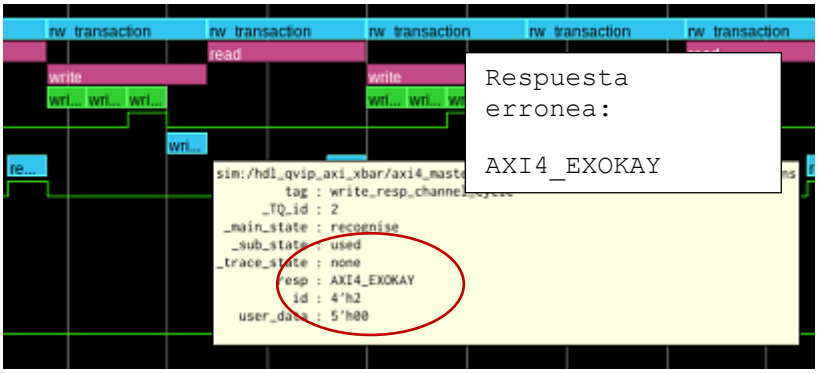


Figura 109 Respuesta al paso 2 de la secuencia 10

En las siguientes figuras, Figura 110 y Figura 111, se muestra el resultado del paso 3 descrito, al enviar dos transacciones seguidas en modo exclusivo a la misma dirección de memoria. La primera respuesta es `AXI4_EXOKAY`, mientras que la respuesta a la última transacción de escritura es `AXI4_OKAY`, lo cual es correcto. Se comprueba de esta forma que para que ocurra un acceso exclusivo, es obligatorio la realización de una primera transacción de lectura y posteriormente una transacción de escritura, independiente del `ID`, y del módulo *master* que lo realice.

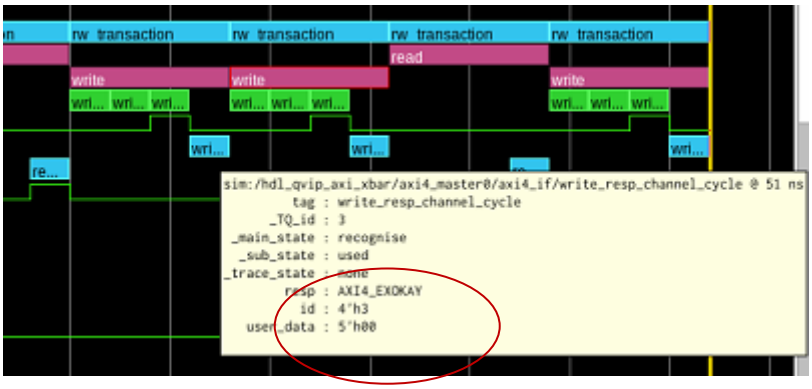


Figura 110 Respuesta al paso 3 de la secuencia 10: primera escritura

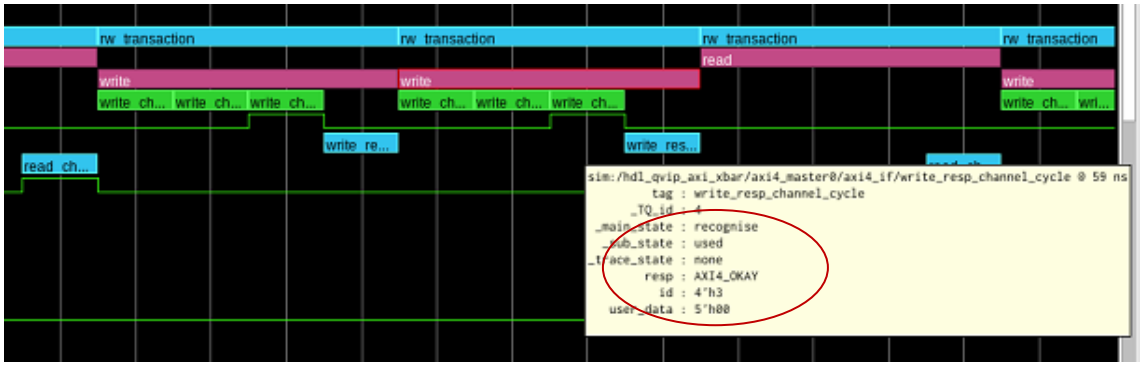


Figura 111 Respuesta al paso 3 de la secuencia 10: segunda escritura

Para concluir esta secuencia se ejecuta el paso 4, Figura 112 y Figura 113. Se trata de un acceso modo *exclusivo* y cuya respuesta ha sido `AXI4_OKAY`. La realización de este hecho es la de mostrar que, para que una transacción sea exclusiva, se han de mantener todos los atributos entre las transacciones de lectura y escritura. No tiene sentido en este caso utilizar la función *`randomize`* para inicializar los atributos de ambas transaciones. Este es el motivo porque se han restringido en todo momento las transacciones utilizadas en esta secuencia, igualando los atributos entre las transacciones de lectura y escritura. En este paso en particular se ha dejado sin restringir algunos atributos, por lo que la respuesta no es exclusiva. Si alguno de los atributos no coincide la respuesta será `EX_OKAY`, lo cual no es correcto.

En la Figura 112 y Figura 113 se puede observar las diferencias en los atributos `PROT`, `REGION` y `CACHE`.

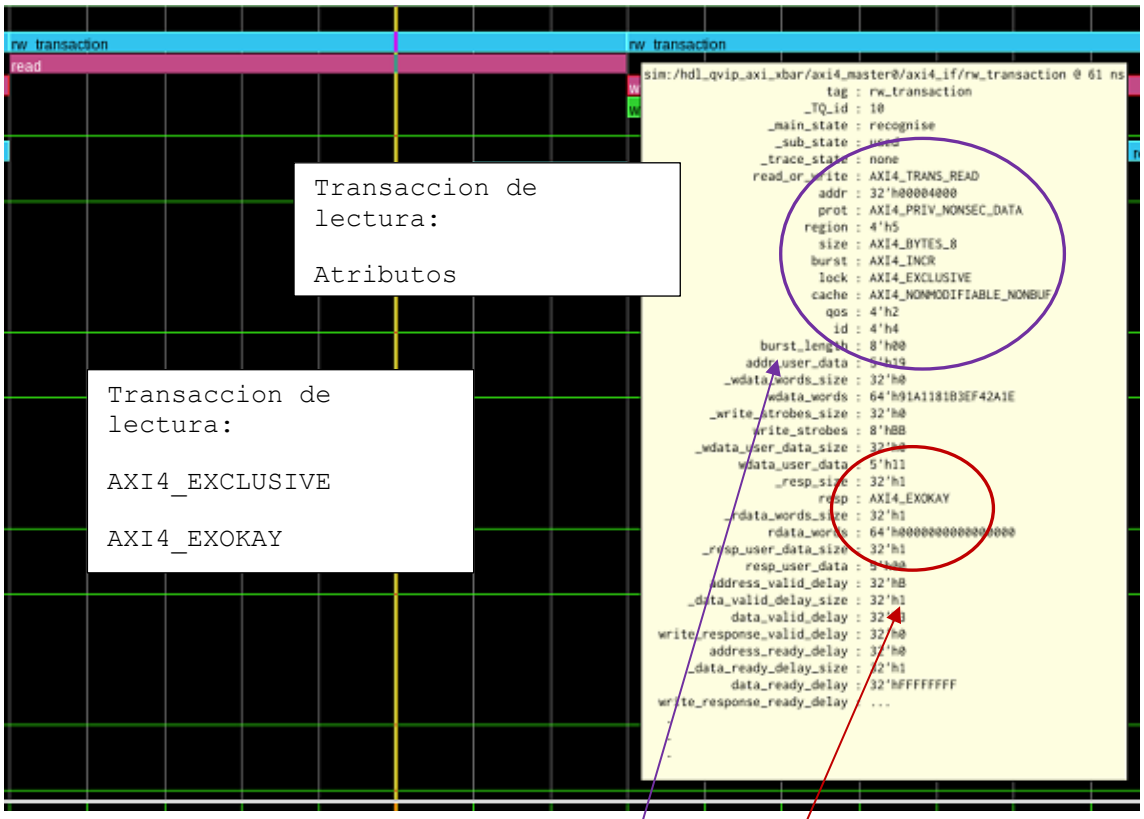


Figura 112 Respuesta al paso 4 de la secuencia 10: transacción de lectura

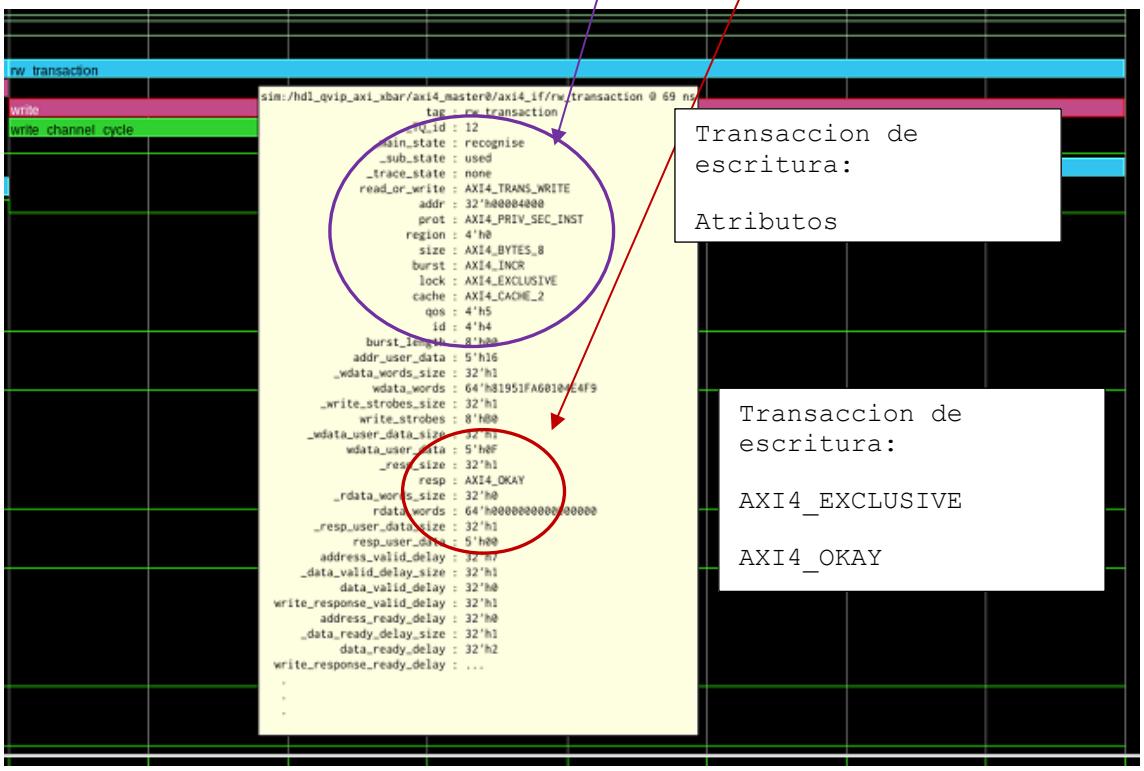


Figura 113 Respuesta al paso 4 de la secuencia 10: transacción de escritura.

Con lo observado en las respuestas obtenidas se puede concluir que el DUV verificado funciona correctamente, si bien el DUV no está diseñado para gestionar situaciones como la descrita en el paso 2 de esta secuencia.

CAPÍTULO 6: DESARROLLO DE UN AGENTE DE CONFIGURACIÓN

Durante la realización de este TFG surgió una dificultad al ejecutar una secuencia con una configuración diferente, debido a la necesidad de compilar todo el sistema con el aumento considerable de tiempo. Para evitar esta necesidad de compilar cada vez que se modifique la configuración del mismo, en este capítulo se describe, como trabajo extra, el desarrollo de un agente de configuración que permita modificar, en tiempo de ejecución, algunas de las características que el DUV que se está verificando. Estas características normalmente tienen que ser configuradas antes de empezar el test. En el diseño original, esto implicaba el tener que compilar todo el sistema cada vez que se quiere cambiar la configuración, lo cual lo convierte en un proceso engorroso y lento.

En concreto se va a actuar sobre el módulo del DUV denominado *decode error master module*. Este módulo se configura para cada *master*, de modo que, si se intenta acceder a un puerto *slave*, que no está ubicado dentro de ninguna regla de dirección de memoria, en vez de dar un error y no realizar ninguna acción perdiendo esa transacción, se direccionará un *slave port* por defecto. El puerto por defecto se debe configurar a través de la entrada. Esta característica es opcional y para habilitarla, hay que activarla a través de la entrada `default_mst_port`.

Los parámetros de configuración sobre los que se va a actuar son:

- `en_default_mst_port`: Este es un parámetro con tantos bits como puertos maestros se tienen configurados. Cada bit representa el *enable* de cada puerto *master.* En el caso de este TFG, al disponer de 6 *master* su tamaño será de 6 bits. Cada uno de esos bits representa 0: *disable* y 1: *enable* de cada *master* configurado. El bit menos significativo corresponde el *master 0*.
- `default_mst_port`: Este parámetro se configura de la siguiente manera. Por cada maestro de entrada hay 3 bits. Esos tres bits representan el puerto *slave* por defecto de cada *master port*. De esta manera, y para la configuración usada en este TFG, se tiene un total de 18 bits (18'b000_000_000_000_000_000), donde cada grupo de tres bits determina el puerto *slave* por defecto de cada puerto *master*.

  Por ejemplo, si se quiere que el puerto *master*0, esté con el `decode error master module` activo, y con el puerto *slave 2* como puerto por defecto, se tendrá que configurarlo de la siguiente manera:
  - `en_default_mst_port= 6'b000001`

- `default_mst_port= 18'b000_000_000_000_000_010`

Como nota importante hay que tener en cuenta que la respuesta que se obtendrá en este caso será del tipo `AXI4_OKAY`, cada vez que una transacción se dirija al puerto *slave* por defecto.

La Figura 114 muestra en las líneas 13 y 14 los tipos de las señales definidas, y utilizadas para la configuración del IP, en el módulo envolvente *`axi_xbar_wrap`*.

```
4   module axi_xbar_wrap#(
5       parameter int NoMasters = 6,
6       parameter int NoSlaves = 8
7   ) (
8     input logic                  clk_i,
9     input logic                  rst_ni,
10    input var logic [NoMasters-1:0][228:0]        slv_ports_req_i,
11    output logic [NoMasters-1:0][91:0]            slv_ports_resp_o,
12    output logic [NoSlaves-1:0][234:0]            mst_ports_req_o,
13    input var logic [NoSlaves-1:0][97:0]          mst_ports_resp_i,
14    input var logic [NoMasters-1:0]               en_default_mst_port,
15    input var logic [NoMasters-1:0][$clog2(NoSlaves)-1:0]  default_mst_port
16  );
```

Figura 114 Definición de las señales de configuración del IP

En la Figura 115 se muestra cómo se conectan los puertos de entrada del módulo envolvente a las señales propias del IP *`axi_xbar`*, líneas 145 y 146.

```
117   // DUT
118   //---------------------------------
119   axi_xbar #(
120     .Cfg          ( xbar_cfg ),
121     .slv_aw_chan_t (  aw_chan_mst_t ),
122     .mst_aw_chan_t (  aw_chan_slv_t ),
123     .w_chan_t      (  w_chan_t      ),
124     .slv_b_chan_t  (  b_chan_mst_t  ),
125     .mst_b_chan_t  (  b_chan_slv_t  ),
126     .slv_ar_chan_t (  ar_chan_mst_t ),
127     .mst_ar_chan_t (  ar_chan_slv_t ),
128     .slv_r_chan_t  (  r_chan_mst_t  ),
129     .mst_r_chan_t  (  r_chan_slv_t  ),
130     .slv_req_t     (  mst_req_t     ),
131     .slv_resp_t    (  mst_resp_t    ),
132     .mst_req_t     (  slv_req_t     ),
133     .mst_resp_t    (  slv_resp_t    ),
134     .rule_t        (  rule_t        )
135   ) i_xbar_dut (
136     .clk_i      ( clk_i      ),
137     .rst_ni     ( rst_ni     ),
138     .test_i     ( 1'b0       ),
139     .slv_ports_req_i  ( masters_req  ),
140     .slv_ports_resp_o ( masters_resp ),
141     .mst_ports_req_o  ( slaves_req   ),
142     .mst_ports_resp_i ( slaves_resp  ),
143     .addr_map_i       ( AddrMap      ),
144     //parametros originales
145     .en_default_mst_port_i ( en_default_mst_port ),
146     .default_mst_port_i    ( default_mst_port )
147
```

Figura 115 Conexionado de las señales de configuración del IP `axi_xbar`.

La Figura 116 muestra el método de configuración original del módulo *`axi_xbar`*. En este caso se muestra, líneas 149 y 150, cómo la configuración del IP se realizaba estableciendo los valores de ambos puertos de forma directa y fijas. Este ejemplo corresponde a la secuencia t5 ejecutada en el anterior capítulo. En esta secuencia se estableció la configuración modificando el módulo envolvente y volviendo a compilar. Este hecho es lo que se va a tratar de modificar en este capítulo.

```
148     //parametros para t5
149     //.en_default_mst_port_i ( 6'b000001 ),
150     //.default_mst_port_i    ( 18'b000_000_000_000_000_010)
151   );
152
```

Figura 116 Asignación de valores enable y mst_port

6.1. MÓDULO WRAP MODIFICADO

El módulo `axi_xbar_wrap` originalmente desarrollado es el que se muestra en la Figura 117 donde se observa que solo se permitía el acceso desde el exterior del módulo `top`, a las señales de los puertos *master* y *slave*. En este caso, todos los parámetros de configuración no son accesibles desde el exterior del módulo `top`, con lo que solo podían ser modificados antes de compilación, y modificando el módulo envolvente.

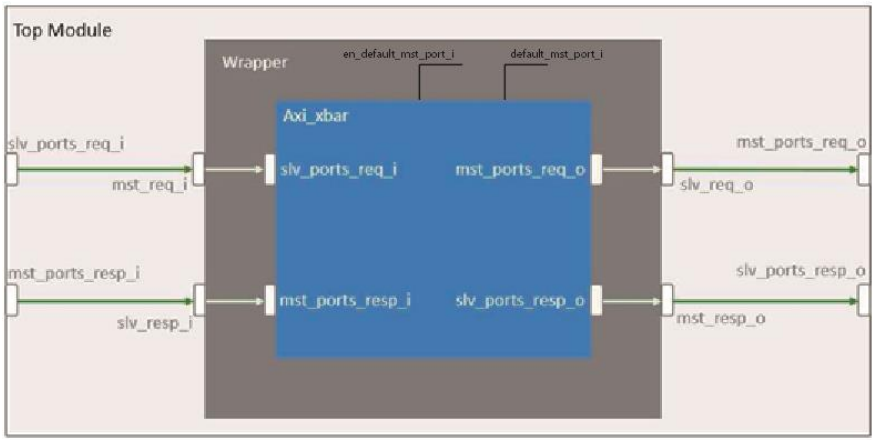


Figura 117 Modulo Wrap actual

Surge entonces la necesidad de modificar el módulo *`axi_xbar_wrap`* para hacer accesible la interfaz de configuración. El módulo modificado queda de la siguiente forma tal y como se presenta en la Figura 118.

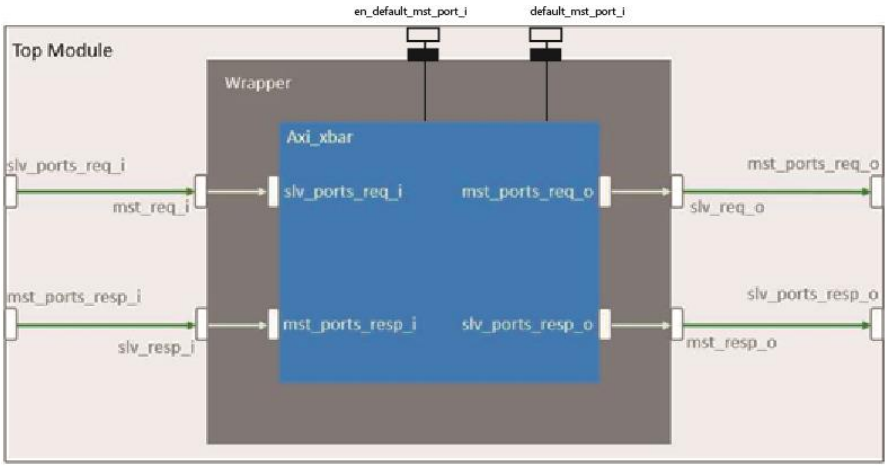


Figura 118 Modulo Wrap con interfaz de configuración.

De esta forma, es posible crear un módulo agente de configuración que sea referenciado

en el *testbench* y cuya finalidad sea la de establecer las señales de configuración como si de una secuencia más se tratara. La configuración del DUV se realiza, en este caso, en tiempo de ejecución, que es lo que se planteó al abordar esta tarea. El esquema, incluyendo el agente de configuración, es el que se muestra en la Figura 119.

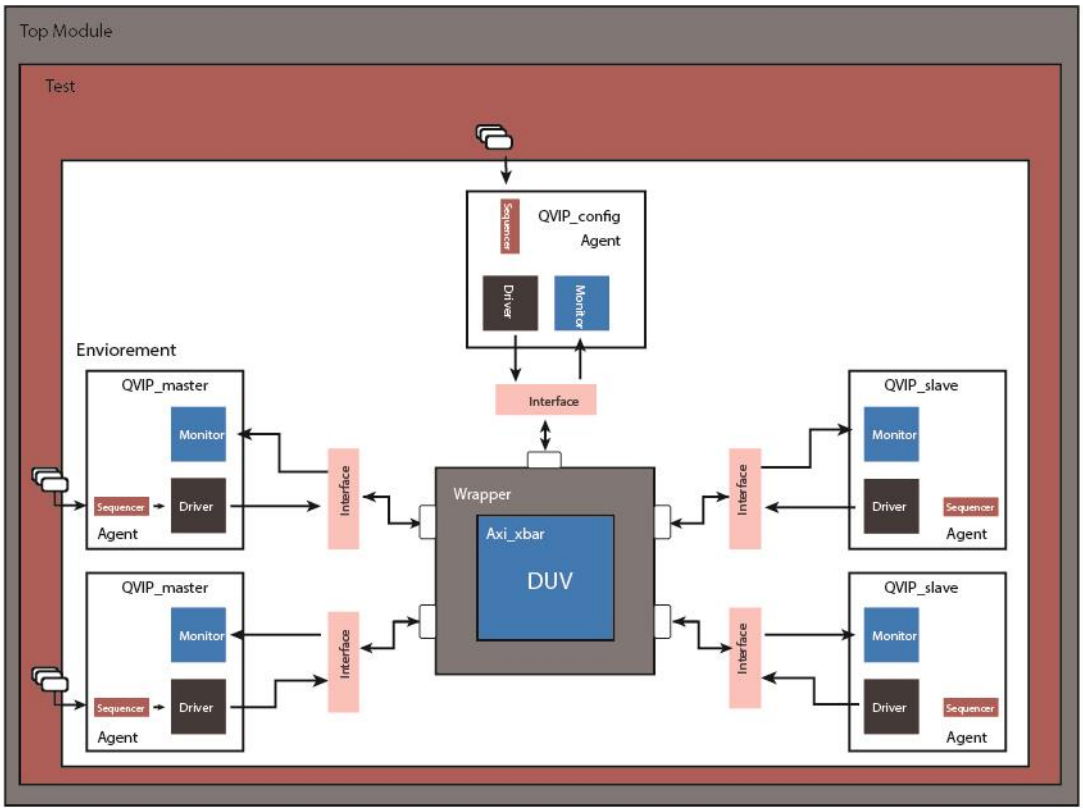


Figura 119 Esquema parcial del entorno de verificación UVM y el módulo DUV modificado

6.2. MÓDULOS UVM PARA LA CONFIGURACIÓN

En este apartado se muestran todos los módulos, con sus correspondientes ficheros, que se han tenido que crear o modificar para la realización de la configuración mediante secuencias.

En primer lugar, ha sido necesario crear un componente *UVM Agent*. Este componente integra tres módulos, como todos los *UVM Agent* de la metodología UVM. En este caso particular no se necesita implementar el módulo *UVM Monitor*, ya que no se va a leer las respuestas a las transacciones generadas.

6.2.1. COMPONENTE INTERFACE

En primer paso para hacer accesible los puertos de configuración del módulo

axi_xbar_wrap desde el *testbench*, es la definición de un interfaz propio, que incluya los puertos de configuración.

La Figura 120 muestra el código de la interfaz creada en el fichero qvip_*axi_xbar*_conf_if.svh. Como se muestra esta interfaz se ha personalizado para el caso particular de este TFG, de 6 *master* y 8 *slaves*. Este módulo contiene únicamente la definición de las señales que intervienen en la configuración del módulo.

```
24  interface qvip_axi_xbar_conf_if #(int NoMasters = 6, int NoSlaves = 8);
25      logic [NoMasters-1:0]                xbar_en_def_mst_port;
26      logic [NoMasters-1:0][$clog2(NoSlaves)-1:0] xbar_def_mst_port;
27  endinterface: qvip_axi_xbar_conf_if
28
```

Figura 120 Módulo Interface de configuración

6.2.2. TRANSACCIÓN DE CONFIGURACIÓN

El primer objeto a definir corresponde a la transacción que el módulo tendrá como parámetro de entrada y que resulta necesaria para poderla convertir en señales. Esta definición se realiza dentro del archivo qvip.*axi_xbar*_conf_tx.svh.

Como se muestra en la Figura 121 se definen los tipos de datos que componen la transacción dentro de una clase que se deriva de *uvm_sequence_item*. En el código mostrado en las líneas 21 y 22, se definen los dos datos a enviar a través del interfaz de configuración. A continuación, se registra la clase en el *Factory*.

```
20  class conf_packet extends uvm_sequence_item;
21      rand logic [5:0]      xbar_en_def_mst_port;
22      rand logic [5:0][2:0] xbar_def_mst_port;
23
24      `uvm_object_utils_begin(conf_packet)
25          `uvm_field_int(xbar_en_def_mst_port, UVM_DEFAULT)
26          `uvm_field_int(xbar_def_mst_port, UVM_DEFAULT)
27      `uvm_object_utils_end
28
29      function new(string name = "conf_packet");
30          super.new(name);
31      endfunction: new
32
33  endclass: conf_packet
34
```

Figura 121 Definición de los tipos de señales

6.2.3. COMPONENTE *UVM AGENT* DE CONFIGURACIÓN.

La nomenclatura que se va a emplear en todos los ficheros de configuración sigue el patrón de `qvip_axi_xbar_conf_xxxxx.svh`, donde el sufijo `xxxxx` será sustituido con el tipo componente que represente.

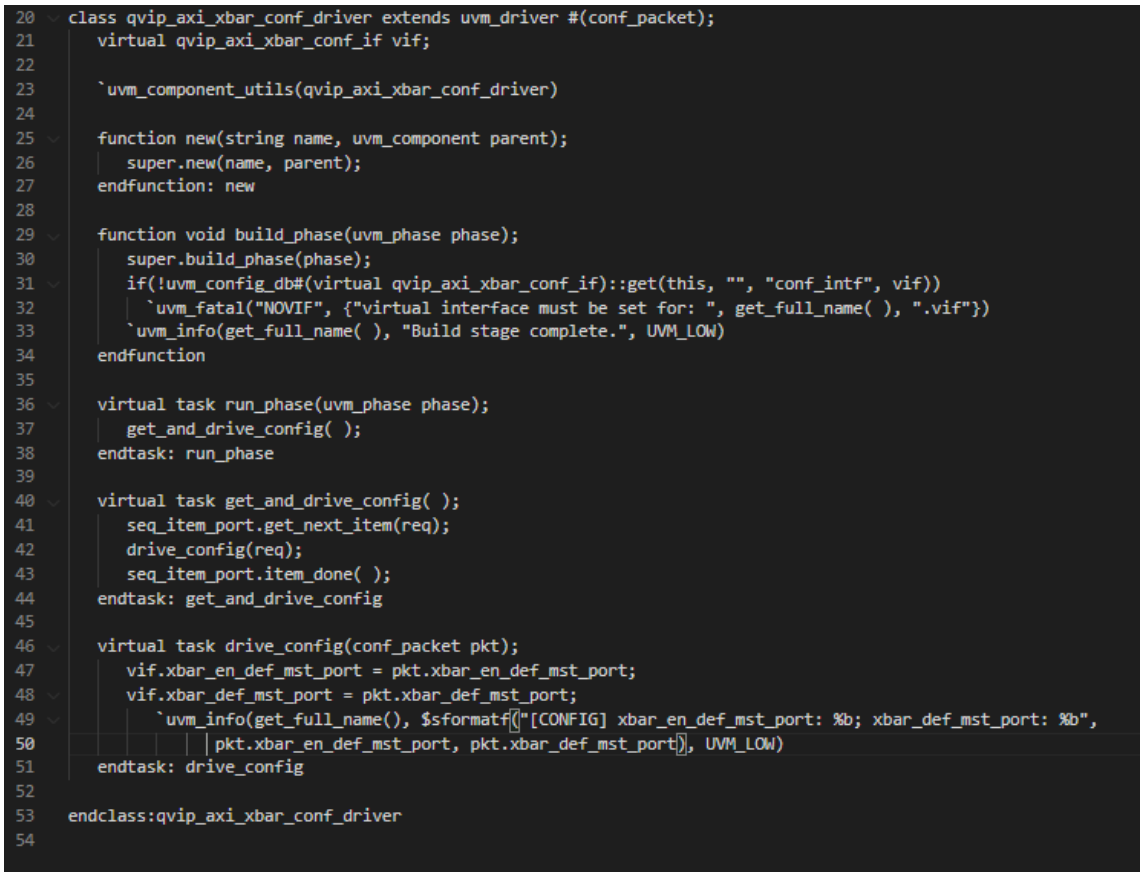
En la Figura 122 se muestra el código del componente *UVM Agent* definido y denominado `qvip_axi_xbar_conf_agent.svh`.

```
20  class qvip_axi_xbar_conf_agent extends uvm_agent;
21      protected uvm_active_passive_enum is_active = UVM_ACTIVE;
22
23      qvip_axi_xbar_conf_sequencer sequencer;
24      qvip_axi_xbar_conf_driver    driver;
25
26      `uvm_component_utils_begin(qvip_axi_xbar_conf_agent)
27          `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
28      `uvm_component_utils_end
29
30      function new(string name, uvm_component parent);
31          super.new(name, parent);
32      endfunction
33
34      function void build_phase(uvm_phase phase);
35          super.build_phase(phase);
36          if(is_active == UVM_ACTIVE) begin
37              sequencer = qvip_axi_xbar_conf_sequencer::type_id::create("sequencer", this);
38              driver = qvip_axi_xbar_conf_driver::type_id::create("driver", this);
39          end
40
41          `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
42      endfunction: build_phase
43
44      function void connect_phase(uvm_phase phase);
45          if(is_active == UVM_ACTIVE)
46              driver.seq_item_port.connect(sequencer.seq_item_export);
47          `uvm_info(get_full_name( ), "Connect stage complete.", UVM_LOW)
48      endfunction: connect_phase
49  endclass: qvip_axi_xbar_conf_agent
50
```

Figura 122 Módulo de configuración

Como se comentó anteriormente, el componente *UVM Agent* se compone únicamente de los módulos *UVM Sequencer* y *UVM Driver*, tal y como se muestra en las líneas 23 y 24. Comentar que este componente se ha declarado tipo *UVM_ACTIVE*, línea 21, para que incluya siempre a los componentes *UVM Sequence* y *UVM Drive*r.

La Figura 123 muestra el código correspondiente al módulo *UVM Driver*, definido en el fichero qvip_*axi_xbar*_conf_*Driver*.svh. Hay que destacar el uso que se hace de la interface virtual, línea 31, y las diferentes fases propias de UVM para la creación del componente *UVM driver*, tales como `build_phase` y `run_phase`.



```
class qvip_axi_xbar_conf_driver extends uvm_driver #(conf_packet);
   virtual qvip_axi_xbar_conf_if vif;

   `uvm_component_utils(qvip_axi_xbar_conf_driver)

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

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if(!uvm_config_db#(virtual qvip_axi_xbar_conf_if)::get(this, "", "conf_intf", vif))
        `uvm_fatal("NOVIF", {"virtual interface must be set for: ", get_full_name( ), ".vif"})
      `uvm_info(get_full_name( ), "Build stage complete.", UVM_LOW)
   endfunction

   virtual task run_phase(uvm_phase phase);
      get_and_drive_config( );
   endtask: run_phase

   virtual task get_and_drive_config( );
      seq_item_port.get_next_item(req);
      drive_config(req);
      seq_item_port.item_done( );
   endtask: get_and_drive_config

   virtual task drive_config(conf_packet pkt);
      vif.xbar_en_def_mst_port = pkt.xbar_en_def_mst_port;
      vif.xbar_def_mst_port = pkt.xbar_def_mst_port;
         `uvm_info(get_full_name(), $sformatf("[CONFIG] xbar_en_def_mst_port: %b; xbar_def_mst_port: %b",
                pkt.xbar_en_def_mst_port, pkt.xbar_def_mst_port), UVM_LOW)
   endtask: drive_config

endclass:qvip_axi_xbar_conf_driver

```

Figura 123 Componete UVM Driver de configuración

En la Figura 124 se muestra el código del componente *UVM Sequencer*, definido en el fichero `qvip_axi_xbar_conf_Sequencer.svh`

```
class qvip_axi_xbar_conf_sequencer extends uvm_sequencer #(conf_packet);

   `uvm_sequencer_utils(qvip_axi_xbar_conf_sequencer)

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

```

Figura 124 Componente UVM Sequencer de configuración

6.2.4. COMPONENTE ENVIOREMENT CONFIG

En este módulo, que se muestra en la Figura 125, se detalla el código del componente *UVM Enviorement* de configuración, en el fichero `qvip_axi_xbar_env_config.svh`. En este componente se realiza una inicialización de los rangos de dirección de memoria y se listan los marcadores *handles* de configuración que se usarán en el proceso.

```
class qvip_axi_xbar_env_config extends uvm_object;
    `uvm_object_utils(qvip_axi_xbar_env_config)

    // Handles for vip config for each of the QVIP instances
    axi4_master0_cfg_t axi4_master0_cfg;
    axi4_master1_cfg_t axi4_master1_cfg;
    axi4_master2_cfg_t axi4_master2_cfg;
    axi4_master3_cfg_t axi4_master3_cfg;
    axi4_master4_cfg_t axi4_master4_cfg;
    axi4_master5_cfg_t axi4_master5_cfg;
    axi4_slave0_cfg_t axi4_slave0_cfg;
    axi4_slave1_cfg_t axi4_slave1_cfg;
    axi4_slave2_cfg_t axi4_slave2_cfg;
    axi4_slave3_cfg_t axi4_slave3_cfg;
    axi4_slave4_cfg_t axi4_slave4_cfg;
    axi4_slave5_cfg_t axi4_slave5_cfg;
    axi4_slave6_cfg_t axi4_slave6_cfg;
    axi4_slave7_cfg_t axi4_slave7_cfg;

    // Handles for address maps
    address_map addr_map;

    function new
    ( string name = "qvip_axi_xbar_env_config"  );
        super.new(name);
    endfunction

   extern function void initialize;

endclass: qvip_axi_xbar_env_config

function void qvip_axi_xbar_env_config::initialize;
    begin
        addr_map_entry_s addr_map_entries[] = new [8];
        addr_map_entries = '{'{MAP_NORMAL,"RANGE_1",0,MAP_NS,4'h0,32'h0000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_2",0,MAP_NS,4'h0,32'h1000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_3",0,MAP_NS,4'h0,32'h2000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_4",0,MAP_NS,4'h0,32'h3000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_5",0,MAP_NS,4'h0,32'h4000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_6",0,MAP_NS,4'h0,32'h5000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_7",0,MAP_NS,4'h0,32'h6000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
                            '{MAP_NORMAL,"RANGE_8",0,MAP_NS,4'h0,32'h7000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA}
                           };
        addr_map = address_map::type_id::create("addr_map_addr_map");
        addr_map.addr_mask = 64'hFFF;
        addr_map.set( addr_map_entries );
    end
endfunction: initialize
```

Figura 125 Componente UVM Environment de configuración

En total, se listan 6 manejadores de configuración para los maestros y un total de 8 manejadores de configuración para los esclavos. Cada uno de estos objetos definen los parámetros particulares de su puerto. El cuerpo de esta clase contiene, además del constructor `new`, su función de inicialización, denominada *`Initialize`*, que establecerá los rangos de dirección, y sus respectivos atributos, para cada entrada en el mapa de direcciones.

6.2.5. COMPONENTE TEST_BASE

La Figura 126 y la Figura 127 muestra el código modificado del fichero `qvip_axi_xbar_test_base.svh`. En primer lugar, en la línea 219 se referencia la secuencia de verificación que se ejecutará en el código inicial para la verificación del DUT. En segundo lugar, ahora se referencia en la línea 220, la nueva secuencia de configuración, `qvip_axi_xbar_data_conf_seq`.

```
213  task qvip_axi_xbar_test_base::run_phase
214    (
215     uvm_phase phase
216     );
217
218     string seq_names[$];
219     qvip_axi_xbar_vseq_base vseq[];
220     qvip_axi_xbar_data_conf_seq cseq;
221
222     uvm_cmdline_processor clp;
223
224     uvm_factory factory = uvm_factory::get();
225
226     clp = uvm_cmdline_processor::get_inst();
227
228     if (clp.get_arg_values("+SEQ=", seq_names) == 0)
229       `uvm_fatal("TEST/SEQ/ARG_MISSING",
230       "You must specify at least one sequence to run using the +SEQ plusarg")
231
232     vseq = new[seq_names.size()];
233     // Qualify that each +SEQ specifies a sequence...
234
235     foreach (seq_names[i]) begin
236        uvm_object obj;
237        obj = factory.create_object_by_name(seq_names[i]);
238
239        if (obj == null)
240     `uvm_fatal("TEST/SEQ/NOT_IN_FACTORY",
241           {"Sequence '",seq_names[i],"' not found in factory.",
242            " Was it declared with a `uvm_*_utils macro?"})
243        if (!$cast(vseq[i], obj))
244     `uvm_fatal("TEST/SEQ/NOT_A_SEQ",
245           {"Sequence '",seq_names[i],"' is not a sequence"})
246     end // foreach (seq_names[i])
247
```

Figura 126 Módulo test base con secuencia de configuración

A continuación, dentro del `foreach` de la línea 235, se crean las secuencias y se

comprueba que las secuencias son correctas, y que están definidas en la *Factory.* Posteriormente, y en el `foreach` de la línea 254 de la Figura 127 se ejecuta la secuencia invocando al método *`start`* de la secuencia. En el caso de la secuencia de configuración solo se tiene una secuencia, mostrada en las líneas 251 y 252, donde se ejecuta el método `start` de la misma. A continuación, en las líneas 254 a 260, se invoca el método `start` de las secuencias de datos, asegurando de esta manera que la configuración ha sido correctamente establecida.

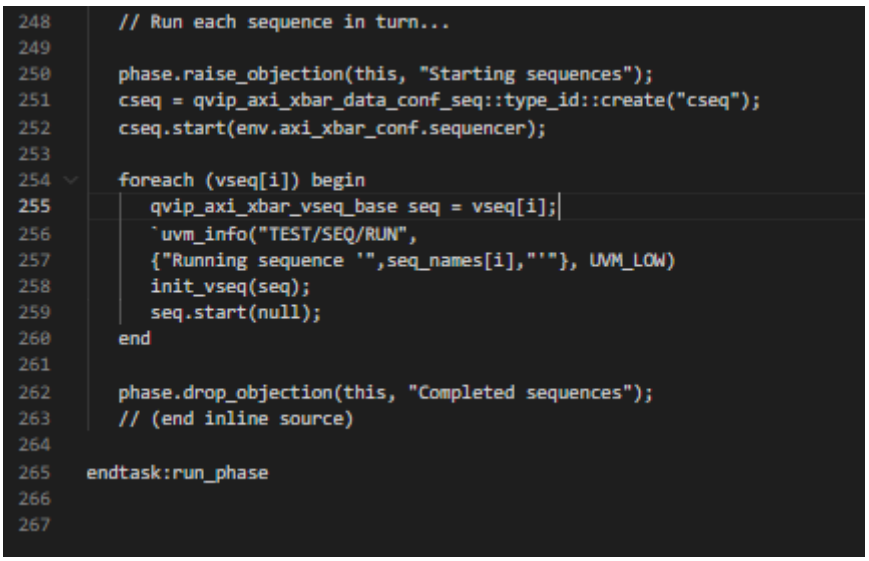
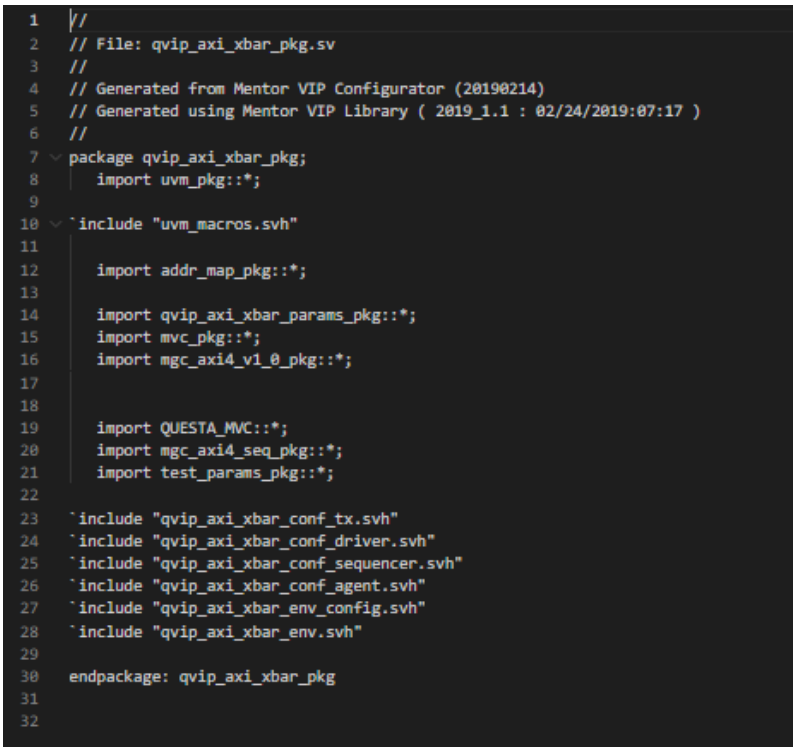


Figura 127 Módulo test_base con secuencia de configuración cont.

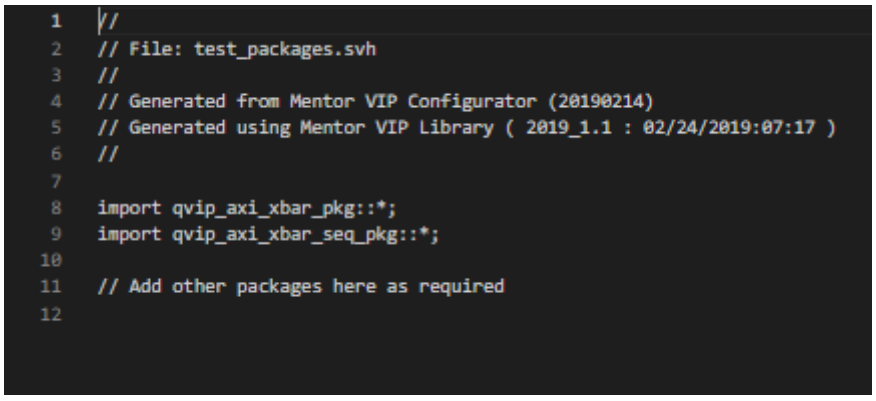
9.2.6. FICHEROS NECESARIOS

Para la ejecución de este nuevo *testbech* se deben modificar dos archivos importantes, como son el archivo `qvip_axi_xbar_pkg.sv` y el archivo `test_packages.svh` que se muestran en la Figura 128 y Figura 129. Estos ficheros deben incluir e importar todos los módulos anteriormente mostrados.



```
//
// File: qvip_axi_xbar_pkg.sv
//
// Generated from Mentor VIP Configurator (20190214)
// Generated using Mentor VIP Library ( 2019_1.1 : 02/24/2019:07:17 )
//
package qvip_axi_xbar_pkg;
   import uvm_pkg::*;

`include "uvm_macros.svh"

   import addr_map_pkg::*;

   import qvip_axi_xbar_params_pkg::*;
   import mvc_pkg::*;
   import mgc_axi4_v1_0_pkg::*;


   import QUESTA_MVC::*;
   import mgc_axi4_seq_pkg::*;
   import test_params_pkg::*;

`include "qvip_axi_xbar_conf_tx.svh"
`include "qvip_axi_xbar_conf_driver.svh"
`include "qvip_axi_xbar_conf_sequencer.svh"
`include "qvip_axi_xbar_conf_agent.svh"
`include "qvip_axi_xbar_env_config.svh"
`include "qvip_axi_xbar_env.svh"

endpackage: qvip_axi_xbar_pkg
```

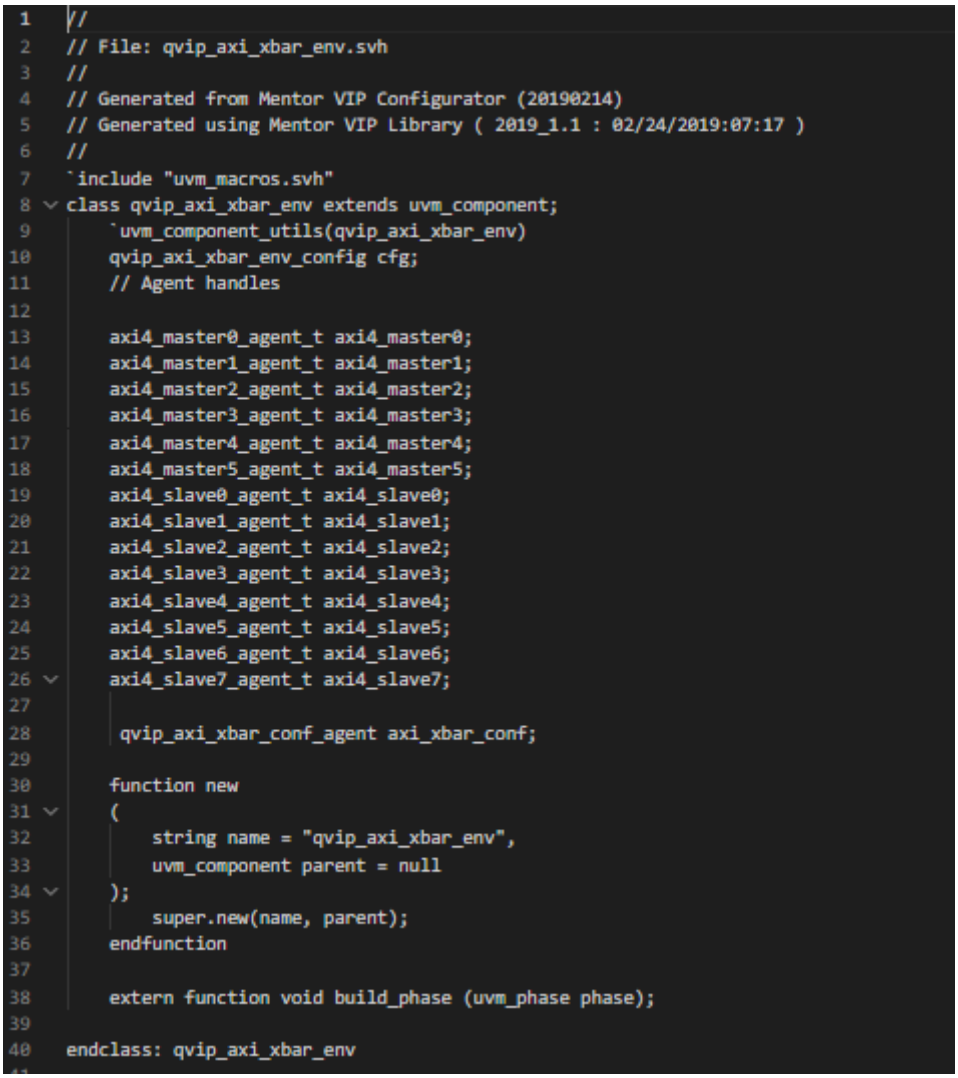
Figura 128 Código modificado del ficheroqvip_axi_xbar_pkg.sv



```
//
// File: test_packages.svh
//
// Generated from Mentor VIP Configurator (20190214)
// Generated using Mentor VIP Library ( 2019_1.1 : 02/24/2019:07:17 )
//

import qvip_axi_xbar_pkg::*;
import qvip_axi_xbar_seq_pkg::*;

// Add other packages here as required
```

Figura 129 Código modificado del fichero test_packages.svh

Como último paso se debe añadir el agente de configuración en la descripción del entorno realizado en el archivo `qvip_axi_xbar_env.svh`, tal y como se muestra en la línea 28 de la Figura 130.



```
//
// File: qvip_axi_xbar_env.svh
//
// Generated from Mentor VIP Configurator (20190214)
// Generated using Mentor VIP Library ( 2019_1.1 : 02/24/2019:07:17 )
//
`include "uvm_macros.svh"
class qvip_axi_xbar_env extends uvm_component;
    `uvm_component_utils(qvip_axi_xbar_env)
    qvip_axi_xbar_env_config cfg;
    // Agent handles

    axi4_master0_agent_t axi4_master0;
    axi4_master1_agent_t axi4_master1;
    axi4_master2_agent_t axi4_master2;
    axi4_master3_agent_t axi4_master3;
    axi4_master4_agent_t axi4_master4;
    axi4_master5_agent_t axi4_master5;
    axi4_slave0_agent_t axi4_slave0;
    axi4_slave1_agent_t axi4_slave1;
    axi4_slave2_agent_t axi4_slave2;
    axi4_slave3_agent_t axi4_slave3;
    axi4_slave4_agent_t axi4_slave4;
    axi4_slave5_agent_t axi4_slave5;
    axi4_slave6_agent_t axi4_slave6;
    axi4_slave7_agent_t axi4_slave7;

    qvip_axi_xbar_conf_agent axi_xbar_conf;

    function new
    (
        string name = "qvip_axi_xbar_env",
        uvm_component parent = null
    );
        super.new(name, parent);
    endfunction

    extern function void build_phase (uvm_phase phase);

endclass: qvip_axi_xbar_env
```

Figura 130 Código modificado del fichero `qvip_axi_xbar_env.svh`

6.2.6. SECUENCIA DE CONFIGURACION

A continuación, se muestra el código desarrollado para describir la secuencia de configuración. En la Figura 131 se muestra el código de las secuencias de configuración base, denominada `qvip_axi_xbar_conf_base_seq`. Esta secuencia se deriva de la secuencia base `uvm_sequence`. Tiene como parámetro la transacción de configuración descrita al comienzo de este apartado. La secuencia base únicamente registra el tipo en el *Factory*, define su constructor y declara su tarea principal `body`().

```
19  class qvip_axi_xbar_conf_base_seq extends uvm_sequence #(conf_packet);
20      `uvm_object_utils(qvip_axi_xbar_conf_base_seq)
21
22      function new(string name = "qvip_axi_xbar_conf_base_seq");
23          super.new(name);
24      endfunction: new
25
26      virtual task body();
27      endtask: body
28
29  endclass: qvip_axi_xbar_conf_base_seq
30
```

Figura 131 Código de la secuencia

La Figura 132 muestra una de las tramas de configuración que deriva de la tarea base, en la línea 59. Una vez creada la transacción, se definen las señales de habilitación y de puerto por defecto. Estas variables fueron definidas del tipo aleatorio y lo que se hace es la llamada al método `ramdomize` restringiendo sus valores con el uso del comando `with`.

```
48  class qvip_axi_xbar_data_conf_seq extends qvip_axi_xbar_conf_base_seq;
49      `uvm_object_utils(qvip_axi_xbar_data_conf_seq)
50
51      function new(string name = "qvip_axi_xbar_data_conf_seq");
52          super.new(name);
53      endfunction: new
54
55      virtual task body();
56          super.body();
57          req = conf_packet::type_id::create("req");
58          start_item(req);
59          assert(req.randomize() with {xbar_en_def_mst_port == 6'h01; xbar_def_mst_port == 18'h0001;});
60          finish_item(req);
61      endtask: body
62
63  endclass: qvip_axi_xbar_data_conf_seq
64
```

Figura 132 Código de la secuencia de configuración derivada.

6.2.7. RESULTADOS OBTENIDOS PARA LA SECUENCIA DE CONFIGURACIÓN

En este apartado se muestran los resultados de ejecutar la secuencia de configuración. En este caso, se ha ejecutado la secuencia listada en la Figura 134. En esta secuencia se asignan a las señales de configuración los siguientes valores.

- `xbar_en_def_mst_port =6'h01`
- `xbar_def_mst_port = 18'h0001`

```
48  class qvip_axi_xbar_data_conf_seq extends qvip_axi_xbar_conf_base_seq;
49      `uvm_object_utils(qvip_axi_xbar_data_conf_seq)
50
51      function new(string name = "qvip_axi_xbar_data_conf_seq");
52          super.new(name);
53      endfunction: new
54
55      virtual task body();
56          super.body();
57          req = conf_packet::type_id::create("req");
58          start_item(req);
59          assert(req.randomize() with {xbar_en_def_mst_port == 6'h01; xbar_def_mst_port == 18'h0001;});
60          finish_item(req);
61      endtask: body
62
63  endclass: qvip_axi_xbar_data_conf_seq
64
```

Figura 133 Secuencia de configuración ejecutada

Para comprobar el resultado de la ejecución de la secuencia de configuración, basta con analizar dos señales en el interfaz de configuración. La Figura 134 muestra las señales del interfaz de configuración. Se comprueba que los valores que aparecen corresponden a los asignados en la secuencia.

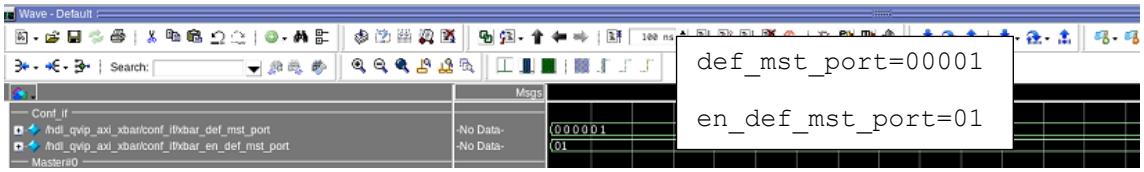


Figura 134 Señales del interfaz de configuración

A continuación, se muestran los resultados de ejecutar una segunda secuencia de configuración, en la que se asignan los siguientes valores:

- `xbar_en_def_mst_port =6'b000011`
- `xbar_def_mst_port = 18'b000_000_000_000_010_001`

La Figura 135, muestra la definición de la secuencia.

```
48  class qvip_axi_xbar_data_conf_seq extends qvip_axi_xbar_conf_base_seq;
49      `uvm_object_utils(qvip_axi_xbar_data_conf_seq)
50
51      function new(string name = "qvip_axi_xbar_data_conf_seq");
52          super.new(name);
53      endfunction: new
54
55      virtual task body();
56          super.body();
57          req = conf_packet::type_id::create("req");
58          start_item(req);
59          assert(req.randomize() with {xbar_en_def_mst_port == 6'b000011; xbar_def_mst_port == 18'b000_000_000_000_010_001;});
60          finish_item(req);
61      endtask: body
62
63  endclass: qvip_axi_xbar_data_conf_seq
64
```

Figura 135 Secuencia 2 de configuración ejecutada

La Figura 136 muestra las señales del interfaz de configuración. Se comprueba una vez más que los valores que aparecen corresponden a los asignados en la secuencia 2.

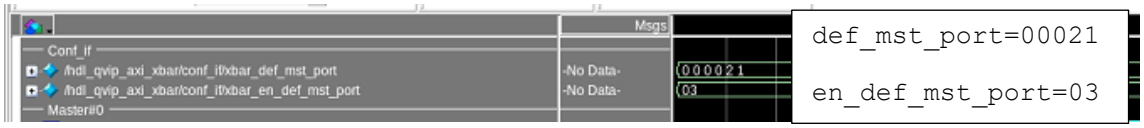


Figura 136 Señales de interfaz de configuración para la secuencia 2

CAPÍTULO 7: CONCLUSIONES Y LÍNEAS FUTURAS

CONCLUSIONES

Una vez finalizada la elaboración del Trabajo Fin de Grado denominado “Desarrollo de una librería de secuencias UVM para la verificación de un módulo *crossbar* con interfaz AXI-4 usando IP de verificación de Mentor Graphics”, se comprueba que se han cumplido los objetivos prefijados en la propuesta del mismo. El principal objetivo de este TFG suponía la implementación de la librería de secuencia UVM que verificaran las características principales definidas en su hoja de características, y utilizando un entorno de verificación UVM con la reutilización del código de los componentes programados.

El DUV estudiado por sus características, cuyo comportamiento se verificó a lo largo del desarrollo del presente TFG, es un módulo proporcionado por un tercero, en concreto, un módulo IP (*crossbar*). La principal función de este sistema es la conexión simultánea de todos los puertos de entrada y salida. El procedimiento seguido se basó en la estimulación de las entradas del DUV y en el exhaustivo análisis de las respuestas de la salida. De esta manera, se comprobó el comportamiento del dispositivo ante unos estímulos controlados. Para la realización de este TFG se partió de un entorno de referencia, desarrollado en el Trabajo Fin de Grado de Álvaro Moreno Florido [1], en el que se reutilizaban todas sus interfaces creadas y se procedió a generar las secuencias de test que permiten verificar todas y cada una de las características y propiedades detalladas en las especificaciones del módulo IP (*crossbar*).

La verificación del comportamiento del módulo *crossbar* se llevó a cabo siguiendo un enfoque *black-box*, por lo que no se profundizó en el contenido del módulo ni en su implementación.

Respecto a los objetivos parciales establecidos inicialmente, se considera haberlos logrado en su totalidad. Los conocimientos adquiridos en el estudio de la metodología UVM han sido suficientes para el desarrollo de la librería de secuencias especificada. Se realizó un análisis en profundidad de los elementos que llevaban a cabo las transacciones entre los componentes del entorno de verificación, lo que resultó en un conocimiento más profundo de dichos elementos, y elcomprender las funciones de cada uno de ellos en el proceso de verificación UVM.

Toda la documentación generada en la elaboración este TFG resultó de gran utilidad en el estudio del IP y del entorno de referencia, lo que permitió realizar correctamente la

librería de secuencias, objetivo principal de este TFG.

El procedimiento seguido en la realización de este Trabajo Fin de Grado ha sido el eje y método de la redacción del presente documento, memoria, siendo la secuencia definida por los capítulos similar a la seguida en el desarrollo real. De esta manera, en primer lugar, se estudiaron los aspectos generales de la métodología UVM para, en capítulos posteriores, estar capacitado para profundizar en el estudio y manejo del módulo Questa Verification IP de verificación que facilitó la realización del *testbench* de forma más sencilla. Una vez finalizada esta etapa, el siguiente paso fue conocer con detalle el DUV a verificar en este trabajo y analizar en detalle las principales características del protocolo de comunicación utilizado.

Hay que destacar que gran parte del tiempo destinado a la realización del TFG se invirtió en el estudio de la metodología, al tratarse de conocimientos complejos y a la vez muy abstractos, que solo tienen sentido al tener una concepción global de ellos. Tras conocer la metodología, esta se aplicó en el estudio del entorno de verificación de referencia, así como en el estudio del IP partiendo de la documentación generada para la realización del TFG [1] .

Una vez finalizada la etapa de estudio, se implementó la librería de secuencias con su posterior comprobación, etapa que centraba el objetivo principal de este TFG. Ante la necesidad de poder controlar señales del DUV que no intervenían directamente como parte del protocolo de comunicación sino, que estaban asociadas a la configuración inicial que gestionaba dicha comunicación, se decidió analizar la forma de poder llevar a cabo la configuración del DUV sin necesidad de volver a modificar el código cada vez que se introducía una nueva configuración. Este hecho permitiría, además, reducir el tiempo de compilación al realizar la configuración en tiempo de ejecución. Tras analizar la interfaz de configuración, se propuso una mejora del TFG, que derivó en la tarea descrita en el capítulo 6. Esta adaptación se propone como una mejora, no contemplada en un comienzo dentro de los objetivos iniciales de este TFG, pero que se realizó con éxito también.

A la vista de los resultados obtenidos, el planteamiento propuesto desde un principio sobre el Trabajo Fin de Grado ha sido correcto, trabajando en todo momento sobre la planificación y no teniendo que realizar acciones correctoras sobre dicha planificación, incluso habiendo incluido la mejora propuesta, pudiendo adaptar los tiempos sin ningún problema. Según lo comentado, en el tiempo designado se ha logrado el objetivo principal del TFG.

En todo momento se ha seguido una correcta aproximación al problema. Siguiendo las indicaciones de los tutores de este TFG, como se constató por la complejidad del aprendizaje del estudio de la metodología UVM y del IP de referencia, siendo los conocimientos adquiridos fundamentales posteriormente en el desarrollo final de este TFG.

Líneas Futuras

El presente TFG supone la continuación del TFG de Álvaro Moreno Florido [1]. El objetivo principal de ese TFG avanza en una línea que se considera “terminal” en el ciclo de diseño de un producto. Por este motivo, en el presente trabajo no se consideran propuestas de líneas futuras.

BIBLIOGRAFÍA

[1] Á. José Moreno Florido, “Desarrollo de un testbench UVM integrando IP de verificación de Mentor Graphics (QVIP)”. TFG Universidad de Las Palmas de Gran Canaria, 2021.

[2] Siemens,
“The 2020 Wilson Research Group Functional Verification Study | Verification Horizons”. Accessed: Oct.17.2021. [Online]. Available: *https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020- wilson-research-group-functional-verification-study/ .2021*

[3] D. I. Rich, “The evolution of systemverilog”. *IEEE Design Test of Computers*, vol. 20, no. 4, pp. 82–84, 2003, doi: 10.1109/MDT.2003.1214355. Accessed: Nov.20.2021.

[4] Siemens, “Part 6: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons”. Accessed: Oct.20.2021. [Online]. Available: *https://blogs.sw.siemens.com/verificationhorizons/2020/12/16/part-6-the-2020-wilson-research-group-functional-verification-study/* , 2021.

[5] Siemens, “Verification IP - Siemens EDA”. Accessed: Jan.10.2022. [Online]. Available: *https://www.mentor.com/products/fv/verification-ip* , 2022.

[6] Asicguru,
“Introduction To System Verilog | System Verilog Tutorial | System Verilog”. Accessed: Jan.1.2022. [Online]. Available: *http://www.asicguru.com/system-verilog/tutorial/introduction/1/* , 2022.

[7] Alvaro Ravelo Medero, “Desarrollo de la capa RAL en un Entorno UVM para la verificación funcional de un IP multi-interfaz orientado a la compresión de imágenes”. TFG Universidad de Las Palmas de Gran Canaria, 2021

[8] Accellera.org, “Universal Verification Methodology (UVM) 1.2 User’s Guide” 2015. Accessed: Feb.17.2022. [Online]. Available: *https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2. pdf.*, 2022.

[9] Vanessa R. cooper, “Getting started with UVM”. *Verilab Publishing*, Austin Texas,USA. 2013. ISBN-13:978-0615819976

[10] “UVM Phasing - The Art of Verification”. Accessed: Feb.28.2022. [Online]. Available: *https://www.theartofverification.com/uvm- phasing/*, 2022.

[11] Accelera.org, “System C TLM”. Accessed: Mar.14.2022. [Online]. Available:*” https://www.accellera.org/community/systemc/about-systemc-tlm*, 2022.

[12] Chipverify.com, “UVM Introduction”. Accessed: Feb.28.2022. [Online]. Available: *https://www.chipverify.com/uvm/uvm-introduction* , 2022.

[13] AMBA AXI, “AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite”. Accessed: Mar.10.2022. [Online]. Available: *https://developer.arm.com/documentation/ihi0022/e/*, 2022.

[14] ARM, “AMBA AXI and ACE Protocol Specification”. Accessed: Mar.01.2022. [Online]. Available: *www.arm.com*, 2022.

[15] M. Graphics Corporation, “Questa ® Verification IP User Guide for the ARM ® AMBA ® 4 AXI Protocol” 2019. Accessed: Feb.28.2022. [Online].

[16] Siemens, “Getting Started with Questa Verification IP for Protocols | Verification Horizons”. Accessed: Apr.10.2022. [Online]. Available: *https://blogs.sw.siemens.com/verificationhorizons/2021/02/18/getting-started-with- questa-verification-ip-for-protocols/* , 2022.

[17] A. | E. Z. Kurth, “Kurth, Andreas | ETH Zurich”. Accessed: Jan.15.2022. [Online]. Available: *https://ee.ethz.ch/the-department/people-a-z/person-detail.MTgwNzQz.TGlzdC8zMjc5LC0xNjUwNTg5ODIw.html* , 2022.

[18] GitHub - pulp-platform/axi, “GitHub - pulp-platform/axi: AXI SystemVerilog synthesizable IP modules and verification infrastructure for high-performance on-chip communication”. Accessed: Feb.28.2022. [Online]. Available: *https://github.com/pulp-platform/axi* , 2022.

[19] Andreas kurth, “AXI4+ATOP Fully-Connected Crossbar”. Accessed: Feb. 28, 2022. [Online]. Available: https://github.com/pulp-platform/axi

[20] Siemens, “Questa Advanced Simulator | Siemens Digital Industries Software”. Accessed: Feb.28.2022. [Online]. Available: *https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/* , 2022.

[21] WordPress, “UVM Driver and Sequencer Communication | Universal Verification Methodology”. Accessed: Feb.28.2022. [Online]. Available: *https://www.learnuvmverification.com/index.php/2015/07/07/*, 2022.

PLIEGO DE CONDICIONES

PC1. CONDICIONES HARDWARE

A continuación, se muestran los equipos y dispositivos *hardware* utilizados en el desarrollo de este TFG, en la Tabla 10.

Tabla 10 Equipos y dispositivos hardware

Equipo/Dispositivo	Modelo	Fabricante/Comerciante
Ordenador Personal Portátil	Lenovo V15-ADA • CPU AMD 3020E. • 8 GB RAM • 520 GB SSD	Lenovo
Ordenador Personal Sobremesa	All in one HP 22-DF0082 • CPU intel Celeron • 8 GB RAM • 256 GB SSD	HP
Estación de Trabajo	Acer Predator G3: • CPU Intel Core i7-4790. • 8 GB RAM • 1 TB HDD • 4 puertos USB	Acer Inc.

PC2. CONDICIONES SOFTWARE

A continuación, se expone el software utilizado en la Tabla 11:

Tabla 11 Tabla de software utilizado

Equipo/Dispositivo	Modelo	Fabricante/Comerciante
Sistema Operativo	Microsoft Windows 10 Home 64 bits.	Microsoft Corporation
Microsoft Office	365	Microsoft Corporation
Microsoft Project	2016	Microsoft Corporation
Google Chrome	Versión 98.0.4758.102 (Build oficial) (64 bits)	Google LLC
Sistema Operativo	Red Hat Enterprise Linux Server	Red Hat Inc.
Simulador y compilador QuestaSim	v10.4b - Linux	Mentor Graphics Corporation
UVM	v1.1d	Cadence, Mentor Graphics, Aldec, Synopsys.
Adobe Acrobat e Pro	Versión 10.1.16	Adobe Systems Incorporated
Adobe Ilustrator	Version CC	Adobe Systems Incorporated

PRESUPUESTO

Para llevar a cabo este Trabajo Fin de Grado, al igual que cualquier otro proyecto, se hace necesario el uso de diferentes recursos. Para cada uno de ellos se puede estimar la cuantía de su valor en referencia a su naturaleza y a la aportación que realiza al proyecto. Se exponen en esta sección del documento los recursos materiales y humanos empleados.

El presupuesto total se desglosa en los siguientes conceptos:

- Coste de Recursos Humanos.
- Coste de Recursos Materiales.
- Coste de Recursos Hardware.
- Coste de Recursos Software.
- Coste de Material Fungible.
- Coste de Redacción del Trabajo.
- Derechos de Visado del COITT.
- Gastos de Tramitación y Envío.
- Coste Total.

P.1. RECURSOS HUMANOS

Para determinar el coste asociado a las horas de trabajo empleadas en la realización del Trabajo Fin de Grado se utiliza la siguiente ecuación, correspondiente al cálculo de honorarios en función de las horas trabajadas dentro y fuera de la jornada laboral convencional. Esta ecuación es la recomendada por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT) para el cálculo de los costes de Recursos Humanos (RRHH):

$$\text{Honorarios}\ (€) = Ct \cdot (74{,}88 \cdot Hn + 96{,}72 \cdot He)$$

- El término Hn define el número de horas trabajadas dentro de la normalidad, es decir, en la jornada laboral convencional.
- El parámetro He hace referencia a las horas especiales u horas extra fuera de la jornada laboral.
- El término Ct hace referencia a un factor de corrección aplicado según la totalidad de horas invertidas al proyecto.

Los factores de corrección posibles para el término Ct se muestran a continuación en la

Tabla 12 de acuerdo con el procedimiento de cálculo establecido por el COITT.

Tabla 12 Factor de corrección calculo horas invertidas al proyecto

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

Teniendo en cuenta que el número de horas destinado al TFG según el Plan de Estudios del título Grado en Ingeniería en Tecnologías de la Telecomunicación es de 300 horas (12 ECTS), se aplicará un factor de corrección de 0,6. Estas horas se cuentan dentro de la jornada laboral, basando esta decisión en la planificación horaria elaborada al principio del desarrollo del correspondiente proyecto. La ecuación antes mostrada daría el siguiente resultado.

$$Honorarios(€) = 0{,}6 \cdot (74{,}88 \cdot 300 + 96{,}72 \cdot 0) = 13.478{,}40\ €$$

Los honorarios resultantes del tiempo invertido en términos de RRHH se eleva hasta los trece mil cuatrocientos setenta y ocho euros con cuarenta céntimos (13.478,40 €), sin haberse aplicado impuestos ni retenciones.

P.2. RECURSOS MATERIALES

En la realización del presente TFG se ha hecho uso tanto de herramientas software como de herramientas hardware. Para realizar un cálculo del coste de cada uno de estos recursos se debe relacionar su coste total con el tiempo útil de los mismos. Para este cálculo se hace uso de la siguiente ecuación (fijando como referencia para el sistema de amortización lineal una vida útil de 3 años).

$$Cuota = (Valor\ de\ la\ adquisición - Valor\ residual)/Tiempo\ de\ vida\ útil$$

P.2.1. RECURSOS HARDWARE

En la Tabla 13 se expone la amortización de los recursos hardware utilizados en el desarrollo del presente TFG. Para todos ellos el tiempo de uso ha sido de 4 meses.

Tabla 13 Recursos hardware

Equipo/Dispositivo	Valor de adquisición	Amortización
Ordenador Personal portátil	450,00 €	50,00 €
Ordenador Personal Sobremesa	500,00 €	55,55 €
Estación de trabajo	994,00 €	110,44
	TOTAL	215,99 €

El coste de los recursos materiales relativos a herramientas hardware se eleva entonces a los doscientos quince euros con noventa y nueve céntimos (215,99 €).

P.2.2. RECURSOS SOFTWARE

En la Tabla 14 se expone la amortización de los recursos software utilizados en el desarrollo del presente TFG. Para todos ellos el tiempo de uso ha sido de 4 meses. Se considera relevante señalar que para la gran mayoría de programas de Microsoft la licencia ha sido proporcionada de forma gratuita por la ULPGC.

Tabla 14 Amortización recurso software

Equipo/Dispositivo	Valor de adquisición	Amortización
Sistema Operativo	145,00 €	16,11 €
Microsoft Office	0,00 € (licencia)	0,00 €
Microsoft Project	0,00 € (licencia)	0,00 €
Google Chrome	0,00 € (Software libre)	0,00 €
Sistema Operativo	0,00 € Software libre)	0,00 €
Simulador y compilador QuestaSim	20.000,00 €	2.222,22 €
UVM	0,00 € (Librería de libre distribución)	0,00 €
Adobe Acrobat e Pro	0,00 € (licencia)	0,00 €
Adobe Ilustrator	0,00 € (licencia)	0,00 €
	TOTAL	2.238,33 €

El coste de los recursos materiales relativos a herramientas software se eleva entonces a los dos mil doscientos treinta y ocho euros con treinta y tres céntimos (2.238,33 €).

P.3. MATERIAL FUNGIBLE

Los costes derivados del material fungible empleado en el desarrollo de este Trabajo Fin de Grado se exponen en la Tabla 15.

Tabla 15 Material Fungible

Concepto	**Coste**
Folios	10,00 €
Impresión de TFG	70,00 €
Encuadernado	10,00 €
Tres CD- Rom	7,50 €
TOTAL	97,50 €

Así, los costes relativos al Material Fungible ascienden a los noventa y siete euros con cincuenta centimos (97,50 €).

P.4. REDACCIÓN DEL TRABAJO

Partiendo del presupuesto del proyecto, se pueden calcular los costes asociados a la redacción del trabajo, a partir de la siguiente ecuación.

$$R=0{,}07 \cdot P \cdot Cn$$

- El término P se refiere al presupuesto del proyecto.
- El parámetro Cn se utiliza como el coeficiente de corrección en función del presupuesto calculado.
- El término R representa los honorarios derivados de la redacción del trabajo.

Para obtener el valor del presupuesto (P), se suma el coste relativo a los RRHH y el relacionado con los recursos materiales, tanto software como hardware.

En la Tabla 16 se muestra el resultado obtenido para el presupuesto del proyecto.

Tabla 16 Presupuesto del proyecto

Concepto	Coste
Coste de RRHH empleados en la realización del TFG	13.478,40 €
Costes materiales (hardware)	215,99 €
Costes materiales (software)	2.238,33 €
TOTAL	15.932,72 €

Según el COITT, el ya mencionado coeficiente corrector Cn tendrá por valor la unidad si el presupuesto (P) tiene un valor inferior a los 30.050,00 €. De esta manera, la ecuación queda de la siguiente forma.

$$R = 0{,}07 \cdot 15.932{,}72 \cdot 1 = 1.115{,}29 \text{ €}$$

De esta manera, el coste derivado de la redacción del presente TFG es de mil ciento quince euros con veintinueve céntimos (1.115,29 €).

P.5. DERECHOS DE VISADO DEL COITT

Anualmente, el COITT da valor a los costes asociados a los derechos de visado por la ejecución de proyectos técnicos de carácter general. De esta manera, para el año 2022 dichos costes se calculan mediante la siguiente ecuación.

$$V=0{,}006\cdot P_1\cdot C_1+0{,}003\cdot P_2\cdot C_2$$

P_1 es el término referido al presupuesto del proyecto.

C_1 es un coeficiente reductor en función del presupuesto.

P_2 hace referencia al presupuesto de ejecución material correspondiente a la obra civil.

C_2 es un coeficiente de corrección en función del presupuesto P_2.

V hace referencia al coste final del visado.

Teniendo en cuenta las características particulares del presente TFG, se puede determinar que el coste asociado al presupuesto P_2 es de 0,00 €, por tanto, no se le aplicará el coeficiente C_2. Respecto al coeficiente de corrección C_1, este es nuevamente 1, debido a que el presupuesto P_1 no supera los 30.050,00 €. En este caso sumamos al presupuesto P_1 el coste asociado a la redacción del trabajo, obteniendo un valor de 17.048,01 €

$$V=0{,}006\cdot\ 17.048{,}01\cdot\ 1\ =102{,}29\ €$$

Por tanto, el coste asociado a los derechos de visado del presupuesto asciende a los ciento dos euros con veintinueve céntimos (102,29 €).

P.6. GASTOS DE TRAMITACIÓN Y ENVÍO

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

P.7. COSTE TOTAL DEL PROYECTO

Una vez obtenidos todos los costes asociados a las diferentes actividades que conforman el 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%), mostrándose el cálculo correspondiente en la Tabla 17.

Tabla 17 Coste total

Concepto	**Coste**
Honorarios por tiempo empleado	13.478,40 €
Recursos materiales Hardware	215,99 €
Recursos materiales Software	2.238,33 €
Material Fungibles	97,50 €
Costes de redacción	1.115,29 €
Derechos de visado del COITT	102,99 €
Costes de tramitación y envío	6,00 €
Subtotal	17.254,50 €
IGIC (7%)	1.207,82 €
TOTAL	18.462,32 €

El importe final al que asciende el presupuesto del Trabajo Fin de Grado “Desarrollo de una librería de secuencias UVM para la verificación de un módulo *crossbar* con interfaz AXI-4 usando IP de verificación de Mentor Graphics” es de dieciocho mil cuatrocientos sesenta y dos euros con treinta y dos céntimos (18.462,32 €).

Fdo.: D. Miguel Quevedo Rodríguez

En Las Palmas de Gran Canaria a 17 de mayo de 2022

ANEXOS

ANEXO I Hoja de características del *Crossbar*

pulp-platform / axi

Code | Issues 5 | Pull requests 9 | Actions | Projects | Security | Insights

master

axi / doc / axi_xbar.md

accuminium doc/axi_xbar: Improve doc on `LatencyMode` for multiple connected cro... ... History

1 contributor

Raw | Blame

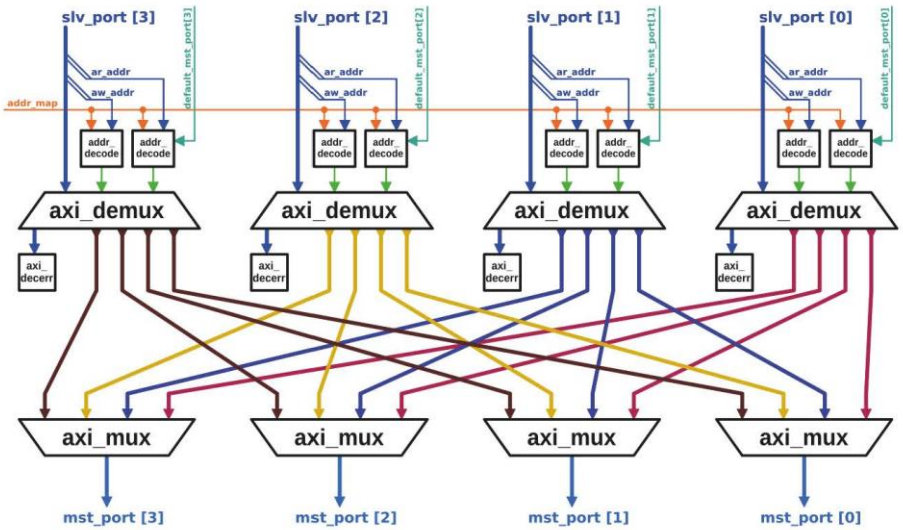
108 lines (65 sloc) | 10.8 KB

AXI4+ATOP Fully-Connected Crossbar

`axi_xbar` is a fully-connected crossbar that implements the full AXI4 specification plus atomic operations (ATOPs) from AXI5.

Design Overview

`axi_xbar` is a fully-connected crossbar, which means that each master module that is connected to a *slave port* for of the crossbar has direct wires to all slave modules that are connected to the *master ports* of the crossbar. A block-diagram of the crossbar is shown below:



The crossbar has a configurable number of slave and master ports.

The ID width of the master ports is wider than that of the slave ports. The additional ID bits are used by the internal multiplexers to route responses. The ID width of the master ports must be `AxiIdWidthSlvPorts + $clog_2(NoSlvPorts)`.

Address Map

One address map is shared by all master ports. The *address map* contains an arbitrary number of rules (but at least one). Each *rule* maps one address range to one master port. Multiple rules can map to the same master port. The address ranges of two rules may overlap: in case two address ranges overlap, the rule at the higher (more significant) position in the address map prevails.

Each address range includes the start address but does **not** include the end address. That is, an address *matches* an address range if and only if

```
addr >= start_addr && addr < end_addr
```

The start address must be less than or equal to the end address.

The address map can be defined and changed at run time (it is an input signal to the crossbar). However, the address map must not be changed while any AW or AR channel of any slave port is valid.

Decode Errors and Default Slave Port

Each slave port has its own internal *decode error slave* module. If the address of a transaction does not match any rule, the transaction is routed to that decode error slave module. That module absorbs each transaction and responds with a decode error (with the proper number of beats). The data of each read response beat is `32'hBADCAB1E` (zero-extended or truncated to match the data width).

Each slave port can have a default master port. If the default master port is enabled for a slave port, any address on that slave port that does not match any rule is routed to the default master port instead of the decode error slave. The default master port can be enabled and changed at run time (it is an input signal to the crossbar), and the same restrictions as for the address map apply.

Configuration

The crossbar is configured through the `Cfg` parameter with a `axi_pkg::xbar_cfg_t` struct. That struct has the following fields:

Name	Type	Definition
`NoSlvPorts`	`int unsigned`	The number of AXI slave ports of the crossbar (in other words, how many AXI master modules can be attached).
`NoMstPorts`	`int unsigned`	The number of AXI master ports of the crossbar (in other words, how many AXI slave modules can be attached).
`MaxMstTrans`	`int unsigned`	Each slave port can have at most this many transactions in flight.
`MaxSlvTrans`	`int unsigned`	Each master port can have at most this many transactions per ID in flight.
`FallThrough`	`bit`	Routing decisions on the AW channel fall through to the W channel. Enabling this allows the crossbar to accept a W beat in the same cycle as the corresponding AW beat, but it increases the combinatorial path of the W channel with logic from the AW channel.

Name	Type	Definition
`LatencyMode`	`enum logic [9:0]`	Latency on the individual channels, defined in detail in section *Pipelining and Latency* below.
`AxiIdWidthSlvPorts`	`int unsigned`	The AXI ID width of the slave ports.
`AxiIdUsedSlvPorts`	`int unsigned`	The number of slave port ID bits (starting at the least significant) the crossbar uses to determine the uniqueness of an AXI ID (see section *Ordering and Stalls* below). This value has to be less or equal than `AxiIdWidthSlvPorts`.
`AxiAddrWidth`	`int unsigned`	The AXI address width.
`AxiDataWidth`	`int unsigned`	The AXI data width.
`NoAddrRules`	`int unsigned`	The number of address map rules.

The other parameters are types to define the ports of the crossbar. The `*_chan_t` and `*_req_t` / `*_resp_t` types must be bound in accordance to the configuration with the `AXI_TYPEDEF` macros defined in `axi/typedef.svh`. The `rule_t` type must be bound to an address decoding rule with the same address width as in the configuration, and `axi_pkg` contains definitions for 64- and 32-bit addresses.

Pipelining and Latency

The `LatencyMode` parameter allows to insert spill registers after each channel (AW, W, B, AR, and R) of each master port (i.e., each multiplexer) and before each channel of each slave port (i.e., each demultiplexer). Spill registers cut combinatorial paths, so this parameter reduces the length of combinatorial paths through the crossbar.

Some common configurations are given in the `xbar_latency_e` `enum`. The recommended configuration (`CUT_ALL_AX`) is to have a latency of 2 on the AW and AR channels because these channels have the most combinatorial logic on them. Additionally, `FallThrough` should be set to `0` to prevent logic on the AW channel from extending combinatorial paths on the W channel. However, it is possible to run the crossbar in a fully combinatorial configuration by setting `LatencyMode` to `NO_LATENCY` and `FallThrough` to `1`.

If two crossbars are connected in both directions, meaning both have one of their master ports connected to a slave port of the other, the `LatencyMode` of both crossbars must be set to either `CUT_SLV_PORTS`, `CUT_MST_PORTS`, or `CUT_ALL_PORTS`. Any other latency mode will lead to timing loops on the uncut channels between the two crossbars. The latency mode of the two crossbars does not have to be identical.

Ports

Name	Description
`clk_i`	Clock to which all other signals (except `rst_ni`) are synchronous.
`rst_ni`	Reset, asynchronous, active-low.
`test_i`	Test mode enable (active-high).

Name	Description
`slv_ports_*`	Array of slave ports of the crossbar. The array index of each port is the index of the slave port. This index will be prepended to all requests at one of the master ports.
`mst_ports_*`	Array of master ports of the crossbar. The array index of each port is the index of the master port.
`addr_map_i`	Address map of the crossbar (see section *Address Map* above).
`en_default_mst_port_i`	One bit per slave port that defines whether the default master port is active for that slave port (see section *Decode Errors and Default Slave Port* above).
`default_mst_port_i`	One master port index per slave port that defines the default master port for that slave port (if active).

Ordering and Stalls

When one slave port receives two transactions with the same ID and direction (i.e., both read or both write) but targeting two different master ports, it will not accept the second transaction until the first has completed. During this time, the crossbar stalls the AR or AW channel of that slave port. To determine whether two transactions have the same ID, the `AxiIdUsedSlvPorts` least-significant bits are compared. That parameter can be set to the full `AxiIdWidthSlvPorts` to avoid false ID conflicts, or it can be set to a lower value to reduce area and delay at the cost of more false conflicts.

The reason for this ordering constraint is that AXI transactions with the same ID and direction must remain ordered. If this crossbar would forward both transactions described above, the second master port could get a response before the first one, and the crossbar would have to reorder the responses before returning them on the master port. However, for efficiency reasons, this crossbar does not have reorder buffers.

Verification Methodology

This module has been verified with a directed random verification testbench, described and implemented in `test/tb_axi_xbar.sv`.

Design Rationale for No Pipelining Inside Crossbar

Inserting spill registers between demuxers and muxers seems attractive to further reduce the length of combinatorial paths in the crossbar. However, this can lead to deadlocks in the W channel where two different muxes at the master ports would circular wait on two different demuxes (TODO). In fact, spill registers between the switching modules causes all four deadlock criteria to be met. Recall that the criteria are:

1. Mutual Exclusion
2. Hold and Wait
3. No Preemption
4. Circular Wait

The first criterion is given by the nature of a multiplexer on the W channel: all W beats have to arrive in the same order as the AW beats regardless of the ID at the slave module. Thus, the different master ports of the multiplexer exclude each other because the order is given by the arbitration tree of the AW channel.

The second and third criterion are inherent to the AXI protocol: For (2), the valid signal has to be held high until the ready signal goes high. For (3), AXI does not allow interleaving of W beats and requires W bursts to be in the same order as AW beats.

The fourth criterion is thus the only one that can be broken to prevent deadlocks. However, inserting a spill register between a master port of the demultiplexer and a slave port of the multiplexer can lead to a circular dependency inside the W FIFOs. This comes from the particular way the round robin arbiter from the AW channel in the multiplexer defines its priorities. It is constructed in a way by giving each of its slave ports an increasing priority and then comparing pairwise down till a winner is chosen. When the winner gets transferred, the priority state is advanced by one position, preventing starvation.

The problem can be shown with an example. Assume an arbitration tree with 10 inputs. Two requests want to be served in the same clock cycle. The one with the higher priority wins and the priority state advances. In the next cycle again the same two inputs have a request waiting. Again it is possible that the same port as last time wins as the priority shifted only one position further. This can lead in conjunction with the other arbitration trees in the other muxes of the crossbar to the circular dependencies inside the FIFOs. Removing the spill register between the demultiplexer and multiplexer forces the switching decision into the W FIFOs in the same clock cycle. This leads to a strict ordering of the switching decision, thus preventing the circular wait.

ANEXO II Secuencias del TFG

Secuencia 1

```
class qvip_axi_xbar_vseq_t1 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t1)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t1"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t1

task qvip_axi_xbar_vseq_t1::body;

    incr_wr_t   incr_wr0 = incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t   incr_rd0 = incr_rd_t::type_id::create("incr_rd0");

    incr_wr_t   incr_wr1 = incr_wr_t::type_id::create("incr_wr1");
    incr_rd_t   incr_rd1 = incr_rd_t::type_id::create("incr_rd1");

 super.body();


  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 0;
              if(!incr_wr0.randomize() with {addr == 32'h0000_4500;
      wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
    //master1 write
       incr_wr1.set_Sequencer(axi4_master1);
       incr_wr1.id = 1;
```

```
            if(!incr_wr1.randomize() with {addr == 32'h0000_4500;
       wr_data.size == 16;})
         `uvm_error(this.get_full_name(),"Randomisation failure");

    fork
      incr_wr0.start(axi4_master0);
      incr_wr1.start(axi4_master1);


    join
  end
endtask: body
```

Secuencia 2

```
class qvip_axi_xbar_vseq_t2 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t2)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t2"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t2

task qvip_axi_xbar_vseq_t2::body;

    incr_wr_t          incr_wr0          =
incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t          incr_rd0          =
incr_rd_t::type_id::create("incr_rd0");

    incr_wr_t          incr_wr1          =
incr_wr_t::type_id::create("incr_wr1");
    incr_rd_t          incr_rd1          =
incr_rd_t::type_id::create("incr_rd1");

    incr_wr_t          incr_wr2          =
incr_wr_t::type_id::create("incr_wr2");
    incr_rd_t          incr_rd2          =
incr_rd_t::type_id::create("incr_rd2");

    incr_wr_t          incr_wr3          =
incr_wr_t::type_id::create("incr_wr3");
    incr_rd_t          incr_rd3          =
incr_rd_t::type_id::create("incr_rd3");

    incr_wr_t          incr_wr4          =
incr_wr_t::type_id::create("incr_wr4");
```

```
    incr_rd_t          incr_rd4          =
incr_rd_t::type_id::create("incr_rd4");


 super.body();


  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 0;
       if(!incr_wr0.randomize() with {addr == 32'h0000_4500; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
    //master1 write
       incr_wr1.set_Sequencer(axi4_master1);
       incr_wr1.id = 1;
       if(!incr_wr1.randomize() with {addr == 32'h0000_4500; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");

    //master2 write
       incr_wr2.set_Sequencer(axi4_master2);
       incr_wr2.id = 2;
       if(!incr_wr2.randomize() with {addr == 32'h0000_4500; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");

    //master1 write
       incr_wr3.set_Sequencer(axi4_master1);
       incr_wr3.id = 3;
       if(!incr_wr3.randomize() with {addr == 32'h0000_8500; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
   //master2 write
       incr_wr4.set_Sequencer(axi4_master2);
       incr_wr4.id = 4;
       if(!incr_wr4.randomize() with {addr == 32'h0000_8500; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");


    fork
      incr_wr0.start(axi4_master0);
      incr_wr1.start(axi4_master1);
      incr_wr2.start(axi4_master2);
    join

    fork
```

```
      incr_wr3.start(axi4_master1);
      incr_wr4.start(axi4_master2);
    join


  end
endtask: body
```

Secuencia 3

```
class qvip_axi_xbar_vseq_t3 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t3)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t3"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t3

task qvip_axi_xbar_vseq_t3::body;

    incr_wr_t          incr_wr0          =
incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t          incr_rd0          =
incr_rd_t::type_id::create("incr_rd0");



 super.body();


  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 3;
       if(!incr_wr0.randomize() with {addr == 32'h0000_8000; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");


    fork
      incr_wr0.start(axi4_master0);

    join
```

```
  end
endtask: body
```

Secuencia 4

```
class qvip_axi_xbar_vseq_t4 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t4)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t4"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t4

task qvip_axi_xbar_vseq_t4::body;

    incr_wr_t          incr_wr0          =
incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t          incr_rd0          =
incr_rd_t::type_id::create("incr_rd0");


 super.body();


  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 3;
       if(!incr_wr0.randomize() with {addr == 32'h0000_7050; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");


    fork
      incr_wr0.start(axi4_master0);

  end
endtask: body
```

Secuencia 5

```
class qvip_axi_xbar_vseq_t5 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t5)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t5"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t5

task qvip_axi_xbar_vseq_t5::body;

    incr_wr_t          incr_wr0          =
incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t          incr_rd0          =
incr_rd_t::type_id::create("incr_rd0");


 super.body();


  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 3;
       if(!incr_wr0.randomize() with {addr == 32'h0000_7050; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");


    fork
      incr_wr0.start(axi4_master0);

    join


  end
```

```
endtask: body
```

Secuencia 6

```
class qvip_axi_xbar_vseq_t6 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t6)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t6"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t6

task qvip_axi_xbar_vseq_t6::body;

    incr_wr_t          incr_wr0          =
incr_wr_t::type_id::create("incr_wr0");
    incr_rd_t          incr_rd0          =
incr_rd_t::type_id::create("incr_rd0");

    incr_wr_t          incr_wr1          =
incr_wr_t::type_id::create("incr_wr1");
    incr_rd_t          incr_rd1          =
incr_rd_t::type_id::create("incr_rd1");

 super.body();

  begin
    //master0 write
       incr_wr0.set_Sequencer(axi4_master0);
       incr_wr0.id = 1;
       if(!incr_wr0.randomize() with {addr == 32'h0000_3500;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
    //master1 write
       incr_wr1.set_Sequencer(axi4_master0);
       incr_wr1.id = 1;
       if(!incr_wr1.randomize() with {addr == 32'h1000_3500;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
```

```
    fork
      incr_wr0.start(axi4_master0);
      incr_wr1.start(axi4_master0);
    join
  end
endtask: body
```

Secuencia 7

```
class qvip_axi_xbar_vseq_t7 #(int AXI4_ADDRESS_WIDTH  = 32,
               int AXI4_RDATA_WIDTH   = 64,
               int AXI4_WDATA_WIDTH   = 64,
               int AXI4_ID_WIDTH    = 4,
               int AXI4_USER_WIDTH   = 5,
               int AXI4_REGION_MAP_SIZE = 16
               ) extends qvip_axi_xbar_vseq_base;

 typedef qvip_axi_xbar_vseq_t7         #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) this_t;

 typedef axi4_master_rw_transaction     #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) rw_t;

 typedef axi4_master_write_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
                         AXI4_RDATA_WIDTH,
                         AXI4_WDATA_WIDTH,
                         AXI4_ID_WIDTH,
                         AXI4_USER_WIDTH,
                         AXI4_REGION_MAP_SIZE
                        ) waddr_t;

 typedef axi4_master_write_data_burst     #(AXI4_ADDRESS_WIDTH,
                         AXI4_RDATA_WIDTH,
                         AXI4_WDATA_WIDTH,
                         AXI4_ID_WIDTH,
                         AXI4_USER_WIDTH,
                         AXI4_REGION_MAP_SIZE
                        ) wdata_burst_t;

 typedef axi4_master_read_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
```

```
                        ) raddr_t;
/*
 typedef axi4_vip_config           #(AXI4_ADDRESS_WIDTH,
                          AXI4_RDATA_WIDTH,
                          AXI4_WDATA_WIDTH,
                          AXI4_ID_WIDTH,
                          AXI4_USER_WIDTH,
                          AXI4_REGION_MAP_SIZE
                          ) cfg_t;
*/

 typedef axi4_incr_rd_deparam_seq  incr_rd_t;

  axi4_master0_cfg_t cfg;

 `uvm_object_param_utils(this_t)

 //cfg_t cfg;

 function new (string name = "qvip_axi_xbar_vseq_t7");
  super.new( name );
 endfunction

 extern task body();
 extern task execute_rd_addr();
 extern task execute_wr_addr_data();

endclass

// Task: body
//
// This is standard UVM body task.
//
// It executes write address and data phase followed by read address
phase
// and waits for the read and write transactions to complete.
//
// (begin inline source)

task qvip_axi_xbar_vseq_t7::body();

 int num_txn = 1;
 // Get the config object so that you can wait for a clock after reset
 // to issue the first sequence_item.
 cfg = axi4_master0_cfg_t::get_config(axi4_master0);

 cfg.wait_for_reset();
 cfg.wait_for_clock();
```

```
 fork
  begin
   for(int i=0;i<num_txn;++i)
   begin
    execute_wr_addr_data();
    execute_rd_addr();
   end
  end
 join

  // Just to ensure that all transactions complete
  // before the sequence ends
   repeat(num_txn*2)
    begin
      rw_t rw_txn = rw_t::type_id::create("rw_txn");
      rw_txn.m_receive_id = get_stream_id(this);
      rw_txn.receive(cfg);
    end

endtask

// (end inline source)

// Task: execute_rd_addr
//
// It executes read address phase

task qvip_axi_xbar_vseq_t7::execute_rd_addr();

 raddr_t raddr0 = raddr_t::type_id::create("raddr0");
 raddr_t raddr1 = raddr_t::type_id::create("raddr1");
 raddr_t raddr2 = raddr_t::type_id::create("raddr2");


 raddr0.set_Sequencer(axi4_master0);


// secuencia de lectra de direccion de master 0
 start_item(raddr0);
 `uvm_info("rd_addr", "start_item", UVM_LOW)
  if(!raddr0.randomize() with { raddr0.lock == AXI4_NORMAL;
                 raddr0.burst_length <= 2;
                }) `uvm_error("axi4_master_phase_seq", "Read addres
randomization failure")
 finish_item(raddr0);
 `uvm_info("rd_addr", "finish_item", UVM_LOW)

endtask
```

```
// Task: execute_wr_addr_data
//
// It executes write address and data phase

task qvip_axi_xbar_vseq_t7::execute_wr_addr_data();

 waddr_t    waddr0 = waddr_t::type_id::create("waddr0");
 waddr_t    waddr01 = waddr_t::type_id::create("waddr01");
 waddr_t    waddr1 = waddr_t::type_id::create("waddr1");
 waddr_t    waddr2 = waddr_t::type_id::create("waddr2");

 wdata_burst_t wdata0 = wdata_burst_t::type_id::create("wdata0");
 wdata_burst_t wdata01 = wdata_burst_t::type_id::create("wdata01");
 wdata_burst_t wdata1 = wdata_burst_t::type_id::create("wdata1");
 wdata_burst_t wdata2 = wdata_burst_t::type_id::create("wdata2");

 waddr0.set_Sequencer(axi4_master0);
 waddr01.set_Sequencer(axi4_master0);
 waddr1.set_Sequencer(axi4_master1);
 waddr2.set_Sequencer(axi4_master2);

 wdata0.set_Sequencer(axi4_master0);
 wdata01.set_Sequencer(axi4_master0);
 wdata1.set_Sequencer(axi4_master1);
 wdata2.set_Sequencer(axi4_master2);
fork
 begin
// ****************************************
// secuencia del master 0 escritura de direccion
 start_item( waddr0 );
 `uvm_info("wr_addr0", "start_item", UVM_LOW)
  if(!waddr0.randomize() with {waddr0.lock == AXI4_NORMAL;
                waddr0.addr == 32'h0000_1000;
                waddr0.burst_length <= 4;
                waddr0.id == 3;
                }) `uvm_error("axi4_master_phase_seq","Write address
randomization failure")
 finish_item(waddr0);
 `uvm_info("wr_addr0", "finish_item", UVM_LOW)
// ****************************************
 end

 begin
// ****************************************
// secuencia del master 01 escritura de direccion
 start_item( waddr01 );
 `uvm_info("wr_addr01", "start_item", UVM_LOW)
  if(!waddr01.randomize() with {waddr01.lock == AXI4_NORMAL;
                waddr01.addr == 32'h0000_1500;
```

```
                waddr01.burst_length <= 20;
                waddr01.id == 2;
                }) `uvm_error("axi4_master_phase_seq","Write address
randomization failure")
 finish_item(waddr01);
 `uvm_info("wr_addr01", "finish_item", UVM_LOW)
// ****************************************
 end

 begin
// ****************************************
// secuencia del master 1 escritura de direccion
  start_item( waddr1 );
 `uvm_info("wr_addr1", "start_item", UVM_LOW)
  if(!waddr1.randomize() with {waddr1.lock == AXI4_NORMAL;
                waddr1.addr == 32'h0000_2000;
                waddr1.burst_length <= 4;
                waddr1.id == 1;
               }) `uvm_error("axi4_master_phase_seq", "Write address
randomization failure")
 finish_item(waddr1);
 `uvm_info("wr_addr1", "finish_item", UVM_LOW)
// ****************************************
 end
 begin
// ****************************************
// secuencia del master 2 escritura de direccion
  start_item( waddr2 );
 `uvm_info("wr_addr2", "start_item", UVM_LOW)
  if(!waddr2.randomize() with {waddr2.lock == AXI4_NORMAL;
                waddr2.addr == 32'h0000_2000;
                waddr2.burst_length <= 4;
                waddr2.id == 1;
               }) `uvm_error("axi4_master_phase_seq", "Write address
randomization failure")
 finish_item(waddr2);
 `uvm_info("wr_addr2", "finish_item", UVM_LOW)
// ****************************************
 end


 begin
// ****************************************
// escritura de datos del master 0

 start_item( wdata0 );
 `uvm_info("wr_data0", "start_item", UVM_LOW)
  if(!wdata0.randomize() with {wdata0.burst_length ==
waddr0.burst_length;
```

```
                 foreach(wdata0.write_strobes[i]) wdata0.write_strobes[i]
== '1;

                }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
 finish_item( wdata0 );
 `uvm_info("wr_data0", "finish_item", UVM_LOW)
// ****************************************
 end


 begin
// ****************************************
// escritura de datos del master 01

 start_item( wdata01 );
 `uvm_info("wr_data01", "start_item", UVM_LOW)
  if(!wdata01.randomize() with {wdata01.burst_length ==
waddr01.burst_length;
                 foreach(wdata01.write_strobes[i])
wdata01.write_strobes[i] == '1;

                }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
 finish_item( wdata01 );
 `uvm_info("wr_data01", "finish_item", UVM_LOW)
// ****************************************
 end

 begin
// ****************************************
// escritura de datos del master 1

 start_item( wdata1 );
 `uvm_info("wr_data1", "start_item", UVM_LOW)
  if(!wdata1.randomize() with {wdata1.burst_length ==
waddr1.burst_length;
                foreach(wdata1.write_strobes[i]) wdata1.write_strobes[i]
== '1;

                //foreach ( wdata1.write_data_beats_delay[i]
){wdata1.write_data_beats_delay[i] == 5;}
                }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
 finish_item( wdata1 );
 `uvm_info("wr_data1", "finish_item", UVM_LOW)
// ****************************************
 end
 begin
```

```
// ****************************************
// escritura de datos del master 2

 start_item( wdata2 );
 `uvm_info("wr_data2", "start_item", UVM_LOW)
  if(!wdata2.randomize() with {wdata2.burst_length ==
waddr2.burst_length;
                foreach(wdata2.write_strobes[i]) wdata2.write_strobes[i]
== '1;

               }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
 finish_item( wdata2 );
 `uvm_info("wr_data2", "finish_item", UVM_LOW)
// ****************************************
 end
join
endtask
```

Secuencia 8

```
class qvip_axi_xbar_vseq_t7 #(int AXI4_ADDRESS_WIDTH  = 32,
               int AXI4_RDATA_WIDTH   = 64,
               int AXI4_WDATA_WIDTH   = 64,
               int AXI4_ID_WIDTH    = 4,
               int AXI4_USER_WIDTH   = 5,
               int AXI4_REGION_MAP_SIZE = 16
               ) extends qvip_axi_xbar_vseq_base;

 typedef qvip_axi_xbar_vseq_t7         #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) this_t;

 typedef axi4_master_rw_transaction     #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) rw_t;

 typedef axi4_master_write_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
                         AXI4_RDATA_WIDTH,
                         AXI4_WDATA_WIDTH,
                         AXI4_ID_WIDTH,
                         AXI4_USER_WIDTH,
                         AXI4_REGION_MAP_SIZE
                        ) waddr_t;

 typedef axi4_master_write_data_burst     #(AXI4_ADDRESS_WIDTH,
                         AXI4_RDATA_WIDTH,
                         AXI4_WDATA_WIDTH,
                         AXI4_ID_WIDTH,
                         AXI4_USER_WIDTH,
                         AXI4_REGION_MAP_SIZE
                        ) wdata_burst_t;

 typedef axi4_master_read_addr_channel_phase #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
```

```
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) raddr_t;


 typedef axi4_incr_rd_deparam_seq  incr_rd_t;

  axi4_master0_cfg_t cfg;

 `uvm_object_param_utils(this_t)


 function new (string name = "qvip_axi_xbar_vseq_t7");
  super.new( name );
 endfunction

 extern task body();
 extern task execute_rd_addr();
 extern task execute_wr_addr_data();

endclass

// Task: body
//
// This is standard UVM body task.
//
// It executes write address and data phase followed by read address
phase
// and waits for the read and write transactions to complete.
//
// (begin inline source)

task qvip_axi_xbar_vseq_t7::body();

 int num_txn = 1;
 // Get the config object so that you can wait for a clock after reset
 // to issue the first sequence_item.
 cfg = axi4_master0_cfg_t::get_config(axi4_master0);

 cfg.wait_for_reset();
 cfg.wait_for_clock();

 fork
  begin
   for(int i=0;i<num_txn;++i)
   begin
    execute_wr_addr_data();
    execute_rd_addr();
   end
```

```
  end
 join

  // Just to ensure that all transactions complete
  // before the sequence ends
   repeat(num_txn*2)
    begin
      rw_t rw_txn = rw_t::type_id::create("rw_txn");
      rw_txn.m_receive_id = get_stream_id(this);
      rw_txn.receive(cfg);
    end

endtask

// (end inline source)

// Task: execute_rd_addr
//
// It executes read address phase

task qvip_axi_xbar_vseq_t7::execute_rd_addr();

 raddr_t raddr0 = raddr_t::type_id::create("raddr0");
 raddr_t raddr1 = raddr_t::type_id::create("raddr1");
 raddr_t raddr2 = raddr_t::type_id::create("raddr2");


 raddr0.set_Sequencer(axi4_master0);


// secuencia de lectra de direccion de master 0
 start_item(raddr0);
 `uvm_info("rd_addr", "start_item", UVM_LOW)
  if(!raddr0.randomize() with { raddr0.lock == AXI4_NORMAL;
                 raddr0.burst_length <= 2;
                }) `uvm_error("axi4_master_phase_seq", "Read addres
randomization failure")
 finish_item(raddr0);
 `uvm_info("rd_addr", "finish_item", UVM_LOW)

endtask


// Task: execute_wr_addr_data
//
// It executes write address and data phase

task qvip_axi_xbar_vseq_t7::execute_wr_addr_data();
```

```
 waddr_t    waddr0 = waddr_t::type_id::create("waddr0");
 waddr_t    waddr01 = waddr_t::type_id::create("waddr01");
 waddr_t    waddr1 = waddr_t::type_id::create("waddr1");
 waddr_t    waddr2 = waddr_t::type_id::create("waddr2");

 wdata_burst_t wdata0 = wdata_burst_t::type_id::create("wdata0");
 wdata_burst_t wdata01 = wdata_burst_t::type_id::create("wdata01");
 wdata_burst_t wdata1 = wdata_burst_t::type_id::create("wdata1");
 wdata_burst_t wdata2 = wdata_burst_t::type_id::create("wdata2");

 waddr0.set_Sequencer(axi4_master0);
 waddr01.set_Sequencer(axi4_master0);
 waddr1.set_Sequencer(axi4_master1);
 waddr2.set_Sequencer(axi4_master2);

 wdata0.set_Sequencer(axi4_master0);
 wdata01.set_Sequencer(axi4_master0);
 wdata1.set_Sequencer(axi4_master1);
 wdata2.set_Sequencer(axi4_master2);

 begin
// ****************************************
// secuencia del master 0 escritura de direccion
 start_item( waddr0 );
 `uvm_info("wr_addr0", "start_item", UVM_LOW)
  if(!waddr0.randomize() with {waddr0.lock == AXI4_NORMAL;
                waddr0.addr == 32'h0000_1000;
                waddr0.burst_length <= 4;
                waddr0.id == 3;
                }) `uvm_error("axi4_master_phase_seq","Write address
randomization failure")
 finish_item(waddr0);
 `uvm_info("wr_addr0", "finish_item", UVM_LOW)
// ****************************************
 end

 begin
// ****************************************
// escritura de datos del master 0

 start_item( wdata0 );
 `uvm_info("wr_data0", "start_item", UVM_LOW)
  if(!wdata0.randomize() with {wdata0.burst_length ==
waddr0.burst_length;
                foreach(wdata0.write_strobes[i]) wdata0.write_strobes[i]
== '1;

               }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
```

```
 finish_item( wdata0 );
 `uvm_info("wr_data0", "finish_item", UVM_LOW)
// ****************************************
 end


 begin
// ****************************************
// escritura de datos del master 01

 start_item( wdata01 );
 `uvm_info("wr_data01", "start_item", UVM_LOW)
  if(!wdata01.randomize() with {wdata01.burst_length ==
waddr0.burst_length;
                foreach(wdata01.write_strobes[i])
wdata01.write_strobes[i] == '1;

               }) `uvm_error("axi4_master_phase_seq", "Write data
randomization failure")
 finish_item( wdata01 );
 `uvm_info("wr_data01", "finish_item", UVM_LOW)
// ****************************************
 end


endtask
```

Secuencia 9

```
class qvip_axi_xbar_vseq_t9 extends qvip_axi_xbar_vseq_base;
  `uvm_object_utils(qvip_axi_xbar_vseq_t9)

   typedef axi4_incr_wr_deparam_seq  incr_wr_t;
   typedef axi4_incr_rd_deparam_seq  incr_rd_t;
   typedef axi4_wrap_wr_deparam_seq  wrap_wr_t;
   typedef axi4_wrap_rd_deparam_seq  wrap_rd_t;
   typedef axi4_excl_rw_deparam_seq  excl_rw_t;

  function new
  (
    string name = "qvip_axi_xbar_vseq_t9"
  );
    super.new(name);
  endfunction

  extern task body;

endclass: qvip_axi_xbar_vseq_t9

task qvip_axi_xbar_vseq_t9::body;

    incr_wr_t          incr_wr00          =
incr_wr_t::type_id::create("incr_wr00");
    incr_wr_t          incr_wr01          =
incr_wr_t::type_id::create("incr_wr01");
    incr_wr_t          incr_wr02          =
incr_wr_t::type_id::create("incr_wr02");
    incr_wr_t          incr_wr03          =
incr_wr_t::type_id::create("incr_wr03");
    incr_wr_t          incr_wr04          =
incr_wr_t::type_id::create("incr_wr04");
    incr_wr_t          incr_wr1          =
incr_wr_t::type_id::create("incr_wr1");
    incr_wr_t          incr_wr2          =
incr_wr_t::type_id::create("incr_wr2");
    incr_wr_t          incr_wr3          =
incr_wr_t::type_id::create("incr_wr3");
    incr_wr_t          incr_wr4          =
incr_wr_t::type_id::create("incr_wr4");
```

```
 super.body();


  begin
    //master0 write
       incr_wr00.set_Sequencer(axi4_master0);
       incr_wr00.id = 0;
       if(!incr_wr00.randomize() with {addr == 32'h0000_4500;
wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
    //master0 write
       incr_wr01.set_Sequencer(axi4_master0);
       incr_wr01.id = 1;
       if(!incr_wr01.randomize() with {addr == 32'h1000_4000;
wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
   //master0 write
       incr_wr02.set_Sequencer(axi4_master0);
       incr_wr02.id = 2;
       if(!incr_wr01.randomize() with {addr == 32'h0000_1000;
wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
   //master0 write
       incr_wr03.set_Sequencer(axi4_master0);
       incr_wr03.id = 3;
       if(!incr_wr03.randomize() with {addr == 32'h1000_1000;
wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
   //master0 write
       incr_wr04.set_Sequencer(axi4_master0);
       incr_wr04.id = 4;
       if(!incr_wr04.randomize() with {addr == 32'h0000_2000;
wr_data.size == 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");

   //master1 write
       incr_wr1.set_Sequencer(axi4_master1);
       incr_wr1.id = 5;
       if(!incr_wr1.randomize() with {addr == 32'h0000_4000; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
   //master2 write
       incr_wr2.set_Sequencer(axi4_master2);
       incr_wr2.id = 6;
       if(!incr_wr2.randomize() with {addr == 32'h1000_4000; wr_data.size
== 16;})
        `uvm_error(this.get_full_name(),"Randomisation failure");
    //master3 write
       incr_wr3.set_Sequencer(axi4_master3);
```

```
        incr_wr3.id = 7;
        if(!incr_wr3.randomize() with {addr == 32'h2000_4000; wr_data.size
== 16;})
         `uvm_error(this.get_full_name(),"Randomisation failure");
   //master1 write
        incr_wr4.set_Sequencer(axi4_master1);
        incr_wr4.id = 8;
        if(!incr_wr4.randomize() with {addr == 32'h0000_3500; wr_data.size
== 16;})
         `uvm_error(this.get_full_name(),"Randomisation failure");

    fork
      incr_wr00.start(axi4_master0);
      incr_wr01.start(axi4_master0);
   incr_wr02.start(axi4_master0);
      incr_wr03.start(axi4_master0);
      incr_wr04.start(axi4_master0);
      incr_wr1.start(axi4_master1);
      incr_wr2.start(axi4_master2);
      incr_wr3.start(axi4_master3);
      incr_wr4.start(axi4_master1);
 join
  end
endtask: body
```

Secuencia 10

```
class qvip_axi_xbar_vseq_t10 #(int AXI4_ADDRESS_WIDTH  = 32,
               int AXI4_RDATA_WIDTH   = 64,
               int AXI4_WDATA_WIDTH   = 64,
               int AXI4_ID_WIDTH    = 4,
               int AXI4_USER_WIDTH   = 5,
               int AXI4_REGION_MAP_SIZE = 16
               ) extends qvip_axi_xbar_vseq_base;

 typedef qvip_axi_xbar_vseq_t10         #(AXI4_ADDRESS_WIDTH,
                        AXI4_RDATA_WIDTH,
                        AXI4_WDATA_WIDTH,
                        AXI4_ID_WIDTH,
                        AXI4_USER_WIDTH,
                        AXI4_REGION_MAP_SIZE
                        ) this_t;

 typedef axi4_vip_config     #(AXI4_ADDRESS_WIDTH,
                  AXI4_RDATA_WIDTH,
                  AXI4_WDATA_WIDTH,
                  AXI4_ID_WIDTH,
                  AXI4_USER_WIDTH,
                  AXI4_REGION_MAP_SIZE
                  ) axi4_vip_config_t;



 typedef axi4_master_write   #( AXI4_ADDRESS_WIDTH,
                  AXI4_RDATA_WIDTH,
                  AXI4_WDATA_WIDTH,
                  AXI4_ID_WIDTH,
                  AXI4_USER_WIDTH,
                  AXI4_REGION_MAP_SIZE
                  ) write_item_t;

 typedef axi4_master_read   #( AXI4_ADDRESS_WIDTH,
                  AXI4_RDATA_WIDTH,
                  AXI4_WDATA_WIDTH,
                  AXI4_ID_WIDTH,
                  AXI4_USER_WIDTH,
                  AXI4_REGION_MAP_SIZE
                  ) read_item_t;
```

```
 `uvm_object_param_utils( this_t)

 axi4_master0_cfg_t cfg;
 axi4_slave0_cfg_t  slv_cfg;

 function new (string name = "qvip_axi_xbar_vseq_t10");
  super.new( name );
 endfunction

 extern task body();

endclass

// Task: body
//
// This is standard UVM body task.
//
// It pre-loads random data into a slave via backdoor write and
// reads back the slave memory contents via the axi4 protocol.
// It also over-writes the slave memory contents via the axi4 protocol
// and then reads the slave memory via the backdoor read.
// (begin inline source)

task qvip_axi_xbar_vseq_t10::body();

 bit [AXI4_ADDRESS_WIDTH-1:0] addr  [32];
 bit [7:0]          wr_data [32];
 bit [7:0]          rd_data [32];

 read_item_t    read_item     =
read_item_t::type_id::create("read_item");
 write_item_t   write_item    =
write_item_t::type_id::create("write_item");

 cfg = axi4_vip_config_t::get_config(axi4_master0);
 slv_cfg = axi4_slave0_cfg_t::get_config(axi4_slave0);

 cfg.wait_for_reset();
 cfg.wait_for_clock();

 //   **********************PASO 1 ******************************

 // Reading the slave memory via axi4 protocol

 read_item.set_Sequencer(axi4_master0);
 start_item( read_item );
  if(!read_item.randomize() with
   {
```

```
read_item.burst_length == 0;
   read_item.id      == 1;
   read_item.burst    == AXI4_INCR;
   read_item.addr     == 32'h00001000;
   read_item.lock     == AXI4_EXCLUSIVE;
   read_item.size     == AXI4_BYTES_8;


  }
 ) `uvm_error(this.get_full_name(),"Randomisation failure");
 finish_item( read_item );

// writing the slave memory via axi4 protocol

write_item.set_Sequencer(axi4_master0);
start_item( write_item );
 if(!write_item.randomize() with
  {
   write_item.prot  == read_item.prot;
   write_item.cache == read_item.cache;
   write_item.addr     == 32'h00001000;
   write_item.burst_length == 0 ;
   write_item.id      == 1;
   write_item.write_strobes[0] == 8'hFF;

   write_item.burst    == AXI4_INCR;
   write_item.lock     == AXI4_EXCLUSIVE;
   write_item.size     == AXI4_BYTES_8;

  }

 ) `uvm_error(this.get_full_name(),"Randomisation failure");

 finish_item( write_item );

//   **********************PASO 2 ******************************

// Reading the slave memory via axi4 protocol

read_item.set_Sequencer(axi4_master0);
start_item( read_item );
 if(!read_item.randomize() with
  {

read_item.id      == 2;
   read_item.burst_length == 0;
   read_item.burst    == AXI4_INCR;
   read_item.addr     == 32'h00001000;
   read_item.lock     == AXI4_EXCLUSIVE;
```

```
    read_item.size     == AXI4_BYTES_8;


   }
  ) `uvm_error(this.get_full_name(),"Randomisation failure");
  finish_item( read_item );

// Programming the slave memory through backdoor write

 `uvm_info("SEQ/BACKDOOR/WR","Starting backdoor write access" ,
UVM_MEDIUM)
 for(int i = 0; i < 16; i++)
  slv_cfg.slv_mem.backdoor_write(32'h00001000 + i, 8'hff);

 `uvm_info("SEQ/BACKDOOR/WR","Backdoor write access done" , UVM_MEDIUM)



 // writing the slave memory via axi4 protocol

 write_item.set_Sequencer(axi4_master0);
 start_item( write_item );
  if(!write_item.randomize() with
   {
 write_item.id      == 2;
    write_item.addr     == 32'h00001000;
    write_item.burst_length == 0 ;
 write_item.prot  == read_item.prot;
    write_item.cache == read_item.cache;
    write_item.region    == read_item.region;
    write_item.burst    == AXI4_INCR;
    write_item.lock     == AXI4_EXCLUSIVE;
    write_item.size     == AXI4_BYTES_8;

   }

  ) `uvm_error(this.get_full_name(),"Randomisation failure");

  finish_item( write_item );

//   ********************** PASO 3 ****************************

// Reading the slave memory via axi4 protocol

 read_item.set_Sequencer(axi4_master0);
 start_item( read_item );
  if(!read_item.randomize() with
   {
```

```
 read_item.id      == 3;
    read_item.burst_length == 0;
    read_item.burst    == AXI4_INCR;
    read_item.addr     == 32'h00003000;
    read_item.lock     == AXI4_EXCLUSIVE;
    read_item.size     == AXI4_BYTES_8;

   }
  ) `uvm_error(this.get_full_name(),"Randomisation failure");
  finish_item( read_item );

 // writing the slave memory via axi4 protocol

 write_item.set_Sequencer(axi4_master0);
 start_item( write_item );
  if(!write_item.randomize() with
   {
 write_item.id      == 3;
    write_item.addr     == 32'h00003000;
    write_item.burst_length == 0 ;
 write_item.prot  == read_item.prot;
    write_item.cache == read_item.cache;
    write_item.region    == read_item.region;
    write_item.burst    == AXI4_INCR;
    write_item.lock     == AXI4_EXCLUSIVE;
    write_item.size     == AXI4_BYTES_8;

   }

  ) `uvm_error(this.get_full_name(),"Randomisation failure");

  finish_item( write_item );

// writing the slave memory via axi4 protocol

 write_item.set_Sequencer(axi4_master0);
 start_item( write_item );
  if(!write_item.randomize() with
   {
 write_item.id      == 3;
    write_item.addr     == 32'h00003000;
    write_item.burst_length == 0 ;
 write_item.prot  == read_item.prot;
    write_item.cache == read_item.cache;
    write_item.region    == read_item.region;
    write_item.burst    == AXI4_INCR;
    write_item.lock     == AXI4_EXCLUSIVE;
    write_item.size     == AXI4_BYTES_8;
```

```
   }

  ) `uvm_error(this.get_full_name(),"Randomisation failure");

  finish_item( write_item );

//   ********************* PASO 4 ****************************

// Reading the slave memory via axi4 protocol

 read_item.set_Sequencer(axi4_master0);
 start_item( read_item );
  if(!read_item.randomize() with
   {

 read_item.id      == 4;
    read_item.burst_length == 0;
    read_item.burst    == AXI4_INCR;
    read_item.addr     == 32'h00004000;
    read_item.lock     == AXI4_EXCLUSIVE;
    read_item.size     == AXI4_BYTES_8;

   }
  ) `uvm_error(this.get_full_name(),"Randomisation failure");
  finish_item( read_item );

 // writing the slave memory via axi4 protocol

 write_item.set_Sequencer(axi4_master0);
 start_item( write_item );
  if(!write_item.randomize() with
   {
 write_item.id      == 4;
    write_item.addr     == 32'h00004000;
    write_item.burst_length == 0 ;
 //write_item.prot  == read_item.prot;
    //write_item.cache == read_item.cache;
    //write_item.region    == read_item.region;
    write_item.burst    == AXI4_INCR;
    write_item.lock     == AXI4_EXCLUSIVE;
    write_item.size     == AXI4_BYTES_8;

   }

  ) `uvm_error(this.get_full_name(),"Randomisation failure");

  finish_item( write_item );

endtask
```