UNIVERSIDAD DE LAS PALMAS
DE GRAN CANARIA

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de un *testbench* UVM integrando IP de verificación de *Mentor Graphics* (QVIP)

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación

Mención: Sistemas electrónicos

Autor: D. Álvaro José Moreno Florido

Tutores: Dr. D. Valentín de Armas Sosa

Dr. D. Félix Bernardo Tobajas Guerrero

Fecha: Julio de 2021

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Desarrollo de un *testbench* UVM integrando IP de verificación de *Mentor Graphics* (QVIP)

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.: _____

Vocal **Secretario/a**

Fdo.:_____ **Fdo.:**_____

Las Palmas de Gran Canaria, a 21 de julio de 2021

Resumen

En la actualidad, la etapa de verificación ocupa un lugar imprescindible para el desarrollo de sistemas *hardware* digitales. Debido al aumento de la demanda de los sistemas electrónicos y a la mejora de la tecnología asociada al sector de la electrónica, los sistemas desarrollados se diseñan con una estructura de mayor complejidad, ya que se integra un mayor número de componentes y, además, se incrementan las funcionalidades asociadas a dicho sistema. Por estos motivos, las tareas de verificación son complejas y consumen mucho tiempo a lo largo del proceso descrito. 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 del presente Trabajo Fin de Grado (TFG) consiste en el diseño de un *testbench* UVM usando un IP de verificación de *Mentor Graphics, a Siemens Business*. Este módulo IP está desarrollado con la metodología UVM y utiliza el protocolo de comunicación AMBA AXI4 y es totalmente configurable.

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. Finalmente, se diseñará un *testbench* UVM que verificará el correcto comportamiento de dicho sistema.

Abstract

Nowadays, the verification stage occupies an essential place in the development of digital hardware systems. Due to the increasing demand for electronic systems and the improvement of the technology associated with the electronics sector, the systems developed are designed with a more complex structure, since a greater number of components are integrated and, in addition, the functionalities associated with the system are increased. Consequently, verification tasks are complex and time-consuming throughout the aforementioned process described. The Universal Verification Methodology (UVM), which is based on the SystemVerilog hardware description and verification language, was developed to solve these problems. The main objective of this Final Degree Project is the design of a UVM testbench using a verification IP from Mentor Graphics, a Siemens Business. This IP module is developed with the UVM methodology and uses the AMBA AXI4 communication protocol and is fully configurable.

Firstly, a study stage will be carried out where the operation of the AXI4 protocol and the necessary concepts of the UVM methodology will be studied in depth. Secondly, we will study the Mentor Graphics verification IP and the QVIP configurator tool, which will create the UVM verification environment to be used for the stimulation of the IP module to be verified. After this, the device to be verified (DUV) will be studied in depth to understand its operation. Finally, a UVM testbench will be designed to verify the correct behaviour of the system.

Índice de contenidos

Capítulo 1. Introducción 1

1.1. Antecedentes 1

1.2. Objetivos 5

1.3. Peticionario 5

1.4. Estructura del documento 6

Capítulo 2. Protocolo AXI4 9

2.1. Canales y señales 9

2.2. Protocolo de *handshake* en AXI4 14

2.3. Relaciones entre los canales 16

2.3.1. Dependencias en las transacciones de lectura 16

2.3.2. Dependencias en las transacciones de escritura 17

2.4. Configuración de las transacciones 18

2.4.1. Parámetro *BURST* 18

2.4.2. Parámetro *PROT* 19

2.4.3. Parámetro LEN 20

2.4.4. Parámetro SIZE 20

2.4.5. Parámetro *CACHE* 21

2.4.6. Parámetro *REGION* 22

2.4.7. Parámetro *USER* 23

2.4.8. Parámetro *ID* 23

2.4.8.1. Requisitos de orden 23

2.4.8.2. Ancho de los campos *ID* de transacción 24

Capítulo 3. *Universal Verification Methodology* 25

3.1. Verificación de sistemas electrónicos digitales 25

3.1.1. Verificación *Black-Box* 27

3.1.2. Verificación *White-Box* 27

3.1.3. Verificación *Grey-Box* 27

3.2. Introducción a la metodología UVM 28

3.3. El lenguaje de descripción *hardware SystemVerilog* 30
3.4. Biblioteca de clases básicas 32
3.5. Mecanismo de fases de UVM 34
3.6. El mecanismo *Factory* en UVM 36
3.7. Sistema de mensajes e informes 36
3.8. Modelado a nivel de transacción TLM 37
3.9. Entorno de verificación UVM 38
3.9.1. Componente UVM *Agent* 39
3.9.1.1. Componente UVM *Sequencer* 41
3.9.1.1.1. Secuencias UVM 41
3.9.1.1.2. Transacciones en UVM 42
3.9.1.1.3. Protocolo *handshake Sequencer-Driver* 42
3.9.1.2. Componente UVM *Driver* 44
3.9.1.3. Componente UVM *Monitor* 44
3.9.2. Componente UVM *Scoreboard* 45
3.9.3. Componente UVM *Environment* 47
3.9.4. Componente UVM *Test* 47
3.9.5. Componente UVM *Testbench* 48
Capítulo 4. Diseño a verificar (DUV) 49
4.1. Módulo *axi_xbar* 49
4.1.1. Características básicas 50
4.1.2. Visión general del diseño 51
4.1.3. Configuración del sistema 53
4.1.4. *Testbench* 55
4.2. Módulo *wrapper* 56
4.2.1. Funcionalidades 57
4.2.2. Código implementado 58
Capítulo 5. Desarrollo de un entorno de verificación UVM mediante la herramienta QVIP *configurator* 65

5.1. Módulo *Questa Verification* IP 65
5.1.1. Principios básicos de la herramienta QVIP *configurator* 66
5.1.2. Diagrama de flujo para el uso de QVIP *configurator* 68
5.1.3. Guía de usuario de la herramienta QVIP *configurator* 70
5.1.3.1. Integración de los módulos de verificación IP 70
5.1.3.2. Conexionado de los módulos IP con el DUV 73
5.1.3.3. Definición del mapa de direcciones 76
5.2. Entorno de verificación UVM 77
5.2.1. Esquema general del *testbench* UVM generado 77
5.2.2. Componente UVM *Testbench* 78
5.2.2.1. Referencia de los módulos maestros y esclavos 79
5.2.2.2. Referencia del módulo *wrapper* 83
5.2.2.2.1. Puertos de la señal de reloj y la señal *reset* 83
5.2.2.2.2. Puerto esclavo de entrada 83
5.2.2.2.3. Puerto maestro de salida 86
5.2.2.2.4. Puerto maestro de entrada.....88
5.2.2.2.5. Puerto esclavo de salida 89
5.2.3. Componente UVM *Test*.....91
5.2.4. Componente UVM *Environment*.....97
5.2.4.1. Configuración del componente UVM *Environment*.....97
5.2.4.2. Descripción del componente UVM *Environment* 98
5.2.5. Objeto UVM *Sequence* 101
5.2.6. Archivo de simulación 102
Capítulo 6. Desarrollo del *testbench* UVM 103
6.1. Biblioteca de estímulos.....103
6.1.1. Biblioteca de secuencias básicas.....104
6.1.2. Interfaces API para la generación de estímulos 105
6.2. Código implementado.....107
6.2.1. Secuencia *axi4_sequence_lib_seq_incr* 109

6.2.2. Secuencia *axi4_sequence_lib_seq_wrap* 111
6.2.3. Secuencia *axi4_sequence_lib_seq* 112
6.2.4. Secuencia *axi4_generic_ api_excl_seq* 113
6.2.5. Secuencia *axi4_generic_ api_normal_seq* 114
6.2.6. Secuencia *axi4_generic_ api_wrap_seq* 115
Capítulo 7. Resultados 117
7.1. Topología resultante 117
7.2. Simulación de las secuencias 119
7.2.1. Fase de ejecución del *test* base 119
7.2.2. Secuencias descritas y ejecutadas en el proceso de verificación 121
7.2.2.1. Secuencia *axi4_sequence_lib_seq_incr* 121
7.2.2.2. Secuencia *axi4_sequence_lib_seq_wrap* 125
7.2.2.3. *Secuencia axi4_sequence_lib_seq* 128
7.2.2.4. Secuencia *axi4_generic_ api_excl_seq* 133
7.2.2.5. Secuencia *axi4_generic_ api_normal_seq* 135
7.2.2.6. Secuencia *axi4_generic_ api_wrap_seq* 137
Capítulo 8. Conclusiones y líneas futuras 143
8.1. Conclusiones 143
8.2. Líneas futuras 144
Pliego de condiciones 151
PC.1 Condiciones *hardware* 151
PC.2 Condiciones *software* 152
Presupuesto 153
P.1. Recursos humanos 153
P.2. Recursos materiales 155
P.2.1. Recursos *hardware* 155
P2.2. Recursos *software* 156
P.3. Material fungible 157
P.4. Redacción de la Memoria 157

P.5. Derechos de visado del COITT 158
P.6. Gastos de tramitación y envío.....159
P.7. Coste total del proyecto 159

Índice de Figuras

Figura 1. Porcentaje de tiempo del proyecto ASIC/IC dedicado a la verificación [1]..... 2
Figura 2. Tendencias en metodologías de verificación FPGA (adaptada de [2]) 3
Figura 3. Arquitectura de una transacción de escritura [6] 10
Figura 4. Arquitectura de una transacción de lectura [6] 10
Figura 5. Protocolo handshake con la activación previa de la señal VALID [6].....15
Figura 6. Protocolo handshake con la activación previa de la señal READY [6].....15
Figura 7. Protocolo handshake con la activación simultánea de las señales READY y VALID [6] 15
Figura 8. Dependencias en las transacciones de lectura [6] 17
Figura 9. Dependencias en las transacciones de escritura [6] 18
Figura 10. Diferencias entre los tipos de verificación [7] 26
Figura 11. Tendencias en metodologías de verificación ASIC/IC [8].....29
Figura 12. Funcionalidades de SystemVerilog [9] 30
Figura 13. Evolución temporal de los lenguajes utilizados en la etapa de verificación [8] 32
Figura 14. Jerarquía parcial de la biblioteca de clases de UVM [11] 33
Figura 15. Fases UVM y orden de ejecución [12].....35
Figura 16. Arquitectura básica de un entorno de verificación [14] 39
Figura 17. Ejemplo de conexionado de tres UVC, dos activos y uno pasivo, a un DUV con tres interfaces [15] 40
Figura 18. Esquemático del protocolo handshake Sequencer-Driver [16] 43
Figura 19. Protocolo de handshake Sequencer-Driver [16].....44
Figura 20. Ejemplo de arquitectura de un entorno de verificación conectado a un DUV mediante dos interfaces [17].....46
Figura 21. Diagrama de bloques del crossbar (Fuente: [20]).....51
Figura 22. Estructura de configuración del módulo axi_xbar [19] 53
Figura 23. Generación de estímulos en el testbench ad-hoc del módulo crossbar original [21] 55
Figura 24. Función de monitorización en el testbench ad-hoc del módulo crossbar original [21] 56
Figura 25. Declaración de los párametros del módulo axi_xbar [19] 57
Figura 26. Definición del módulo axi_xbar_wrap.....58
Figura 27. Parámetros de configuración del módulo wrapper 59
Figura 28. Estructura de configuración 59
Figura 29. Concatenación de señales del protocolo AXI4 60
Figura 30. Definición del address map.....61

Figura 31. Referencia del DUV con su interfaz de entrada/salida y los parámetros de configuración 62
Figura 32. Asignación de las estructuras a su respectivo puerto de entrada o salida 63
Figura 33. Esquema parcial de los módulos crossbar y wrapper conectados al componente UVM Top Module 63
Figura 34. Protocolos de comunicación compatibles con el Questa Verification IP [22] ... 67
Figura 35. Flujo de trabajo para la creación de un testbench con la herramienta de configuración QVIP [24] 70
Figura 36. Interfaz de la herramienta configurator QVIP [24] 71
Figura 37. Módulo wrapper importado al configurator QVIP 72
Figura 38. Módulo QVIP maestro con interfaz AXI4 73
Figura 39. Gestión de las conexiones de los módulos QVIP 74
Figura 40. Conexionado parcial del módulo wrapper con un módulo QVIP maestro 75
Figura 41.Ventana de configuración del Address Map 76
Figura 42. Esquema parcial del entorno de verificación UVM y el módulo DUV 78
Figura 43. Declaración de las señales del protocolo AXI4 para el QVIP maestro 0 80
Figura 44. Definición de los parámetros básicos de un módulo QVIP maestro 81
Figura 45. Conexión de las señales con la interfaz AXI4 de un QVIP maestro 82
Figura 46. Definición de los puertos de la señal de reloj y la señal reset 83
Figura 47. Representación gráfica de las señales concatenadas del puerto esclavo de entrada 84
Figura 48. Canales del protocolo AXI4 utilizados en el puerto esclavo de entrada 84
Figura 49. Concatenación de las señales AXI4 en el puerto esclavo de entrada 85
Figura 50. Representación gráfica de las señales concatenadas del puerto maestro de salida 86
Figura 51. Canales del protocolo AXI4 utilizados en el puerto maestro de salida 86
Figura 52. Concatenación de las señales AXI4 en el puerto maestro de salida 87
Figura 53. Representación gráfica de las señales concatenadas del puerto maestro de entrada 88
Figura 54. Canales del protocolo AXI4 utilizados en el puerto maestro de entrada 88
Figura 55. Concatenación de las señales AXI4 en el puerto maestro de entrada 89
Figura 56. Representación gráfica de las señales concatenadas del puerto esclavo de salida 90
Figura 57. Canales del protocolo AXI4 utilizados en el puerto esclavo de salida 90
Figura 58. Concatenación de las señales AXI4 en el puerto esclavo de salida 91
Figura 59. Fragmento del código del componente UVM Test 92
Figura 60. Función que inicializa la secuencia 93

Figura 61. Fragmento 1 del código de la fase build_phase() del componente UVM Test .94
Figura 62. Fragmento 2 del código de la fase build_phase() del componente UVM Test .95
Figura 63. Fragmento 3 del código de la fase build_phase() del componente UVM Test .96
Figura 64. Tarea que ejecuta la fase run_phase()....96
Figura 65. Configuración del componente UVM Environment....98
Figura 66. Fragmento del código del componente UVM Environment....99
Figura 67. Variable de configuración de un IP AXI4....99
Figura 68. Declaración de la fase build_phase() en el componente UVM Environment..100
Figura 69. Código de un objeto UVM Sequence base....101
Figura 70. Código del archivo de simulación questa_run....102
Figura 71. Proporción del tiempo invertido en tareas por los ingenieros de verificación.104
Figura 72. Clase qvip_axi_xbar_vseq_wr_rd 108
Figura 73. Secuencia UVM axi4_sequence_lib_seq_incr....110
Figura 74. Secuencia UVM axi4_sequence_lib_seq_wrap....111
Figura 75. Secuencia UVM axi4_sequence_lib_seq 112
Figura 76. Secuencia UVM axi4_generic_api_excl_seq....113
Figura 77. Secuencia UVM axi4_generic_api_normal_seq 114
Figura 78. Secuencia UVM axi4_generic_api_wrap_seq 115
Figura 79. Topología resultante del entorno de verificación UVM 118
Figura 80. Ejemplo de implementación de la fase run_phase()....119
Figura 81. Tarea run_phase() del componente UVM Test 120
Figura 82. Señales handshake de la secuencia ejecutada por el maestro 0 123
Figura 83. Operaciones de lectura y escritura del maestro 0 en el esclavo 0....124
Figura 84. Operaciones de lectura y escritura del maestro 1 en el esclavo 6....127
Figura 85. Señales handshake de la secuencia ejecutada por el maestro 2 130
Figura 86. Operación de escritura ejecutada por el maestro 2 en los esclavos 2 y 4 131
Figura 87. Operación de lectura ejecutada por el maestro 2 en los esclavos 2 y 4 132
Figura 88. Operaciones de lectura y escritura del maestro 3 en el esclavo 1....134
Figura 89. Operaciones de escritura y lectura del maestro 4 en el esclavo 7....136
Figura 90. Primera transacción de escritura de los datos 64'h1122334455667788.....138
Figura 91. Segunda transacción de escritura de los datos 64'h1122334455667788.....138
Figura 92. Tercera transacción de escritura de los datos 64'h1122334455667788.....138
Figura 93. Cuarta transacción de escritura de los datos 64'h1122334455667788 139
Figura 94. Primera transacción de escritura del campo de datos de 128 bits....139
Figura 95. Segunda transacción de escritura del campo de datos de 128 bits....140
Figura 96. Operaciones de escritura del maestro 5 en los esclavos 3 y 5....141
Figura 97. Operaciones de lectura del maestro 5 en los esclavos 3 y 5....142

Índice de Tablas

Tabla 1. Señales asociadas al canal write data....11
Tabla 2. Señales asociadas al canal write address....11
Tabla 3. Señales asociadas al canal read data....12
Tabla 4. Señales asociadas al canal read address....13
Tabla 5. Señales asociadas al canal write response....14
Tabla 6. Codificación de los tipos de burst....19
Tabla 7. Codificación del parámetro PROT....20
Tabla 8. Codificación del parámetro SIZE....21
Tabla 9. Codificación del parámetro CACHE....22
Tabla 10. Puertos de entrada y salida del DUV....53
Tabla 11. Parámetros de configuración del módulo crossbar....54
Tabla 12. Biblioteca de secuencias de estímulos para el protocolo AXI4....105
Tabla 13. Interfaces API para la generación de transacciones de lectura....106
Tabla 14. Interfaces API para la generación de transacciones de escritura....106
Tabla 15. Lista de transacciones de cada módulo QVIP maestro....109
Tabla 16. Resultados obtenidos en la secuencia ejecutada por el maestro 1....126
Tabla 17. Resultados obtenidos de la secuencia enviada por el maestro 2 al esclavo 2 129
Tabla 18. Resultados obtenidos de la secuencia enviada por el maestro 2 al esclavo 4 129
Tabla 19. Configuración de la secuencia ejecutada por el maestro 3....133
Tabla 20. Configuración de la secuencia ejecutada por el maestro 4....135
Tabla 21. Configuración de la secuencia ejecutada por el maestro 5....137
Tabla 22. Condiciones hardware....151
Tabla 23. Condiciones software....152
Tabla 24. Factor de corrección según horas trabajadas....154
Tabla 25. Coste total de los recursos hardware....155
Tabla 26. Coste total de los recursos software....156
Tabla 27. Costes del material fungible....157
Tabla 28. Coste total del Trabajo Fin de Grado....159

ACRÓNIMOS

- ABV – *Assertion-Based Verification*
- AHB – *Advanced High-performance Bus*
- AMBA – *Advanced Microcontroller Bus Architecture*
- APB – *Advanced Peripheral Bus*
- API – *Application Programming Interfaces*
- ARM – *Advanced RISC Machine*
- ASIC – *Application-Specific Integrated Circuit*
- AVM – *Advanced Verification Methodology*
- AXI3 – *Advanced eXtensible Interface* 3
- AXI4 – *Advanced eXtensible Interface* 4
- BFM – *Bus Functional Model*
- CD-ROM – *Compact Disc-Read Only Memory*
- COITT – Colegio Oficial de Ingenieros Técnicos de Telecomunicación
- DRAM – *Dynamic Random-Access Memory*
- 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*
- ETHZ – Swiss Federal Institute of Technology Zurich
- FIFO – *First In First Out*
- FPGA – *Field-Programmable Gate Array*
- GITT – Grado en Ingeniería en Tecnologías de la Telecomunicación
- HDL – *Hardware Description Language*
- HVL – *Hardware Verification Language*
- I2C – *Inter-Integrated Circuit*
- IEEE – *Institute of Electrical and Electronics Engineers*
- IGIC – Impuesto General Directo Canario
- IP – *Intellectual Property*
- OOP – *Object-Oriented Programming*
- OVM – *Open Verification Methodology*
- PCIe – *Peripheral Component Interconnect Express*

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

Capítulo 1. Introducción

Durante este primer capítulo se detallarán los motivos y las necesidades que han dado lugar a la elaboración de este Trabajo Fin de Grado (TFG). Además, se presentarán los objetivos que persigue este TFG y la estructura del presente documento, con el propósito de proporcionar una idea general del trabajo y de esclarecer los diferentes puntos tratados a lo largo de la memoria.

1.1. Antecedentes

En la actualidad, el procedimiento de verificación es imprescindible para el correcto desarrollo de un producto y, más en concreto, en el diseño de sistemas *hardware* digitales. 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 como se muestra en la Figura 1.

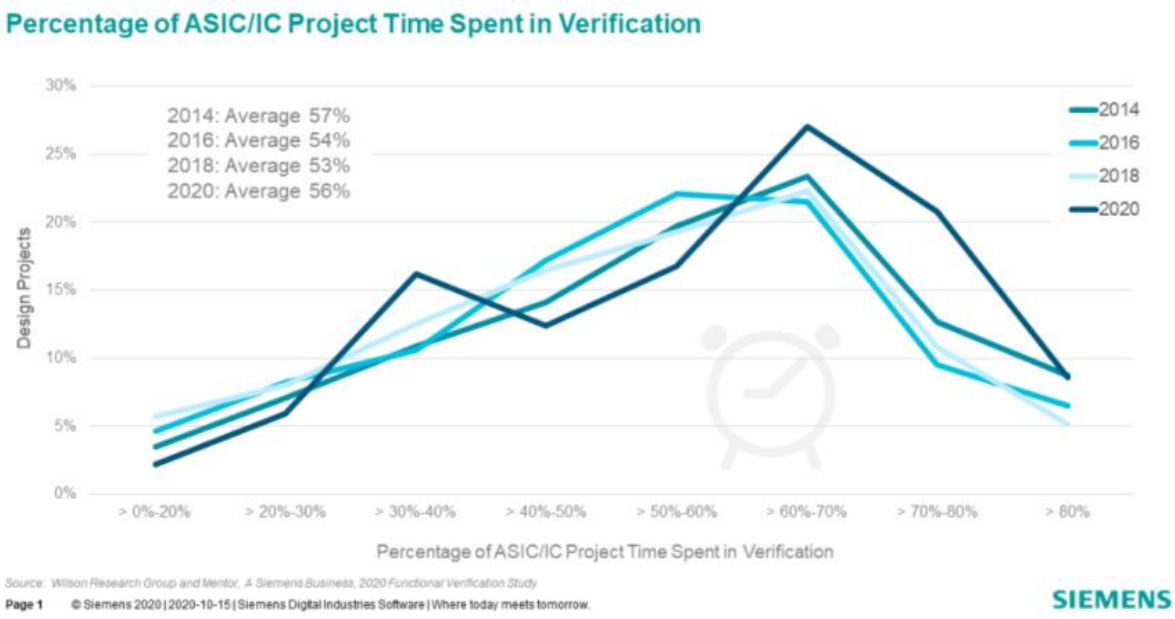

Figura 1. Porcentaje de tiempo del proyecto ASIC/IC dedicado a la verificación [1]

En los últimos años, debido al crecimiento de la demanda de sistemas electrónicos y a la mejora de la tecnología, estos sistemas se integran con un mayor número de componentes y se pretende que desempeñen más funcionalidades. Esta situación desemboca en sistemas mucho más complejos y, por lo tanto, también conlleva un procedimiento de verificación de mayor complejidad. Para solucionar este inconveniente nace la metodología UVM (*Universal Verification Methodology*), que se basa en el lenguaje de descripción y de verificación hardware (HDL, *Hardware Description Language* y, HVL, *Hardware Verification Language*) *SystemVerilog*. En la Figura 2 se observa la tendencia de adopción de varios estándares de metodologías de verificación, siendo *Acellera* UVM la metodología predominante para crear bancos de pruebas, tanto para diseños basados en FPGA como para diseños ASIC, en todo el mundo.

Hoy en día, debido a las exigencias de las principales compañías de diseño electrónico, las compañías EDA (*Electronics Design Automation*) tales como *Mentor Graphics, a Siemens Business, Cadence* o, incluso, *Xilinx*, han optado por proporcionar una librería de módulos de verificación que, si bien no exime del uso del lenguaje *SystemVerilog* para la realización de un *testbench*, sí que facilita enormemente la implementación de este. Estos módulos de verificación, denominados VIP (*Verification Intellectual Property*), 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).

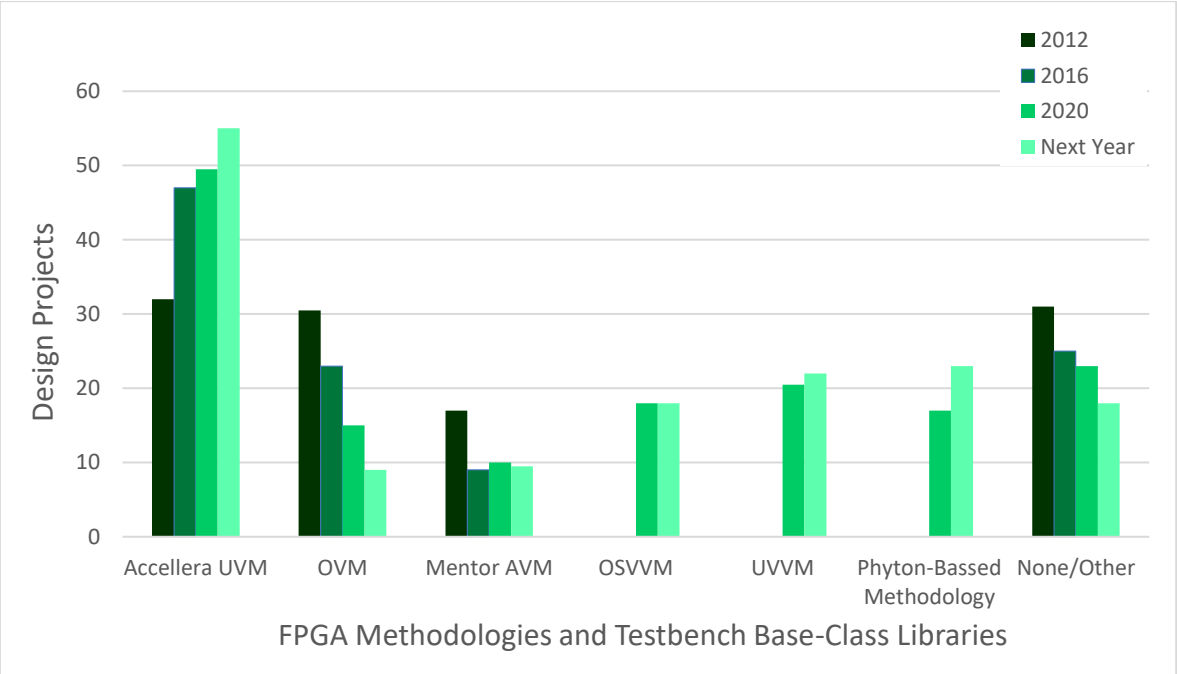

Figura 2. Tendencias en metodologías de verificación FPGA (adaptada de [2]*)*

La metodología UVM surge a partir de la metodología OVM (*Open Verification Methology*) de *Mentor Graphics* y está siendo desarrollada por el consorcio *Accellera* con el fin de fomentar la interoperabilidad de los módulos VIP (*Verification Intellectual Property*) y así estandarizar esta metodología de verificación. La adopción de UVM como metodología de verificación funcional conlleva una serie de características y beneficios para la comunidad que se detallan a continuación:

- Ofrece una biblioteca de clases abierta y unificada.
- Elimina la necesidad de interoperabilidad entre múltiples bibliotecas de verificación.
- Establece un flujo de trabajo común que optimiza el desarrollo e implementación del Plan de verificación.
- Permite la generación de componentes y objetos reutilizables que mejora la automatización de la verificación de diferentes dispositivos.

A diferencia de las técnicas de verificación tradicionales en las que se define un entorno para cada IP a verificar, la metodología UVM proporciona herramientas que permiten al ingeniero de verificación la creación de un entorno de verificación donde se garantiza la reusabilidad del código desarrollado [3]. De esta manera, muchos módulos UVM se pueden reutilizar en otro proyecto con ligeras adaptaciones. Esto conlleva la disminución del tiempo de verificación del comportamiento de un sistema, a pesar del esfuerzo inicial en las primeras etapas del proceso de verificación. Por lo tanto, en este

sentido, la metodología UVM se desmarca de la verificación *ad-hoc*, que es una verificación ajustada únicamente a los requisitos del sistema o del IP que se pretenda verificar.

Para desarrollar correctamente un entorno de verificación UVM se deben poseer amplios conocimientos del estándar UVM y una gran experiencia en diseño HDL. Unida a la dificultad innata de aprendizaje de la metodología UVM, se suma el límite de tiempo para la realización del presente Trabajo Fin de Grado. Por estas razones, en principio resultaría imposible el desarrollo completo de un *testbench* UVM a lo largo de este trabajo. Esto mismo ocurre a nivel empresarial, donde los ingenieros de verificación deben emplear un tiempo considerable en adaptarse a la metodología UVM. Por este motivo, las principales empresas EDA del sector del diseño *hardware,* han puesto a disposición de los equipos de verificación los módulos IP de verificación, también denominados VIP. En el caso particular de la empresa *Mentor Graphics*, *a Siemens Business*, estos módulos reciben el nombre de *Questa Verification* IP [4]. Estos IP permiten el desarrollo casi automático de un *testbench* UVM. Esto hará posible alcanzar, en tiempo y forma, el objetivo principal de este Trabajo Fin de Grado, centrado en el desarrollo de un *testbench* UVM. Por lo tanto, este TFG no se centrará en el desarrollo de los elementos activos que componen el entorno de verificación UVM, por lo que se subirán niveles de abstracción del sistema. Como se explicará posteriormente, el QVIP permitirá la configuración del entorno de verificación UVM mediante el uso de un módulo de configuración proporcionado por la herramienta. Esta configuración se realizará según las necesidades del sistema que se desea verificar y se hará teniendo en cuenta el protocolo de comunicación utilizado para el intercambio de información entre el QVIP y el DUV.

En el caso que concierne a este Trabajo Fin de Grado, el DUV posee una interfaz AXI4 (*Advanced eXtensible Interface* 4), por lo tanto, el entorno de verificación se configurará siguiendo ese mismo protocolo. El uso de la metodología UVM supone dos grandes ventajas:

- En primer lugar, se crearán módulos que se encargarán de traducir, a nivel de señales, las operaciones descritas a nivel de transacciones. Esto supone una gran ventaja para el ingeniero de verificación porque aísla todas las señales asociadas al protocolo de comunicación en el entorno de verificación, por lo que no se tendrán en cuenta las señales concretas de un protocolo durante el desarrollo de los *test*.
- Relacionado con la ventaja anterior, también se analiza la situación en la que, por un improvisto, se decide cambiar el protocolo de comunicación en medio de un proyecto. En este supuesto, sólo se deben sustituir los módulos QVIP del entorno

de verificación UVM, sin necesidad de tocar el *testbench* elaborado durante el mismo, y configurar el entorno con el nuevo protocolo elegido.

1.2. Objetivos

El objetivo principal de este Trabajo Fin de Grado consiste en el diseño de un *testbench* UVM usando un IP de verificación de *Mentor Graphics*. Las principales características de este IP son que está desarrollado con la metodología UVM, utiliza un protocolo de comunicación AXI4 y es totalmente configurable. Por estos motivos, este objetivo final se puede desgranar en los siguientes objetivos:

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

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

O3. Estudiar y comprender el IP de verificación de *Mentor Graphics*.

O4. Estudiar y entender el módulo IP (DUV) que se va a verificar, lo que va a permitir describir los *test* necesarios para realizar la comprobación del correcto funcionamiento del sistema.

O5. Realizar un *testbench* UVM para la verificación del DUV estudiado.

1.3. 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).

1.4. Estructura del documento

El presente documento está dividido en cuatro secciones: Memoria, Pliego de Condiciones, Presupuesto y Anexo. Respecto a la Memoria, va a estar estructurada en ocho capítulos y la bibliografía utilizada. A continuación, se va a especificar el contenido de cada uno de estos capítulos:

- **Capítulo 1. Introducción.** Con este primer capítulo se inicia la Memoria y se exponen los antecedentes y las necesidades que justifican la realización de este Trabajo Fin de Grado, también se hablará de los objetivos que persigue este TFG, el peticionario y la estructura del presente documento.

- **Capítulo 2. Protocolo AXI4.** En este segundo capítulo se profundizará en los conceptos básicos relacionados con el protocolo AXI4 debido a que es el protocolo de comunicación que interconectará el entorno de verificación y el dispositivo que se desea verificar y permitirá la comunicación entre ellos. Se analizarán factores claves del protocolo como los canales que lo forman y las señales asociadas a los mismos, el protocolo *handshake*, las relaciones entre los diferentes canales y, por último, se expondrá la configuración de las transacciones a través de diferentes parámetros.

- **Capítulo 3. *Universal Verification Methodology*.** El objetivo del tercer 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, como también sus principios básicos de funcionamiento y los diferentes componentes que forman parte de un entorno de verificación UVM.

- **Capítulo 4. Diseño a verificar (DUV).** Durante este capítulo se analizarán los conceptos claves del dispositivo a verificar durante la realización del presente Trabajo Fin de Grado, enfatizando la adaptación de este módulo a la herramienta *Questa Verification* IP *configurator*.

- **Capítulo 5. Desarrollo de un entorno de verificación UVM mediante la herramienta QVIP *configurator*.** En el quinto capítulo se presentarán los aspectos claves de los módulos QVIP y de la herramienta QVIP *configurator*. Además, se

mostrará y se explicará detalladamente el entorno de verificación UVM generado por esta herramienta.

- **Capítulo 6. Desarrollo del *testbench* UVM.** Antes de explicar el código implementado relativo al *testbench* UVM, se presentará brevemente la biblioteca de secuencias UVM básicas que incorpora la herramienta QVIP para la estimulación del DUV y la posterior comprobación del correcto funcionamiento del mismo.

- **Capítulo 7. Resultados.** Durante el séptimo capítulo se mostrarán los resultados obtenidos al estimular de manera controlada al DUV.

- **Capítulo 8. 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 2. Protocolo AXI4

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

Es una interfaz paralela de alto rendimiento, síncrona y multimaestro orientada a sistemas de altas prestaciones funcionando a alta frecuencia. Este protocolo es adecuado para diseños de alto ancho de banda y baja latencia, proporciona flexibilidad en la implementación de arquitecturas de interconexión, cumple con los requisitos de interfaz de una amplia gama de componentes y es compatible con versiones anteriores de las interfaces AHB (*Advanced High-performance Bus*) y APB (*Advanced Peripheral Bus*) [5]. Durante este capítulo se van a destacar aquellos aspectos de mayor relevancia para el desarrollo de este Trabajo Fin de Grado. Para profundizar en mayor medida se aconseja consultar la especificación oficial del protocolo [6].

2.1. Canales y señales

Una característica clave de este protocolo es que las fases de dirección, control y datos están separadas. Por lo tanto, existen 5 canales diferentes: *write data, write address, read data, read address* y *write response*. Cada uno de estos canales tiene una serie de

señales asociadas, de las cuales se estudiarán las más importantes para este trabajo. Se debe tener en cuenta que hay señales que sólo pueden ser enviadas por el maestro o por el esclavo.

Como se deduce fácilmente por los nombres de los canales, una transacción de escritura se realizará sobre los canales *write address, write data* y *write response*. La Figura 3 muestra la arquitectura básica de una transferencia de escritura entre un maestro y un esclavo.

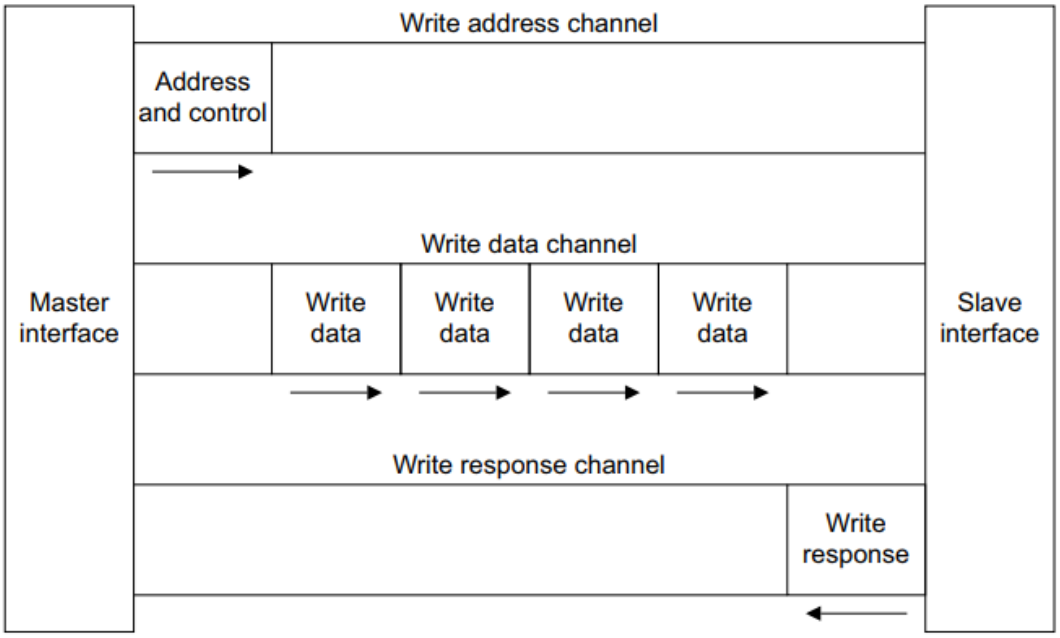

Figura 3. Arquitectura de una transacción de escritura [6]

En la Figura 4 se puede observar la arquitectura básica de una transferencia de lectura, la cual se apoya en los canales *read address* y *read data*.

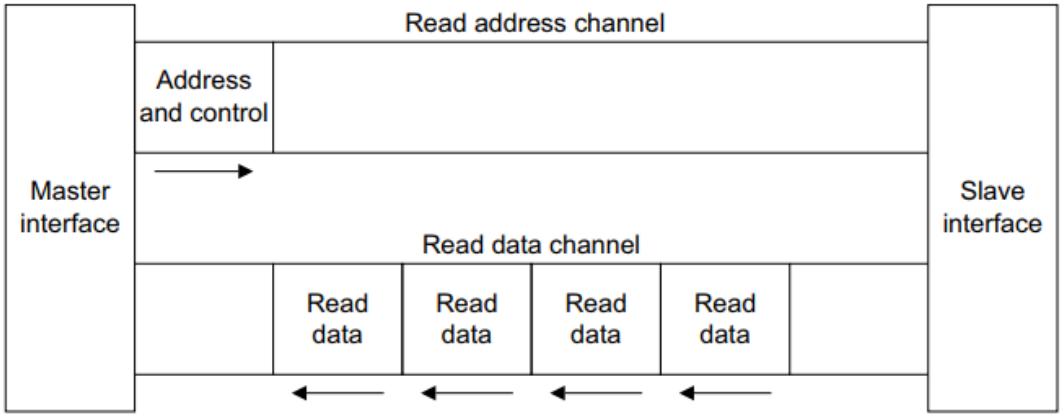

Figura 4. Arquitectura de una transacción de lectura [6]

Cada uno de los canales consta de un conjunto de señales de información, *bus* de datos/direcciones y señales de control de protocolo *VALID* y *READY*. Por un lado, la fuente de información utiliza la señal de control *VALID* para indicar cuándo hay un dato o información de control disponible en el canal. Por otro lado, el destino utiliza la señal de control *READY* para mostrar cuándo puede aceptar esa información. Por último, la fuente de datos, en transferencia de datos, utiliza una señal denominada *LAST*, presente en los canales de escritura y de lectura, la cual indica la finalización de la transferencia de datos.

- ***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.

- ***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 asociados 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* es opcional 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.

- ***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 *RDATA*, así como la información asociada a la misma.
RREADY	Maestro	Esta señal indica que el maestro puede aceptar nuevos datos de lectura.

- ***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.
ARSIZE	Maestro	Tamaño de la ráfaga. Este *bus* indica el tamaño de cada transferencia en la ráfaga.
ARBURST	Maestro	Indica el tipo de ráfaga y 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 lectura válida de la información de dirección y control.
ARREADY	Esclavo	Señal *READY* para las direcciones de lectura. Esta señal indica que el esclavo puede aceptar una dirección y señales de control asociadas.

- ***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 cana 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. Es *bus* de datos opcional 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. Protocolo de *handshake* en AXI4

Se debe tener en cuenta que los 5 canales explicados en el apartado anterior utilizan el mismo protocolo de *handshake* con señales de *VALID* y *READY* para transferir la dirección, los datos y la información de control. El protocolo AXI4 posee un mecanismo de control de flujo bidireccional, lo que significa que tanto el maestro como el esclavo pueden controlar la velocidad a la que la información se transmite entre ellos.

Por un lado, la fuente genera una señal *VALID* para indicar que la dirección o los datos, así como toda la información de control relacionada, están disponible. Por otro lado, el destino genera una señal *READY*, con el propósito de indicar que puede aceptar la información. La transferencia tiene lugar una vez que ambas señales se encuentran a nivel alto.

En la Figura 5 se puede ver un ejemplo del protocolo de *handshake* implementado donde, en primer lugar, la fuente activa la señal VALID después de T1. Por lo tanto, en ese instante, deposita en el bus la dirección o datos, e información de control. En T2, el destino genera la señal READY indicando que puede aceptar la información generada por la fuente. Para que la transferencia se pueda producir, el destino debe mantener estable su información hasta que se produzca dicha transferencia. En caso de no proporcionar nueva

información, la fuente deberá desactivar la señal *VALID* en T3. De lo contrario, se estará validando una nueva información.

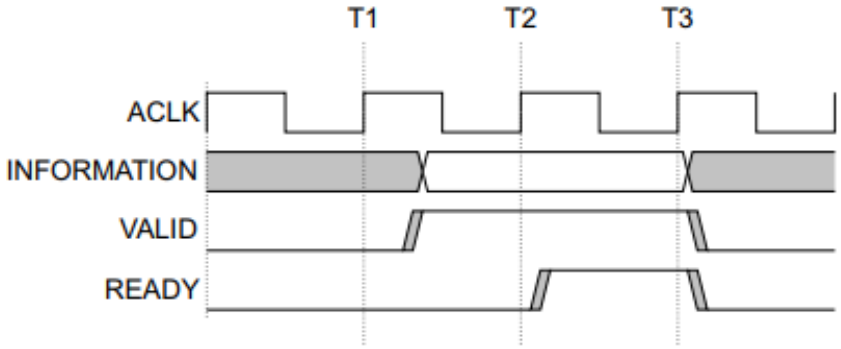

Figura 5. Protocolo handshake con la activación previa de la señal VALID [6]

También se puede dar el caso en el que el receptor tenga activada la señal *READY* antes de que la fuente valide un nuevo dato activando la señal *VALID* a nivel alto. Por lo tanto, ahora la transferencia tiene lugar en un único ciclo. Esta situación se puede analizar en la Figura 6.

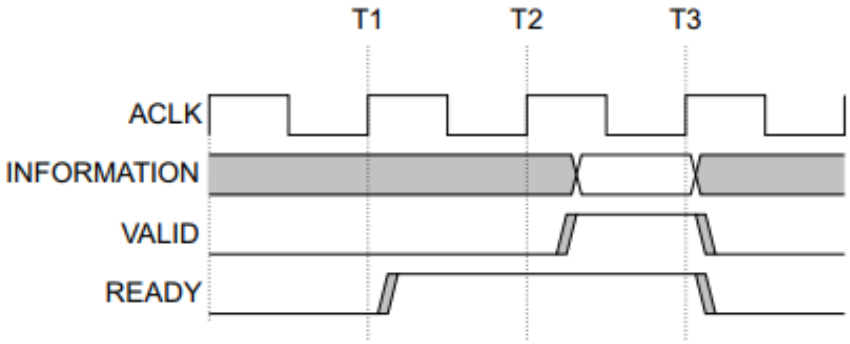

Figura 6. Protocolo handshake con la activación previa de la señal READY [6]

Otro posible caso, similar al anterior, es la activación de ambas señales al mismo tiempo. Al igual que en el caso anterior, la transferencia ocurre en un único ciclo. Este último caso descrito se puede visualizar en la Figura 7.

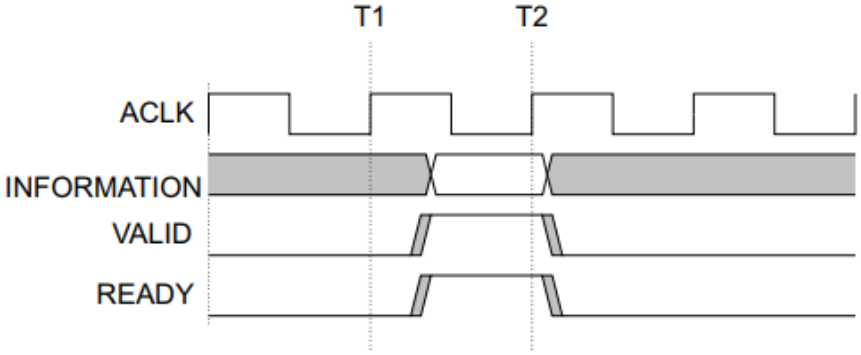

Figura 7. Protocolo handshake con la activación simultánea de las señales READY y VALID [6]

2.3. Relaciones entre los canales

Para que funcione correctamente, en el protocolo AXI4 existen algunas reglas de dependencia entre las señales de *handshake*. De ahí que este protocolo requiera de ciertas relaciones entre estas señales:

- La respuesta de escritura (en el canal *write response*) siempre debe ir detrás de la última transferencia de escritura.
- Los datos de lectura siempre deben estar después de la dirección a la cual se dirigen estos datos de lectura.

Por lo tanto, existe una serie de dependencias entre algunas señales comunes de los diferentes canales. En primer lugar, la señal *VALID* de la interfaz AXI que valida la información nunca debe depender de la señal *READY* de la interfaz AXI que recibe información. En segundo lugar, otra dependencia general es que la interfaz que está recibiendo la información puede esperar para activar su señal *READY* hasta que detecte una señal *VALID*. Aunque esto puede ser así, un diseño más eficiente se daría si la señal *READY* estuviese activa antes de detectar la activación de la correspondiente señal *VALID*.

En los sucesivos apartados se mostrarán unos diagramas de dependencia para las operaciones de lectura y escritura. En estos diagramas, las flechas de una sola cabeza apuntan a señales que pueden afirmarse antes o después de la señal que se encuentra al comienzo de la flecha. Por otro lado, las flechas de dos puntas apuntan a señales que deben afirmarse sólo después de la afirmación de la señal al comienzo de dicha flecha.

2.3.1. Dependencias en las transacciones de lectura

En este apartado se estudiarán las dependencias de las señales en las transacciones de lectura. Estas dependencias se muestran en la Figura 8. Sabiendo que las señales *ARVALID* y *RREADY* pertenecen al maestro y, *ARREADY* y *RVALID* al esclavo, del presente diagrama se pueden sacar las siguientes conclusiones:

- Por un lado, el maestro no debe esperar a que el esclavo active la señal *ARREADY* antes de activar la señal *ARVALID*. Por otro lado, el esclavo puede activar *ARREADY* antes de que *ARVALID*, pero también puede esperar a que el maestro la active antes.
- El esclavo no puede activar la señal *RVALID* para indicar que el dato válido está disponible hasta que *ARVALID* y *ARREADY* hayan sido activadas. El esclavo no debe esperar a que la señal *RREADY* esté a nivel alto antes de activar la señal

RVALID. Eso sí, el maestro puede esperar o no a que *RVALID* se encuentre activa, para así activar su señal *RREADY*.

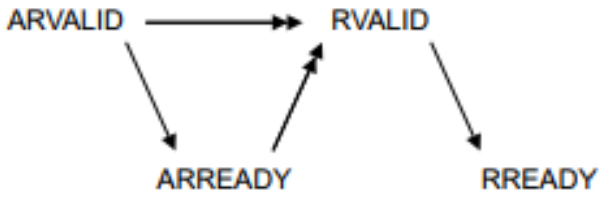

Figura 8. Dependencias en las transacciones de lectura [6]

2.3.2. Dependencias en las transacciones de escritura

Durante este apartado se analizarán las dependencias existentes entre las señales en las transacciones de escritura, esto es lo que se muestra en la Figura 9. Sabiendo que las señales *AWVALID*, *WVALID* y *BREADY* pertenecen al maestro y, *AWREADY*, *WREADY* y *BVALID* al esclavo, del presente diagrama se pueden sacar las siguientes conclusiones:

- El maestro no debe esperar a que el esclavo active las señales *AWREADY* o *WREADY* para poder activar *AWVALID* o *WVALID*. Aunque no se recomienda, también se puede dar el caso de que el esclavo espere a que se activen dichas señales antes de notificar que puede recibir información. Por lo tanto, se recomienda que la señal *READY* del esclavo esté activado para realizar el *handshake* en un solo ciclo. Es importante tener en cuenta que, a su vez, la señal *WVALID* también depende de la activación de la señal *WLAST* emitida por el propio esclavo, indicando mediante su activación la finalización de la transferencia de datos de escritura en modo *burst*.
- Para que el esclavo pueda activar la señal BVALID, avisando por el canal *write response* que tiene un dato válido para enviar, deben estar activas las señales WVALID, WREADY, AWVALID y AWREADY con anterioridad, pero no debe esperar por la señal BREADY del maestro. El maestro podría activar esta señal antes de que el esclavo active la señal BVALID, aunque puede esperar por ella.

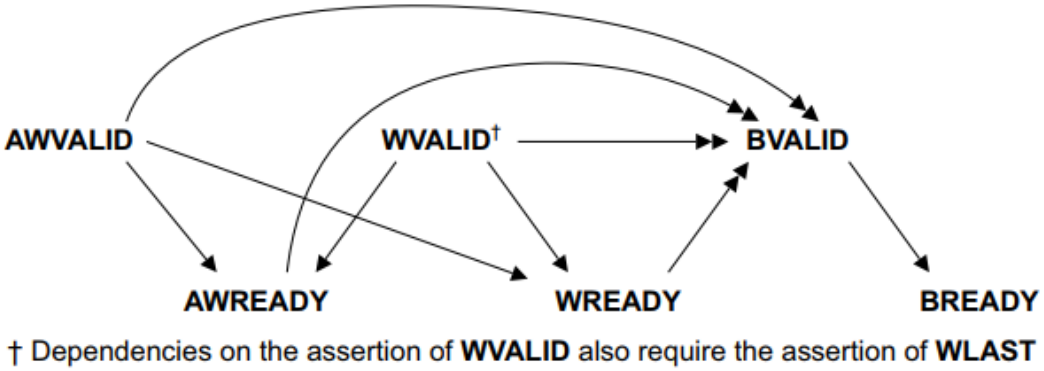

Figura 9. Dependencias en las transacciones de escritura [6]

2.4. Configuración de las transacciones

Desde el punto de vista de la verificación, unos de los aspectos más importantes del protocolo AXI, es la caracterización de las transacciones mediante los parámetros *SIZE*, *LEN*, *BURST*, *CACHE*, *PROT*, *REGION*, *USER* e *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 han sido definidos de forma detallada en la Tabla 2 y en la Tabla 4.

A continuación, se especificarán los diferentes valores que pueden adoptar cada uno de estos parámetros. Se debe tener en cuenta que como se analizarán parámetros que, como ya se ha comentado, se configuran durante la fase de direcciones (*write address* y *read address)*, para abreviar se utilizará la siguiente nomenclatura: *AxSIGNAL*. De esta manera, se englobaría la correspondiente señal tanto en el canal de escritura como en el de lectura.

2.4.1. Parámetro *BURST*

El protocolo AXI4 define 3 tipos de ráfagas (*burst*) y en la Tabla 6 se muestra la codificación de la señal *AxBURST* (tanto *AWBURST* como *ARBURST*), dependiendo del tipo de ráfaga a enviar.

Tabla 6. Codificación de los tipos de burst

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

A continuación se describen las principales características de cada tipo de ráfaga existente en el protocolo AXI4:

- **FIXED**: también denominada ráfaga fija. En este tipo, la dirección es la misma para todas las transferencias de la ráfaga. Además, los *bits* de validación de cada *byte* son constantes para todos los pulsos de la ráfaga. Sin embargo, los *bytes* reales que tiene *WSTRB* activados pueden diferir para cada escritura de la ráfaga. Este tipo de ráfaga se utiliza para accesos repetidos a la misma ubicación, como ocurre al cargar o vaciar un FIFO.
- **INCR:** este modo es el incremental. En una ráfaga incremental, la dirección de cada transferencia en la ráfaga es un incremento de la dirección de la transferencia anterior. El valor de incremento depende del tamaño de la transferencia. A modo de ejemplo, la dirección para cada transferencia en una ráfaga con un tamaño de cuatro *bytes* es la dirección anterior más cuatro. Este tipo de ráfaga se utiliza para acceder a la memoria secuencial normal.
- **WRAP:** una ráfaga de envoltura tiene cierto parecido a la ráfaga incremental anteriormente explicada, pero cuenta con una diferencia clave: la dirección se envuelve a una dirección más baja si se alcanza un límite de dirección superior. Este tipo de ráfaga se utiliza para accesos a la línea de datos de una memoria caché.

2.4.2. Parámetro *PROT*

El protocolo AXI4 proporciona señales de permiso de acceso con el fin de protegerse de transacciones ilegales. La Tabla 7 presenta las diferentes combinaciones para el parámetro *AxPROT*.

Tabla 7. Codificación del parámetro PROT

AxPROT[2:0]	Valor	Función
0	0	Acceso no privilegiado
	1	Acceso privilegiado
1	0	Acceso seguro
	1	Acceso no seguro
2	0	Acceso a los datos
	1	Acceso a las instrucciones

Seguidamente se describen, brevemente, cada uno de los tipos de accesos:

- **Acceso privilegiado y no privilegiado.** Un maestro AXI puede admitir más de un nivel de privilegio operativo y extender este concepto de privilegio al acceso a la memoria.
- **Acceso seguro y no seguro.** En un maestro AXI se pueden configurar estados operativos seguros y no seguros y, de esta manera, extender este concepto de seguridad al acceso a la memoria.
- **Acceso a los datos y a las instrucciones.** Con este *bit* se indica si la transacción realizada se trata de un acceso a datos o a instrucciones.

2.4.3. Parámetro LEN

El parámetro *LEN* marca el número exacto de transferencias en una ráfaga. Esta información determina el número de transferencias de datos asociados con la dirección validada. El protocolo AXI3 sólo soporta una longitud de *burst* de 1 a 16 transferencias para todos los tipos de *burst*. En cambio, AXI4 amplía este límite de manera significativa para el modo incremental del *burst*, dejando enviar de 1 a 256 transferencias, aunque en el resto de los modos mantiene el límite marcado por el protocolo AXI3.

2.4.4. Parámetro SIZE

El parámetro *SIZE* especifica el número máximo de *bytes* que se pueden enviar por transferencia, tanto por el canal de escritura como por el de lectura. En la Tabla 8 se muestra la manera de codificar los campos de configuración del parámetro *SIZE* según las necesidades del sistema. De esta manera, el tamaño de cualquier transferencia no debe

superar el ancho del *bus* de datos de ninguno de los agentes que intervienen en dicha transacción.

Tabla 8. Codificación del parámetro SIZE

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

2.4.5. Parámetro *CACHE*

El parámetro *CACHE* está definido con un ancho de 4 *bits* y cada uno de ellos configura de alguna manera dicho parámetro. A continuación, se analizará el significado de cada uno de estos *bits*:

- **AxCACHE[0]:** hace referencia al *bufferable bit*, el cual indica que en caso de estar activado, la interconexión o cualquier componente puede retrasar un número de ciclos la llegada de una transacción a su respectivo destino final. Por lo general, es relevante sólo en escrituras.
- **AxCACHE[1]:** en este *bit* el protocolo AXI4 hace una modificación respecto al AXI3, ya que originalmente se llamaba *cacheable bit* y con AXI4 se pasó a llamar *modifiable bit*, pero no cambia la funcionalidad requerida. Con este *bit* se indica que las características de la transacción pueden ser modificadas en caso de estar este *bit* a nivel 1. En caso negativo, sería una memoria no modificable y las señales que no serían susceptibles de ser modificadas son las siguientes: *AxADDR, AxREGION, AxSIZE, AxLEN, AxBURST, AxLOCK* y *AxPROT*.
- **AxCACHE[2]:** se configura el *read-allocate bit*. Este *bit* define la política de acceso a memoria caché en operaciones de lectura.
- **AxCACHE[3]:** asigna el valor del *write-allocate bit*. Este *bit* define la política de acceso a memoria caché en operaciones de escritura.

Como ya se ha comentado, el protocolo AXI4 realiza ciertos cambios de relevancia en el parámetro *CACHE* y se desmarca de algunos conceptos ya desfasados. Siguiendo

esta línea, también cambia la codificación de configuración del parámetro *CACHE*, en la Tabla 9 se detalla la codificación que propone el protocolo AXI4, tanto para el canal de escritura como el de lectura.

Tabla 9. Codificación del parámetro CACHE

ARCACHE[3:0]	AWCACHE[3:0]	Tipo de memoria
0000	0000	*Device Non-bufferable*
0001	0001	*Device Bufferable*
0010	0010	*Normal Non-cacheable Non-bufferable*
0011	0011	*Normal Non-cacheable Bufferable*
1010	0110	*Write-through No-allocate*
1110	0110	*Write-through Read-allocate*
1010	1110	*Write-through Write-allocate*
1110	1110	*Write-through Read and Write-allocate*
1011	0111	*Write-back No-allocate*
1111	0111	*Write-back Read-allocate*
1011	1111	*Write-back Write-allocate*
1111	1111	*Write-back Read and Write-allocate*

2.4.6. Parámetro *REGION*

De manera opcional, el conjunto de señales asociadas a la interfaz AXI4 puede ampliarse para soportar dos identificadores de región de 4 *bits*, que se utilizan para identificar de forma exclusiva hasta 16 regiones distintas.

El uso del parámetro *REGION* permite que una interfaz física de un esclavo pueda proporcionar múltiples interfaces lógicas, cada una ubicada en zonas diferentes del mapa de direcciones del sistema. Por lo tanto, el esclavo no tiene que soportar la decodificación de direcciones entre las diferentes interfaces lógicas.

Existen múltiples modelos de uso para el parámetro *REGION*, de los que destacan los siguientes:

- Un periférico puede ser configurado con su ruta de datos y sus registros de control en diferentes ubicaciones del mapa de direcciones y, para acceder a la información se pueden hacer a través de una única interfaz, por lo que no es necesario que el esclavo realice una decodificación de direcciones.

- Permite la opción de programar un esclavo y que se comporte de manera diferente en función de la región de memoria en la que opere. Esto permite, por ejemplo, que un esclavo proporciona acceso de lectura y escritura en una región concreta, pero en otra región sólo permite acceso de lectura.

Al utilizar un mismo esclavo con dos o más regiones se debe asegurar la correcta señalización del protocolo y el orden correcto de las transacciones. Por ello, el esclavo debe proporcionar las respuestas a dos transacciones a diferentes regiones con el mismo *ID* en el orden correspondiente. En el caso de que un esclavo implemente menos de 16 regiones también se debe asegurar la señalización del protocolo cuando se intente acceder a una región no soportada por dicho esclavo. Se proponen dos posibles soluciones para asegurar el correcto funcionamiento del protocolo en estos casos y son las siguientes:

- Notificar un error cada vez que una transacción acceda a una región no soportada.
- Asignar un alias a las regiones reconocidas en todas las regiones no soportadas, con el fin de garantizar una respuesta conforme al protocolo para cada uno de los accesos.

2.4.7. Parámetro *USER*

Aunque el protocolo AXI4 pone a disposición del usuario el parámetro *USER*, no es del todo recomendable hacer uso de estas señales. Esta recomendación se debe a que el protocolo AXI no define las funciones que deberán cumplir estas señales y su uso incorrecto podría conllevar problemas de interoperabilidad en el caso de que dos componentes usaran las mismas señales del parámetro *USER* de manera incompatible.

El parámetro *USER* está presente en todos los canales (*AWUSER*, *ARUSER*, *WUSER*, *RUSER* y *BUSER*), pero esto no quiere decir que exista una dependencia entre dichos parámetros. De hecho, no es necesario que el parámetro *USER* sea compatible entre ellos. Por esta razón, la decisión de definir o no el parámetro *USER* en un diseño concreto es independiente para cada canal.

2.4.8. Parámetro *ID*

2.4.8.1. Requisitos de orden

Las señales *AWID* y *ARID* de los canales *address write* y *address read*, respectivamente, se utilizan para indicar el identificador de la transacción. Así, por ejemplo, si un maestro está compuesto por varios sub-bloques, este identificador podría indicar qué

sub-bloque inició la transferencia. En lo relativo a la ordenación de las transacciones existen una serie de reglas, las cuales se exponen a continuación:

- Las transacciones de diferentes maestros carecen de restricciones de orden, por lo que se pueden completar en cualquier orden.
- Las transacciones del mismo maestro y con diferente parámetro *ID* tampoco cuenta con ningún tipo de restricción de orden.
- Las transferencias de datos para una secuencia con transacciones de lectura o escritura con el mismo valor del parámetro *ID* deben completarse en el orden en el que el maestro emitió las direcciones.
- El protocolo AXI4 no tiene restricciones de orden entre las transacciones de lectura y escritura, es decir, pueden completarse en cualquier orden, incluso si el valor de la señal *ARID* (propio de una transacción de lectura) es el mismo que el valor de la señal *AWID* (utilizado en una transacción de escritura).

2.4.8.2. Ancho de los campos *ID* de transacción

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

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

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

Capítulo 3. *Universal Verification Methodology*

A lo largo del presente capítulo se detallarán las principales características de la metodología UVM, resaltando las cuestiones más importantes para el desarrollo de este Trabajo Fin de Grado. En primer lugar, se profundizará en el concepto de la verificación de los sistemas electrónicos digitales, haciendo hincapié en la importancia de esta etapa en el flujo de diseño de sistemas *hardware* digitales. En segundo lugar, con la finalidad de proporcionar una idea general de la metodología, se realizará una pequeña introducción de la metodología UVM, en la que se contextualizarán los factores claves de la creación de dicha metodología y la necesidad real que cubre a pesar de la costosa curva de aprendizaje que supone. Después, se introducirán algunos conceptos básicos del lenguaje HVL (*Hardware Verification Language*) *SystemVerilog*, que están intrínsecamente relacionados con la metodología UVM. Por último, se profundizará en la metodología propiamente dicha prestando especial atención a la creación del entorno de verificación y de todos los componentes que lo conforman, su interconexión, sus funciones y cómo se comunican entre ellos.

3.1. Verificación de sistemas electrónicos digitales

La verificación es el proceso por el cual se determina el correcto funcionamiento de un diseño. De esta manera, la verificación pretende asegurar que el diseño en cuestión

cumple las especificaciones de diseño marcadas, satisfaciendo la necesidad que se desea cubrir.

Como se comentó en el primer capítulo, 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. Además, 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.

Cabe destacar que la verificación es el proceso inverso a la etapa de diseño, ya que durante la verificación se toma una implementación y se confirma si dicho sistema cumple con las especificaciones indicadas. Por esta razón, en cada etapa del diseño de un sistema digital, debe realizarse la correspondiente verificación para asegurar la integridad del diseño en los diferentes niveles de abstracción.

Este proceso se puede elaborar de diferentes maneras: según su objetivo, su estrategia de seguimiento o su tipo de ejecución, entre otros. Se puede llevar a cabo mediante técnicas de verificación estáticas (verificación formal) y dinámicas (verificación funcional). Esta última ha cobrado especial relevancia en los últimos años y, de hecho, el presente Trabajo Fin de Grado 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 10, en relación a la información 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*. En los sucesivos epígrafes se profundizará en estos tres enfoques.

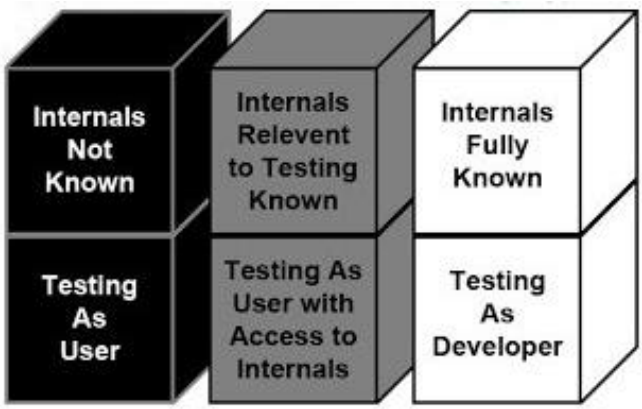

Figura 10. Diferencias entre los tipos de verificación [7]

3.1.1. Verificación *Black-Box*

Cuando un ingeniero de verificación sigue este enfoque, el diseño que se desea verificar se trata como una “caja negra”. Esto quiere decir que no se conoce cómo está diseñado dicho 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 a 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.

3.1.2. Verificación *White-Box*

Siguiendo este enfoque, el ingeniero de verificación conoce en su totalidad el DUV, como se describe en la Figura 10. 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*.

3.1.3. Verificación *Grey-Box*

Las pruebas que se realizan siguiendo el enfoque *Grey-Box* son una combinación de las anteriormente explicadas, conciliando ambas filosofías. 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.

3.2. Introducción a la metodología UVM

UVM es un estándar del consorcio *Accellera* 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 las incesantes 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:

- Como se ha comentado, cabe destacar 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.
- Debido a la inherente popularidad de UVM, la biblioteca de clases básicas es compatible con la mayoría de los simuladores del mercado, por lo tanto no existe dependencia a un simulador específico.

- 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.
- Esta metodología cuenta con 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 visualizando la Figura 11.

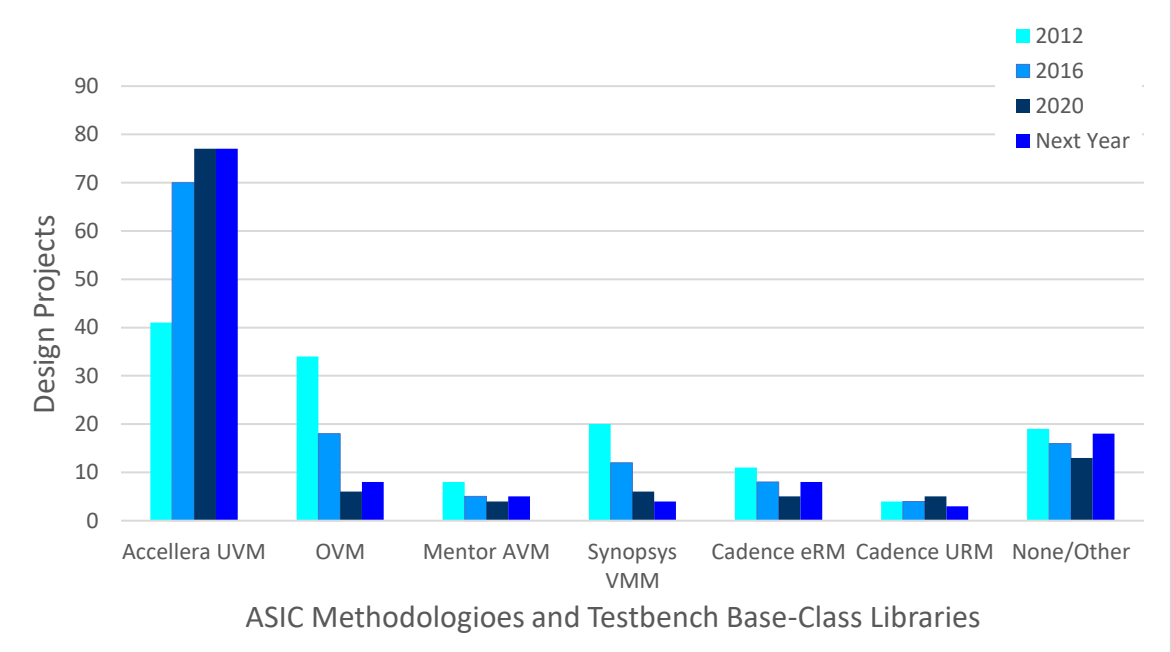

Figura 11. Tendencias en metodologías de verificación ASIC/IC [8]

En lo que resta del presente capítulo se profundizará en los aspectos más relevantes de esta metodología y se comentarán algunas características del lenguaje de

descripción *hardware* que se utiliza para el desarrollo de cualquier proyecto basado en UVM.

3.3. El lenguaje de descripción *hardware SystemVerilog*

SystemVerilog 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 12 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).

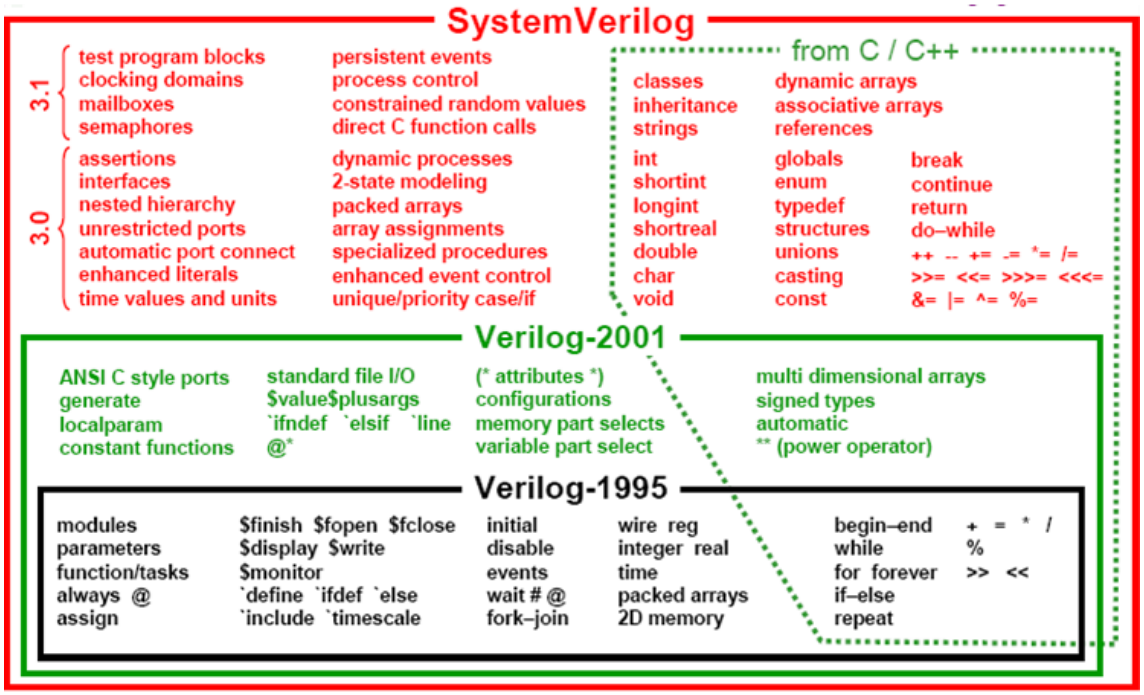

Figura 12. Funcionalidades de SystemVerilog [9]

Como ya se ha comentado, *SystemVerilog* ha evolucionado 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 por varias razones, de las que se destacan las siguientes [10]:

- 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* y 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 13 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).

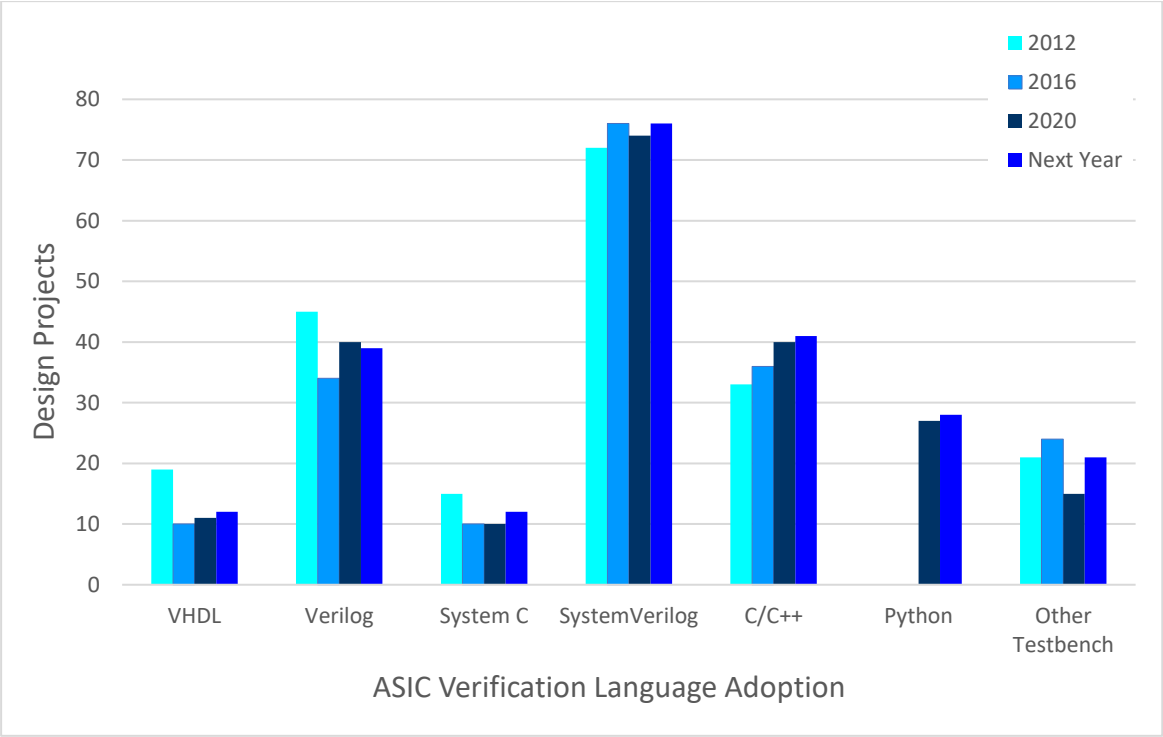

Figura 13. Evolución temporal de los lenguajes utilizados en la etapa de verificación [8]

3.4. Biblioteca de clases básicas

Para 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 14.

- **`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 `uvm_object`, la clase `uvm_component` incorpora más métodos como, por ejemplo, métodos que buscan y recorren la jerarquía de componentes y, además,

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.

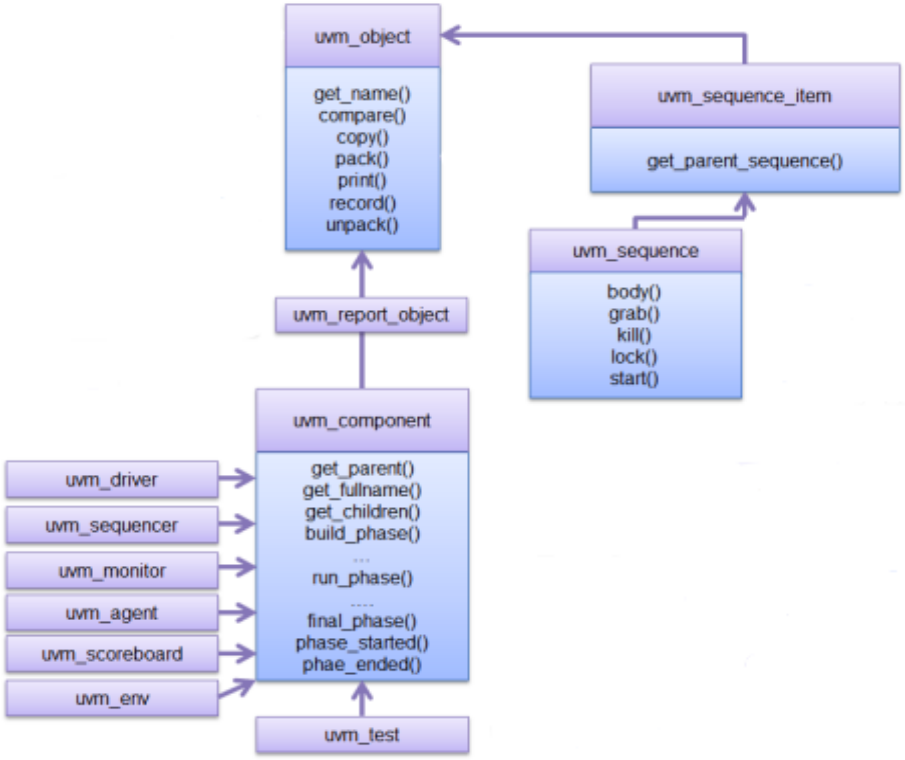

Figura 14. Jerarquía parcial de la biblioteca de clases de UVM [11]

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.

3.5. Mecanismo de fases de UVM

En líneas generales, 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 en UVM se detalla en la Figura 15. 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*. Como se puede observar en la Figura 15, 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 simulación 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.

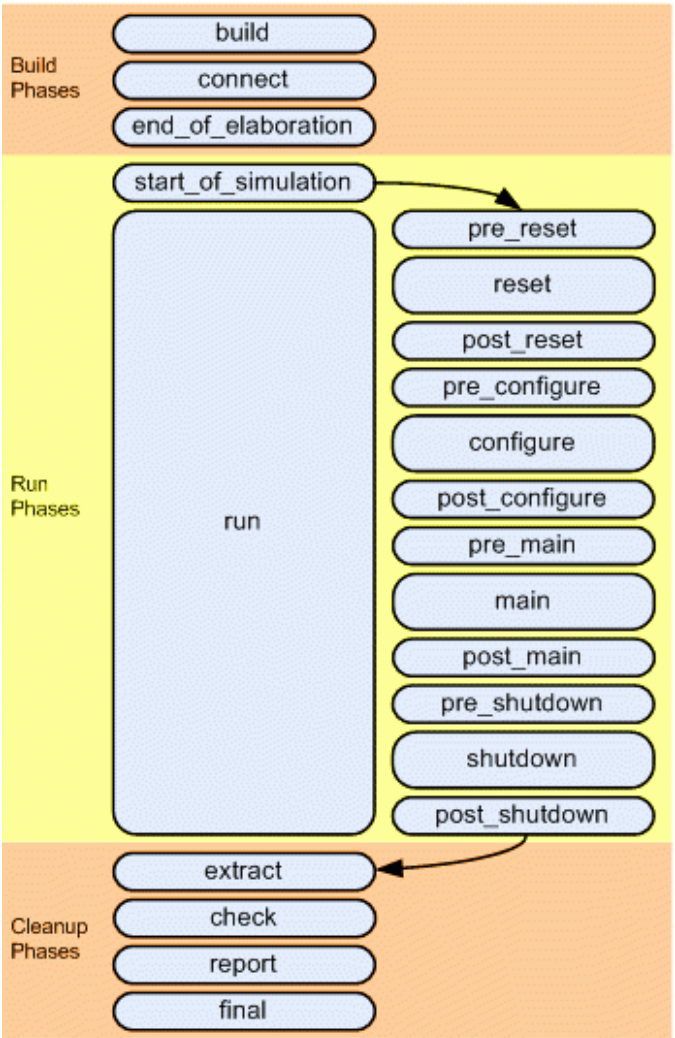

Figura 15. Fases UVM y orden de ejecución [12]

3.6. El mecanismo *Factory* en UVM

Similar al patrón de diseño *Factory* de la programación orientada a objetos, la metodología UVM utiliza un patrón parecido para registrar 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. Además, el patrón *Factory* facilita la sustitución de objetos existentes por cualquiera de sus objetos de clase hijo que heredan todo de la clase padre, todo ello sin modificar el código. Con esto, se consigue una mejora de la flexibilidad y de la escalabilidad del *testbench*.

Todas las clases que derivan de `uvm_object` y `uvm_component` requieren que se registren en la clase *Factory* usando unas macros de UVM específicas. Las macros son, en este caso, `` `uvm_object_utils `` y `` `uvm_component_utils ``, a las cuales se les pasará como argumento el nombre del objeto o componente, respectivamente, que se quiera registrar. Dicho nombre debe coincidir con el nombre de la clase a la que se hace referencia. En caso de registrar un objeto o componente de tipo parametrizable, se deberán utilizar las macros `` `uvm_object_param_utils `` y `` `uvm_component_param_utils ``.

Estas macros, en primer lugar, implementan la función `get_type_name()`, la cual devolverá el objeto o el componente como una variable tipo *string*. En segundo lugar, ejecutan la función `create()` y después registran el objeto o el componente en la *Factory* mediante la variable tipo *string* generada anteriormente.

3.7. Sistema de mensajes e informes

Es necesario que, cuando se ejecuta la simulación de un entorno de verificación, se reporten una serie de mensajes informativos, los cuales son de gran utilidad para la depuración del sistema. 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 que se proporciona 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. Los niveles de `VERBOSITY` en orden de menor a mayor nivel de filtrado son los siguientes: `UVM_NONE`, `UVM_LOW`, `UVM_MEDIUM`, `UVM_HIGH`, `UVM_FULL` y `UVM_DEBUG`. Sólo en un caso se puede aportar este argumento y es en la macro *`uvm_info*, aunque no es obligatorio, ya que, en caso de no rellenar dicho argumento, se le asignará un valor por defecto. El resto de las macros poseen un nivel interno de filtrado con un valor interno de `UVM_NONE` que el usuario no puede modificar, de modo que no se puedan desactivar por error estableciendo el nivel de `VERBOSITY` demasiado bajo.

3.8. Modelado a nivel de transacción TLM

La metodología UVM proporciona una serie de interfaces y canales de comunicación TLM (*Transaction-Level Modeling*) [13], 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 UVM *Driver* y *Sequencer*, 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 complejo con una estimulación más elaborada. Por esta razón, se necesita un nivel de abstracción muy elevado. Para ello, se habilita la comunicación TLM que, como ya se ha explicado, 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.

En determinados procesos de verificación, el ingeniero podría necesitar que parte de la información circulase entre varios componentes, no sólo entre dos, límite que imponen los puertos TLM *ports* y los puertos TLM *exports*, ya que establecen comunicación únicamente entre dos componentes. Por esta razón, la metodología UVM soluciona esta dificultad con la creación de los puertos TLM *analysis ports* y los puertos TLM *analysis exports*. De esta manera, un puerto TLM *analysis ports* permite la conexión de varios puertos TLM *analysis exports*, posibilitando una transmisión *broadcast* mediante el uso del método *write()* .

3.9. Entorno de verificación UVM

Un entorno de verificación UVM está formado por varios niveles de abstracción donde se crean diversos componentes de verificación organizados en una jerarquía específica. Un entorno UVM permite la posibilidad de conseguir un mayor control sobre la generación de los estímulos que se envía al DUV, la recepción y análisis de las señales que se captan a la salida y la respectiva comparación de estos datos con los esperados.

En la Figura 16 se visualiza un ejemplo básico de jerarquía UVM, donde en el módulo principal (*tb_top*) se referencia el sistema que se pretende verificar y el entorno de verificación UVM. Para que ambos módulos se puedan comunicar, con el fin estimular y monitorizar el DUV, se conectan a una interfaz virtual (*interface*). El nivel más alto de la jerarquía es el componente UVM *Test* que contiene todos los componentes UVM *Environment*, ya que se pueden crear uno o varios. En el caso particular de la Figura 16, se incluye un único componente UVM *Environment*. A su vez, el componente UVM *Environment* contiene los diferentes componentes UVM de verificación tales como el UVM *Sequencer*, UVM *Driver*, UVM *Monitor* y el UVM *Scoreboard*.

En los sucesivos epígrafes se procederá a explicar con mayor detalle las características propias de cada uno de los componentes que conforman el entorno de verificación diseñado con la metodología UVM.

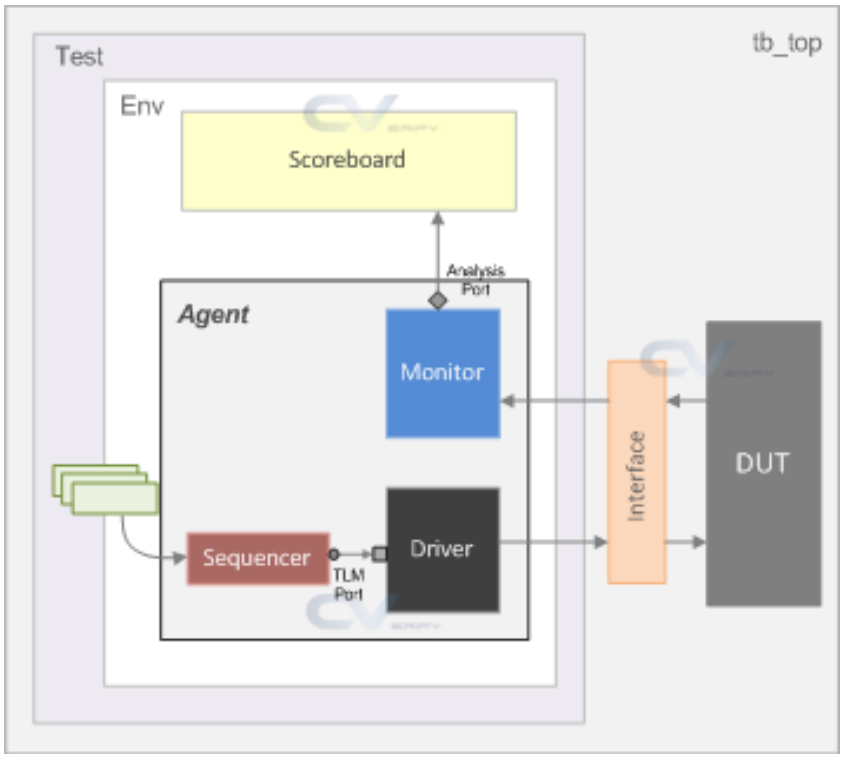

Figura 16. Arquitectura básica de un entorno de verificación [14]

3.9.1. Componente UVM *Agent*

Un componente UVM *Agent* 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 *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. A su vez, el componente UVM *Monitor* tiene la función de muestrear las señales que capte en la interfaz del dispositivo que se desea verificar.

Un componente UVM *Agent* se puede configurar como un elemento activo o pasivo. Por un lado, cuando un componente UVM *Agent* se encuentra en el modo activo tiene la función de conducir los estímulos generados en el *testbench* hacia el DUV. Por el contrario, un componente UVM *Agent* pasivo sólo debe monitorizar la interfaz del dispositivo en cuestión. Por esta razón, un componente UVM *Agent* configurado en modo pasivo no

necesita un componente UVM *Sequencer* ni un componente UVM *Driver*, sólo necesita un componente UVM *Monitor*, ya que debe poder acceder a la interfaz. En la Figura 17 se muestra la disposición de tres componentes UVM *Agent*, de los cuales dos se encargan de la estimulación del sistema a verificar, actuando en modo activo y, el otro se encuentra configurado en modo pasivo, por lo que su función será monitorizar la interfaz correspondiente.

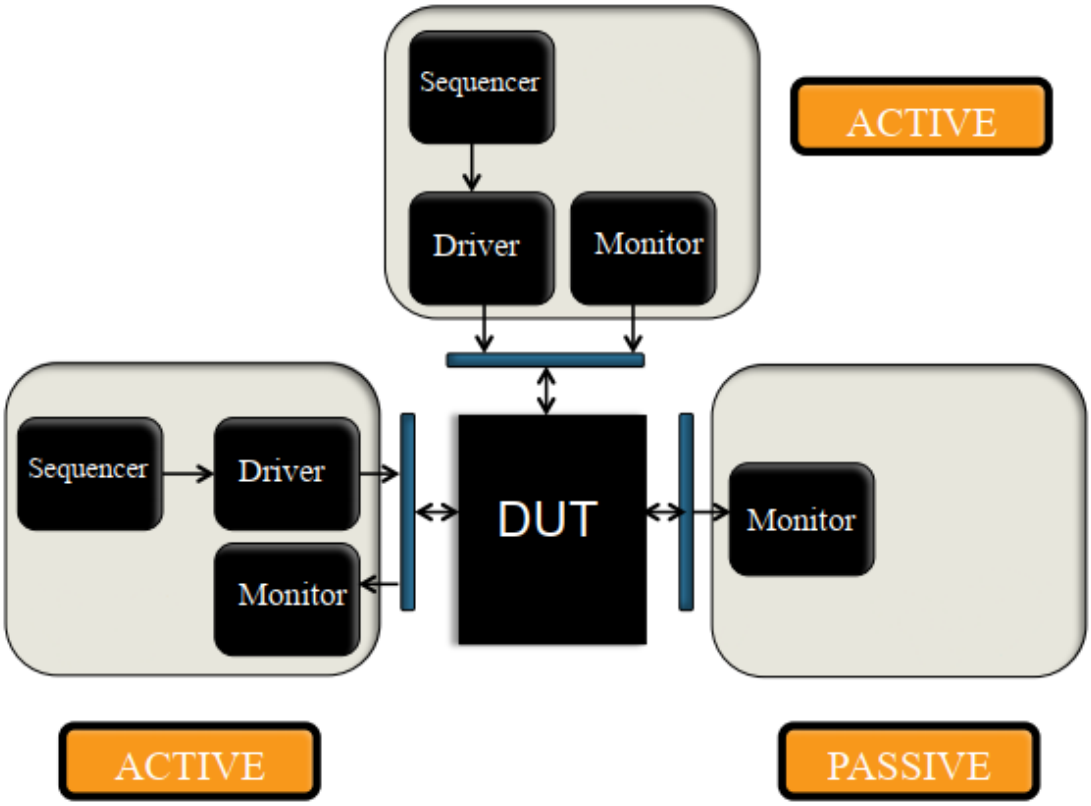

Figura 17. Ejemplo de conexionado de tres UVC, dos activos y uno pasivo, a un DUV con tres interfaces [15]

De forma predeterminada todos los agentes se encuentran en un estado activo. Para configurar un componente UVM *Agent* es necesario modificar una variable interna llamada `is_active` que es del tipo `uvm_active_passive_enum`. Esta variable indicará en qué estado se encontrará el agente en cuestión. De esta manera, al darle el valor *UVM_ACTIVE*, el agente estará en modo activo y cuando se le atribuye el valor *UVM_PASSIVE* se configurará en modo pasivo.

Al ser un componente que deriva de la clase `uvm_agent` tiene un conjunto de funciones heredadas. Una de ellas es la función llamada `get_is_active()`, la cual devolverá el modo en el que opera el agente UVM solicitado.

3.9.1.1. Componente UVM *Sequencer*

Un componente UVM *Sequencer* 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 *Driver* se realiza a través de un protocolo de *handshake* que se explicará más adelante. Este protocolo es sumamente importante ya que libera al ingeniero de verificación 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.

3.9.1.1.1. Secuencias UVM

En un entorno de verificación UVM, 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. Son ejecutados por un componente UVM *Sequencer*, asignado a un componente *Agent*, y 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 (`run_phase`).

En este apartado se van a destacar dos macros que proporciona UVM*:* `` `uvm_do (SEQ_OR_ITEM) `` y `` `uvm_do_with (SEQ_OR_ITEM,CONSTRAINTS) ``. Estas macros permiten 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 que ha sido 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.

3.9.1.1.2. Transacciones en UVM

Una transacción UVM es un ítem 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 la entrada del DUV en cuestión. Por lo general, se suelen aleatorizar los campos de datos que componen una transacción UVM, con posibilidad de que el ingeniero de verificación cree ciertas restricciones durante dicho proceso.

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`.

3.9.1.1.3. Protocolo *handshake Sequencer-Driver*

Existe un mecanismo que permite el envío de transacciones desde el componente UVM *Sequencer* hasta componente UVM *Driver* con el único fin de conseguir estimular el dispositivo que se desea verificar. En la Figura 18 y en la Figura 19 se expone de manera esquemática el protocolo *handshake* entre estos dos componentes. Cuando una secuencia se dirige al componente UVM *Sequencer,* se descompone en una serie de ítems de transacción que, a su vez, serán enviados al componente UVM *Driver* y 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.

Por un lado, 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 *Sequencer* que se enumeran a continuación:

- En primer lugar, 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 le concede permiso al componente UVM *Driver* para acceder a la transacción mediante la función `start_item(transaction_item_handle)`.
- En tercer lugar, se aleatoriza la transacción respetando las restricciones programadas por el ingeniero de verificación.
- 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.

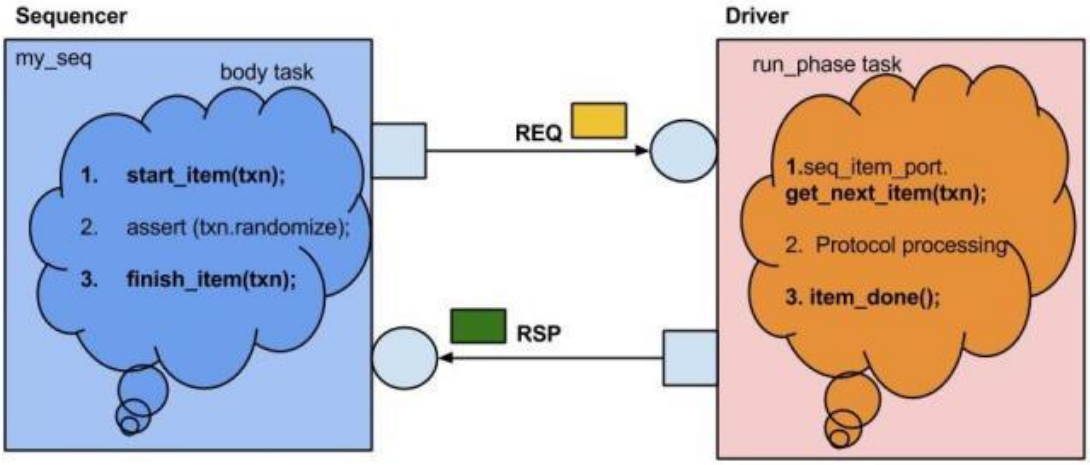

Figura 18. Esquemático del protocolo handshake Sequencer-Driver [16]

Por otro lado, el componente UVM *Driver* también realiza una serie de acciones para completar la comunicación con el componente UVM *Sequencer* que se enumeran en los siguientes puntos:

- La función `get_next_item(transaction_item_handle)` bloquea el procesamiento hasta que el objeto de transacción esté disponible en la solicitud del *Sequencer*. Posteriormente, una vez disponible, `get_next_item()` devuelve un puntero al objeto de la transacción en curso.
- Otra manera que tiene el *Driver* de solicitar un objeto de transacción es haciendo uso del método no bloqueante `try_next_item()`. Este método devolverá un valor *null* en caso de que el objeto de solicitud no esté disponible en el componente UVM *Sequencer*. De lo contrario, devolverá un puntero al objeto.
- Para indicar al componente UVM *Sequencer* que el proceso de transferencia ha sido completado, el componente UVM *Driver* hace uso de la función `item_done()` y la envía por el puerto TLM. El método `item_done()` sólo se debe usar después de finalizar los métodos `get_next_item()` o `try_next_item()`.

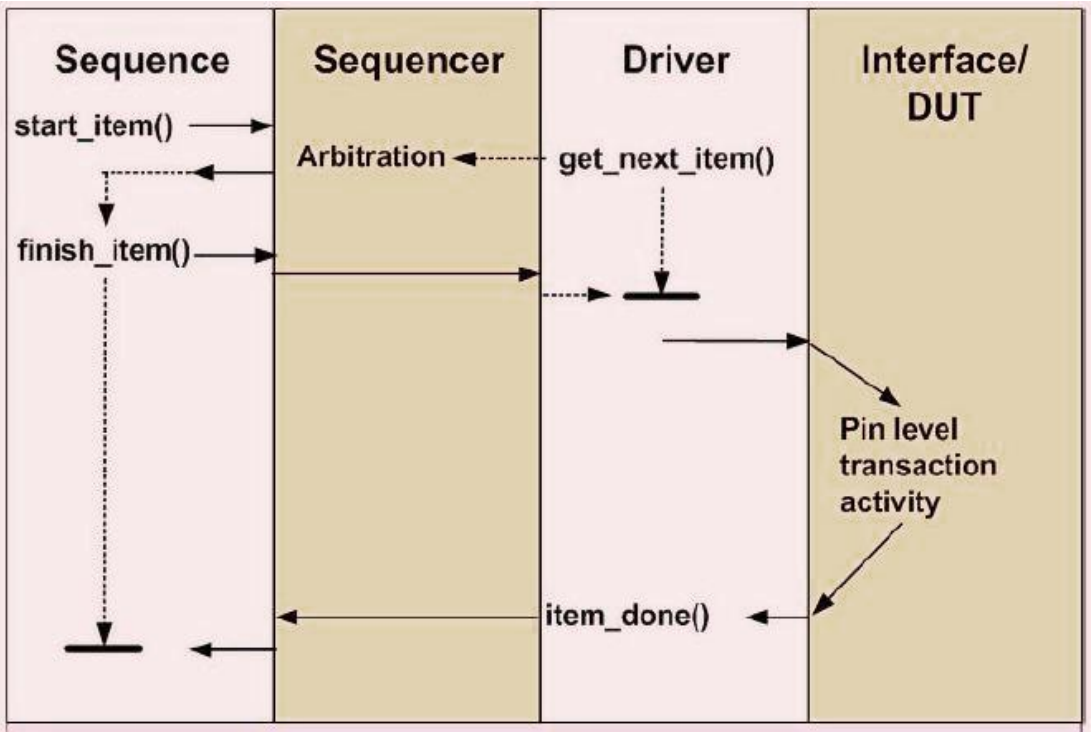

Figura 19. Protocolo de handshake Sequencer-Driver [16]

3.9.1.2. Componente UVM *Driver*

Un UVM *Driver* 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 el *Driver*. Esta interfaz se implementa mediante un objeto tipo *interface* en *SystemVerilog*.

3.9.1.3. Componente UVM *Monitor*

Un UVM *Monitor* es un componente que hereda 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 *TLM Analysis Port* para ponerlos a disposición del resto del entorno de verificación.

Aunque en ocasiones puede resultar útil, por lo general, el componente *Monitor* no realiza ningún procesamiento interno sobre las transacciones generadas. Comúnmente, delega 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.

En el caso de que el ingeniero de verificación diseñe un componente *Monitor* con funcionalidades complejas, se puede introducir una completa abstracción desde el nivel de señales, separando el muestreo de señales del resto de funcionalidades del componente en cuestión. Esto se consigue introduciendo un nuevo componente llamado UVM *Collector* que deriva directamente de la clase `uvm_component`, ya que no tiene clase base directa. Las funciones del UVM *Collector* se limitaría a la captación de la información que se envía a través de la interfaz virtual y a la generación de transacciones para el componente UVM *Monitor*.

3.9.2. Componente UVM *Scoreboard*

Un UVM *Scoreboard* 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.

Un componente UVM *Scoreboard* se puede utilizar en diferentes niveles de la jerarquía del entorno. Lo más común es crearlo dentro de un UVC que compruebe el comportamiento de un DUV como se ve en la Figura 20, pero también se puede implementar para comprobar las comunicaciones entre diferentes componentes *Agent* o a nivel de componente UVM *Testbench*.

Por ejemplo, durante la ejecución del *testbench* se realiza una operación de escritura en un registro, el UVM componente *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 UVM *Scoreboard* notifica un error en la operación de escritura.

Para que el 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* como se presenta en la Figura 20. De esta manera, el *Scoreboard* recibirá todas las transacciones que lleguen al dispositivo que se desea verificar.

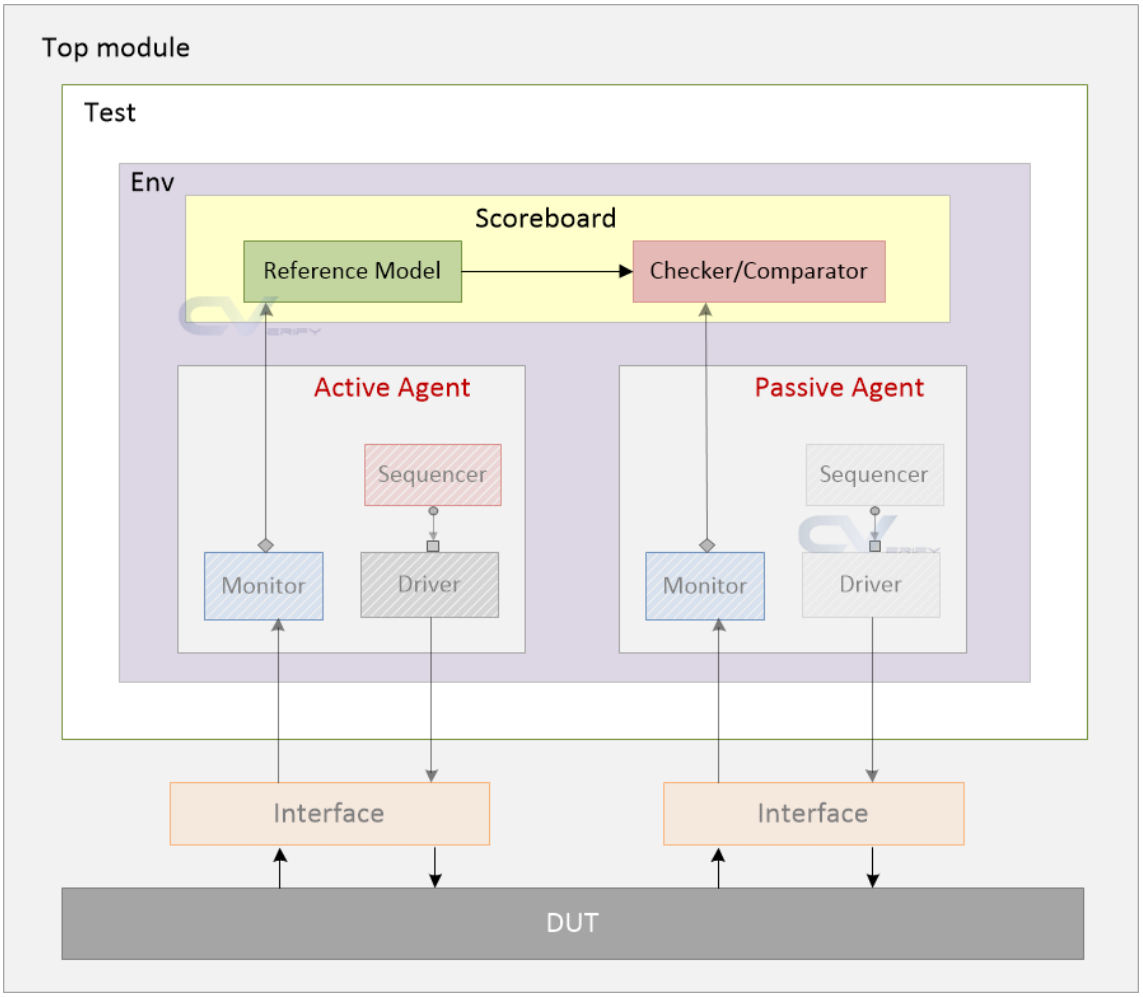

Figura 20. Ejemplo de arquitectura de un entorno de verificación conectado a un DUV mediante dos interfaces [17]

El ingeniero de verificación debe implementar una cola FIFO en el componente UVM *Scoreboard*, donde se guardará la información enviada por el componente *Monitor*. El *Scoreboard* irá consumiendo las transacciones guardadas en la FIFO. Con este mecanismo se evita la pérdida de información cuando la velocidad de transmisión de paquetes del *Monitor* es mayor que la velocidad de consumo del *Scoreboard.* Para mayor facilidad, UVM proporciona dos clases para la implementación de las colas FIFO: `uvm_tlm_fifo` y `uvm_tlm_analysis_fifo`.

3.9.3. Componente UVM *Environment*

Un componente UVM *Environment* es un contenedor que se deriva de la clase `uvm_env`. El componente *Environment* posee propiedades de configuración que permiten personalizar la topología del entorno de verificación y el comportamiento de los UVC que integra, pudiendo enfocarse a diversas tareas de verificación. Como se puede visualizar en la Figura 20, el componente *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*. Teóricamente es posible crear referencias a todos estos componentes sin necesidad de crear el UVM *Environment* y hacerlo directamente en el UVM *Test*, pero esto no se recomienda en ningún caso. Esto se debe por 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.

3.9.4. Componente UVM *Test*

El componente UVM *Test* engloba todo el entorno de verificación UVM y a todos los componentes que lo componen, siendo el nivel más alto de la jerarquía de dicho entorno. Por lo tanto, cuando comienza el proceso de creación del entorno se ejecuta la fase `build_phase()` y 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 *test* 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 *Test* se derivan otros *test* 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*.

También es recomendable que se programe una biblioteca de secuencias con la finalidad de cubrir todos los casos que se contemplan en el Plan de Verificación.

3.9.5. Componente UVM *Testbench*

Todos los componentes de verificación, interfaces y DUV son referenciados en el módulo *Top* (también conocido como UVM *Testbench*), que es del tipo *module* en *SystemVerilog*. Por lo tanto, este componente es un contenedor estático que contiene todos los componentes que conforman el entorno de verificación UVM (incluido el componente UVM *Test*) y el DUV. Este componente juega un papel importante en la metodología UVM y, a continuación, se van a nombrar algunas de sus funciones:

- 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 del mismo. 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*. En un diseño cuyos bloques digitales operan en múltiples frecuencias de reloj, el componente UVM *Testbench* debe generar múltiples relojes. Por lo tanto, la generación de la señal de reloj no siempre será tan sencilla de implementar. Además, para comprobar diferentes funcionalidades del dispositivo se pueden modificar determinados parámetros como la frecuencia, el ciclo de trabajo y la fase de manera dinámica, por lo que el componente UVM *Testbench* necesitaría cierta infraestructura para tolerar estas operaciones dinámicas.
- 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 UVM *Driver* y el UVM *Monitor,* podrán acceder a las interfaces para interactuar con el DUV, con el fin de estimular y monitorizar dicho dispositivo.
- Realiza la llamada del método `run_test()`, el cual inicia la ejecución de las distintas fases que se explicaron en el apartado 3.5, lo que provoca el comienzo de la simulación UVM.

Capítulo 4. Diseño a verificar (DUV)

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. Como el diseño de este DUV no es el objetivo de este TFG, se ha utilizado un sistema elaborado por un tercero. 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 [18]. 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 a lo largo del desarrollo del presente Trabajo Fin de Grado, siempre referenciando a su autor como se ha hecho. 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.

4.1. Módulo *axi_xbar*

Para la realización del presente TFG se ha tomado como referencia un componente de interconexión, cuya utilización está muy generalizada en los protocolos de comunicación. 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 esto 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*, por lo tanto, no importará lo que haya dentro del módulo ni como se haya implementado. Aunque se trate al DUV como una “caja negra”, 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. El conocimiento de estos parámetros justificará, entre otras cosas, la creación de un módulo, denominado *wrapper*, que permitirá adaptar el código original del *crossbar* al entorno de verificación UVM generado por el entorno de configuración utilizado sin necesidad de modificar el código original del DUV.

4.1.1. Características básicas

En *GitHub* se encuentra un repositorio [19] 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 un 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 puede 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, siempre intentando la simplificación del código para extender la compatibilidad con el mayor número posible de herramientas EDA.

4.1.2. Visión general del diseño

De todos los módulos que contiene el repositorio de *GitHub*, el que concierne al presente Trabajo Fin de Grado es el módulo *axi_xbar*, el cual implementa un componente de interconexión llamado *crossbar*. La implementación de este dispositivo resulta muy útil en algunos protocolos de comunicación y su uso está muy extendido. En la Figura 21 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 mayor modularidad y versatilidad del componente de interconexión *crossbar*, el usuario puede configurar el número de puertos maestros y esclavos, entre otros muchos parámetros configurables.

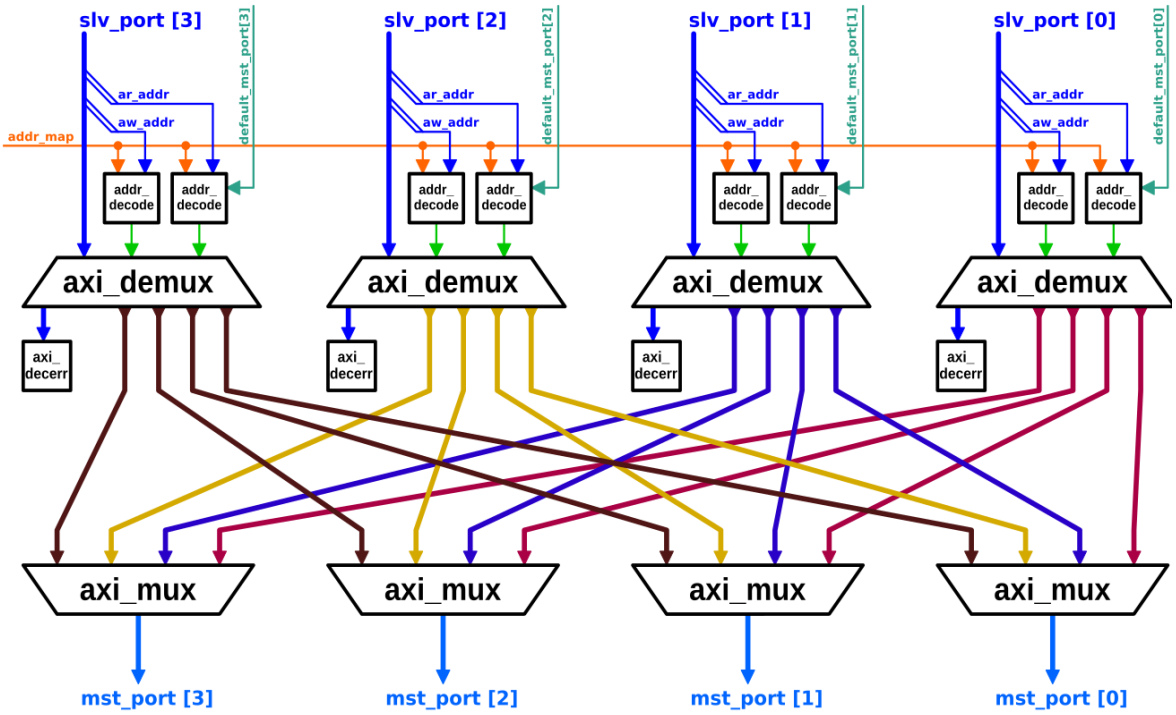

Figura 21. Diagrama de bloques del crossbar (Fuente: [20]*)*

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 maestro. Se deben tener en cuenta las siguientes consideraciones:

- Un *address map* debe contener al menos una regla.
- Varias reglas se pueden asociar 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. Este conflicto se resuelve dándole prioridad a la regla de posición más significativa en el *address map*.

- 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 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 ello, 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 se realice una 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, pero no posee exactamente una interfaz AXI4. En la Tabla 10 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 saber 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 10. 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.

4.1.3. Configuración del sistema

El módulo *crossbar* es parametrizable, es decir, tiene ciertos parámetros que el usuario puede configurar para que el sistema se adapte y cubra sus necesidades. Esta configuración se realiza a través del parámetro *Cfg,* que contiene una estructura denominada *xbar_cfg_t*, definida en el paquete *axi_pkg*. La definición de dicha estructura se describe en la Figura 22 con sus respectivos campos configurables.

```
1   // Configuration for `axi_xbar`.
2   typedef struct packed {
3     int unsigned   NoSlvPorts;
4     int unsigned   NoMstPorts;
5     int unsigned   MaxMstTrans;
6     int unsigned   MaxSlvTrans;
7     bit            FallThrough;
8     xbar_latency_e LatencyMode;
9     int unsigned   AxiIdWidthSlvPorts;
10    int unsigned   AxiIdUsedSlvPorts;
11    int unsigned   AxiAddrWidth;
12    int unsigned   AxiDataWidth;
13    int unsigned   NoAddrRules;
14  } xbar_cfg_t;
```

Figura 22. Estructura de configuración del módulo axi_xbar [19]

A su vez, en la Tabla 11 se resume la utilidad de cada uno de los campos que componen la estructura de configuración *xbar_cfg_t*. Muchos de estos parámetros no serán utilizados durante este trabajo, pero otros son cruciales para la configuración del sistema y su completa adaptación al entorno de verificación UVM.

Tabla 11. Parámetros de configuración del módulo crossbar

Nombre del parámetro	Tipo de dato	Definición
NoSlvPorts	*int unsigned*	Número de puertos esclavos AXI del *crossbar*, es decir, indica cuántos módulos maestros AXI se pueden conectar al DUV.
NoMstPorts	*int unsigned*	Número de puertos maestros AXI del *crossbar*, por lo que indica el número de módulos esclavos AXI que se pueden conectar al DUV.
MaxMstTrans	*int unsigned*	Indica el número máximo de transacciones *in flight* que puede tener cada puerto esclavo.
MaxSlvTrans	*int unsigned*	Indica el número máximo de transacciones por *ID in flight* que puede tener cada puerto maestro.
FallThrough	*bit*	A nivel alto, las decisiones de enrutamiento en el canal *Address Write* pasan al canal *Write Data*. Esto permite que el *crossbar* acepte un pulso del canal *Write Data* en el mismo ciclo que el correspondiente pulso del canal *Address Write*, pero aumenta la ruta combinatoria del canal *Write Data* con la lógica del canal *Address Write*.
LatencyMode	*enum logic [9:0]*	Configuración de la latencia en los canales individuales.
AxiIdWidthSlvPorts	*int unsigned*	Ancho del parámetro *ID* de los puertos esclavos AXI.
AxiIdUsedSlvPorts	*int unsigned*	Número de *bits* de *ID* del puerto esclavo que el *crossbar* utiliza para determinar la singularidad de un *ID* AXI, comenzando por el menos significativo. Este parámetro debe ser menor o igual al parámetro *AxiIdWidthSlvPorts*.
UniqueIds	*bit*	En caso de poder garantizar que el *ID* de cada transacción es siempre único entre todas las transacciones *in flight* en la misma dirección, este parámetro se debe establecer a 1'b1. De este modo, se simplifica el cruce.
AxiAddrWidth	*int unsigned*	Ancho de la dirección AXI.
AxiDataWidth	*int unsigned*	Ancho de los datos AXI.
NoAddrRules	*int unsigned*	Número de reglas contenidas en el *address map*.

4.1.4. *Testbench*

En el repositorio de *GitHub*, el módulo *axi_xbar* incorpora su respectivo *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.

Del análisis del *testbench* original se destacan los puntos más importantes que limitan la reusabilidad de este *testbench* en cualquier otro diseño.

- La generación de estímulos se realiza mediante módulos *ad-hoc*, denominados *axi_rand_master*. Tal y como se muestra en la Figura 23, los estímulos se generan mediante las funciones *add_memory_region*, *reset* y *run* del módulo *axi_rand_master*. Estas funciones son particulares y propias de este diseño. En el diseño propuesto en este Trabajo Fin de Grado, este módulo se sustituye por un IP de verificación actuando en modo maestro.

```
// Masters control simulation run time
for (genvar i = 0; i < NoMasters; i++) begin : gen_rand_master
    static axi_rand_master_t axi_rand_master = new ( master_dv[i] );
  initial begin
    end_of_sim[i] <= 1'b0;
    axi_rand_master.add_memory_region(AddrMap[0].start_addr,
                                      AddrMap[xbar_cfg.NoAddrRules-1].end_addr,
                                      axi_pkg::DEVICE_NONBUFFERABLE);
    axi_rand_master.reset();
    @(posedge rst_n);
    axi_rand_master.run(NoReads, NoWrites);
    end_of_sim[i] <= 1'b1;
  end
end
```

Figura 23. Generación de estímulos en el testbench ad-hoc del módulo crossbar original [21]

- El análisis de las salidas se realiza mediante módulos *ad-hoc*, denominados *axi_rand_slaves*. En la Figura 24 se presenta un fragmento del *testbench ad-hoc*, donde las salidas del módulo se monitorizan mediante las funciones *reset* y *run* del módulo *axi_rand_slave*. Como en el caso anterior, estas funciones son particulares y propias de este diseño. Para mejorar la reusabilidad del código, en este Trabajo Fin de Grado, este módulo se sustituye por un IP de verificación que actúa en modo esclavo. Esto permite, además de realizar la monitorización con fines de depuración, contabilizar todos los parámetros asociados con la medición del

porcentaje de cobertura llevado a cabo. Por las razones expuestas, se podrá tener una medida cuantitativa de cuánto se ha verificado y cuándo se puede dar por finalizada la verificación.

```
for (genvar i = 0; i < NoSlaves; i++) begin : gen_rand_slave
    static axi_rand_slave_t axi_rand_slave = new( slave_dv[i] );
  initial begin
    axi_rand_slave.reset();
    @(posedge rst_n);
    axi_rand_slave.run();
  end
end
```

Figura 24. Función de monitorización en el testbench ad-hoc del módulo crossbar original [21]

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 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 del mismo 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.

4.2. Módulo *wrapper*

El módulo *axi_xbar* 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 diversos motivos que se analizarán más adelante. De ahí surge la necesidad de la creación de un

módulo llamado *wrapper*, el cual se adapte de manera satisfactoria a la herramienta de configuración, encargado de la creación del entorno de verificación UVM. En este Trabajo Fin de Grado se va a utilizar, tal y como se comentó en el capítulo 1, el QVIP *configurator*, de *Mentor Graphics*, *a Siemens Company*.

4.2.1. Funcionalidades

La necesidad de implementar el módulo *wrapper* 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*) en las primeras líneas de código, como se muestra en la Figura 25. 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 *wrapper* que englobe el *axi_xbar* y que cuente con las mismas características estructurales propias del *crossbar*, pero con un lenguaje compatible con la herramienta de *Mentor Graphics*.

```
1   module axi_xbar #(
2     parameter axi_pkg::xbar_cfg_t Cfg = '0,
3     parameter bit  ATOPs              = 1'b1,
4     parameter type slv_aw_chan_t      = logic,
5     parameter type mst_aw_chan_t      = logic,
6     parameter type w_chan_t           = logic,
7     parameter type slv_b_chan_t       = logic,
8     parameter type mst_b_chan_t       = logic,
9     parameter type slv_ar_chan_t      = logic,
10    parameter type mst_ar_chan_t      = logic,
11    parameter type slv_r_chan_t       = logic,
12    parameter type mst_r_chan_t       = logic,
13    parameter type slv_req_t          = logic,
14    parameter type slv_resp_t         = logic,
15    parameter type mst_req_t          = logic,
16    parameter type mst_resp_t         = logic,
17    parameter type rule_t             = axi_pkg::xbar_rule_64_t
18  )
```

Figura 25. Declaración de los párametros del módulo axi_xbar [19]

- Como ya se ha comentado, el módulo *axi_xbar* tiene parámetros que se configuran a través del parámetro *Cfg*, el cual está definido en un módulo diferente llamado

axi_pkg y no interesa 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 del DUV.

- También se adaptarán las entradas y salidas del módulo *axi_xbar*, ya que no posee una interfaz AXI4, aunque la concatenación se realizará en el componente UVM *Top* creado por el QVIP *configurator*.

4.2.2. Código implementado

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 la Figura 26 se muestra la declaración del número de maestros y esclavos y la definición de las entradas y salidas del sistema. En esta implementación se han obviado cuatro señales respecto al diseño *axi_xbar* original: la señal de entrada correspondiente al *address map* porque 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.

```
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 26. Definición del módulo axi_xbar_wrap

Es crucial realizar la configuración de los parámetros específicos del DUV analizados en la Tabla 11, aunque la definición de todos no es necesaria y muchos no se

utilizan en este diseño. Estos parámetros configurables permiten mayor flexibilidad del sistema y se han definido en la Figura 27. La configuración utilizada establece que el campo de dirección de cada transacción tiene 32 *bits* y cuenta con un campo de datos que asciende a 64 *bits*. Cabe destacar que el campo *ID* de los puertos maestros tendrá una longitud de 4 *bits*. En cambio, el campo *ID* de los puertos esclavos dependerá del número de maestros presentes en el sistema y se añadirán tantos *bits* como sean necesarios para la codificación del número de maestros establecidos. El código ha sido diseñado de esta manera para que cuando un maestro envíe con éxito una transacción, el esclavo muestre el número del maestro en cuestión además del *ID* de dicha transacción.

```
13   // Axi4 configuration
14   localparam int unsigned AxiIdWidthMasters =  4;
15   localparam int unsigned AxiIdUsed         =  3;
16   localparam int unsigned AxiIdWidthSlaves  =  AxiIdWidthMasters + $clog2(NoMasters);
17   localparam int unsigned AxiAddrWidth      =  32;
18   localparam int unsigned AxiDataWidth      =  64;
19   localparam int unsigned AxiStrbWidth      =  AxiDataWidth / 8;
20   localparam int unsigned AxiUserWidth      =  5;
```

Figura 27. Parámetros de configuración del módulo wrapper

Todos estos parámetros se deben agrupar en una estructura de configuración que, a su vez, será incorporada como argumento de otra estructura que compone el elemento *axi_xbar*. En la Figura 28 se presenta la configuración de una estructura *xbar_cfg_t*, incluida en el paquete *axi_pkg* y definida en la Figura 22, denominada *xbar_cfg*. Esta estructura de configuración será uno de los parámetros que definirá los elementos para la configuración del DUV.

```
22   // Cfg struct
23   localparam axi_pkg::xbar_cfg_t xbar_cfg = '{
24     NoSlvPorts:         NoMasters,
25     NoMstPorts:         NoSlaves,
26     MaxMstTrans:        10,
27     MaxSlvTrans:        6,
28     FallThrough:        1'b0,
29     LatencyMode:        axi_pkg::CUT_ALL_AX,
30     AxiIdWidthSlvPorts: AxiIdWidthMasters,
31     AxiIdUsedSlvPorts:  AxiIdUsed,
32     AxiAddrWidth:       AxiAddrWidth,
33     AxiDataWidth:       AxiDataWidth,
34     NoAddrRules:        8
35   };
```

Figura 28. Estructura de configuración

En la Figura 29 se muestra la concatenación de las diferentes señales que conforman el protocolo de comunicación AXI4. La concatenación se realiza mediante una serie de macros utilizadas en el *testbench* original, y denominadas *'AXI_TYPEDEF_XX_CHAN_T.* Las macros del tipo *'AXI_TYPEDEF_XX_CHAN_T* están creadas en un paquete llamado *#typedef*, el cual se debe incluir en el módulo *wrapper*. Cada una de estas macros define una estructura por cada canal del protocolo. Respecto a los canales *Address Write*, *Write Data, Address Read* y *Read Data*, estas estructuras se dividen en estructuras de tipo maestro y estructuras de tipo esclavo. Esta asignación se realiza para englobar, por un lado, las señales generadas por los dispositivos maestros y, por otro lado, las señales generadas por los dispositivos esclavos. A las primeras se les denominan señales tipo *req* (*Request*) y a las segundas se les llaman señales tipo *resp* (*Respond*). Por último, las estructuras del tipo *'AXI_TYPEDEF_REQ_T* y *'AXI_TYPEDEF_RESP_T* concatenan las señales según sean del tipo req o *resp* de los canales en sus respectivos puertos esclavos o maestros.

```
37    typedef logic [AxiIdWidthMasters-1:0] id_mst_t;
38    typedef logic [AxiIdWidthSlaves-1:0]  id_slv_t;
39    typedef logic [AxiAddrWidth-1:0]      addr_t;
40    typedef axi_pkg::xbar_rule_32_t       rule_t;
41    typedef logic [AxiDataWidth-1:0]      data_t;
42    typedef logic [AxiStrbWidth-1:0]      strb_t;
43    typedef logic [AxiUserWidth-1:0]      user_t;
44
45    `AXI_TYPEDEF_AW_CHAN_T(aw_chan_mst_t, addr_t, id_mst_t, user_t)
46    `AXI_TYPEDEF_AW_CHAN_T(aw_chan_slv_t, addr_t, id_slv_t, user_t)
47    `AXI_TYPEDEF_W_CHAN_T(w_chan_t, data_t, strb_t, user_t)
48    `AXI_TYPEDEF_B_CHAN_T(b_chan_mst_t, id_mst_t, user_t)
49    `AXI_TYPEDEF_B_CHAN_T(b_chan_slv_t, id_slv_t, user_t)
50
51    `AXI_TYPEDEF_AR_CHAN_T(ar_chan_mst_t, addr_t, id_mst_t, user_t)
52    `AXI_TYPEDEF_AR_CHAN_T(ar_chan_slv_t, addr_t, id_slv_t, user_t)
53    `AXI_TYPEDEF_R_CHAN_T(r_chan_mst_t, data_t, id_mst_t, user_t)
54    `AXI_TYPEDEF_R_CHAN_T(r_chan_slv_t, data_t, id_slv_t, user_t)
55
56    `AXI_TYPEDEF_REQ_T(mst_req_t, aw_chan_mst_t, w_chan_t, ar_chan_mst_t)
57    `AXI_TYPEDEF_RESP_T(mst_resp_t, b_chan_mst_t, r_chan_mst_t)
58    `AXI_TYPEDEF_REQ_T(slv_req_t, aw_chan_slv_t, w_chan_t, ar_chan_slv_t)
59    `AXI_TYPEDEF_RESP_T(slv_resp_t, b_chan_slv_t, r_chan_slv_t)
```

Figura 29. Concatenación de señales del protocolo AXI4

También se deben definir las reglas del *address map* como se muestra en la Figura 30. En este caso en particular, el mapa de direcciones se ha programado con ocho regiones bien diferenciadas. Cada una de estas regiones será asignada a los diferentes módulos QVIP esclavo.

```
61    localparam rule_t [xbar_cfg.NoAddrRules-1:0] AddrMap = '{
62      '{idx: 32'd0, start_addr: 32'h0000_0000, end_addr: 32'h0FFF_FFFF},
63      '{idx: 32'd1, start_addr: 32'h1000_0000, end_addr: 32'h1FFF_FFFF},
64      '{idx: 32'd2, start_addr: 32'h2000_0000, end_addr: 32'h2FFF_FFFF},
65      '{idx: 32'd3, start_addr: 32'h3000_0000, end_addr: 32'h3FFF_FFFF},
66      '{idx: 32'd4, start_addr: 32'h4000_0000, end_addr: 32'h4FFF_FFFF},
67      '{idx: 32'd5, start_addr: 32'h5000_0000, end_addr: 32'h5FFF_FFFF},
68      '{idx: 32'd6, start_addr: 32'h6000_0000, end_addr: 32'h6FFF_FFFF},
69      '{idx: 32'd7, start_addr: 32'h7000_0000, end_addr: 32'h7FFF_FFFF}
70    };
```

Figura 30. Definición del address map

A partir de las señales concatenadas creadas en la Figura 29 (*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.

En la Figura 31 se muestra el código que declara el DUV con diferentes parámetros definidos anteriormente como, por ejemplo, la estructura de configuración. Además, se definen los puertos de entrada y salida del módulo *crossbar* como las señales de reloj y *reset*, el mapa de direcciones, entre otros. Cabe destacar el código de las líneas 101 a 104 de la mencionada figura, ya que también se pasan los *arrays masters_req*, *masters_resp*, *slaves_req* y *slaves_resp* a los puertos maestros y esclavos del sistema: *slv_ports_req_i, slv_ports_resp_o, mst_ports_req_o* y *mst_ports_resp_i*, respectivamente.

```
72   // master structs
73   mst_req_t  [NoMasters-1:0] masters_req;
74   mst_resp_t [NoMasters-1:0] masters_resp;
75
76   // slave structs
77   slv_req_t  [NoSlaves-1:0] slaves_req;
78   slv_resp_t [NoSlaves-1:0] slaves_resp;
79
80   // DUV
81   axi_xbar #(
82     .Cfg           ( xbar_cfg ),
83     .slv_aw_chan_t (  aw_chan_mst_t ),
84     .mst_aw_chan_t (  aw_chan_slv_t ),
85     .w_chan_t      (  w_chan_t      ),
86     .slv_b_chan_t  (  b_chan_mst_t  ),
87     .mst_b_chan_t  (  b_chan_slv_t  ),
88     .slv_ar_chan_t (  ar_chan_mst_t ),
89     .mst_ar_chan_t (  ar_chan_slv_t ),
90     .slv_r_chan_t  (  r_chan_mst_t  ),
91     .mst_r_chan_t  (  r_chan_slv_t  ),
92     .slv_req_t     (  mst_req_t     ),
93     .slv_resp_t    (  mst_resp_t    ),
94     .mst_req_t     (  slv_req_t     ),
95     .mst_resp_t    (  slv_resp_t    ),
96     .rule_t        (  rule_t        )
97   ) i_xbar_dut (
98     .clk_i      ( clk_i      ),
99     .rst_ni     ( rst_ni     ),
100    .test_i     ( 1'b0       ),
101    .slv_ports_req_i  ( masters_req  ),
102    .slv_ports_resp_o ( masters_resp ),
103    .mst_ports_req_o  ( slaves_req   ),
104    .mst_ports_resp_i ( slaves_resp  ),
105    .addr_map_i       ( AddrMap      ),
106    .en_default_mst_port_i ( '0       ),
107    .default_mst_port_i    ( '0       )
108  );
```

Figura 31. Referencia del DUV con su interfaz de entrada/salida y los parámetros de configuración

Por último, lo único que queda es asignar cada estructura a su respectivo puerto de entrada o salida como se observa en la Figura 32, teniendo en cuenta que las señales de los maestros están asociadas a los puertos esclavos y las señales de los esclavos se relacionan con los puertos maestros del DUV.

```
110   for(genvar i = 0; i < NoMasters; i++) begin: gen_connect_NoMasters
111       assign masters_req[i]      =    slv_ports_req_i[i];
112       assign slv_ports_resp_o[i] =    masters_resp[i];
113   end
114
115   for(genvar i = 0; i < NoSlaves; i++) begin: gen_connect_NoSlaves
116       assign mst_ports_req_o[i]  =    slaves_req[i];
117       assign slaves_resp[i]      =    mst_ports_resp_i[i];
118   end
```

Figura 32. Asignación de las estructuras a su respectivo puerto de entrada o salida

Así pues, para una mayor comprensión, se adjunta el esquema de la Figura 33, 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. En la Figura 33 se muestran los diferentes nombres que definen los puertos según cada elemento, aunque realmente están definidos de igual manera.

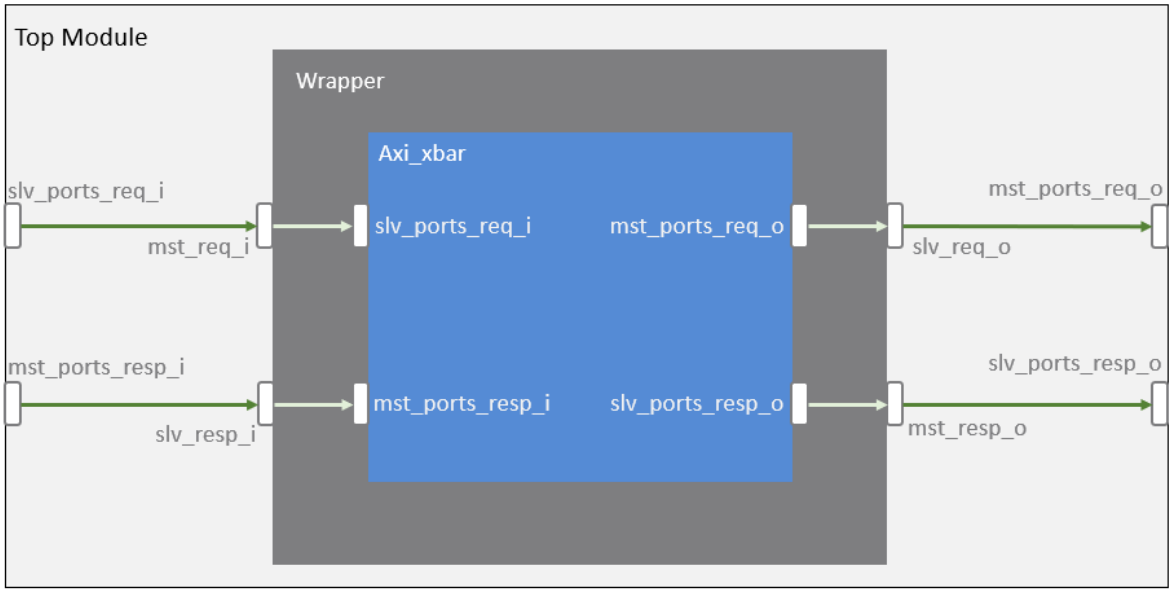

Figura 33. Esquema parcial de los módulos crossbar y wrapper conectados al componente UVM Top Module

Capítulo 5. Desarrollo de un entorno de verificación UVM mediante la herramienta QVIP *configurator*

El principal objetivo de este Trabajo Fin de Grado es el desarrollo completo de un *testbench* UVM para verificar el comportamiento de un DUV proporcionado por un tercero. Inicialmente, este objetivo no podría ser realizable 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 es la utilización de una herramienta que permita reducir el tiempo empleado en la generación de los componentes UVM del *testbench*. Una de las herramientas que ofrece esta posibilidad pertenece a la empresa *Mentor Graphics*, la cual crea y configura de manera dinámica el entorno de verificación UVM. Por lo tanto, esta herramienta llamada QVIP *configurator* cobra especial relevancia.

5.1. 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 [22].

5.1.1. Principios básicos de la herramienta QVIP *configurator*

La herramienta 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.

La herramienta *Questa Verification* IP se encuentra disponible para una amplia gama de protocolos de comunicación, como se visualiza en la Figura 34, 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.

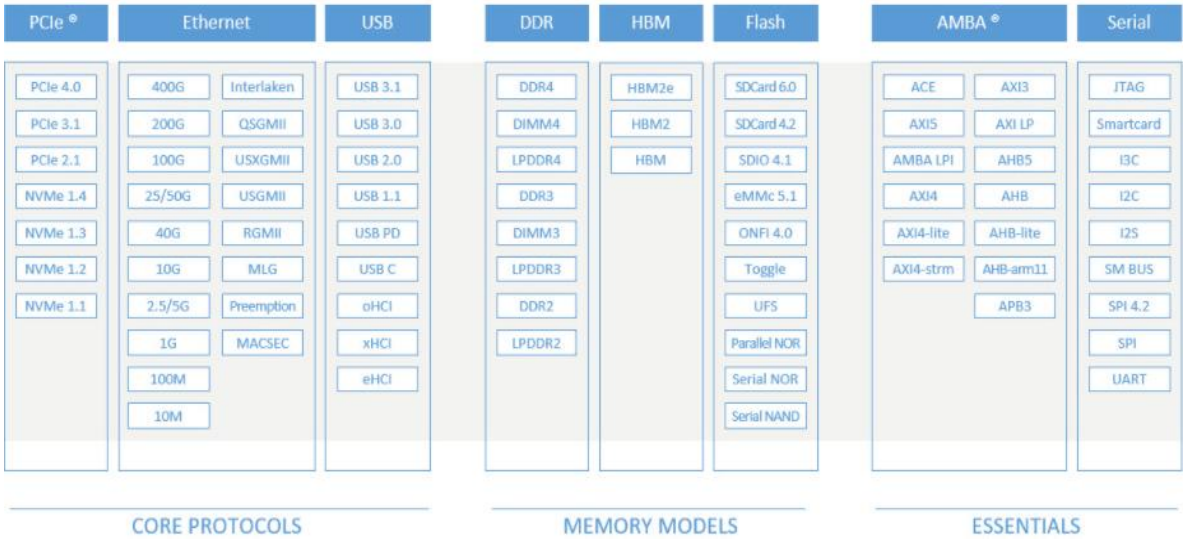

Figura 34. Protocolos de comunicación compatibles con el Questa Verification IP [22]

Con el fin de recapitular los conceptos básicos y explicar la creciente adopción de la herramienta *configurator* QVIP, se aclaran 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 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, QVIP cuenta con una gran ventaja, ya que, si el proceso de verificación del DUV ha sido realizado antes

del cambio del protocolo de comunicación, no afecta al *testbench* UVM diseñado. La herramienta QVIP, junto a la metodología UVM, permite la reusabilidad del código, en especial de las secuencias diseñadas 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.

5.1.2. Diagrama de flujo para el uso de QVIP *configurator*

Mentor Graphics 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 [23]. 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, el tercer paso no se realizará en la herramienta de configuración, pues la generación de estímulos controlados se elaborará manualmente. En la Figura 35 se presentan, a grandes rasgos, las cuatro etapas que se deben completar durante el flujo de integración QVIP. Para mayor detalle, 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 monitorizar las salidas del mismo. 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 favorece 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 *configurator* QVIP 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 entorno de verificación. 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.

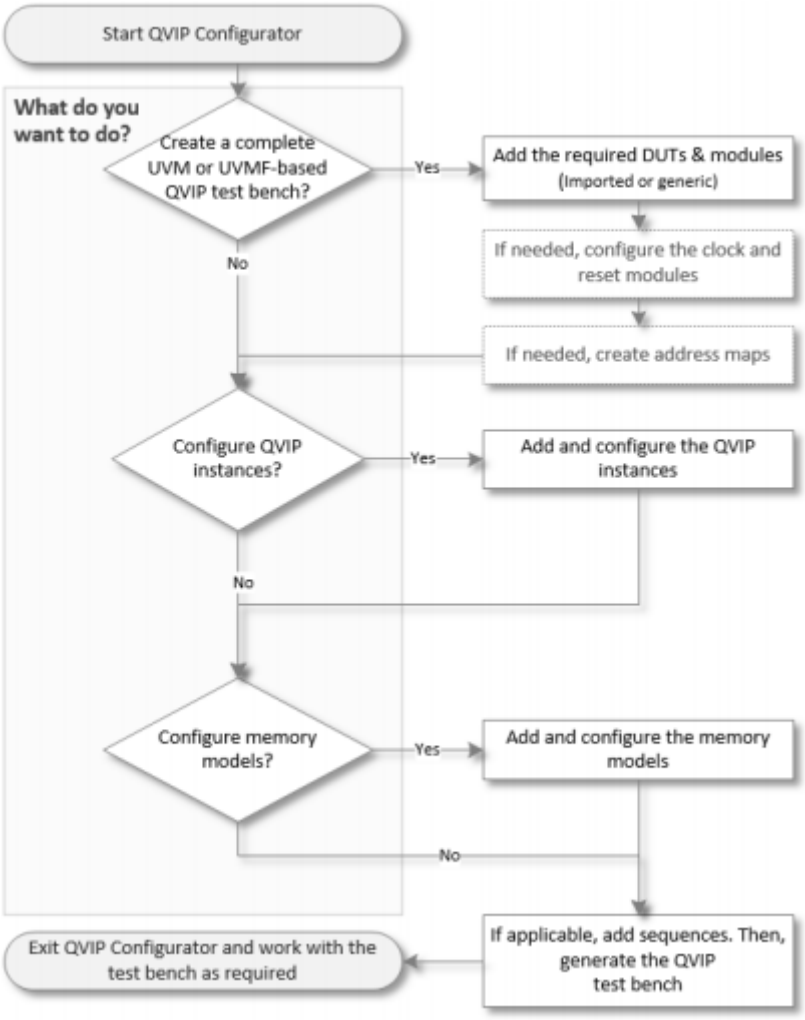

Figura 35. Flujo de trabajo para la creación de un testbench con la herramienta de configuración QVIP [24]

5.1.3. Guía de usuario de la herramienta QVIP *configurator*

Como se explicó en el apartado anterior, la herramienta de configuración [24] juega un importante papel durante la realización del presente Trabajo Fin de Grado, debido a que permite la configuración e integración de los módulos de verificación IP en el entorno de verificación UVM. A lo largo de este apartado se explicará con mayor detalle cada una de las etapas que se deben seguir al utilizar dicha herramienta, y se destacarán aquellos puntos de mayor relevancia para el desarrollo de este TFG.

5.1.3.1. Integración de los módulos de verificación IP

El *software* proporcionado por *Mentor Graphics* permite configurar diferentes módulos de verificación IP y conectarlos al dispositivo que se desea verificar, tanto de manera gráfica como manualmente. Para ello, como se muestra en la Figura 36, la interfaz

del QVIP *configurator* es clara y sencilla, se aleja de cualquier añadido que no sea necesario.

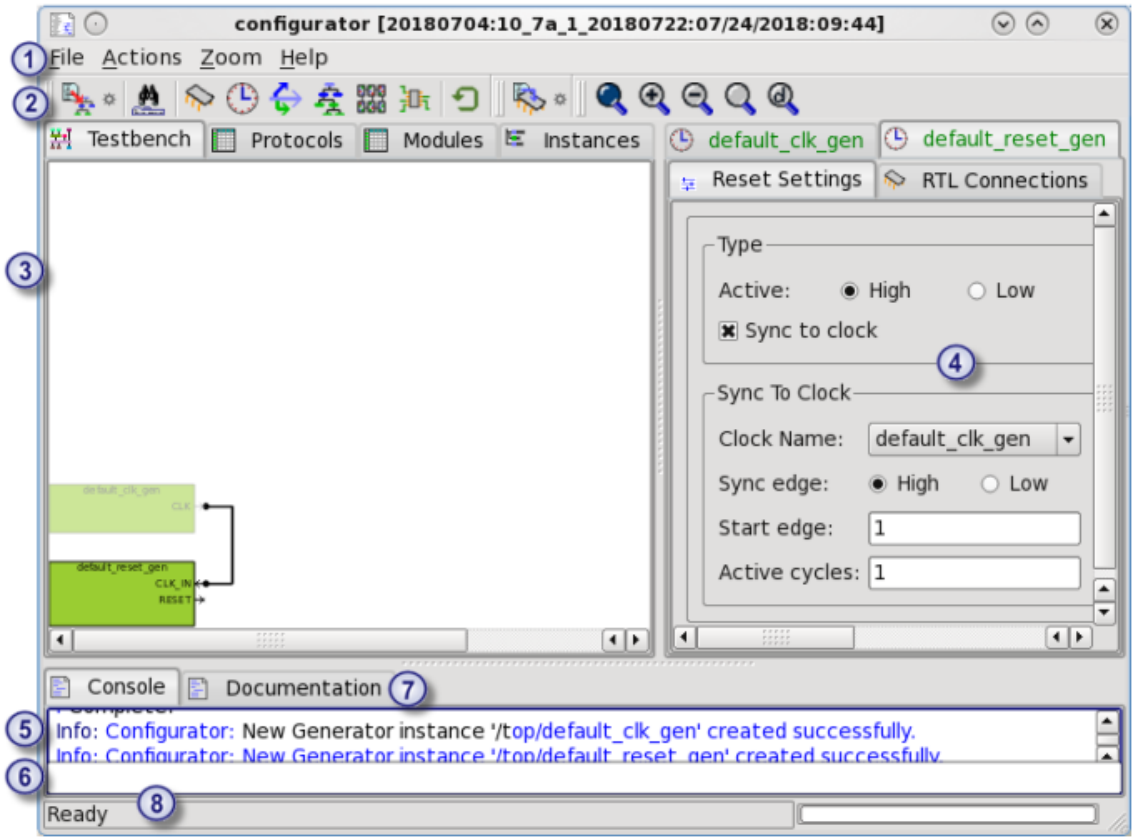

Figura 36. Interfaz de la herramienta configurator QVIP [24]

A continuación, se van a enumerar las diferentes características y funcionalidades de las áreas destacadas de la interfaz del QVIP *configurator* en la Figura 36:

1. Barra de menú. Este menú cuenta con pestañas que permiten salvar e importar un proyecto, acciones como la creación de módulos QVIP o módulos DUV, opciones de *zoom* y también proporciona una pestaña de ayuda donde se pueden consultar todos los comandos disponibles en la herramienta.
2. Barra de herramientas. Esta barra proporciona un acceso rápido a las principales funciones del *testbench* QVIP y las opciones de *zoom*.
3. Panel principal. Cuenta con varias pestañas cuyo contenido se detalla a continuación:
    - *Testbench*: muestra una representación gráfica de todos los módulos y referencias creadas: módulos de reloj y *reset*, módulos QVIP, modelos de memorias y el DUV importado al entorno. Además, también se representan las conexiones que unen todos estos dispositivos.

- *Protocols*: enumera todos los protocolos de comunicación disponibles en la herramienta, con los cuales se puede configurar la interfaz de los módulos de verificación QVIP integrados.
- *Modules*: nombra todos los módulos importados que se pueden utilizar en el *testbench*.
- *Instancies*: enumera las referencias creadas y módulos del *testbench* de manera jerárquica.

4. Panel de configuración. Muestra, en diferentes pestañas, las opciones de configuración del módulo seleccionado en el *testbench* QVIP.
5. *Console*. Presenta en pantalla el registro de los cambios efectuados en la topología del entorno. Además, aparecerán todos aquellos errores, advertencias y mensajes de información generados a medida que se crea el *testbench*. La herramienta de configuración almacena este registro en el archivo *configurator.log*, el cual se ejecutará comando a comando cuando se vuelva a abrir el *software*.
6. Panel de comando. Permite la ejecución de los comandos disponibles en el QVIP *configurator*, como puede ser la modificación de un módulo VIP, la definición o eliminación de un *address map*, la creación de una señal de reloj, entre otros muchos procesos que se pueden ejecutar en el panel de comando.
7. *Documentation*. Muestra la descripción de la configuración de los elementos creados.
8. Barra de estados. Indica el estado del *testbench* que puede ser *ready* o *busy*.

En la Figura 37 se presenta el módulo DUV utilizado en este TFG una vez importado en la herramienta *configurator*. Tal y como se explicó en el capítulo 4, el módulo *crossbar* original está diseñado para ser totalmente compatible con el estándar AMBA AXI4 pero, por cuestiones de eficiencia, no posee una interfaz AXI4 definida de forma explícita. Cuenta con cuatro puertos de entrada y dos de salida, en cuyas características se profundizó durante el capítulo 4 de la presente memoria.

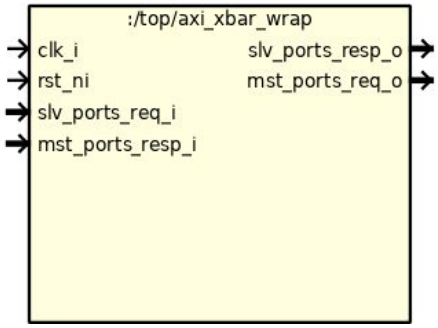

Figura 37. Módulo wrapper importado al configurator QVIP

Los módulos de verificación IP son necesarios para la estimulación controlada del dispositivo a verificar. Por lo tanto, los módulos QVIP se van a comportar, en un entorno UVM, como un componente UVM *Agent* y, como tales, deben contar con una interfaz que permita la comunicación del módulo QVIP con el DUV. Para ello, se deben configurar todos los módulos QVIP, tanto esclavos como maestros, con el protocolo de comunicación AXI4 como se presenta en la Figura 38.

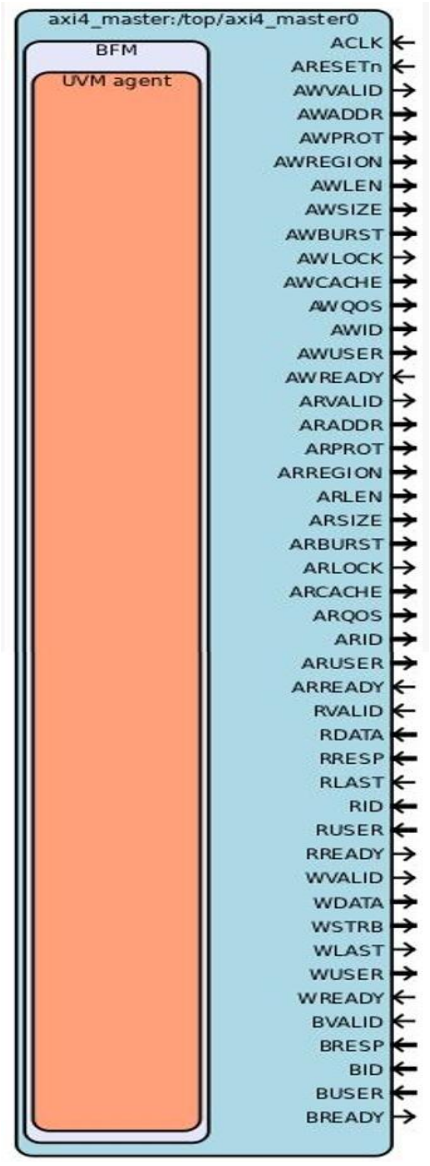

Figura 38. Módulo QVIP maestro con interfaz AXI4

5.1.3.2. Conexionado de los módulos IP con el DUV

El proceso de conexión de todos los módulos que van a componer el entorno de verificación y el DUV es un factor clave para poder realizar posteriormente la simulación

del sistema diseñado. Este proceso se puede llevar cabo de diversas maneras, las cuales se analizarán en los siguientes puntos:

- La manera más cómoda de hacer el conexionado de los puertos de los componentes del *testbench* es de forma automática. Para ello, ambos módulos deben poseer exactamente la misma interfaz. En el caso particular de este TFG, las conexiones no se pueden elaborar automáticamente porque el módulo *crossbar* carece de interfaz AXI4 explícito, como se ha comentado en anteriores ocasiones.
- Otra forma de realizar las pertinentes conexiones es desde la interfaz gráfica de forma manual, seleccionando un puerto y arrastrando el cable que al instante genera el QVIP *configurator*. Después, se guía el cable al destino que se desea y se conecta automáticamente. De esta manera, la herramienta de configuración ejecuta el comando que crea la conexión entre los dos puertos seleccionados. En caso de una conexión errónea, se puede eliminar un cable seleccionado.
- En el panel de la derecha, etiquetado con el número 4 en la Figura 36, el *software* permite gestionar las conexiones de un módulo. Para gestionarlo, basta con seleccionar el módulo y acceder a la pestaña *wires*, tal y como se muestra en la Figura 39.

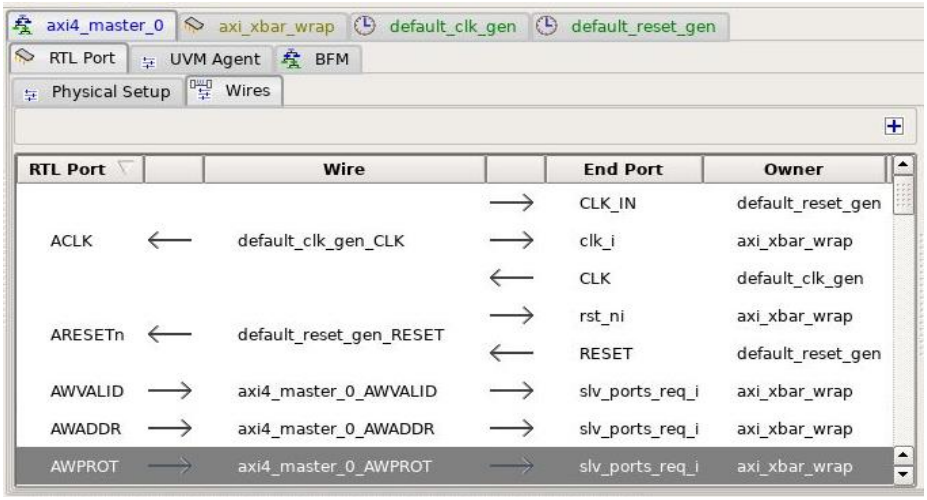

Figura 39. Gestión de las conexiones de los módulos QVIP

- La cuarta manera de efectuar el conexionado es mediante una configuración manual en el código generado. Por cuestiones de eficiencia, en este Trabajo Fin de Grado, se decidió seguir esta filosofía de trabajo, ya que hacer el

conexionado de seis QVIP maestros y ocho QVIP esclavos de manera gráfica ralentiza el proceso de creación del entorno de verificación UVM.

En caso de haber realizado el conexionado siguiendo el enfoque planteado en los tres primeros puntos, el sistema genera una representación gráfica, tal y como se muestra en el ejemplo presentado de la Figura 40. Si el conexionado se realiza de manera manual, es decir, modificando el código generado por el *configurator*, esta representación no se visualizará. Al conectar las señales, se aconseja seguir un orden lógico durante el proceso, por ejemplo, realizando la conexión de las señales según el canal del protocolo de comunicación AXI4 al que pertenezcan.

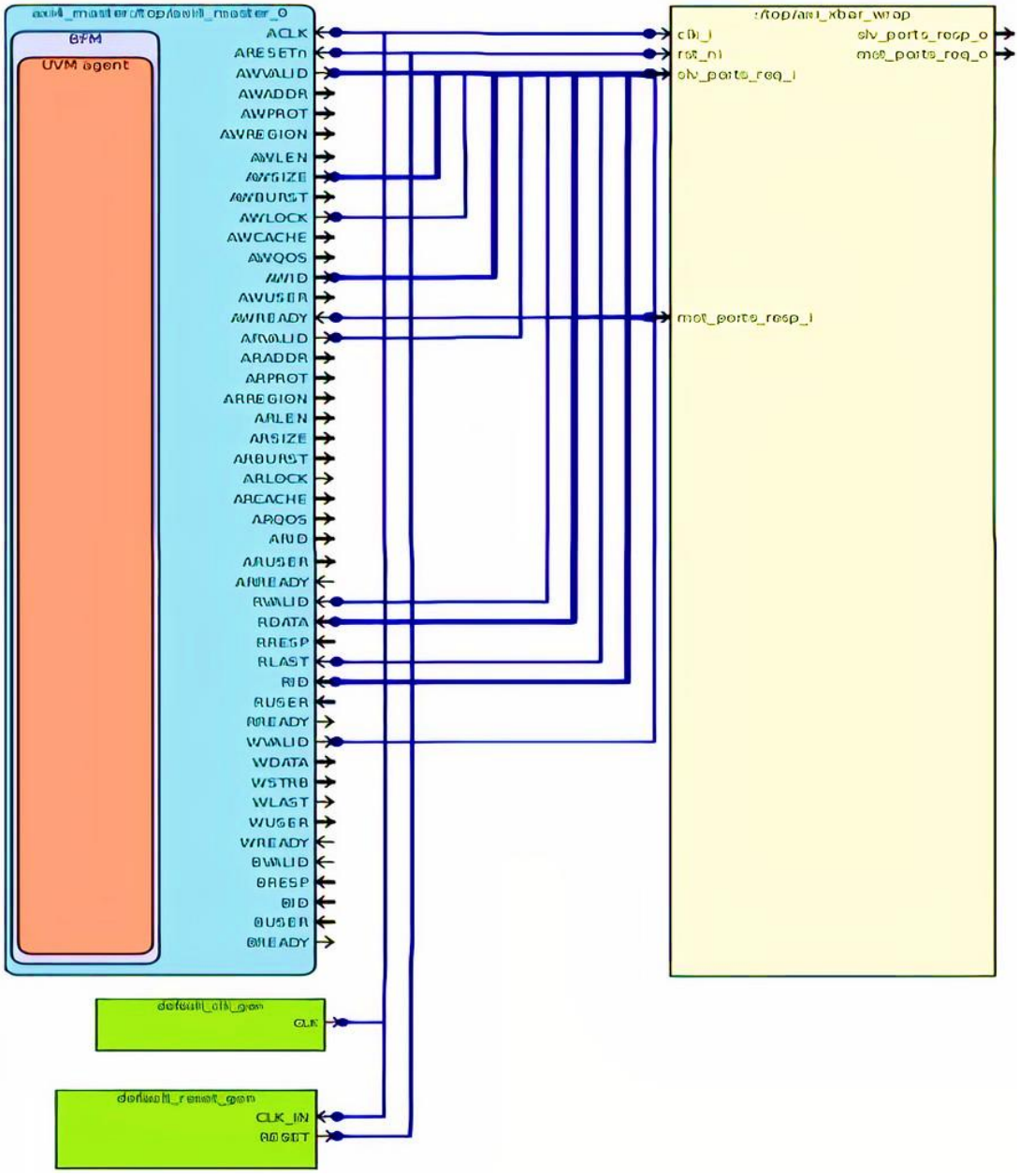

Figura 40. Conexionado parcial del módulo wrapper con un módulo QVIP maestro

5.1.3.3. Definición del mapa de direcciones

La herramienta de configuración permite la definición de mapas de direcciones (*address map*) de una manera sencilla. Estos mapas se pueden incorporar al proyecto si se asignan a algún componente que forme parte del sistema diseñado. Por lo general, una memoria tiene regiones contiguas y no contiguas que se dividen en función de los rangos de direcciones definidos. De esta manera, los mapas de direcciones relacionan las diferentes regiones.

En la Figura 41 se observa la ventana de configuración del *Address Map* propia del QVIP *configurator*. En esta ventana se pueden definir los parámetros de cada región del mapa de direcciones como la máscara, el tipo de memoria y el tipo de protección, parámetros que vienen definidos por el protocolo AXI4. Además, se debe indicar la dirección de comienzo de cada una de las regiones y el tamaño que tendrá dicha región de la memoria. Durante el desarrollo de este Trabajo Fin de Grado, se definió un *address map* con direcciones de 32 *bits*, ya que en el diseño *crossbar* original las direcciones se codifican con ese mismo número de *bits*. El QVIP *configurator* tiene asignado por defecto un mapa de direcciones de 64 *bits* y esta longitud no se puede cambiar en la propia herramienta. Por lo tanto, se ha tenido que modificar posteriormente de manera manual.

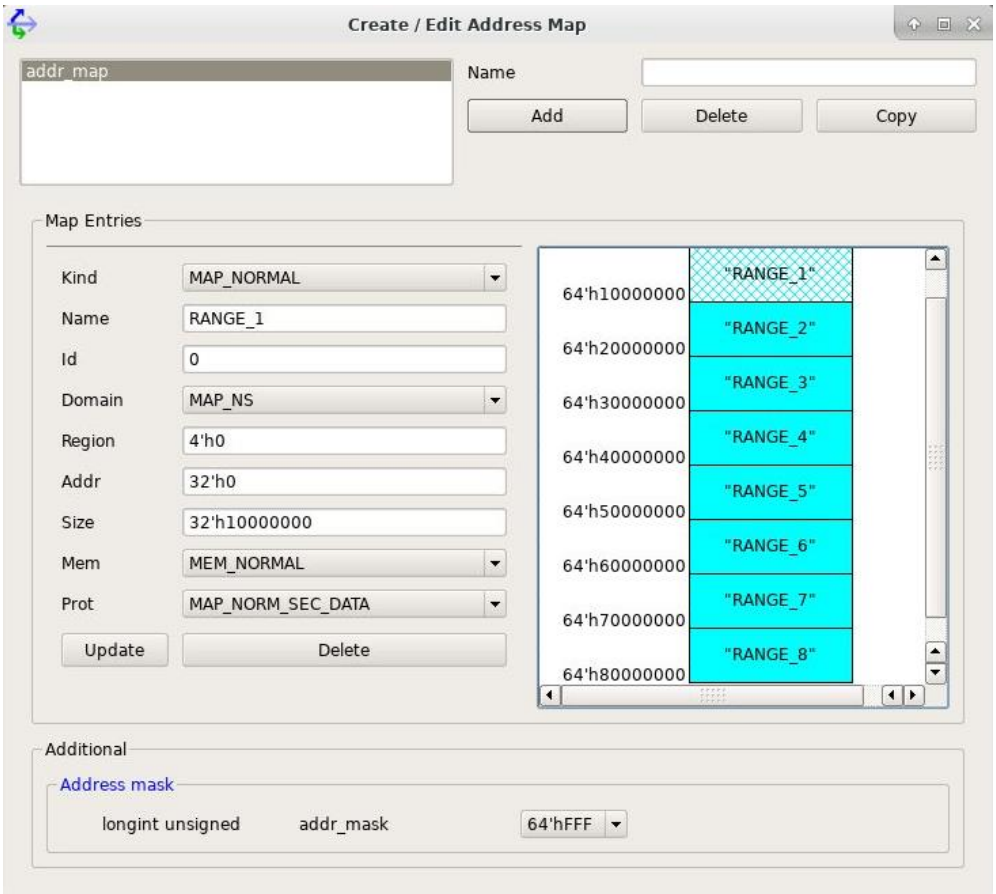

Figura 41.Ventana de configuración del Address Map

Los rangos de memoria no se deben superponer, ya que sólo se tendrá en cuenta la región que primero se haya compilado, es decir, una dirección sólo puede aparecer en una única región de la memoria. Otra característica a tener en cuenta a la hora de realizar el diseño del mapa de direcciones es su nombre, el sistema tiene reservado el nombre "*address_map"*, por lo que no se podrá definir ninguna memoria con ese nombre.

5.2. Entorno de verificación UVM

Una vez finalizada la descripción de todos los componentes que van a formar el *testbench*, se procede a generar, de manera automática, el código del *testbench* UVM desde la herramienta de configuración QVIP. En este apartado se realizará un análisis exhaustivo del código generado. El QVIP *configurator* aporta agilidad al proceso de verificación, ya que a partir de un DUV se crea todo el entorno que engloba los diferentes componentes UVM estudiados en el capítulo 3, elaborando todo el conexionado que se requiere para su funcionamiento.

En el caso particular del presente Trabajo Fin de Grado se ha utilizado la herramienta de configuración 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. Por esta razón, se ha optado por el desarrollo manual de la definición de las señales y de su posterior concatenación en los respectivos puertos de entrada y salida del DUV. El proyecto generado por el QVIP *configurator* se ha llamado *qvip_axi_xbar*.

5.2.1. 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 42 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.

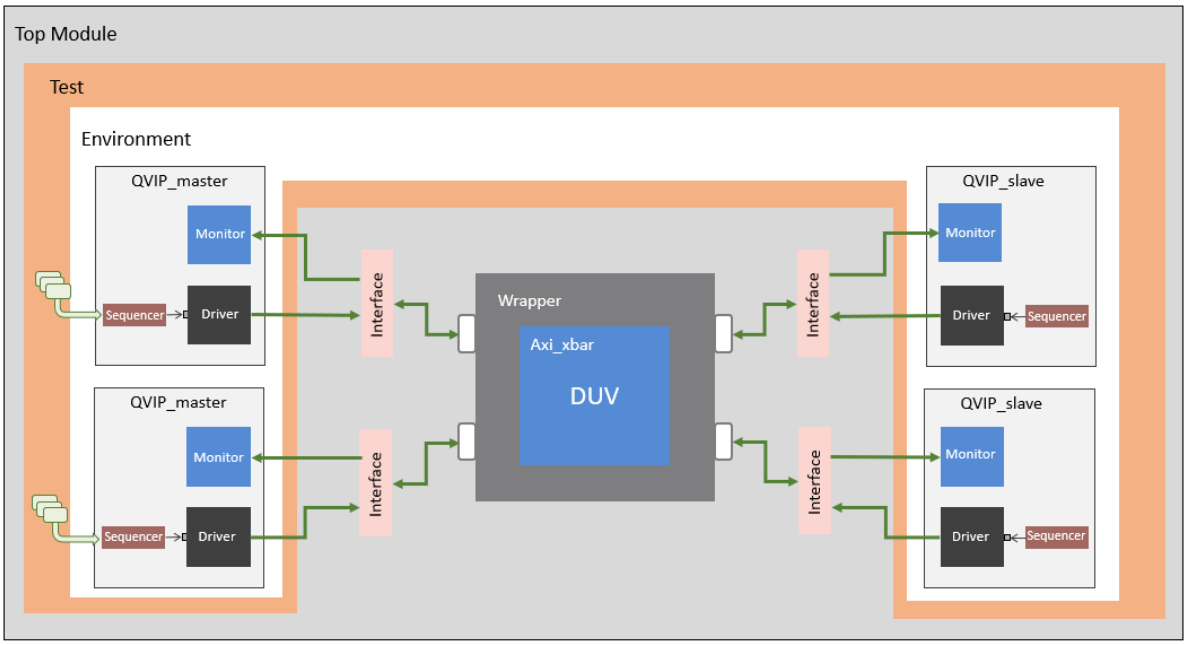

Figura 42. Esquema parcial del entorno de verificación UVM y el módulo DUV

En lo que resta de capítulo, se analizarán las diferentes características de cada uno de los componentes del entorno UVM creado a través de la herramienta de configuración QVIP, profundizando en los aspectos más relevantes de los mismos.

5.2.2. Componente UVM *Testbench*

El módulo *Testbench,* también llamado módulo *Top Module*, es el único módulo común en cualquier *testbench* de un sistema definido en lenguaje HDL. Este módulo, por tanto, es un módulo estático y no es exclusivo de una metodología en particular. Este módulo constituye el nivel más alto de la jerarquía del entorno y se comporta como un contenedor estático que contiene únicamente aquellos módulos (no componentes) que se crean en tiempo de compilación y no en tiempo de ejecución. En concreto, este módulo incluye, en primer lugar, el DUV, el módulo de generación de las señales de reloj y reset y, por último, las interfaces de entrada y salida definidas en el *testbench*. La principal diferencia de este módulo, cuando se utiliza la metodología UVM, es que incluye un proceso que invoca al método `run_test`. Este método es el responsable de activar el mecanismo de fases de UVM, mediante el cual se construye (`build_phase`), se conecta (`connect_phase`) y se ejecuta el *test* (`run_phase`) especificado durante la simulación.

5.2.2.1. Referencia de los módulos maestros y esclavos

En primer lugar, se van a definir todas las señales propias del protocolo AXI4, que van asociadas a cada uno de los maestros y esclavos que forman parte del sistema, para asegurar una comunicación exitosa entre los módulos QVIP y el dispositivo a verificar. En la Figura 43 se muestra la declaración de las señales asociadas al primer maestro, es decir el maestro 0. Estas señales se deben declarar con el correspondiente número de *bits* que marca el protocolo AXI4 para cada campo. Muchos de estos parámetros vienen definidos a partir de parámetros que se encuentran dentro del paquete *axi4_masterX_params* por lo que se debe acceder a dicho paquete para la correcta declaración de todas las señales. Estos parámetros fueron definidos en la herramienta de configuración QVIP. Aunque en la Figura 43 sólo se presentan las señales del maestro 0, este proceso se realiza para todos los módulos QVIP maestros y esclavos presentes en el diseño.

```
1  module hdl_qvip_axi_xbar;
2    wire                                                default_clk_gen_CLK;
3    wire                                                default_reset_gen_RESET;
4    wire                                                axi4_master0_AWVALID;
5    wire [axi4_master0_params::AXI4_ADDRESS_WIDTH-1:0]  axi4_master0_AWADDR;
6    wire [2:0]                                          axi4_master0_AWPROT;
7    wire [3:0]                                          axi4_master0_AWREGION;
8    wire [7:0]                                          axi4_master0_AWLEN;
9    wire [2:0]                                          axi4_master0_AWSIZE;
10   wire [1:0]                                          axi4_master0_AWBURST;
11   wire                                                axi4_master0_AWLOCK;
12   wire [3:0]                                          axi4_master0_AWCACHE;
13   wire [3:0]                                          axi4_master0_AWQOS;
14   wire [axi4_master0_params::AXI4_ID_WIDTH-1:0]       axi4_master0_AWID;
15   wire [axi4_master0_params::AXI4_USER_WIDTH-1:0]     axi4_master0_AWUSER;
16   wire                                                axi4_master0_AWREADY;
17   wire                                                axi4_master0_ARVALID;
18   wire [axi4_master0_params::AXI4_ADDRESS_WIDTH-1:0]  axi4_master0_ARADDR;
19   wire [2:0]                                          axi4_master0_ARPROT;
20   wire [3:0]                                          axi4_master0_ARREGION;
21   wire [7:0]                                          axi4_master0_ARLEN;
22   wire [2:0]                                          axi4_master0_ARSIZE;
23   wire [1:0]                                          axi4_master0_ARBURST;
24   wire                                                axi4_master0_ARLOCK;
25   wire [3:0]                                          axi4_master0_ARCACHE;
26   wire [3:0]                                          axi4_master0_ARQOS;
27   wire [axi4_master0_params::AXI4_ID_WIDTH-1:0]       axi4_master0_ARID;
28   wire [axi4_master0_params::AXI4_USER_WIDTH-1:0]     axi4_master0_ARUSER;
29   wire                                                axi4_master0_ARREADY;
30   wire                                                axi4_master0_RVALID;
31   wire [axi4_master0_params::AXI4_RDATA_WIDTH-1:0]    axi4_master0_RDATA;
32   wire [1:0]                                          axi4_master0_RRESP;
33   wire                                                axi4_master0_RLAST;
34   wire [axi4_master0_params::AXI4_ID_WIDTH-1:0]       axi4_master0_RID;
35   wire [axi4_master0_params::AXI4_USER_WIDTH-1:0]     axi4_master0_RUSER;
36   wire                                                axi4_master0_RREADY;
37   wire                                                axi4_master0_WVALID;
38   wire [axi4_master0_params::AXI4_WDATA_WIDTH-1:0]    axi4_master0_WDATA;
39   wire [(axi4_master0_params::AXI4_WDATA_WIDTH/8)-1:0] axi4_master0_WSTRB;
40   wire                                                axi4_master0_WLAST;
41   wire [axi4_master0_params::AXI4_USER_WIDTH-1:0]     axi4_master0_WUSER;
42   wire                                                axi4_master0_WREADY;
43   wire                                                axi4_master0_BVALID;
44   wire [1:0]                                          axi4_master0_BRESP;
45   wire [axi4_master0_params::AXI4_ID_WIDTH-1:0]       axi4_master0_BID;
46   wire [axi4_master0_params::AXI4_USER_WIDTH-1:0]     axi4_master0_BUSER;
47   wire                                                axi4_master0_BREADY;
```

Figura 43. Declaración de las señales del protocolo AXI4 para el QVIP maestro 0

En segundo lugar, se realiza la configuración básica de todos los componentes QVIP. Como ejemplo, en la Figura 44 se puede observar la configuración del maestro 0, donde se asignan una serie de parámetros que marcarán el ancho de la dirección, de los datos de lectura y escritura, el identificador y el parámetro *USER*, esto determinará el ancho de los cinco canales del protocolo AXI4. Los dos últimos parámetros *IF_NAME* y *PATH_NAME* corresponden al nombre de la interfaz asociada al módulo QVIP y la ruta de búsqueda para dicha interfaz, respectivamente. El nombre de la interfaz virtual (en el ejemplo, “axi4_master0”) será registrado en el *Factory* para su uso en el entorno UVM generado. Esta interfaz será recuperada por un componente UVM *Agent* para generar las señales y datos propios del mismo. Para realizar la búsqueda de esta interfaz, el componente UVM *Agent* utiliza el parámetro PATH_NAME. Cada uno de los módulos QVIP tendrá una interfaz virtual asignada.

```
648   axi4_master
649     #(.ADDR_WIDTH(axi4_master0_params::AXI4_ADDRESS_WIDTH),
650       .RDATA_WIDTH(axi4_master0_params::AXI4_RDATA_WIDTH),
651       .WDATA_WIDTH(axi4_master0_params::AXI4_WDATA_WIDTH),
652       .ID_WIDTH(axi4_master0_params::AXI4_ID_WIDTH),
653       .USER_WIDTH(axi4_master0_params::AXI4_USER_WIDTH),
654       .REGION_MAP_SIZE(axi4_master0_params::AXI4_REGION_MAP_SIZE),
655       .IF_NAME("axi4_master0"),
656       .PATH_NAME("uvm_test_top"))
```

Figura 44. Definición de los parámetros básicos de un módulo QVIP maestro

Por último, se deben conectar todas las señales del protocolo AXI4 con la interfaz virtual del maestro 0 recién asignada. Por lo tanto, se asociará cada señal con el respectivo puerto de la interfaz AXI4, incluidas las señal de reloj y la señal *reset* negada. En la Figura 45 se adjunta el código que implementa esta conexión, necesaria para el correcto funcionamiento del protocolo de comunicación entre el entorno de verificación UVM y el módulo DUV.

```
658 axi4_master0
659   (.ACLK(default_clk_gen_CLK),
660    .ARESETn(default_reset_gen_RESET),
661    .AWVALID(axi4_master0_AWVALID),
662    .AWADDR(axi4_master0_AWADDR),
663    .AWPROT(axi4_master0_AWPROT),
664    .AWREGION(axi4_master0_AWREGION),
665    .AWLEN(axi4_master0_AWLEN),
666    .AWSIZE(axi4_master0_AWSIZE),
667    .AWBURST(axi4_master0_AWBURST),
668    .AWLOCK(axi4_master0_AWLOCK),
669    .AWCACHE(axi4_master0_AWCACHE),
670    .AWQOS(axi4_master0_AWQOS),
671    .AWID(axi4_master0_AWID),
672    .AWUSER(axi4_master0_AWUSER),
673    .AWREADY(axi4_master0_AWREADY),
674    .ARVALID(axi4_master0_ARVALID),
675    .ARADDR(axi4_master0_ARADDR),
676    .ARPROT(axi4_master0_ARPROT),
677    .ARREGION(axi4_master0_ARREGION),
678    .ARLEN(axi4_master0_ARLEN),
679    .ARSIZE(axi4_master0_ARSIZE),
680    .ARBURST(axi4_master0_ARBURST),
681    .ARLOCK(axi4_master0_ARLOCK),
682    .ARCACHE(axi4_master0_ARCACHE),
683    .ARQOS(axi4_master0_ARQOS),
684    .ARID(axi4_master0_ARID),
685    .ARUSER(axi4_master0_ARUSER),
686    .ARREADY(axi4_master0_ARREADY),
687    .RVALID(axi4_master0_RVALID),
688    .RDATA(axi4_master0_RDATA),
689    .RRESP(axi4_master0_RRESP),
690    .RLAST(axi4_master0_RLAST),
691    .RID(axi4_master0_RID),
692    .RUSER(axi4_master0_RUSER),
693    .RREADY(axi4_master0_RREADY),
694    .WVALID(axi4_master0_WVALID),
695    .WDATA(axi4_master0_WDATA),
696    .WSTRB(axi4_master0_WSTRB),
697    .WLAST(axi4_master0_WLAST),
698    .WUSER(axi4_master0_WUSER),
699    .WREADY(axi4_master0_WREADY),
700    .BVALID(axi4_master0_BVALID),
701    .BRESP(axi4_master0_BRESP),
702    .BID(axi4_master0_BID),
703    .BUSER(axi4_master0_BUSER),
704    .BREADY(axi4_master0_BREADY));
```

Figura 45. Conexión de las señales con la interfaz AXI4 de un QVIP maestro

5.2.2.2. Referencia del módulo *wrapper*

El módulo *wrapper* también se debe referenciar en el sistema. Para ello, se definen señal a señal cada uno de los puertos, tanto de entrada como de salida, del DUV. Esto cobra especial relevancia ya que se deben ajustar las señales AXI4 de la interfaz de los módulos QVIP con los puertos del DUV. Por este motivo es necesario realizar la concatenación de todas las señales, ya que el diseño que se desea verificar, si bien es compatible con el protocolo AXI4, carece de una interfaz AXI4 explícitamente definida. Para realizar la concatenación se ha tenido en cuenta el orden específico de las señales marcadas en el diseño original del *crossbar*. Esto se debe a que el DUV dividirá, de manera interna, todas las señales en diferentes campos según los *bits* que correspondan con cada una de estas señales. Si se cambiase este orden, no funcionaría el protocolo de comunicación y no se podría verificar el funcionamiento del *crossbar*.

5.2.2.2.1. Puertos de la señal de reloj y la señal *reset*

A los puertos de entrada de la señal de reloj y la señal de *reset* sólo se les asocia las correspondientes señales que se definieron en la Figura 43, tal y como se muestra en la Figura 46. De esta manera, con la señal de reloj se podrán sincronizar todos los componentes del entorno y sus señales, y con la señal *reset* activa a nivel bajo se contará con la función *reset* del diseño.

```
588 ∨ axi_xbar_wrap (
589       .clk_i(default_clk_gen_CLK),
590       .rst_ni(default_reset_gen_RESET),
```

Figura 46. Definición de los puertos de la señal de reloj y la señal reset

5.2.2.2.2. Puerto esclavo de entrada

El diseño original del módulo *crossbar* contiene un puerto denominado *slv_ports_req_i*. Este puerto agrupa todas las señales *request* de todos los módulos maestros del *testbench*. Por un lado, con el fin de visualizar con mayor claridad estos campos de datos, en la Figura 47 se recopilan todos los campos que componen la estructura concatenada de 229 *bits* asociada al puerto esclavo *slv_ports_req_i*. Por otro lado, como se observa en la Figura 48, los campos se agrupan en función del canal del protocolo AXI4 que se utilice.

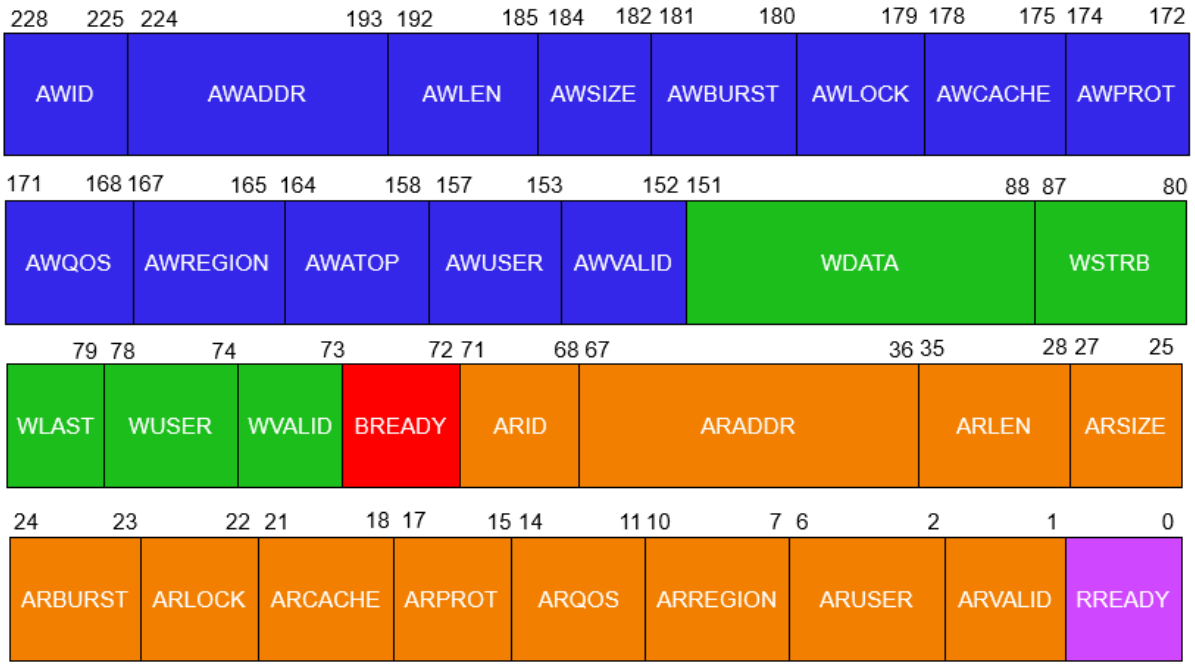

Figura 47. Representación gráfica de las señales concatenadas del puerto esclavo de entrada

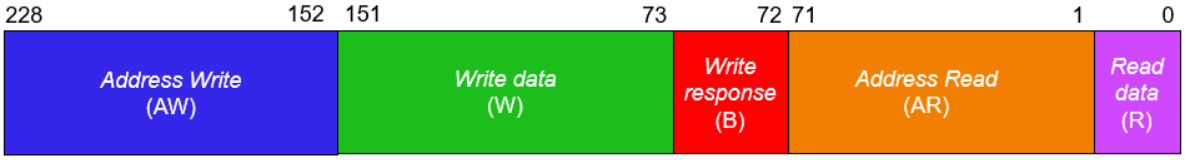

Figura 48. Canales del protocolo AXI4 utilizados en el puerto esclavo de entrada

Por lo tanto, el módulo *Top* debe contener el código que realiza la concatenación de todas estas señales *request* de los maestros. Esta estructura de señales se concatena tantas veces como maestros haya en el sistema. Por esta razón, en el caso que concierne a este proyecto, al haber seis maestros, se crea una estructura compleja de 1374 *bits*. La implementación de dicha estructura se muestra en la Figura 49.

```
592     .slv_ports_req_i( {
        {axi4_master5_AWID, axi4_master5_AWADDR, axi4_master5_AWLEN, axi4_master5_AWSIZE,
        axi4_master5_AWBURST, axi4_master5_AWLOCK, axi4_master5_AWCACHE, axi4_master5_AWPROT,
        axi4_master5_AWQOS, axi4_master5_AWREGION, axi4_master5_AWATOP, axi4_master5_AWUSER,
        axi4_master5_AWVALID, axi4_master5_WDATA, axi4_master5_WSTRB, axi4_master5_WLAST,
        axi4_master5_WUSER, axi4_master5_WVALID, axi4_master5_BVALID, axi4_master5_ARID,
        axi4_master5_ARADDR, axi4_master5_ARLEN, axi4_master5_ARSIZE, axi4_master5_ARBURST,
        axi4_master5_ARLOCK, axi4_master5_ARCACHE, axi4_master5_ARPROT, axi4_master5_ARQOS,
        axi4_master5_ARREGION, axi4_master5_ARUSER, axi4_master5_ARVALID, axi4_master5_RREADY},
593
594     {axi4_master4_AWID, axi4_master4_AWADDR, axi4_master4_AWLEN, axi4_master4_AWSIZE,
        axi4_master4_AWBURST, axi4_master4_AWLOCK, axi4_master4_AWCACHE, axi4_master4_AWPROT,
        axi4_master4_AWQOS, axi4_master4_AWREGION, axi4_master4_AWATOP, axi4_master4_AWUSER,
        axi4_master4_AWVALID, axi4_master4_WDATA, axi4_master4_WSTRB, axi4_master4_WLAST,
        axi4_master4_WUSER, axi4_master4_WVALID, axi4_master4_BVALID, axi4_master4_ARID,
        axi4_master4_ARADDR, axi4_master4_ARLEN, axi4_master4_ARSIZE, axi4_master4_ARBURST,
        axi4_master4_ARLOCK, axi4_master4_ARCACHE, axi4_master4_ARPROT, axi4_master4_ARQOS,
        axi4_master4_ARREGION, axi4_master4_ARUSER, axi4_master4_ARVALID, axi4_master4_RREADY},
595
596     {axi4_master3_AWID, axi4_master3_AWADDR, axi4_master3_AWLEN, axi4_master3_AWSIZE,
        axi4_master3_AWBURST, axi4_master3_AWLOCK, axi4_master3_AWCACHE, axi4_master3_AWPROT,
        axi4_master3_AWQOS, axi4_master3_AWREGION, axi4_master3_AWATOP, axi4_master3_AWUSER,
        axi4_master3_AWVALID, axi4_master3_WDATA, axi4_master3_WSTRB, axi4_master3_WLAST,
        axi4_master3_WUSER, axi4_master3_WVALID, axi4_master3_BVALID, axi4_master3_ARID,
        axi4_master3_ARADDR, axi4_master3_ARLEN, axi4_master3_ARSIZE, axi4_master3_ARBURST,
        axi4_master3_ARLOCK, axi4_master3_ARCACHE, axi4_master3_ARPROT, axi4_master3_ARQOS,
        axi4_master3_ARREGION, axi4_master3_ARUSER, axi4_master3_ARVALID, axi4_master3_RREADY},
597
598     {axi4_master2_AWID, axi4_master2_AWADDR, axi4_master2_AWLEN, axi4_master2_AWSIZE,
        axi4_master2_AWBURST, axi4_master2_AWLOCK, axi4_master2_AWCACHE, axi4_master2_AWPROT,
        axi4_master2_AWQOS, axi4_master2_AWREGION, axi4_master2_AWATOP, axi4_master2_AWUSER,
        axi4_master2_AWVALID, axi4_master2_WDATA, axi4_master2_WSTRB, axi4_master2_WLAST,
        axi4_master2_WUSER, axi4_master2_WVALID, axi4_master2_BVALID, axi4_master2_ARID,
        axi4_master2_ARADDR, axi4_master2_ARLEN, axi4_master2_ARSIZE, axi4_master2_ARBURST,
        axi4_master2_ARLOCK, axi4_master2_ARCACHE, axi4_master2_ARPROT, axi4_master2_ARQOS,
        axi4_master2_ARREGION, axi4_master2_ARUSER, axi4_master2_ARVALID, axi4_master2_RREADY},
599
600     {axi4_master1_AWID, axi4_master1_AWADDR, axi4_master1_AWLEN, axi4_master1_AWSIZE,
        axi4_master1_AWBURST, axi4_master1_AWLOCK, axi4_master1_AWCACHE, axi4_master1_AWPROT,
        axi4_master1_AWQOS, axi4_master1_AWREGION, axi4_master1_AWATOP, axi4_master1_AWUSER,
        axi4_master1_AWVALID, axi4_master1_WDATA, axi4_master1_WSTRB, axi4_master1_WLAST,
        axi4_master1_WUSER, axi4_master1_WVALID, axi4_master1_BVALID, axi4_master1_ARID,
        axi4_master1_ARADDR, axi4_master1_ARLEN, axi4_master1_ARSIZE, axi4_master1_ARBURST,
        axi4_master1_ARLOCK, axi4_master1_ARCACHE, axi4_master1_ARPROT, axi4_master1_ARQOS,
        axi4_master1_ARREGION, axi4_master1_ARUSER, axi4_master1_ARVALID, axi4_master1_RREADY},
601
602     {axi4_master0_AWID, axi4_master0_AWADDR, axi4_master0_AWLEN, axi4_master0_AWSIZE,
        axi4_master0_AWBURST, axi4_master0_AWLOCK, axi4_master0_AWCACHE, axi4_master0_AWPROT,
        axi4_master0_AWQOS, axi4_master0_AWREGION, axi4_master0_AWATOP, axi4_master0_AWUSER,
        axi4_master0_AWVALID, axi4_master0_WDATA, axi4_master0_WSTRB, axi4_master0_WLAST,
        axi4_master0_WUSER, axi4_master0_WVALID, axi4_master0_BVALID, axi4_master0_ARID,
        axi4_master0_ARADDR, axi4_master0_ARLEN, axi4_master0_ARSIZE, axi4_master0_ARBURST,
        axi4_master0_ARLOCK, axi4_master0_ARCACHE, axi4_master0_ARPROT, axi4_master0_ARQOS,
        axi4_master0_ARREGION, axi4_master0_ARUSER, axi4_master0_ARVALID, axi4_master0_RREADY}
        }),
```

Figura 49. Concatenación de las señales AXI4 en el puerto esclavo de entrada

5.2.2.2.3. Puerto maestro de salida

Todas las señales del protocolo AXI4 enviadas a través del puerto *slv_ports_req_i*, una vez atraviesen el módulo *crossbar,* deben ser enviadas a los módulos QVIP esclavos. Estas señales se envían a los esclavos a través del puerto *mst_ports_req_o*. Por lo tanto, el puerto *mst_ports_req_*o es un puerto maestro de salida al que se van a conectar los módulos QVIP esclavos implementados. A este puerto se conectarán las señales *request* enviadas a los esclavos. Al igual que en el caso del puerto *slv_ports_req_*i, este puerto debe conducir una concatenación de las señales que, enviadas por los QVIP maestros, reciben los QVIP esclavos. Como en el caso anterior, la Figura 50 y Figura 51 muestran la concatenación realizada y visualizan con mayor facilidad las señales que componen la estructura básica que se enviarán por el puerto maestro de salida y los *bits* que abarcan cada uno de los canales del protocolo AXI4. La estructura básica de señales contará, en este caso, con 235 *bits*.

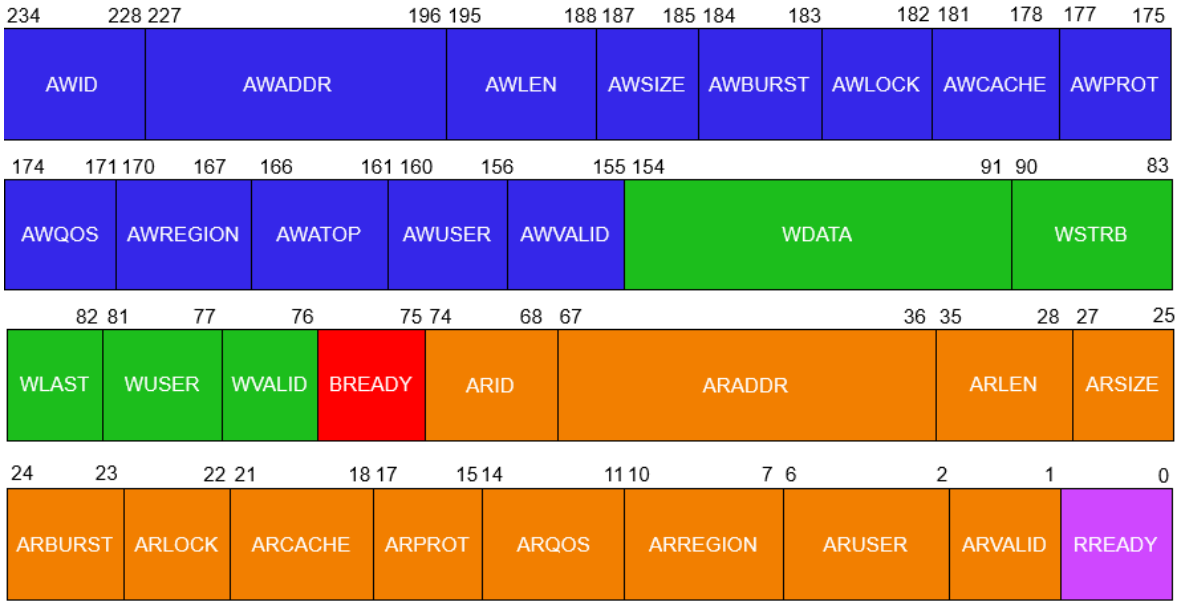

Figura 50. Representación gráfica de las señales concatenadas del puerto maestro de salida

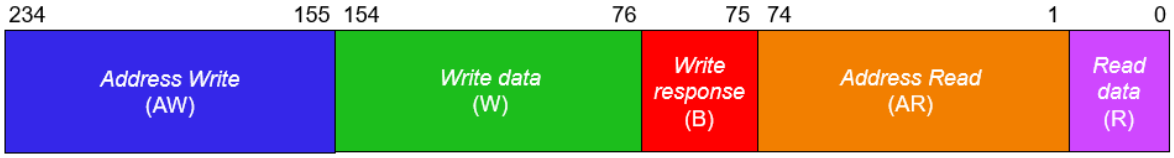

Figura 51. Canales del protocolo AXI4 utilizados en el puerto maestro de salida

Para la concatenación de estas señales se seguirá el *modus operandi* empleado anteriormente. El resultado de esta concatenación se visualiza en la Figura 52, donde se implementan ocho estructuras por cada uno de los esclavos del sistema, por lo que esta estructura compleja tendrá un ancho total de 1880 *bits*.

```
616     .mst_ports_req_o( {
        {axi4_slave7_AWID, axi4_slave7_AWADDR, axi4_slave7_AWLEN, axi4_slave7_AWSIZE,
        axi4_slave7_AWBURST, axi4_slave7_AWLOCK, axi4_slave7_AWCACHE, axi4_slave7_AWPROT,
        axi4_slave7_AWQOS, axi4_slave7_AWREGION, axi4_slave7_AWATOP, axi4_slave7_AWUSER,
        axi4_slave7_AWVALID, axi4_slave7_WDATA, axi4_slave7_WSTRB, axi4_slave7_WLAST,
        axi4_slave7_WUSER, axi4_slave7_WVALID, axi4_slave7_BREADY, axi4_slave7_ARID,
        axi4_slave7_ARADDR, axi4_slave7_ARLEN, axi4_slave7_ARSIZE, axi4_slave7_ARBURST,
        axi4_slave7_ARLOCK, axi4_slave7_ARCACHE, axi4_slave7_ARPROT, axi4_slave7_ARQOS,
        axi4_slave7_ARREGION, axi4_slave7_ARUSER, axi4_slave7_ARVALID, axi4_slave7_RREADY},
617
618     {axi4_slave6_AWID, axi4_slave6_AWADDR, axi4_slave6_AWLEN, axi4_slave6_AWSIZE,
        axi4_slave6_AWBURST, axi4_slave6_AWLOCK, axi4_slave6_AWCACHE, axi4_slave6_AWPROT,
        axi4_slave6_AWQOS, axi4_slave6_AWREGION, axi4_slave6_AWATOP, axi4_slave6_AWUSER,
        axi4_slave6_AWVALID, axi4_slave6_WDATA, axi4_slave6_WSTRB, axi4_slave6_WLAST,
        axi4_slave6_WUSER, axi4_slave6_WVALID, axi4_slave6_BREADY, axi4_slave6_ARID,
        axi4_slave6_ARADDR, axi4_slave6_ARLEN, axi4_slave6_ARSIZE, axi4_slave6_ARBURST,
        axi4_slave6_ARLOCK, axi4_slave6_ARCACHE, axi4_slave6_ARPROT, axi4_slave6_ARQOS,
        axi4_slave6_ARREGION, axi4_slave6_ARUSER, axi4_slave6_ARVALID, axi4_slave6_RREADY},
619
620     {axi4_slave5_AWID, axi4_slave5_AWADDR, axi4_slave5_AWLEN, axi4_slave5_AWSIZE,
        axi4_slave5_AWBURST, axi4_slave5_AWLOCK, axi4_slave5_AWCACHE, axi4_slave5_AWPROT,
        axi4_slave5_AWQOS, axi4_slave5_AWREGION, axi4_slave5_AWATOP, axi4_slave5_AWUSER,
        axi4_slave5_AWVALID, axi4_slave5_WDATA, axi4_slave5_WSTRB, axi4_slave5_WLAST,
        axi4_slave5_WUSER, axi4_slave5_WVALID, axi4_slave5_BREADY, axi4_slave5_ARID,
        axi4_slave5_ARADDR, axi4_slave5_ARLEN, axi4_slave5_ARSIZE, axi4_slave5_ARBURST,
        axi4_slave5_ARLOCK, axi4_slave5_ARCACHE, axi4_slave5_ARPROT, axi4_slave5_ARQOS,
        axi4_slave5_ARREGION, axi4_slave5_ARUSER, axi4_slave5_ARVALID, axi4_slave5_RREADY},
621
622     {axi4_slave4_AWID, axi4_slave4_AWADDR, axi4_slave4_AWLEN, axi4_slave4_AWSIZE,
        axi4_slave4_AWBURST, axi4_slave4_AWLOCK, axi4_slave4_AWCACHE, axi4_slave4_AWPROT,
        axi4_slave4_AWQOS, axi4_slave4_AWREGION, axi4_slave4_AWATOP, axi4_slave4_AWUSER,
        axi4_slave4_AWVALID, axi4_slave4_WDATA, axi4_slave4_WSTRB, axi4_slave4_WLAST,
        axi4_slave4_WUSER, axi4_slave4_WVALID, axi4_slave4_BREADY, axi4_slave4_ARID,
        axi4_slave4_ARADDR, axi4_slave4_ARLEN, axi4_slave4_ARSIZE, axi4_slave4_ARBURST,
        axi4_slave4_ARLOCK, axi4_slave4_ARCACHE, axi4_slave4_ARPROT, axi4_slave4_ARQOS,
        axi4_slave4_ARREGION, axi4_slave4_ARUSER, axi4_slave4_ARVALID, axi4_slave4_RREADY},
623
624     {axi4_slave3_AWID, axi4_slave3_AWADDR, axi4_slave3_AWLEN, axi4_slave3_AWSIZE,
        axi4_slave3_AWBURST, axi4_slave3_AWLOCK, axi4_slave3_AWCACHE, axi4_slave3_AWPROT,
        axi4_slave3_AWQOS, axi4_slave3_AWREGION, axi4_slave3_AWATOP, axi4_slave3_AWUSER,
        axi4_slave3_AWVALID, axi4_slave3_WDATA, axi4_slave3_WSTRB, axi4_slave3_WLAST,
        axi4_slave3_WUSER, axi4_slave3_WVALID, axi4_slave3_BREADY, axi4_slave3_ARID,
        axi4_slave3_ARADDR, axi4_slave3_ARLEN, axi4_slave3_ARSIZE, axi4_slave3_ARBURST,
        axi4_slave3_ARLOCK, axi4_slave3_ARCACHE, axi4_slave3_ARPROT, axi4_slave3_ARQOS,
        axi4_slave3_ARREGION, axi4_slave3_ARUSER, axi4_slave3_ARVALID, axi4_slave3_RREADY},
625
626     {axi4_slave2_AWID, axi4_slave2_AWADDR, axi4_slave2_AWLEN, axi4_slave2_AWSIZE,
        axi4_slave2_AWBURST, axi4_slave2_AWLOCK, axi4_slave2_AWCACHE, axi4_slave2_AWPROT,
        axi4_slave2_AWQOS, axi4_slave2_AWREGION, axi4_slave2_AWATOP, axi4_slave2_AWUSER,
        axi4_slave2_AWVALID, axi4_slave2_WDATA, axi4_slave2_WSTRB, axi4_slave2_WLAST,
        axi4_slave2_WUSER, axi4_slave2_WVALID, axi4_slave2_BREADY, axi4_slave2_ARID,
        axi4_slave2_ARADDR, axi4_slave2_ARLEN, axi4_slave2_ARSIZE, axi4_slave2_ARBURST,
        axi4_slave2_ARLOCK, axi4_slave2_ARCACHE, axi4_slave2_ARPROT, axi4_slave2_ARQOS,
        axi4_slave2_ARREGION, axi4_slave2_ARUSER, axi4_slave2_ARVALID, axi4_slave2_RREADY},
627
628     {axi4_slave1_AWID, axi4_slave1_AWADDR, axi4_slave1_AWLEN, axi4_slave1_AWSIZE,
        axi4_slave1_AWBURST, axi4_slave1_AWLOCK, axi4_slave1_AWCACHE, axi4_slave1_AWPROT,
        axi4_slave1_AWQOS, axi4_slave1_AWREGION, axi4_slave1_AWATOP, axi4_slave1_AWUSER,
        axi4_slave1_AWVALID, axi4_slave1_WDATA, axi4_slave1_WSTRB, axi4_slave1_WLAST,
        axi4_slave1_WUSER, axi4_slave1_WVALID, axi4_slave1_BREADY, axi4_slave1_ARID,
        axi4_slave1_ARADDR, axi4_slave1_ARLEN, axi4_slave1_ARSIZE, axi4_slave1_ARBURST,
        axi4_slave1_ARLOCK, axi4_slave1_ARCACHE, axi4_slave1_ARPROT, axi4_slave1_ARQOS,
        axi4_slave1_ARREGION, axi4_slave1_ARUSER, axi4_slave1_ARVALID, axi4_slave1_RREADY},
629
630     {axi4_slave0_AWID, axi4_slave0_AWADDR, axi4_slave0_AWLEN, axi4_slave0_AWSIZE,
        axi4_slave0_AWBURST, axi4_slave0_AWLOCK, axi4_slave0_AWCACHE, axi4_slave0_AWPROT,
        axi4_slave0_AWQOS, axi4_slave0_AWREGION, axi4_slave0_AWATOP, axi4_slave0_AWUSER,
        axi4_slave0_AWVALID, axi4_slave0_WDATA, axi4_slave0_WSTRB, axi4_slave0_WLAST,
        axi4_slave0_WUSER, axi4_slave0_WVALID, axi4_slave0_BREADY, axi4_slave0_ARID,
        axi4_slave0_ARADDR, axi4_slave0_ARLEN, axi4_slave0_ARSIZE, axi4_slave0_ARBURST,
        axi4_slave0_ARLOCK, axi4_slave0_ARCACHE, axi4_slave0_ARPROT, axi4_slave0_ARQOS,
        axi4_slave0_ARREGION, axi4_slave0_ARUSER, axi4_slave0_ARVALID, axi4_slave0_RREADY}
        }),
```

Figura 52. Concatenación de las señales AXI4 en el puerto maestro de salida

5.2.2.2.4. Puerto maestro de entrada

Cuando un QVIP esclavo recibe una petición *request* debe responder a dicha petición, tanto de escritura como de lectura. El módulo DUV original dispone de un puerto específico para ello, denominado *mst_ports_resp_i*. Este puerto se crea con el propósito de enviar las señales *response* de los módulos QVIP esclavos. En la Figura 54 se presenta la agrupación de cada canal del protocolo AXI4 que se utilizan en este puerto, donde se puede destacar que el canal *Address Write*, el canal *Address Read* y el canal *Write data* sólo poseen un único *bit* que pertenece a la señal *READY* de todos estos canales, que indicará cuando está preparado el módulo esclavo para recibir una transacción de escritura o lectura.

Todas las señales que se concatenan en este puerto se encuentran representadas gráficamente en la Figura 53, donde se dividen todos los campos que componen la estructura básica de 98 *bits*.

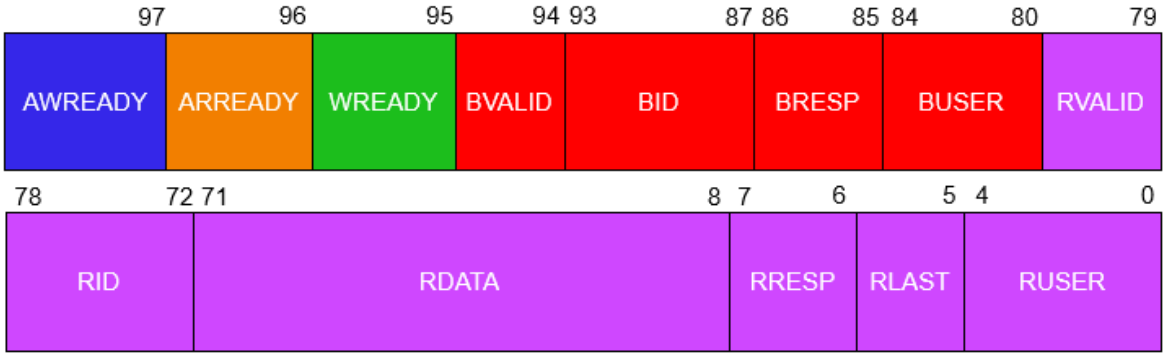

Figura 53. Representación gráfica de las señales concatenadas del puerto maestro de entrada

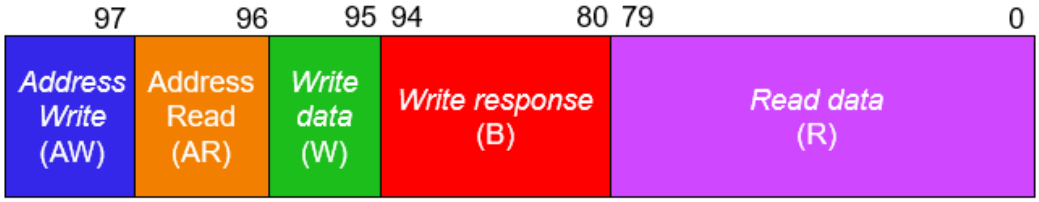

Figura 54. Canales del protocolo AXI4 utilizados en el puerto maestro de entrada

La Figura 53 representa únicamente la estructura básica de uno de los esclavos. Como el sistema diseñado contiene ocho módulos QVIP esclavos, en el código se implementa ocho veces esta estructura con las respectivas señales de cada uno de los módulos. Por lo tanto, la estructura compleja que contiene las señales de los ocho esclavos

contará con un total de 784 *bits* y el código que implementa esta estructura es el que se muestra en la Figura 55.

```
632     .mst_ports_resp_i( {
        {axi4_slave7_AWREADY, axi4_slave7_ARREADY, axi4_slave7_WREADY, axi4_slave7_BVALID,
        axi4_slave7_BID, axi4_slave7_BRESP, axi4_slave7_BUSER, axi4_slave7_RVALID, axi4_slave7_RID,
        axi4_slave7_RDATA, axi4_slave7_RRESP, axi4_slave7_RLAST, axi4_slave7_RUSER},
633
634     {axi4_slave6_AWREADY, axi4_slave6_ARREADY, axi4_slave6_WREADY, axi4_slave6_BVALID,
        axi4_slave6_BID, axi4_slave6_BRESP, axi4_slave6_BUSER, axi4_slave6_RVALID, axi4_slave6_RID,
        axi4_slave6_RDATA, axi4_slave6_RRESP, axi4_slave6_RLAST, axi4_slave6_RUSER},
635
636     {axi4_slave5_AWREADY, axi4_slave5_ARREADY, axi4_slave5_WREADY, axi4_slave5_BVALID,
        axi4_slave5_BID, axi4_slave5_BRESP, axi4_slave5_BUSER, axi4_slave5_RVALID, axi4_slave5_RID,
        axi4_slave5_RDATA, axi4_slave5_RRESP, axi4_slave5_RLAST, axi4_slave5_RUSER},
637
638     {axi4_slave4_AWREADY, axi4_slave4_ARREADY, axi4_slave4_WREADY, axi4_slave4_BVALID,
        axi4_slave4_BID, axi4_slave4_BRESP, axi4_slave4_BUSER, axi4_slave4_RVALID, axi4_slave4_RID,
        axi4_slave4_RDATA, axi4_slave4_RRESP, axi4_slave4_RLAST, axi4_slave4_RUSER},
639
640     {axi4_slave3_AWREADY, axi4_slave3_ARREADY, axi4_slave3_WREADY, axi4_slave3_BVALID,
        axi4_slave3_BID, axi4_slave3_BRESP, axi4_slave3_BUSER, axi4_slave3_RVALID, axi4_slave3_RID,
        axi4_slave3_RDATA, axi4_slave3_RRESP, axi4_slave3_RLAST, axi4_slave3_RUSER},
641
642     {axi4_slave2_AWREADY, axi4_slave2_ARREADY, axi4_slave2_WREADY, axi4_slave2_BVALID,
        axi4_slave2_BID, axi4_slave2_BRESP, axi4_slave2_BUSER, axi4_slave2_RVALID, axi4_slave2_RID,
        axi4_slave2_RDATA, axi4_slave2_RRESP, axi4_slave2_RLAST, axi4_slave2_RUSER},
643
644     {axi4_slave1_AWREADY, axi4_slave1_ARREADY, axi4_slave1_WREADY, axi4_slave1_BVALID,
        axi4_slave1_BID, axi4_slave1_BRESP, axi4_slave1_BUSER, axi4_slave1_RVALID, axi4_slave1_RID,
        axi4 slave1 RDATA, axi4 slave1 RRESP, axi4 slave1 RLAST, axi4 slave1 RUSER},
645
646     {axi4_slave0_AWREADY, axi4_slave0_ARREADY, axi4_slave0_WREADY, axi4_slave0_BVALID,
        axi4_slave0_BID, axi4_slave0_BRESP, axi4_slave0_BUSER, axi4_slave0_RVALID, axi4_slave0_RID,
        axi4_slave0_RDATA, axi4_slave0_RRESP, axi4_slave0_RLAST, axi4_slave0_RUSER}
        })
    );
```

Figura 55. Concatenación de las señales AXI4 en el puerto maestro de entrada

5.2.2.2.5. Puerto esclavo de salida

El puerto esclavo de salida del módulo *crossbar* se llama *slv_ports_resp_o* y se utiliza para el envío de las señales *response* de todos los módulos QVIP esclavos a los módulos QVIP maestros conectados. La estructura básica de las señales se parece mucho a la estructura del puerto maestro de entrada, pero cuenta con una diferencia que cambia sustancialmente la estructura, ya que cambia el ancho del parámetro ID en los canales *Write* response y *Read data* como se observa en la Figura 56. Por lo tanto, la extensión de la estructura básica disminuye a 92 *bits*. La Figura 57 muestra los *bits* de cada uno de los canales, cuyas señales se concatenan en el puerto esclavo de salida.

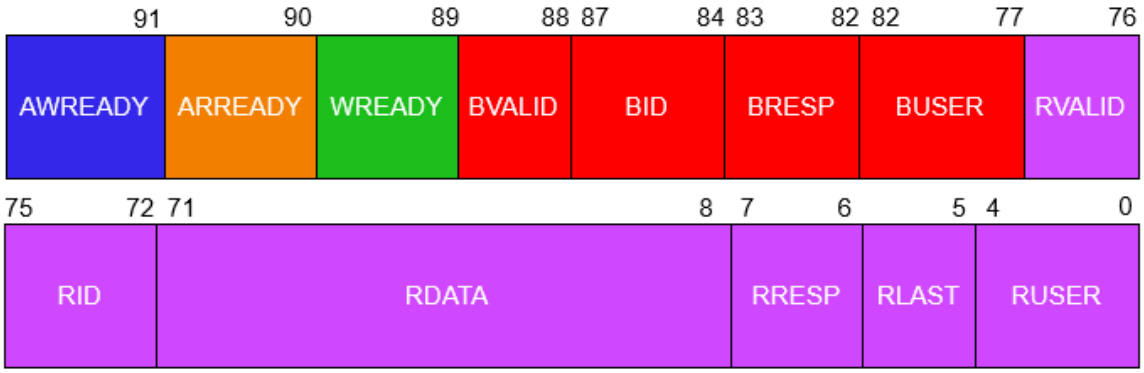

Figura 56. Representación gráfica de las señales concatenadas del puerto esclavo de salida

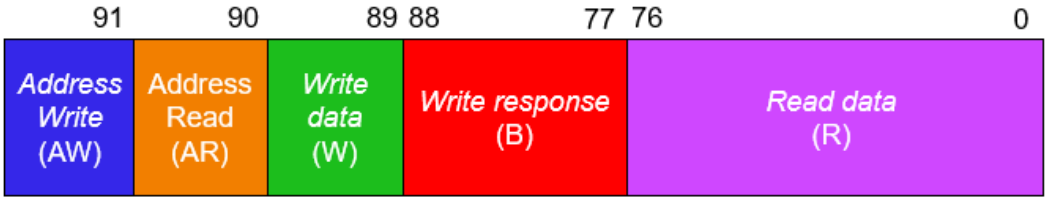

Figura 57. Canales del protocolo AXI4 utilizados en el puerto esclavo de salida

Como se ha comentado en anteriores ocasiones, es importante realizar la concatenación de todas las señales de la Figura 56 en el orden específico que marca el módulo *crossbar*. Además, se deben concatenar seis veces la estructura básica antes descrita en relación con los seis módulos QVIP maestros que existen en el entorno de verificación UVM, por lo que, en total, se concatenarán 552 *bits*. En la Figura 58 se presenta el código que representa la concatenación de todas las señales.

```
604     .slv_ports_resp_o( {
        {axi4_master5_AWREADY, axi4_master5_ARREADY, axi4_master5_WREADY, axi4_master5_BVALID,
        axi4_master5_BID, axi4_master5_BRESP, axi4_master5_BUSER, axi4_master5_RVALID,
        axi4_master5_RID, axi4_master5_RDATA, axi4_master5_RRESP, axi4_master5_RLAST,
        axi4_master5_RUSER},
605
606     {axi4_master4_AWREADY, axi4_master4_ARREADY, axi4_master4_WREADY, axi4_master4_BVALID,
        axi4_master4_BID, axi4_master4_BRESP, axi4_master4_BUSER, axi4_master4_RVALID,
        axi4_master4_RID, axi4_master4_RDATA, axi4_master4_RRESP, axi4_master4_RLAST,
        axi4_master4_RUSER},
607
608     {axi4_master3_AWREADY, axi4_master3_ARREADY, axi4_master3_WREADY, axi4_master3_BVALID,
        axi4_master3_BID, axi4_master3_BRESP, axi4_master3_BUSER, axi4_master3_RVALID,
        axi4_master3_RID, axi4_master3_RDATA, axi4_master3_RRESP, axi4_master3_RLAST,
        axi4_master3_RUSER},
609
610     {axi4_master2_AWREADY, axi4_master2_ARREADY, axi4_master2_WREADY, axi4_master2_BVALID,
        axi4_master2_BID, axi4_master2_BRESP, axi4_master2_BUSER, axi4_master2_RVALID,
        axi4_master2_RID, axi4_master2_RDATA, axi4_master2_RRESP, axi4_master2_RLAST,
        axi4_master2_RUSER},
611
612     {axi4_master1_AWREADY, axi4_master1_ARREADY, axi4_master1_WREADY, axi4_master1_BVALID,
        axi4_master1_BID, axi4_master1_BRESP, axi4_master1_BUSER, axi4_master1_RVALID,
        axi4_master1_RID, axi4_master1_RDATA, axi4_master1_RRESP, axi4_master1_RLAST,
        axi4_master1_RUSER},
613
614     {axi4_master0_AWREADY, axi4_master0_ARREADY, axi4_master0_WREADY, axi4_master0_BVALID,
        axi4_master0_BID, axi4_master0_BRESP, axi4_master0_BUSER, axi4_master0_RVALID,
        axi4_master0_RID, axi4_master0_RDATA, axi4_master0_RRESP, axi4_master0_RLAST,
        axi4_master0_RUSER}
        }),
```

Figura 58. Concatenación de las señales AXI4 en el puerto esclavo de salida

5.2.3. Componente UVM *Test*

El componente UVM *Test* engloba todos los componentes UVM *Environment* del entorno de verificación y, a su vez, todos los componentes UVM que se encuentran dentro de dichos componentes UVM *Environment*. La principal función del componente UVM *Test* es definir una librería de *test*, la cual estará compuesta por múltiples secuencias que serán enviadas al componente UVM *Sequencer*, integrado en cada componente UVM *Agent*, con el único propósito de estimular las interfaces del DUV de manera controlada. La monitorización de las salidas del dispositivo a verificar se realiza mediante los componentes UVM *Monitor*, integrados en los componentes UVM *Agent*.

En este apartado se detalla el código para la implementación del componente UVM *Test*. En primer lugar, cada uno de los *test* a desarrollar se deben definir con nombre único. En este caso particular, el *test* que se presenta, se ha denominado *qvip_axi_xbar_test*. Este *test*, cuya descripción se muestra en la Figura 59, se deriva de la clase `uvm_test` y así hereda todas las funciones propias de esta clase. Mediante el uso de la macro

`uvm_component_utils la clase *qvip_axi_xbar_test_base* se registra en la *Factory* (línea 2 del código de la Figura 59). En el componente UVM *Test* se debe referenciar el componente UVM *Environment* de más alto nivel (línea 21 del código) y, a partir del mismo, se referencian el resto de los componentes del entorno de verificación. Además, se crean todos los objetos de configuración de los módulos maestros y esclavos (líneas 4 a 17 del código), que servirán para configurarlos con los parámetros diseñados en el *Environment*, como posteriormente se explicará. Además del constructor de la clase *function new*, el cuerpo de este objeto contiene dos funciones y una tarea. Por un lado, una de las funciones corresponde a la fase de creación del objeto, denominada `build_phase()`. Por otro lado, la tarea corresponde con la fase de ejecución del *test*, conocida como `run_phase()`. En esta fase, el *test* iniciará la secuencia o secuencias a enviar al DUV.

```
1  class qvip_axi_xbar_test_base extends uvm_test;
2    `uvm_component_utils(qvip_axi_xbar_test_base)
3
4    axi4_master0_cfg_t axi4_master0_cfg;
5    axi4_master1_cfg_t axi4_master1_cfg;
6    axi4_master2_cfg_t axi4_master2_cfg;
7    axi4_master3_cfg_t axi4_master3_cfg;
8    axi4_master4_cfg_t axi4_master4_cfg;
9    axi4_master5_cfg_t axi4_master5_cfg;
10   axi4_slave0_cfg_t axi4_slave0_cfg;
11   axi4_slave1_cfg_t axi4_slave1_cfg;
12   axi4_slave2_cfg_t axi4_slave2_cfg;
13   axi4_slave3_cfg_t axi4_slave3_cfg;
14   axi4_slave4_cfg_t axi4_slave4_cfg;
15   axi4_slave5_cfg_t axi4_slave5_cfg;
16   axi4_slave6_cfg_t axi4_slave6_cfg;
17   axi4_slave7_cfg_t axi4_slave7_cfg;
18
19   qvip_axi_xbar_env_config env_cfg;
20
21   qvip_axi_xbar_env env;
22
23   function new (string name = "qvip_axi_xbar_test_base_test_base", uvm_component parent = null);
24     super.new(name, parent);
25   endfunction
26
27   extern function void init_vseq (qvip_axi_xbar_vseq_wr_rd vseq);
28
29   extern function void build_phase (uvm_phase phase);
30
31   extern task run_phase (uvm_phase phase);
32
33 endclass: qvip_axi_xbar_test_base
```

Figura 59. Fragmento del código del componente UVM Test

En la metodología UVM, las secuencias UVM son objetos que tienen una vida útil limitada, es decir, una vez que una secuencia se completa al generar todos los estímulos, se descarta. El componente UVM *Test* crea y ejecuta todas y cada una de las secuencias

que componente el *test*. En este caso, las secuencias son creadas y ejecutadas dentro de la tarea `run_phase`, pero antes de todo esto, se debe ejecutar la función `init_vseq`, la cual asigna a todos los módulos maestros y esclavos el componente UVM *Sequencer* correspondiente a cada uno de ellos. La asignación, tal y como se muestra en la Figura 60, se realiza asignando un identificador, denominado `m_sequencer`, a cada uno de los módulos QVIP maestros y esclavos. Este identificador, de tipo `uvm_sequencer_base`, está disponible por defecto en cada secuencia. Por lo tanto, el identificador `m_sequencer` se comporta como un puntero al componente UVM *Sequencer* específico que identificará la secuencia indicada a la interfaz correspondiente del DUV.

```
35  function void qvip_axi_xbar_test_base::init_vseq (qvip_axi_xbar_vseq_wr_rd vseq);
36
37    vseq.axi4_master0 = env.axi4_master0.m_sequencer;
38    vseq.axi4_master1 = env.axi4_master1.m_sequencer;
39    vseq.axi4_master2 = env.axi4_master2.m_sequencer;
40    vseq.axi4_master3 = env.axi4_master3.m_sequencer;
41    vseq.axi4_master4 = env.axi4_master4.m_sequencer;
42    vseq.axi4_master5 = env.axi4_master5.m_sequencer;
43    vseq.axi4_slave0 = env.axi4_slave0.m_sequencer;
44    vseq.axi4_slave1 = env.axi4_slave1.m_sequencer;
45    vseq.axi4_slave2 = env.axi4_slave2.m_sequencer;
46    vseq.axi4_slave3 = env.axi4_slave3.m_sequencer;
47    vseq.axi4_slave4 = env.axi4_slave4.m_sequencer;
48    vseq.axi4_slave5 = env.axi4_slave5.m_sequencer;
49    vseq.axi4_slave6 = env.axi4_slave6.m_sequencer;
50    vseq.axi4_slave7 = env.axi4_slave7.m_sequencer;
51
52  endfunction: init_vseq
```

Figura 60. Función que inicializa la secuencia

Durante la fase `build_phase()` se crean todos los componentes UVM, siendo esta fase la primera en el flujo de ejecución. Por esta razón, en el componente UVM *Test* es donde se crean todos estos componentes porque es el nivel más alto de la jerarquía UVM. En la línea 56 de la Figura 61 se crea un componente para la configuración del componente UVM *Environment*, donde se guarda el mapa de direcciones con el que se definirá las diferentes regiones de los módulos maestros y esclavos dentro del mapa. Después se debe crear un objeto para la configuración de cada uno de los módulos QVIP que sentarán las bases de la configuración de todos ellos en el componente UVM *Environment*. Este procedimiento se llevará a cabo con los seis maestros y los ocho esclavos del entorno.

Para facilitar la etapa de depuración en caso de fallo, cada vez que se crea un componente de configuración, se valora la posibilidad de encontrar un error. Si esto ocurre, se saca un mensaje de error usando el mecanismo de mensajes de UVM, como se

muestra, por ejemplo, en la línea 52 del código. Este mensaje, debido a su importancia, detendrá la ejecución del *test*. Una vez finalizada la creación de todos los objetos de configuración que se realiza en líneas 59 a 183 de la Figura 61, de la Figura 62 y de la Figura 63, se crea un componente UVM *Environment*, el cual se configura con el objeto `env_cfg`. Este último objeto guarda toda la topología del entorno de verificación.

```
54  function void qvip_axi_xbar_test_base::build_phase(uvm_phase phase);
55
56    env_cfg = qvip_axi_xbar_env_config::type_id::create("env_cfg");
57    env_cfg.initialize();
58
59    axi4_master0_cfg = axi4_master0_cfg_t::type_id::create("axi4_master0_cfg");
60    if(!uvm_config_db #(axi4_master0_bfm_t)::get(this,"","axi4_master0", axi4_master0_cfg.m_bfm))
61      begin
62        `uvm_error("build_phase", "Unable to get virtual interface axi4_master0 for
63                                   axi4_master0_cfg from uvm_config_db")
64      end
65    axi4_master0_config_policy::configure(axi4_master0_cfg, env_cfg.addr_map);
66    env_cfg.axi4_master0_cfg = axi4_master0_cfg;
67
68    axi4_master1_cfg = axi4_master1_cfg_t::type_id::create("axi4_master1_cfg");
69    if(!uvm_config_db #(axi4_master1_bfm_t)::get(this,"","axi4_master1", axi4_master1_cfg.m_bfm))
70      begin
71        `uvm_error("build_phase", "Unable to get virtual interface axi4_master1 for
72                                   axi4_master1_cfg from uvm_config_db")
73      end
74    axi4_master1_config_policy::configure(axi4_master1_cfg, env_cfg.addr_map);
75    env_cfg.axi4_master1_cfg = axi4_master1_cfg;
76
77    axi4_master2_cfg = axi4_master2_cfg_t::type_id::create("axi4_master2_cfg");
78    if(!uvm_config_db #(axi4_master2_bfm_t)::get(this,"","axi4_master2", axi4_master2_cfg.m_bfm))
79      begin
80        `uvm_error("build_phase", "Unable to get virtual interface axi4_master2 for
81                                   axi4_master2_cfg from uvm_config_db")
82      end
83    axi4_master2_config_policy::configure(axi4_master2_cfg, env_cfg.addr_map);
84    env_cfg.axi4_master2_cfg = axi4_master2_cfg;
85
86    axi4_master3_cfg = axi4_master3_cfg_t::type_id::create("axi4_master3_cfg");
87    if(!uvm_config_db #(axi4_master3_bfm_t)::get(this,"","axi4_master3", axi4_master3_cfg.m_bfm))
88      begin
89        `uvm_error("build_phase", "Unable to get virtual interface axi4_master3 for
90                                   axi4_master3_cfg from uvm_config_db")
91      end
92    axi4_master3_config_policy::configure(axi4_master3_cfg, env_cfg.addr_map);
93    env_cfg.axi4_master3_cfg = axi4_master3_cfg;
94
95    axi4_master4_cfg = axi4_master4_cfg_t::type_id::create("axi4_master4_cfg");
96    if(!uvm_config_db #(axi4_master4_bfm_t)::get(this,"","axi4_master4", axi4_master4_cfg.m_bfm))
97      begin
98        `uvm_error("build_phase", "Unable to get virtual interface axi4_master4 for
99                                   axi4_master4_cfg from uvm_config_db")
100     end
101   axi4_master4_config_policy::configure(axi4_master4_cfg, env_cfg.addr_map);
102   env_cfg.axi4_master4_cfg = axi4_master4_cfg;
103
104   axi4_master5_cfg = axi4_master5_cfg_t::type_id::create("axi4_master5_cfg");
105   if(!uvm_config_db #(axi4_master5_bfm_t)::get(this,"","axi4_master5", axi4_master5_cfg.m_bfm))
106     begin
107       `uvm_error("build_phase", "Unable to get virtual interface axi4_master5 for
108                                  axi4_master5_cfg from uvm_config_db")
109     end
110   axi4_master5_config_policy::configure(axi4_master5_cfg, env_cfg.addr_map);
111   env_cfg.axi4_master5_cfg = axi4_master5_cfg;
112
```

Figura 61. Fragmento 1 del código de la fase build_phase() del componente UVM Test

```
113     axi4_slave0_cfg = axi4_slave0_cfg_t::type_id::create("axi4_slave0_cfg");
114     if(!uvm_config_db #(axi4_slave0_bfm_t)::get(this,"","axi4_slave0", axi4_slave0_cfg.m_bfm))
115       begin
116         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave0 for
117                                    axi4_slave0_cfg from uvm_config_db")
118       end
119     axi4_slave0_config_policy::configure(axi4_slave0_cfg, env_cfg.addr_map);
120     env_cfg.axi4_slave0_cfg = axi4_slave0_cfg;
121
122     axi4_slave1_cfg = axi4_slave1_cfg_t::type_id::create("axi4_slave1_cfg");
123     if(!uvm_config_db #(axi4_slave1_bfm_t)::get(this,"","axi4_slave1", axi4_slave1_cfg.m_bfm))
124       begin
125         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave1 for
126                                    axi4_slave1_cfg from uvm_config_db")
127       end
128     axi4_slave1_config_policy::configure(axi4_slave1_cfg, env_cfg.addr_map);
129     env_cfg.axi4_slave1_cfg = axi4_slave1_cfg;
130
131     axi4_slave2_cfg = axi4_slave2_cfg_t::type_id::create("axi4_slave2_cfg");
132     if(!uvm_config_db #(axi4_slave2_bfm_t)::get(this,"","axi4_slave2", axi4_slave2_cfg.m_bfm))
133       begin
134         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave2 for
135                                    axi4_slave2_cfg from uvm_config_db")
136       end
137     axi4_slave2_config_policy::configure(axi4_slave2_cfg, env_cfg.addr_map);
138     env_cfg.axi4_slave2_cfg = axi4_slave2_cfg;
139
140     axi4_slave3_cfg = axi4_slave3_cfg_t::type_id::create("axi4_slave3_cfg");
141     if(!uvm_config_db #(axi4_slave3_bfm_t)::get(this,"","axi4_slave3", axi4_slave3_cfg.m_bfm))
142       begin
143         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave3 for
144                                    axi4_slave3_cfg from uvm_config_db")
145       end
146     axi4_slave3_config_policy::configure(axi4_slave3_cfg, env_cfg.addr_map);
147     env_cfg.axi4_slave3_cfg = axi4_slave3_cfg;
148
149     axi4_slave4_cfg = axi4_slave4_cfg_t::type_id::create("axi4_slave4_cfg");
150     if(!uvm_config_db #(axi4_slave4_bfm_t)::get(this,"","axi4_slave4", axi4_slave4_cfg.m_bfm))
151       begin
152         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave4 for
153                                    axi4_slave4_cfg from uvm_config_db")
154       end
155     axi4_slave4_config_policy::configure(axi4_slave4_cfg, env_cfg.addr_map);
156     env_cfg.axi4_slave4_cfg = axi4_slave4_cfg;
157
158     axi4_slave5_cfg = axi4_slave5_cfg_t::type_id::create("axi4_slave5_cfg");
159     if(!uvm_config_db #(axi4_slave5_bfm_t)::get(this,"","axi4_slave5", axi4_slave5_cfg.m_bfm))
160       begin
161         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave5 for
162                                    axi4_slave5_cfg from uvm_config_db")
163       end
164     axi4_slave5_config_policy::configure(axi4_slave5_cfg, env_cfg.addr_map);
165     env_cfg.axi4_slave5_cfg = axi4_slave5_cfg;
166
167     axi4_slave6_cfg = axi4_slave6_cfg_t::type_id::create("axi4_slave6_cfg");
168     if(!uvm_config_db #(axi4_slave6_bfm_t)::get(this,"","axi4_slave6", axi4_slave6_cfg.m_bfm))
169       begin
170         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave6 for
171                                    axi4_slave6_cfg from uvm_config_db")
172       end
173     axi4_slave6_config_policy::configure(axi4_slave6_cfg, env_cfg.addr_map);
174     env_cfg.axi4_slave6_cfg = axi4_slave6_cfg;
175
```

Figura 62. Fragmento 2 del código de la fase build_phase() del componente UVM Test

```
176     axi4_slave7_cfg = axi4_slave7_cfg_t::type_id::create("axi4_slave7_cfg");
177     if(!uvm_config_db #(axi4_slave7_bfm_t)::get(this,"","axi4_slave7", axi4_slave7_cfg.m_bfm))
178       begin
179         `uvm_error("build_phase", "Unable to get virtual interface axi4_slave7 for
180                                    axi4_slave7_cfg from uvm_config_db")
181       end
182     axi4_slave7_config_policy::configure(axi4_slave7_cfg, env_cfg.addr_map);
183     env_cfg.axi4_slave7_cfg = axi4_slave7_cfg;
184
185     env = qvip_axi_xbar_env::type_id::create("env", this);
186     env.cfg = env_cfg;
187
188   endfunction: build_phase
```

Figura 63. Fragmento 3 del código de la fase build_phase() del componente UVM Test

Respecto a la fase `run_phase()`, se ejecuta después de la creación del entorno de verificación UVM. Durante esta fase se realizan todas las operaciones para la generación de estímulos, de monitorización y verificación de datos en las interfaces del DUV. En este caso y en relación con la generación de estímulos, como se presenta en la Figura 64, durante esta tarea se creará la secuencia UVM que el ingeniero de verificación haya introducido al efectuar la simulación. En caso de errores graves saltará un `` `uvm_fatal `` que saldrá por pantalla si, por ejemplo, no detecta la secuencia introducida en la *Factory*. Por último, se inicia la secuencia UVM.

```
190   task qvip_axi_xbar_test_base::run_phase(uvm_phase phase);
191
192     string sequence_name;
193     qvip_axi_xbar_vseq_wr_rd vseq;
194     uvm_object obj;
195     uvm_cmdline_processor clp;
196     clp = uvm_cmdline_processor::get_inst();
197
198     if(clp.get_arg_value("+SEQ=",sequence_name) == 0)
199       begin
200         `uvm_fatal(get_type_name(), "You must specify a virtual sequence to run using
201                                      the +SEQ plusarg")
202       end
203     obj = factory.create_object_by_name(sequence_name);
204     if (obj == null)
205       begin
206         factory.print();
207         `uvm_fatal(get_type_name(),{"Virtual sequence'",sequence_name,"'not found in factory"})
208       end
209
210     if (!$cast(vseq, obj) )
211       begin
212         `uvm_fatal(get_type_name(),{"Virtual sequence'",sequence_name,"'is not derived
213                                      from qvip_axi_xbar_vseq_wr_rd"})
214       end
215
216     `uvm_info(get_type_name(),{"Running virtual sequence'",sequence_name,"'"},UVM_LOW)
217
218     phase.raise_objection(this);
219     init_vseq(vseq);
220     vseq.start(null);
221     phase.drop_objection(this);
222
223   endtask: run_phase
```

Figura 64. Tarea que ejecuta la fase run_phase()

5.2.4. Componente UVM *Environment*

A simple vista, el componente UVM *Environment* carece de importancia, pero es un componente de gran utilidad en el entorno de verificación. Sin este componente la reusabilidad del código se reduciría casi por completo, debido a que el *testbench* UVM programado dejaría de ser reutilizable porque se basaría en una estructura del entorno específica para cada caso en concreto. Además, los cambios en la topología del sistema supondrían la necesidad de actualización de varios módulos del componente UVM *Test*.

Un factor clave es que la configuración del entorno de verificación se realiza en el componente UVM *Environment*. La filosofía de la herramienta QVIP *configurator* incluye las referencias de los componentes UVM *Agent* en el componente UVM *Environment*. Dentro de la descripción del código correspondiente a este componente, se ha decidido incluir un primer apartado para detallar la configuración asociada al componente UVM *Environment*. Esta configuración se realiza mediante el objeto de configuración referenciado en la línea 21 del código de la Figura 59.

5.2.4.1. Configuración del componente UVM *Environment*

La clase destinada a configurar el componente UVM *Environment* se deriva de la clase raíz `uvm_object` y se registra en la *Factory* con el nombre, en este caso particular, *qvip_axi_xbar_env_config*, tal y como se observa en la Figura 65. Esta clase define un objeto de configuración para cada uno de los módulos maestros y esclavos que usará el componente UVM *Environment*. Además, en esta clase se define un mapa de direcciones que será utilizado por el *testbench* para obtener y verificar las propiedades de cada una de las transacciones de datos enviadas o recibidas. En este caso, en la línea 19 de la Figura 65, se define un objeto, que deriva de la clase *Address Map*, y que contendrá todos los rangos de memoria de cada uno de los módulos QVIP, así como las características básicas de este mapa de direcciones, la asignación de rangos y la descripción de los mismos, esto se realiza en la función *initialize*. En el caso particular de este Trabajo Fin de Grado, se ha seguido la pauta del diseño original del *crossbar*, cuyo mapa de direcciones está dividido en 8 regiones bien diferenciadas.

```
1  class qvip_axi_xbar_env_config extends uvm_object;
2      `uvm_object_utils(qvip_axi_xbar_env_config)
3
4      axi4_master0_cfg_t axi4_master0_cfg;
5      axi4_master1_cfg_t axi4_master1_cfg;
6      axi4_master2_cfg_t axi4_master2_cfg;
7      axi4_master3_cfg_t axi4_master3_cfg;
8      axi4_master4_cfg_t axi4_master4_cfg;
9      axi4_master5_cfg_t axi4_master5_cfg;
10     axi4_slave0_cfg_t axi4_slave0_cfg;
11     axi4_slave1_cfg_t axi4_slave1_cfg;
12     axi4_slave2_cfg_t axi4_slave2_cfg;
13     axi4_slave3_cfg_t axi4_slave3_cfg;
14     axi4_slave4_cfg_t axi4_slave4_cfg;
15     axi4_slave5_cfg_t axi4_slave5_cfg;
16     axi4_slave6_cfg_t axi4_slave6_cfg;
17     axi4_slave7_cfg_t axi4_slave7_cfg;
18
19     address_map addr_map;
20
21     function new (string name = "qvip_axi_xbar_env_config");
22         super.new(name);
23     endfunction
24
25     extern function void initialize;
26
27 endclass: qvip_axi_xbar_env_config
28
29 function void qvip_axi_xbar_env_config::initialize;
30     begin
31         addr_map_entry_s addr_map_entries[] = new [8];
32         addr_map_entries = '{
33             '{MAP_NORMAL,"RANGE_1",0,MAP_NS,4'h0,32'h0000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
34             '{MAP_NORMAL,"RANGE_2",0,MAP_NS,4'h0,32'h1000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
35             '{MAP_NORMAL,"RANGE_3",0,MAP_NS,4'h0,32'h2000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
36             '{MAP_NORMAL,"RANGE_4",0,MAP_NS,4'h0,32'h3000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
37             '{MAP_NORMAL,"RANGE_5",0,MAP_NS,4'h0,32'h4000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
38             '{MAP_NORMAL,"RANGE_6",0,MAP_NS,4'h0,32'h5000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
39             '{MAP_NORMAL,"RANGE_7",0,MAP_NS,4'h0,32'h6000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA},
40             '{MAP_NORMAL,"RANGE_8",0,MAP_NS,4'h0,32'h7000_0000,32'h1000_0000,MEM_NORMAL,MAP_NORM_SEC_DATA}
41         };
42         addr_map = address_map::type_id::create("addr_map_addr_map");
43         addr_map.addr_mask = 64'hFFF;
44         addr_map.set( addr_map_entries );
45     end
46 endfunction: initialize
```

Figura 65. Configuración del componente UVM Environment

5.2.4.2. Descripción del componente UVM *Environment*

En la Figura 66 se muestra el código que define el componente UVM *Environment* del entorno de verificación propuesto. Como se comentó en el apartado anterior, la clase *qvip_axi_xbar_env_config* es absolutamente necesaria para la correcta configuración de los módulos QVIP maestros y esclavos presentes en el sistema. En la citada figura, en las líneas 6 a 19, se muestra cómo se referencian los agentes que integran este componente y que corresponden a los seis agentes maestros y ocho agentes esclavos presentes en el diseño propuesto. Este componente, además de su constructor *new*, incluye la función `build_phase()`.

```
1  class qvip_axi_xbar_env extends uvm_component;
2    `uvm_component_utils(qvip_axi_xbar_env)
3
4    qvip_axi_xbar_env_config cfg;
5
6    axi4_master0_agent_t axi4_master0;
7    axi4_master1_agent_t axi4_master1;
8    axi4_master2_agent_t axi4_master2;
9    axi4_master3_agent_t axi4_master3;
10   axi4_master4_agent_t axi4_master4;
11   axi4_master5_agent_t axi4_master5;
12   axi4_slave0_agent_t axi4_slave0;
13   axi4_slave1_agent_t axi4_slave1;
14   axi4_slave2_agent_t axi4_slave2;
15   axi4_slave3_agent_t axi4_slave3;
16   axi4_slave4_agent_t axi4_slave4;
17   axi4_slave5_agent_t axi4_slave5;
18   axi4_slave6_agent_t axi4_slave6;
19   axi4_slave7_agent_t axi4_slave7;
20
21   function new ( string name = "qvip_axi_xbar_env", uvm_component parent = null );
22     super.new(name, parent);
23   endfunction
24
25   extern function void build_phase (uvm_phase phase);
26
27 endclass: qvip_axi_xbar_env
```

Figura 66. Fragmento del código del componente UVM Environment

En la Figura 67 se muestra la variable de configuración de un IP que utiliza el protocolo de comunicación AXI4. El ingeniero de verificación puede particularizar cada maestro o esclavo del sistema, haciendo uso de su objeto de configuración. Así, por ejemplo, si el ingeniero de verificación quiere particularizar el agente correspondiente al maestro 0, puede configurarlo a través del objeto *axi4_master0_cfg*, que se asigna a dicho maestro en la línea 4 de la Figura 65.

```
1   // Variable: agent_cfg
2   //
3   // Used to configure the AXI4 agent.
4   // See also: <axi4_agent_cfg_s> struct for detailed description of
5   // the configuration options available.
6   //
7   axi4_agent_cfg_s agent_cfg = '{is_active   :  1'b1,
8                                  agent_type  :  AXI4_MASTER,
9                                  if_type     :  AXI4,
10                                 ext_clock   :  1'b1,
11                                 ext_reset   :  1'b1,
12                                 en_cvg      : '{func  : 1'b0,
13                                                 wr_ch_toggle : 1'b0,
14                                                 rd_ch_toggle : 1'b0},
15                                 en_logger   : '{txn_log  : 0,
16                                                 beat_log : 0,
17                                                 txn_log_name : "txn.log",
18                                                 beat_log_name : "beat.log",
```

Figura 67. Variable de configuración de un IP AXI4

El código presentado en la Figura 67 muestra un fragmento de la declaración del objeto de configuración de un agente. En la estructura *agent_cfg* se pueden distinguir los diferentes campos de configuración, como por ejemplo: el campo *is_active*, que determina si el agente es activo o pasivo; el campo *agent_type*, el cual determina si se trata de un maestro o un esclavo; el campo *if_type*, que especifica el tipo de interfaz, etc.

La Figura 68 presenta el código correspondiente a la fase `build_phase()`, donde, en primer lugar, se referencia el objeto de configuración del componente UVM *Environment* y, en segundo lugar, todos los módulos de configuración de los agentes maestros y esclavos que lo conforman.

```
29  function void qvip_axi_xbar_env::build_phase (uvm_phase phase);
30    if ( cfg == null )
31      begin
32        if ( !uvm_config_db #(qvip_axi_xbar_env_config)::get(this, "", "env_config", cfg) )
33          begin
34            `uvm_error("build_phase", "Unable to find the env config object in the uvm_config_db")
35          end
36      end
37    axi4_master0 = axi4_master0_agent_t::type_id::create("axi4_master0", this );
38    axi4_master0.set_mvc_config(cfg.axi4_master0_cfg);
39
40    axi4_master1 = axi4_master1_agent_t::type_id::create("axi4_master1", this );
41    axi4_master1.set_mvc_config(cfg.axi4_master1_cfg);
42
43    axi4_master2 = axi4_master2_agent_t::type_id::create("axi4_master2", this );
44    axi4_master2.set_mvc_config(cfg.axi4_master2_cfg);
45
46    axi4_master3 = axi4_master3_agent_t::type_id::create("axi4_master3", this );
47    axi4_master3.set_mvc_config(cfg.axi4_master3_cfg);
48
49    axi4_master4 = axi4_master4_agent_t::type_id::create("axi4_master4", this );
50    axi4_master4.set_mvc_config(cfg.axi4_master4_cfg);
51
52    axi4_master5 = axi4_master5_agent_t::type_id::create("axi4_master5", this );
53    axi4_master5.set_mvc_config(cfg.axi4_master5_cfg);
54
55    axi4_slave0 = axi4_slave0_agent_t::type_id::create("axi4_slave0", this );
56    axi4_slave0.set_mvc_config(cfg.axi4_slave0_cfg);
57
58    axi4_slave1 = axi4_slave1_agent_t::type_id::create("axi4_slave1", this );
59    axi4_slave1.set_mvc_config(cfg.axi4_slave1_cfg);
60
61    axi4_slave2 = axi4_slave2_agent_t::type_id::create("axi4_slave2", this );
62    axi4_slave2.set_mvc_config(cfg.axi4_slave2_cfg);
63
64    axi4_slave3 = axi4_slave3_agent_t::type_id::create("axi4_slave3", this );
65    axi4_slave3.set_mvc_config(cfg.axi4_slave3_cfg);
66
67    axi4_slave4 = axi4_slave4_agent_t::type_id::create("axi4_slave4", this );
68    axi4_slave4.set_mvc_config(cfg.axi4_slave4_cfg);
69
70    axi4_slave5 = axi4_slave5_agent_t::type_id::create("axi4_slave5", this );
71    axi4_slave5.set_mvc_config(cfg.axi4_slave5_cfg);
72
73    axi4_slave6 = axi4_slave6_agent_t::type_id::create("axi4_slave6", this );
74    axi4_slave6.set_mvc_config(cfg.axi4_slave6_cfg);
75
76    axi4_slave7 = axi4_slave7_agent_t::type_id::create("axi4_slave7", this );
77    axi4_slave7.set_mvc_config(cfg.axi4_slave7_cfg);
78
79  endfunction: build_phase
```

Figura 68. Declaración de la fase build_phase() en el componente UVM Environment

5.2.5. Objeto UVM *Sequence*

Una vez finalizada la 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 se deriva de la clase `mvc_sequence` y esta, a su vez, de la clase `uvm_sequence_item`, que corresponde al nivel más bajo en la jerarquía UVM. Lo primero que hace este objeto es registrase en la *Factory*, tal y como se muestra en la línea 2 del código de la Figura 69. Esta secuencia UVM base no hace nada, es decir cuando ejecuta su tarea *body*, no se estimula ninguna señal, debido a que esta tarea está vacía. El motivo de la creación de esta secuencia base no es otro que proveer al ingeniero de verificación de una secuencia base a partir de la cual van a derivar sus propias secuencias para la estimulación de los agentes maestros y que contenga una referencia a los *sequencer* de cada agente, 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 69. Código de un objeto UVM Sequence base

5.2.6. Archivo de simulación

Por último, la herramienta de configuración QVIP también genera un archivo de simulación que se puede ejecutar en un terminal *Linux*. Es un archivo sencillo que se presenta en la Figura 70 y que ejecuta todos los comandos y paquetes que se referencian. 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.

```
1   echo "Setting up work directory"
2   vlib work
3
4   vlog ${QUESTA_MVC_HOME}/include/questa_mvc_svapi.svh \
5       +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv +define+MAP_PROT_ATTR
6   ${QUESTA_MVC_HOME}/questa_mvc_src/sv/mvc_pkg.sv \
7       +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv
8   +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4
9   ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/mgc_axi4_v1_0_pkg.sv \
10      +incdir+../config_policies ../config_policies/qvip_axi_xbar_params_pkg.sv \
11      qvip_axi_xbar_pkg.sv \
12      +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules
13  ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules/axi4_master.sv \
14      +incdir+${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules
15  ${QUESTA_MVC_HOME}/questa_mvc_src/sv/axi4/modules/axi4_slave.sv \
16
17  /home/users/PFC/divdsi/amflorido/TFG/axi-master/qvip/axi_xbar_wrap.sv \
18      default_clk_gen.sv \
19      default_reset_gen.sv \
20      hdl_qvip_axi_xbar.sv \
21      hvl_qvip_axi_xbar.sv
22
23  echo "Optimisation step"
24  vopt -o top_opt hdl_qvip_axi_xbar hvl_qvip_axi_xbar
25
26  echo "Running test"
27  vsim -c -mvchome ${QUESTA_MVC_HOME} top_opt +nowarnTSCALE -t 1ns -do
28  "run -all; quit" +UVM_TESTNAME=qvip_axi_xbar_test_base +SEQ=qvip_axi_xbar_vseq_base
```

Figura 70. Código del archivo de simulación questa_run

Dentro del archivo que se muestra en la Figura 70, se invocan los comandos *vlib*, *vlog*, *vopt* y *vsim*, propios del entorno *QuestaSim* de *Mentor Graphics*, *a Siemens Business*. Si el lector está interesado en profundizar en las diferentes opciones con las que se invocan estos comandos, puede consultarlos en [25].

Capítulo 6. Desarrollo del *testbench* UVM

Durante el capítulo 6 se procederá, en primer lugar, a la explicación de la biblioteca de estímulos básicos que proporciona la herramienta QVIP *configurator* para la elaboración de un *testbench* UVM rápido y efectivo. En segundo lugar, se presentará y explicará el código desarrollado durante el Trabajo Fin de Grado para la verificación del comportamiento del DUV, lo cual es el objetivo principal de este TFG. Para ello, se trabajará sobre el entorno de verificación UVM expuesto en el capítulo anterior, adaptando el código generado por el *Questa Verification* IP *configurator*.

6.1. Biblioteca de estímulos

Como se ha comentado en capítulos anteriores, resulta especialmente útil que la herramienta QVIP *configurator* incorpore una serie de estímulos básicos para facilitar la labor del ingeniero de verificación, ya que agiliza una etapa clave que consume mucho tiempo durante el desarrollo de cualquier *testbench* para un sistema *hardware* digital.

La Figura 71 representa en qué tareas invierten su tiempo los ingenieros de verificación en promedio. Por un lado, el 11% del tiempo se emplea en diseñar el Plan de Verificación, donde se construye una línea de trabajo para comprobar el correcto funcionamiento del dispositivo en cuestión. Por otro lado, para desarrollar el entorno de verificación y el *testbench* se utilizan, en promedio, el 20% y el 18% del tiempo,

respectivamente. Este 38% abarca tanto la corrección de los errores del código que describe el entorno de verificación como el del *testbench*, por lo que en estas etapas se resuelven los problemas detectados por el ingeniero de verificación. Cabe destacar que el 46% del tiempo se gasta en tareas de depuración (*debug*), las cuales se centran en la depuración del dispositivo en sí mismo. En este caso, se resuelven los errores derivados del código del DUV realizado por el ingeniero de diseño.

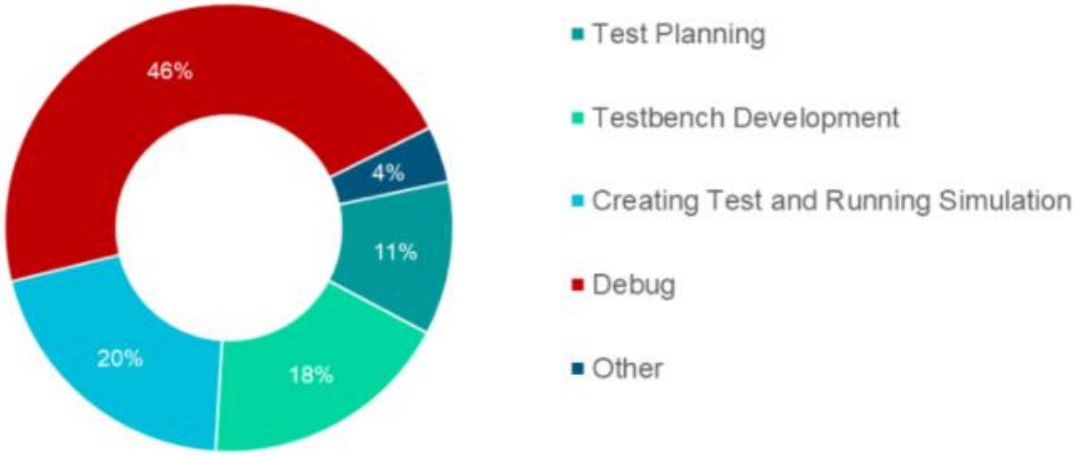

Figura 71. Proporción del tiempo invertido en tareas por los ingenieros de verificación

Por lo tanto, debido a las técnicas utilizadas a lo largo de este Trabajo Fin de Grado, se ha podido dedicar mayor tiempo a la etapa de depuración para encontrar posibles errores en el diseño del módulo *crossbar*.

6.1.1. Biblioteca de secuencias básicas

La biblioteca de secuencias aporta un conjunto de secuencias con las que se puede estimular el dispositivo a verificar. Son secuencias básicas de alcance limitado que abarcan una funcionalidad esencial, ya que permiten operaciones de lectura y escritura y configurar determinados parámetros de las distintas transacciones a enviar al DUV. Por lo tanto, resultan de gran utilidad para el ingeniero de verificación, que puede hacer uso completo de todas estas secuencias. Por esta razón, en este capítulo se expondrán brevemente los conceptos básicos asociados a estas secuencias. Las funciones y características de las secuencias que se encuentran en esta biblioteca se exponen en la Tabla 12.

Tabla 12. Biblioteca de secuencias de estímulos para el protocolo AXI4

Secuencia	Descripción
Atomic	Contiene secuencias que ejecutan transacciones de lectura y escritura con acceso exclusivo.
Burst	Contiene secuencias que ejecutan transacciones de lectura y escritura con distintos tipos de ráfagas. Se puede realizar una transacción tipo incremental, tipo *wrap* o tipo fijo.
Concurrent	Contiene secuencias que ejecutan transacciones de lectura y escritura concurrentes con el mismo o diferentes ID.
Delay	Contiene secuencias que ejecutan transacciones de lectura y escritura con diferentes retardos.

6.1.2. Interfaces API para la generación de estímulos

La librería de secuencias proporcionada por la herramienta QVIP *configurator* también incorpora una serie de API que permiten operaciones básicas de lectura y escritura de una manera sencilla y eficaz. Estas API no son dependientes del protocolo de comunicación utilizado en el entorno de verificación, es decir, están disponibles para todos los protocolos.

Con el uso de estas API, el ingeniero de verificación puede modificar diferentes atributos propios del protocolo AXI4, configurando características claves de las transacciones enviadas. Así, durante el proceso de verificación se pueden controlar los parámetros de los estímulos tales como el tamaño de las transacciones (parámetro *SIZE*), el tipo de ráfaga utilizada (parámetro *BURST*), el tipo de acceso al registro (parámetro *LOCK*), entre otros.

En la Tabla 13 se presentan las API disponibles para realizar operaciones, en este caso, de lectura. Se debe tener en cuenta el ancho del canal de los datos diseñados ya que, si se realiza una operación de lectura o de escritura de un número mayor de *bytes* de lo que soporta dicho canal, la operación se dividirá en tantas transacciones como sean necesarias.

Tabla 13. Interfaces API para la generación de transacciones de lectura

API	Descripción
read8	Operación de lectura de un dato de 8 *bits* en una dirección especificada.
read16	Operación de lectura de un dato de 16 *bits* en una dirección especificada.
read32	Operación de lectura de un dato de 32 *bits* en una dirección especificada.
read64	Operación de lectura de un dato de 64 *bits* en una dirección especificada.
read	Operación de lectura de un determinado número de *bytes* en una dirección especificada.

A continuación, en la Tabla 14 se adjuntan las API de escritura con su correspondiente descripción:

Tabla 14. Interfaces API para la generación de transacciones de escritura

API	Descripción
write8	Operación de escritura de un dato de 8 *bits* en una dirección especificada.
write16	Operación de escritura de un dato de 16 *bits* en una dirección especificada.
write32	Operación de escritura de un dato de 32 *bits* en una dirección especificada.
write64	Operación de escritura de un dato de 64 *bits* en una dirección especificada.
write	Operación de escritura de un *array* de longitud variable en una dirección especificada.

En caso de configurar una API con un campo de datos menor que el parámetro que se le pasa, sólo se enviará el número de *bits* que defina la API. De manera que si se realiza la siguiente operación de escritura: *write8('h1000, 32'h11223344)*, donde el primer campo corresponde a la dirección donde se realizará la escritura y el segundo parámetro hace referencia al campo de datos, en este caso un dato de 32 *bits*, la API utilizada sólo realizará la escritura de los 8 *bits* menos significativos (8'h44).

6.2. Código implementado

El módulo *crossbar*, objeto de verificación de este Trabajo Fin de Grado, es un componente de interconexión utilizado de manera habitual en diferentes protocolos de comunicación, cuyo principal propósito es la conexión simultánea de todos los puertos de entrada del mismo. Cuando un módulo QVIP maestro envía una transacción a través del *crossbar*, ésta llega a la interfaz de salida que esté asociado a la dirección de la transacción enviada.

En los sucesivos apartados se presentará el *testbench* UVM implementado. Se han utilizado los estímulos descritos en el primer apartado del presente capítulo, enviando estímulos controlados al módulo *wrapper* que, a su vez, está conectado en serie al módulo *crossbar*, cuyo comportamiento se desea verificar. De esta manera, cada uno de los módulos QVIP maestros estimulará las entradas del DUV. En la Figura 72 se muestra el código de la clase *qvip_axi_xbar_vseq_wr_rd*, la cual ejecutará todas las secuencias programadas. Cada uno de los maestros tiene asignado un tipo de secuencia, cuyas características se analizarán más adelante. Se ha utilizado una estructura *fork-join* que permite la creación de hilos que se ejecutarán de manera paralela.

En el código de la Figura 72 se muestra, entre las líneas 28 y 35, como se ejecuta cada una de las secuencias creadas sobre el componente UVM *Sequencer* correspondiente. Estas transacciones se enviarán de forma consecutiva. En este ejemplo, no se ha definido ningún tipo de restricciones ni prioridades. Sin embargo, una opción viable sería otorgar una serie de prioridades para gestionar el orden del envío de cada una de estas transacciones. En este caso no se ha elaborado este sistema de prioridades, por lo que será el propio entorno de verificación el que gestione dicho orden.

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

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

  extern task body;

endclass: qvip_axi_xbar_vseq_wr_rd

task qvip_axi_xbar_vseq_wr_rd::body;

  axi4_sequence_lib_seq_incr axi4_master0_seq_0  =  axi4_sequence_lib_seq_incr::type_id::create
                                                    ("axi4_sequence_lib_seq_incr");
  axi4_sequence_lib_seq_wrap axi4_master0_seq_1  =  axi4_sequence_lib_seq_wrap::type_id::create
                                                    ("axi4_sequence_lib_seq_wrap");
  axi4_sequence_lib_seq axi4_master0_seq_2       =  axi4_sequence_lib_seq::type_id::create
                                                    ("axi4_sequence_lib_seq");
  axi4_generic_api_excl_seq axi4_master0_seq_3   =  axi4_generic_api_excl_seq::type_id::create
                                                    ("axi4_generic_api_excl_seq");
  axi4_generic_api_normal_seq axi4_master0_seq_4 =  axi4_generic_api_normal_seq::type_id::create
                                                    ("axi4_generic_api_normal_seq");
  axi4_generic_api_wrap_seq axi4_master0_seq_5   =  axi4_generic_api_wrap_seq::type_id::create
                                                    ("axi4_generic_api_wrap_seq");

  begin
    fork
      axi4_master0_seq_0.start(axi4_master0);
      axi4_master0_seq_1.start(axi4_master1);
      axi4_master0_seq_2.start(axi4_master2);
      axi4_master0_seq_3.start(axi4_master3);
      axi4_master0_seq_4.start(axi4_master4);
      axi4_master0_seq_5.start(axi4_master5);
    join
  end
endtask: body
```

Figura 72. Clase qvip_axi_xbar_vseq_wr_rd

En lo que resta de capítulo se detallarán las características de cada una de las secuencias diseñadas a partir de la biblioteca de secuencias básicas y las API proporcionadas por la herramienta QVIP. En la Tabla 15 se especifican las transacciones que se ejecutarán a lo largo de la simulación del *testbench* UVM, comprobando algunas de las principales funciones del módulo *crossbar*.

Tabla 15. Lista de transacciones de cada módulo QVIP maestro

Fuente	Destino	Dirección de destino	Descripción de la transacción
Maestro 0	Esclavo 0	32'h0000_3500	Operación de lectura y escritura de la dirección especificada con el parámetro *BURST* en modo *INCR*.
Maestro 1	Esclavo 6	32'h6500_0000	Operación de lectura y escritura de la dirección especificada con el parámetro *BURST* en modo *WRAP*.
Maestro 2	Esclavo 2	32'h2500_0000	Operación de lectura y escritura de la dirección especificada con el parámetro *BURST* en modo *INCR*.
	Esclavo 4	32'h4500_4500	Operación de lectura y escritura de la dirección especificada con el parámetro *BURST* en modo *WRAP*.
Maestro 3	Esclavo 0	32'h1000_2000	Operación de lectura y escritura en modo exclusivo usando las API.
Maestro 4	Esclavo 7	32'h7000_2000	Operación de lectura y escritura en modo normal usando las API.
Maestro 5	Esclavo 3	32'h3000_1000	Operación de lectura y escritura con las API con el parámetro *BURST* en modo *WRAP*.
	Esclavo 5	32'h5000_5000	Operación de lectura y escritura con las API con el parámetro *BURST* en modo *WRAP*.

6.2.1. Secuencia *axi4_sequence_lib_seq_incr*

La secuencia *axi4_sequence_lib_seq_incr* es una secuencia básica que realiza una operación de lectura y escritura con el parámetro *BURST* configurado en modo incremental. Por lo tanto, una vez que se realiza una de las operaciones de lectura y escritura en memoria, la siguiente operación de lectura/escritura se realiza en la siguiente posición de memoria. El incremento de la dirección viene determinado por el tamaño del dato a escribir.

Como se muestra en la Figura 73 dentro de la tarea *body*, lo primero que se debe hacer es asignar el `m_sequencer` a la secuencia creada. Todas las secuencias están definidas con la capacidad de aleatorizar los principales parámetros del protocolo AXI4. El ingeniero de verificación puede escribir una serie de restricciones a la aleatorización de los atributos de una secuencia. De esta manera, se cuenta con un mayor control a la hora de configurar los estímulos. En este caso, la secuencia *axi4_sequence_lib_seq_incr* se ha aleatorizado con dos restricciones. La primera de estas restricciones es la dirección de

memoria a la que se desea acceder, tanto para la operación de lectura como de escritura. Esto se debe a que resulta interesante que, después de realizar la operación de escritura, se compruebe si la operación ha sido completada con éxito; dicha comprobación se efectúa haciendo una operación de lectura en la misma posición de memoria. La segunda restricción es el número de *bytes* que se va a enviar, lo que condiciona el número de transacciones que realizarán las operaciones de lectura y escritura. Mediante el mecanismo de mensajes UVM se notificará cualquier error en la aleatorización de los parámetros de las transacciones.

En este caso en particular, inicialmente se va a realizar una operación de lectura, la cual se espera que lea un dato o datos asignados por defecto en el modelo de memoria. Después, se realizará una operación de escritura en esas posiciones de memoria y, por último, se comprobará, realizando una última lectura, que los datos se han actualizado correctamente.

```
1   class axi4_sequence_lib_seq_incr extends axi4_master_deparam_seq;
2
3     `uvm_object_utils( axi4_sequence_lib_seq_incr )
4
5     typedef axi4_incr_wr_deparam_seq incr_wr_t;
6     typedef axi4_incr_rd_deparam_seq incr_rd_t;
7
8     function new (string name = "axi4_sequence_lib_seq_incr");
9       super.new( name );
10    endfunction
11
12    extern task body();
13  endclass
14
15  task axi4_sequence_lib_seq_incr::body();
16
17    incr_wr_t      incr_wr      =      incr_wr_t::type_id::create("incr_wr");
18    incr_rd_t      incr_rd      =      incr_rd_t::type_id::create("incr_rd");
19
20    super.body();
21
22    incr_rd.set_sequencer(m_sequencer);
23    incr_rd.id = 0;
24    if(!incr_rd.randomize() with {addr == 32'h0000_3500; rd_bytes == 16;})
25      `uvm_error(this.get_full_name(),"Randomisation failure");
26
27    incr_wr.set_sequencer(m_sequencer);
28    incr_wr.id = 0;
29    if(!incr_wr.randomize() with {addr == 32'h0000_3500; wr_data.size == 16;})
30      `uvm_error(this.get_full_name(),"Randomisation failure");
31
32    incr_rd.start(m_sequencer);
33    incr_wr.start(m_sequencer);
34    incr_rd.start(m_sequencer);
35
36  endtask
```

Figura 73. Secuencia UVM axi4_sequence_lib_seq_incr

6.2.2. Secuencia *axi4_sequence_lib_seq_wrap*

La secuencia *axi4_sequence_lib_seq_wrap* será enviada por el módulo QVIP maestro 1 con el fin de estimular el DUV y que la transacción llegue al módulo QVIP esclavo 6, tal y como se muestra en la Figura 74. Para enviar las transacciones de datos al esclavo número 6, la secuencia tiene que ser enviada a una dirección comprendida en el rango de memoria 32'6000_0000 y 32'6FFF_FFFF, que es el rango asociado a este esclavo. En este caso, se ha establecido la dirección 32'h6500_0000 y, por lo tanto, este parámetro no será aleatorizado por la clase padre de la secuencia en cuestión.

Como en el caso anterior, en primer lugar, se realizará una operación de lectura cuyo resultado será la lectura de datos no inicializados. Posteriormente se efectuará una operación de escritura y posterior lectura de las mismas posiciones de memoria. Todo esto con el parámetro *BURST* en modo *WRAP*.

```
1  class axi4_sequence_lib_seq_wrap extends axi4_master_deparam_seq;
2
3    `uvm_object_utils( axi4_sequence_lib_seq_wrap )
4
5    typedef axi4_wrap_wr_deparam_seq wrap_wr_t;
6    typedef axi4_wrap_rd_deparam_seq wrap_rd_t;
7
8    function new (string name = "axi4_sequence_lib_seq_wrap");
9      super.new( name );
10   endfunction
11
12   extern task body();
13 endclass
14
15 task axi4_sequence_lib_seq_wrap::body();
16
17   wrap_rd_t     wrap_rd     =     wrap_rd_t::type_id::create("wrap_rd");
18   wrap_wr_t     wrap_wr     =     wrap_wr_t::type_id::create("wrap_wr");
19
20   super.body();
21
22   wrap_rd.set_sequencer(m_sequencer);
23   wrap_rd.id = 1;
24   if(!wrap_rd.randomize() with {addr == 32'h6500_0000; rd_bytes == 16;})
25     `uvm_error(this.get_full_name(),"Randomisation failure");
26
27   wrap_wr.set_sequencer(m_sequencer);
28   wrap_wr.id = 1;
29   if(!wrap_wr.randomize() with {addr == 32'h6500_0000; wr_data.size == 16;})
30     `uvm_error(this.get_full_name(),"Randomisation failure");
31
32   wrap_rd.start(m_sequencer);
33   wrap_wr.start(m_sequencer);
34   wrap_rd.start(m_sequencer);
35
36 endtask
```

Figura 74. Secuencia UVM axi4_sequence_lib_seq_wrap

6.2.3. Secuencia *axi4_sequence_lib_seq*

El segundo módulo QVIP maestro llevará a cabo una transacción de escritura y lectura, cuyo destino será los módulos QVIP esclavos 2 y 4. De esta manera, se pondrá a prueba el módulo *crossbar* y se comprobará si un maestro puede estimular de forma consecutiva dos esclavos diferentes. Las operaciones se realizarán en modo incremental para uno de los esclavos, y en modo *WRAP* para el otro, tal y como se describe en la Figura 75. Las líneas 27 a 35 del código presentado, establecen los parámetros de las secuencias de lectura y escritura tipo *INCR* que será enviada al esclavo 2, mientras que las líneas 40 a 48 establecen los parámetros de las secuencias de lectura y escritura tipo *WRAP* a enviar al esclavo 4.

```
1  class axi4_sequence_lib_seq extends axi4_master_deparam_seq;
2
3    `uvm_object_utils( axi4_sequence_lib_seq )
4
5    typedef axi4_incr_wr_deparam_seq incr_wr_t;
6    typedef axi4_incr_rd_deparam_seq incr_rd_t;
7    typedef axi4_wrap_wr_deparam_seq wrap_wr_t;
8    typedef axi4_wrap_rd_deparam_seq wrap_rd_t;
9    typedef axi4_excl_rw_deparam_seq excl_rw_t;
10
11   function new (string name = "axi4_sequence_lib_seq");
12     super.new( name );
13   endfunction
14
15   extern task body();
16 endclass
17
18 task axi4_sequence_lib_seq::body();
19
20   incr_wr_t     incr_wr     =     incr_wr_t::type_id::create("incr_wr");
21   incr_rd_t     incr_rd     =     incr_rd_t::type_id::create("incr_rd");
22   wrap_rd_t     wrap_rd     =     wrap_rd_t::type_id::create("wrap_rd");
23   wrap_wr_t     wrap_wr     =     wrap_wr_t::type_id::create("wrap_wr");
24
25   super.body();
26
27   incr_rd.set_sequencer(m_sequencer);
28   incr_rd.id = 2;
29   if(!incr_rd.randomize() with {addr == 32'h2500_0000; rd_bytes == 16;})
30     `uvm_error(this.get_full_name(),"Randomisation failure");
31
32   incr_wr.set_sequencer(m_sequencer);
33   incr_wr.id = 2;
34   if(!incr_wr.randomize() with {addr == 32'h2500_0000; wr_data.size == 16;})
35     `uvm_error(this.get_full_name(),"Randomisation failure");
36
37   incr_wr.start(m_sequencer);
38   incr_rd.start(m_sequencer);
39
40   wrap_rd.set_sequencer(m_sequencer);
41   wrap_rd.id = 2;
42   if(!wrap_rd.randomize() with {addr == 32'h4500_0000; rd_bytes == 16;})
43     `uvm_error(this.get_full_name(),"Randomisation failure");
44
45   wrap_wr.set_sequencer(m_sequencer);
46   wrap_wr.id = 2;
47   if(!wrap_wr.randomize() with {addr == 32'h4500_0000; wr_data.size == 16;})
48     `uvm_error(this.get_full_name(),"Randomisation failure");
49
50   wrap_wr.start(m_sequencer);
51   wrap_rd.start(m_sequencer);
52
53 endtask
```

Figura 75. Secuencia UVM axi4_sequence_lib_seq

6.2.4. Secuencia *axi4_generic_ api_excl_seq*

La secuencia *axi4_generic_ api_excl_seq*, mostrada en la Figura 76, hace uso de las API anteriormente descritas y, mediante ellas, realizará un acceso tipo exclusivo en una región de memoria seleccionada por la dirección de memoria especificada en las API. En este caso el módulo QVIP maestro enviará una transacción de lectura y escritura con el tamaño de dato de 8 *bits.* Para ejecutar la secuencia, en primer lugar, se especifica el tipo de acceso mediante la API *set_lock*, cuyo primer parámetro indica el tipo *AXI4_EXCLUSIVE*, asignado a las transacciones con *ID* 3. Posteriormente, mediante el uso de las API *read8*, *write8* y *read8* se analizan las operaciones de lectura y escritura correspondientes. En este caso, el dato enviado tiene valor 8'h11 definido en la línea 15, y dicho valor será el esperado en la segunda operación de lectura.

```
1   import rw_txn_pkg::*;
2
3   class axi4_generic_api_excl_seq extends axi4_master_deparam_seq;
4     `uvm_object_utils(axi4_generic_api_excl_seq)
5
6     function new(string name="");
7       super.new(name);
8     endfunction
9
10    virtual task body();
11
12      byte unsigned rd_data[];
13      byte unsigned wr_data[] = new[16];
14
15      byte unsigned    wr_data_8  = 'h11;
16      shortint wr_data_16 = 'h1122;
17      int      wr_data_32 = 'h11223344;
18      longint  wr_data_64 = 64'h1122334455667788;
19
20      byte unsigned    rd_data_8;
21      shortint rd_data_16;
22      int      rd_data_32;
23      longint  rd_data_64;
24
25      super.body();
26
27      wr_data = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
28
29      set_lock(AXI4_EXCLUSIVE, 3);
30      read8('h1000_2000, rd_data_8, 3);
31      write8('h1000_2000, wr_data_8, 3);
32      read8('h1000_2000, rd_data_8, 3);
33
34    endtask
35
36  endclass
```

Figura 76. Secuencia UVM axi4_generic_api_excl_seq

6.2.5. Secuencia *axi4_generic_ api_normal_seq*

La secuencia *axi4_generic_ api_normal_seq* se presenta en la Figura 77. Esta secuencia es muy parecida a la anterior, tal y como se deduce de su normbre El principal cambio que se introduce es que se trata de una operación de acceso normal, no exclusivo. Además, se utiliza la API *write32*, por lo que se transferirán datos de 32 *bits*. Este dato a enviar se define en la línea 17 del código de la Figura 77 y dicho dato se envía a la dirección 32'h7000_2000, que se encuentra dentro del rango de memoria del módulo QVIP esclavo 7. Para especificar el tipo de acceso, se ha utilizado la API *set_lock*, estableciendo un tipo AXI4_NORMAL para las transacciones con identificador 4.

```
1   import rw_txn_pkg::*;
2
3   class axi4_generic_api_normal_seq extends axi4_master_deparam_seq;
4     `uvm_object_utils(axi4_generic_api_normal_seq)
5
6     function new(string name="");
7       super.new(name);
8     endfunction
9
10    virtual task body();
11
12      byte unsigned rd_data[];
13      byte unsigned wr_data[] = new[16];
14
15      byte unsigned wr_data_8  = 8'h11;
16      shortint      wr_data_16 = 16'h1122;
17      int           wr_data_32 = 32'h11223344;
18      longint       wr_data_64 = 64'h1122334455667788;
19
20      byte unsigned rd_data_8;
21      shortint      rd_data_16;
22      int           rd_data_32;
23      longint       rd_data_64;
24
25      super.body();
26
27      wr_data = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
28
29      set_lock(AXI4_NORMAL, 4);
30      write32('h7000_2000, wr_data_32, 4);
31      read32('h7000_2000, rd_data_32, 4);
32
33    endtask
34
35  endclass
```

Figura 77. Secuencia UVM axi4_generic_api_normal_seq

6.2.6. Secuencia *axi4_generic_ api_wrap_seq*

La Figura 78 hace referencia a la secuencia *axi4_generic_ api_wrap_seq* que será la secuencia enviada por el módulo QVIP maestro 5. Esta secuencia enviará una transacción de escritura y lectura a los esclavos 3 y 5, configuradas de manera diferente. En el primer caso, las líneas 23 a 26, se configuran las transacciones a enviar al esclavo 3. Se especifica el campo de datos a enviar, definido como 64'h1122334455667788. El parámetro *BURST* de esta transacción se configura en modo *WRAP* y el parámetro *SIZE* con 8 *bytes* de transferencia.

En segundo lugar, en las líneas 28 a 30 de la Figura 78, se configuran las transacciones a enviar al esclavo 5. Estas transacciones incluyen una operación de lectura y otra de escritura. Sin embargo, y a diferencia de las transacciones transferidas al esclavo 3, en este caso, se usan las API *write* y *read*. Estas API utilizan como dato de entrada un *array* de *bytes*. En esta caso, el tipo de acceso se configura en modo incremental, AXI4_INCR, mediante la API *set_burst_type*. Estableciendo la dirección 32'h5000_5000 se envían los datos al esclavo 5.

```
1  import rw_txn_pkg::*;
2
3  class axi4_generic_api_wrap_seq extends axi4_master_deparam_seq;
4    `uvm_object_utils(axi4_generic_api_wrap_seq)
5
6    function new(string name="");
7      super.new(name);
8    endfunction
9
10   virtual task body();
11
12     byte unsigned rd_data[];
13     byte unsigned wr_data[] = new[16];
14
15     longint  wr_data_64 = 64'h1122334455667788;
16
17     longint  rd_data_64;
18
19     super.body();
20
21     wr_data = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
22
23     set_burst_type(AXI4_WRAP, 5);
24     set_size(AXI4_BYTES_8, 5);
25     write64('h3000_1000, wr_data_64, 5);
26     read64('h3000_1000, rd_data_64, 5);
27
28     set_burst_type(AXI4_INCR, 5);
29     write('h5000_5000, wr_data, 5);
30     read('h5000_5000, rd_data, 5);
31
32   endtask
33
34 endclass
```

Figura 78. Secuencia UVM axi4_generic_api_wrap_seq

Capítulo 7. Resultados

Después de describir los fundamentos teóricos utilizados durante la realización del presente Trabajo Fin de Grado, detallar el funcionamiento de la herramienta de configuración QVIP y presentar el código implementado, a lo largo de este capítulo se mostrarán los resultados obtenidos de la simulación del *testbench* UVM diseñado para la comprobación del funcionamiento del componente de interconexión *crossbar* [20].

7.1. Topología resultante

El primer aspecto que se analizará es la topología del entorno de verificación UVM creado a partir de la herramienta QVIP *configurator*. Como se comentó en el capítulo 3, la metodología UVM facilita una serie de mecanismos de impresión por pantalla de ciertas características y detalles de los objetos y componentes que conforman el entorno de verificación UVM. En este caso, se ha utilizado el método *print()*, que es heredado de la clase `uvm_object`, junto a otros métodos básicos como el *copy()* o el *clone()*. En la Figura 79 se muestra la topología del entorno de verificación UVM diseñado. Al analizar la figura mencionada, la cual muestra únicamente la topología con los dos primeros maestros, se puede comprobar que los resultados obtenidos son satisfactorios.

Para poder imprimir la topología, es necesario que haya finalizado la fase de conexionado. En este caso, mediante la utilización de la fase

`end_of_elaboration_phase()`, se obtiene un alto nivel de detalle de la jerarquía UVM, ya que al ejecutarse esta fase, ya se han creado y conectado todos los componentes que forman parte del entorno de verificación. De manera indirecta, todos los componentes UVM declarados heredan de las clases definidas en la biblioteca de UVM, lo que significa que existe una gran cantidad de código, *a priori* "invisible" para el ingeniero de verificación, que se está ejecutando en todo momento. Este código es imprescindible para el correcto funcionamiento del entorno de verificación UVM. Por la razón expuesta, se crean de forma automática diversos elementos que durante el desarrollo de este Trabajo Fin de Grado no se han creado, como pueden ser el componente UVM *Checker* dentro de todos los módulos QVIP maestros y esclavos.

Si se analiza con más atención la Figura 79, el nivel más alto de la jerarquía lo ocupa el componente UVM *Test*. Durante el desarrollo de este TFG, en ningún momento se ha definido el elemento `uvm_test_top`. Como se comentó anteriormente esto es un claro ejemplo del código heredado de la biblioteca de UVM. El siguiente nivel de la jerarquía es el componente UVM *Environment*, el cual declara los diferentes módulos QVIP maestros y esclavos. Debido a la gran extensión de la topología del sistema, en la Figura 79 solo se presenta la topología parcial del mismo. Todos los módulos QVIP se comportan como componentes UVM *Agent*, en cuyo interior cuentan con diferentes componentes que lo capacitan para la estimulación del dispositivo a verificar.

```
# ----------------------------------------------------------------------
# Name                      Type                          Size  Value
# ----------------------------------------------------------------------
# uvm_test_top              qvip_axi_xbar_test_base       -     @501
#   env                     qvip_axi_xbar_env             -     @622
#     axi4_master0          axi4_agent #(32,64,64,4,5,16) -     @629
#       checker             uvm_component                 -     @909
#         analysis_imp      uvm_analysis_imp              -     @916
#       driver              mvc_driver                    -     @863
#         rsp_port          uvm_analysis_port             -     @878
#         seq_item_port     uvm_seq_item_pull_port        -     @870
#       monitor             mvc_monitor                   -     @894
#         trans_ap          uvm_analysis_port             -     @928
#       sequencer           mvc_sequencer                 -     @754
#         rsp_export        uvm_analysis_export           -     @761
#         seq_item_export   uvm_seq_item_pull_imp         -     @855
#         arbitration_queue array                         0     -
#         lock_queue        array                         0     -
#         num_last_reqs     integral                      32    'd1
#         num_last_rsps     integral                      32    'd1
#       sequencer_export    implementation                -     @886
#       trans_ap            uvm_analysis_port             -     @901
#     axi4_master1          axi4_agent #(32,64,64,4,5,16) -     @640
#       checker             uvm_component                 -     @1107
#         analysis_imp      uvm_analysis_imp              -     @1114
#       driver              mvc_driver                    -     @1061
#         rsp_port          uvm_analysis_port             -     @1076
#         seq_item_port     uvm_seq_item_pull_port        -     @1068
#       monitor             mvc_monitor                   -     @1092
#         trans_ap          uvm_analysis_port             -     @1126
#       sequencer           mvc_sequencer                 -     @952
#         rsp_export        uvm_analysis_export           -     @959
#         seq_item_export   uvm_seq_item_pull_imp         -     @1053
#         arbitration_queue array                         0     -
#         lock_queue        array                         0     -
#         num_last_reqs     integral                      32    'd1
#         num_last_rsps     integral                      32    'd1
#       sequencer_export    implementation                -     @1084
#       trans_ap            uvm_analysis_port             -     @1099
```

Figura 79. Topología resultante del entorno de verificación UVM

7.2. Simulación de las secuencias

Con el propósito de estimular el módulo *crossbar* se han diseñado diferentes *test*. Cada uno de estos *test* se ha descrito en una secuencia, es decir, habrá tantas secuencias como *test* a realizar. Se ha utilizado el componente UVM *Test* base para arrancar todas las secuencias. Esto se debe a la necesidad de ejecutar una secuencia a través de un componente UVM *Agent* y, a su vez, esta debe arrancar desde un *test*. Dentro del *test*, una secuencia se puede ejecutar únicamente dentro de la fase `run_phase()`. Por este motivo se analizará con detalle la fase `run_phase()` mostrada en la Figura 64 del capítulo 5.

7.2.1. Fase de ejecución del *test* base

Existen dos maneras de ejecutar una serie de *test* para la estimulación de un DUV. La primera forma es mediante la definición de una clase diferente para cada *test.* Dentro de la fase `run_phase()` asociado a cada *test*, se ejecuta la secuencia que corresponde. Esta manera tiene como principal ventaja la simplicidad en la descripción de la fase `run_phase()`. Un ejemplo de la definición de la fase `run_phase()` siguiendo esta filosofía se presenta en la Figura 80. Tal y como se observa en la mencionada figura, la fase `run_phase()` referencia una secuencia, en este caso, de tipo *random_sequence.* Posteriormente se crea la secuencia con el método *create()* y, por último, se comienza la secuencia mediante el método *start().* Esta manera tan simplificada de ejecutar un *test* lleva asociada la definición de un nuevo *test* si se desea cambiar la secuencia a ejecutar. Cuando se selecciona otro *test* para su correspondiente ejecución, no es necesaria la compilación de todo el entorno de verificación UVM, simplemente se especifica el nombre del *test* a ejecutar cuando se invoca el simulador, mediante la opción *+TESTNAME=<test_name>* utilizada al invocar a la herramienta de simulación. En el caso expuesto en la Figura 80, bastaría con la sustitución del *test_name* por el nombre del *test* que se desea ejecutar.

```
virtual task run_phase(uvm_phase phase);
   random_sequence seq;

   super.run_phase(phase);
   phase.raise_objection(this);
   seq = random_sequence::type_id::create("seq");
   seq.start(env.penv_in.agent.sequencer);
   phase.drop_objection(this);
endtask: run_phase
```

Figura 80. Ejemplo de implementación de la fase run_phase()

La segunda manera de ejecutar un conjunto de *test* es la utilizada por la herramienta QVIP *configurator*. Esta herramienta crea un *test* genérico, por lo que se abre la posibilidad

a que un *test* ejecute una pila de secuencias. Dichas secuencias se ejecutarán de forma consecutiva y en el orden que especifique el ingeniero de verificación. Para profundizar y entender este concepto se analizará la Figura 81, que implementa el código de la fase `run_phase()`.

```
190  task qvip_axi_xbar_test_base::run_phase(uvm_phase phase);
191
192    string sequence_name;
193    qvip_axi_xbar_vseq_wr_rd vseq;
194    uvm_object obj;
195    uvm_cmdline_processor clp;
196    clp = uvm_cmdline_processor::get_inst();
197
198    if(clp.get_arg_value("+SEQ=",sequence_name) == 0)
199      begin
200        `uvm_fatal(get_type_name(), "You must specify a virtual sequence to run using
201                                     the +SEQ plusarg")
202      end
203    obj = factory.create_object_by_name(sequence_name);
204    if (obj == null)
205      begin
206        factory.print();
207        `uvm_fatal(get_type_name(),{"Virtual sequence'",sequence_name,"'not found in factory"})
208      end
209
210    if (!$cast(vseq, obj) )
211      begin
212        `uvm_fatal(get_type_name(),{"Virtual sequence'",sequence_name,"'is not derived
213                                     from qvip_axi_xbar_vseq_wr_rd"})
214      end
215
216    `uvm_info(get_type_name(),{"Running virtual sequence'",sequence_name,"'"},UVM_LOW)
217
218    phase.raise_objection(this);
219    init_vseq(vseq);
220    vseq.start(null);
221    phase.drop_objection(this);
222
223  endtask: run_phase
```

Figura 81. Tarea run_phase() del componente UVM Test

Al abrirse la posibilidad de ejecutar una serie de secuencias con un único *test*, facilita la simulación de un conjunto de secuencias de una manera eficaz. Cuando se invoca el simulador, solo se debe añadir la opción *+SEQ=<seq_name>*, donde la etiqueta *seq_name* se debe sustituir por el nombre de la secuencia a ejecutar. La opción *+SEQ=<seq_name>* se puede utilizar tantas veces como secuencias se quieran ejecutar. Como se observa en las líneas 198 a 203 del código de la Figura 81, se crea en la *Factory* el objeto que se corresponde con el nombre de la secuencia introducida. El código de las líneas 204 a 214 hace uso del mecanismo de mensajes UVM, por lo que se notificará en pantalla un error si el objeto creado es *null* o no se corresponde con un objeto tipo

uvm_sequence. Por último, en las líneas 219 a 220 se inicializa la secuencia y comienza la ejecución de la misma.

Para el arbitraje de las transacciones se puede diseñar un sistema de prioridades, asignándole a cada una de las secuencias una prioridad mayor o menor respecto a las demás. En el caso particular del presente Trabajo Fin de Grado, no se ha definido dicho sistema de prioridades. Por lo tanto, el sistema internamente será el encargado de decidir el orden de ejecución de las secuencias.

7.2.2. Secuencias descritas y ejecutadas en el proceso de verificación

Durante el proceso de verificación realizado a lo largo del Trabajo Fin de Grado, se ha desarrollado una serie de secuencias mediante la biblioteca de secuencias y API proporcionada por la herramienta QVIP, como se explicó en el capítulo 6. Esta pila de secuencias se ejecutará con una estructura *fork-join*, por lo que las secuencias descritas dentro de esta construcción se ejecutarán de manera paralela.

En este Trabajo Fin de Grado, no es una prioridad diseñar un *testbench* demasiado complejo que incluya una comprobación exhaustiva del componente de interconexión *crossbar*. Esto se debe al tiempo limitado asociado a la realización de este TFG y al tiempo invertido en el desarrollo del entorno de verificación UVM y la herramienta QVIP *configurator*, incluyendo el estudio teórico que conlleva.

En lo que resta de capítulo, se expondrán detalladamente los resultados obtenidos mediante la ejecución del *testbench* UVM diseñado, verificando el funcionamiento del componente de interconexión *crossbar*.

7.2.2.1. Secuencia *axi4_sequence_lib_seq_incr*

Al realizar la simulación del *test*, se ha ejecutado la secuencia *axi4_sequence_lib_seq_incr*. En primer lugar, se analizarán las señales del protocolo de comunicación AXI4 que gestionan el protocolo de *handshake* de los canales de los módulos maestros y esclavos. En segundo lugar, se estudiará si las operaciones de lectura y escritura se han completado de manera satisfactoria.

En este caso, esta secuencia ha sido asignada al módulo QVIP maestro 0, que se comunicará con el módulo QVIP esclavo 0. Tal y como se muestra en la Figura 82, en total, se realizan tres operaciones: lectura, escritura y lectura. Las señales *READY* siempre se encuentran a nivel alto, tanto en el maestro como en el esclavo. Por lo tanto, no se ejecutará

ninguna transacción de lectura hasta que se no valide la señal *RVALID*, pero esta señal cuenta con una dependencia directa hacia la señal *ARVALID*, la cual valida el envío de la dirección de memoria en la que se desea ejecutar la operación de lectura.

A su vez, el esclavo debe aceptar la dirección de lectura enviada por el maestro y pasar a nivel alto la señal *RREADY* como respuesta a la señal *RVALID*. Estas circunstancias se cumplen en las marcas 1 y 2 de la Figura 82. Por lo tanto, en ese instante se realizará la operación de lectura que recogerá los datos empaquetados en el campo *RDATA*. Como previamente no se ha realizado ninguna operación de escritura en ese registro de memoria, lo esperado es que devuelva un dato igual a 0 o un valor aleatorio escrito por defecto. Como se muestra en la marca 1 de la Figura 83, esta suposición se cumple al comprobar el valor del campo de datos *RDATA* del módulo maestro.

En el caso de la operación de escritura, el protocolo de *handshake* sigue los mismos pasos que la operación de lectura. Una vez que las señales *VALID* y *READY* están a nivel alto (marcas 3 y 4 de la Figura 82), se envían dos datos de escritura aleatorios con los siguientes valores: 64'h3868d86402962fa y 64'h1dec55ad94169ca8, como se observa en las marcas 2 y 3 de la Figura 83. Esta escritura se realizará en la dirección 32'h0000_3500, marca 6 de la mencionada figura.

Por último, se ejecuta una operación de lectura con el fin de comprobar si las anteriores transacciones de escritura se completaron con éxito. Al analizar las marcas 4 y 5 de la Figura 83, se corrobora que así es, ya que devuelve exactamente los valores escritos previamente en esa misma dirección.

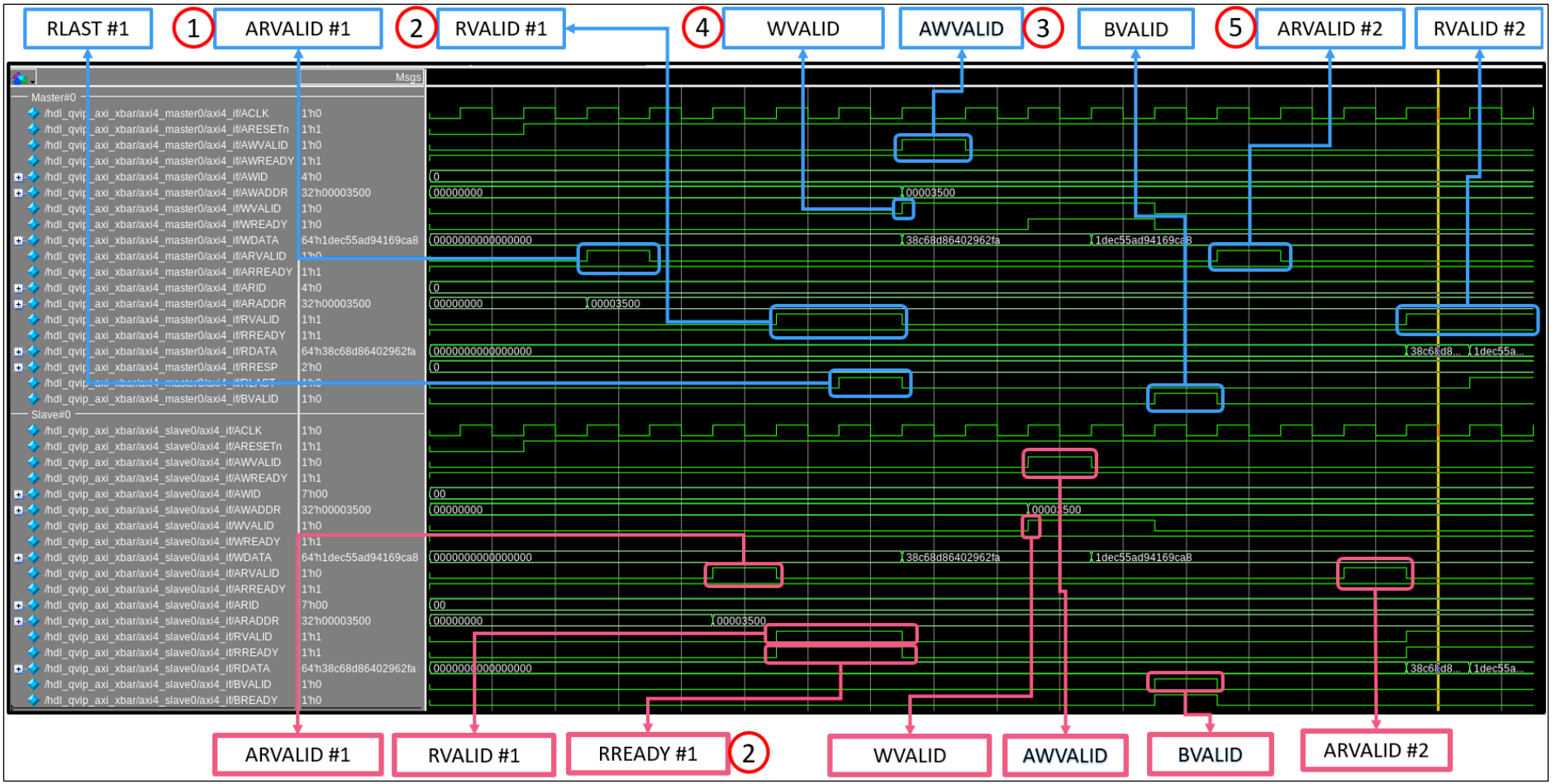

Figura 82. Señales handshake de la secuencia ejecutada por el maestro 0

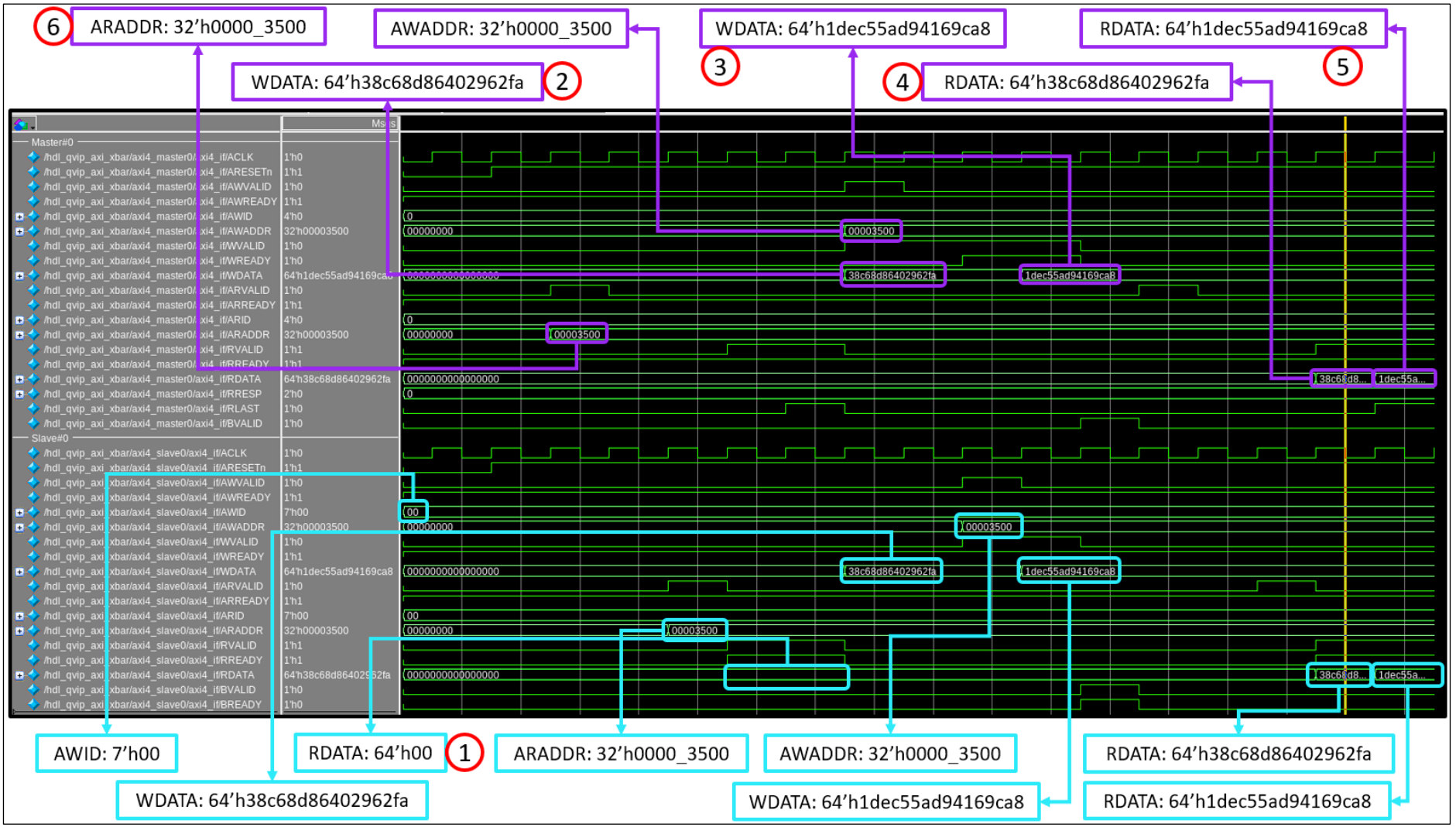

Figura 83. Operaciones de lectura y escritura del maestro 0 en el esclavo 0

7.2.2.2. Secuencia *axi4_sequence_lib_seq_wrap*

A la hora de estimular el DUV con una secuencia, es imprescindible que en todos los casos se cumpla el protocolo de *handshake* del protocolo de comunicación AXI4. Cuando se ejecuta la secuencia *axi4_sequence_lib_seq_wrap*, el *handshake* de las señales *READY* y *VALID* se cumple de manera similar al caso anterior y para todos los canales, tal y como muestra la Figura 84.

Como el módulo *crossbar* es un componente de interconexión cuyo propósito es la conexión simultánea de todos sus puertos, es crucial la comprobación de la comunicación transversal entre los puertos. Para ello, una de las cosas que comprobará esta secuencia es la comunicación de puertos no alineados. Por esta razón esta secuencia es ejecutada por el segundo módulo QVIP maestro y su destino es el sexto módulo QVIP esclavo.

Los resultados más relevantes de la simulación de dicha secuencia están recogidos en la Tabla 16, donde cabe destacar la codificación del parámetro *BURST* que, como está en modo *WRAP*, se codifica con el valor 0x10. Si en cada transacción de la secuencia *axi4_sequence_lib_seq_wrap* no se definiese la dirección en la que se va a acceder, la operación de escritura enviaría el campo de datos a la dirección 32'h6500_0004 debido a que el parámetro *BURST* se encuentra en el modo *WRAP*. En la tercera transacción se leería la dirección 32'h6500_0008 que, en ese caso, también estaría vacía. Pero como se puede observar en el código de la Figura 74, que describe la estructura de esta secuencia, en todos los casos se define la dirección de lectura y escritura, por lo tanto, siempre se accederá a la misma dirección, tal y como se muestra en la Figura 84.

Además, el identificador de las transacciones enviadas se define, en este caso, con el valor 1 pero, por el diseño del módulo *crossbar*, el identificador del esclavo tiene 7 *bits*, donde los 4 *bits* menos significativos representan el identificador de la transacción y los *bits* restantes hacen referencia a la fuente de dichas transacciones, en este caso, el módulo QVIP maestro 1. Por esta razón, el módulo esclavo redefine el parámetro *ID* con el valor 7'h11.

Tabla 16. Resultados obtenidos en la secuencia ejecutada por el maestro 1

Tipo de operación	Señales del maestro 1	Datos obtenidos	Señales del esclavo 6	Datos obtenidos
Primera transacción de lectura	ARADDR	32'h6500_0000	ARADDR	32'h6500_0000
	ARBURST	2'h2	ARBURST	2'h2
	ARID	4'h1	ARID	7'h11
	RDATA	64'h0	RDATA	64'h0
Transacción de escritura	AWADDR	32'h6500_0000	AWADDR	32'h6500_0000
	AWBURST	2'h2	AWBURST	2'h2
	AWID	4'h1	AWID	2'h2
	WDATA	64'h7F30_C55F_0E7E_1E39	WDATA	64'h6EC3_9DF5_D74A_83A6
Segunda transacción de lectura	ARADDR	32'h6500_0000	ARADDR	32'h6500_0000
	ARBURST	2'h2	ARBURST	2'h2
	ARID	4'h1	ARID	7'h11
	RDATA	64'h7F30_C55F_0E7E_1E39	RDATA	64'h6EC3_9DF5_D74A_83A6

Como se observa en la Figura 84, la primera operación de lectura lee un valor nulo en la dirección 32'h6500_0000 (marca 1 de la mencionada figura), debido a que todavía no se ha escrito nada en dicho registro. La operación de escritura se efectúa en las marcas 2 y 3 y, una vez finalizada, se lleva a cabo otra operación de lectura en la misma dirección que, en esta ocasión, devuelve el valor anteriormente escrito, como se observa en las marcas 4 y 5 de la Figura 84. Como los valores de escritura y lectura coinciden se puede afirmar que la secuencia se ha ejecutado con éxito.

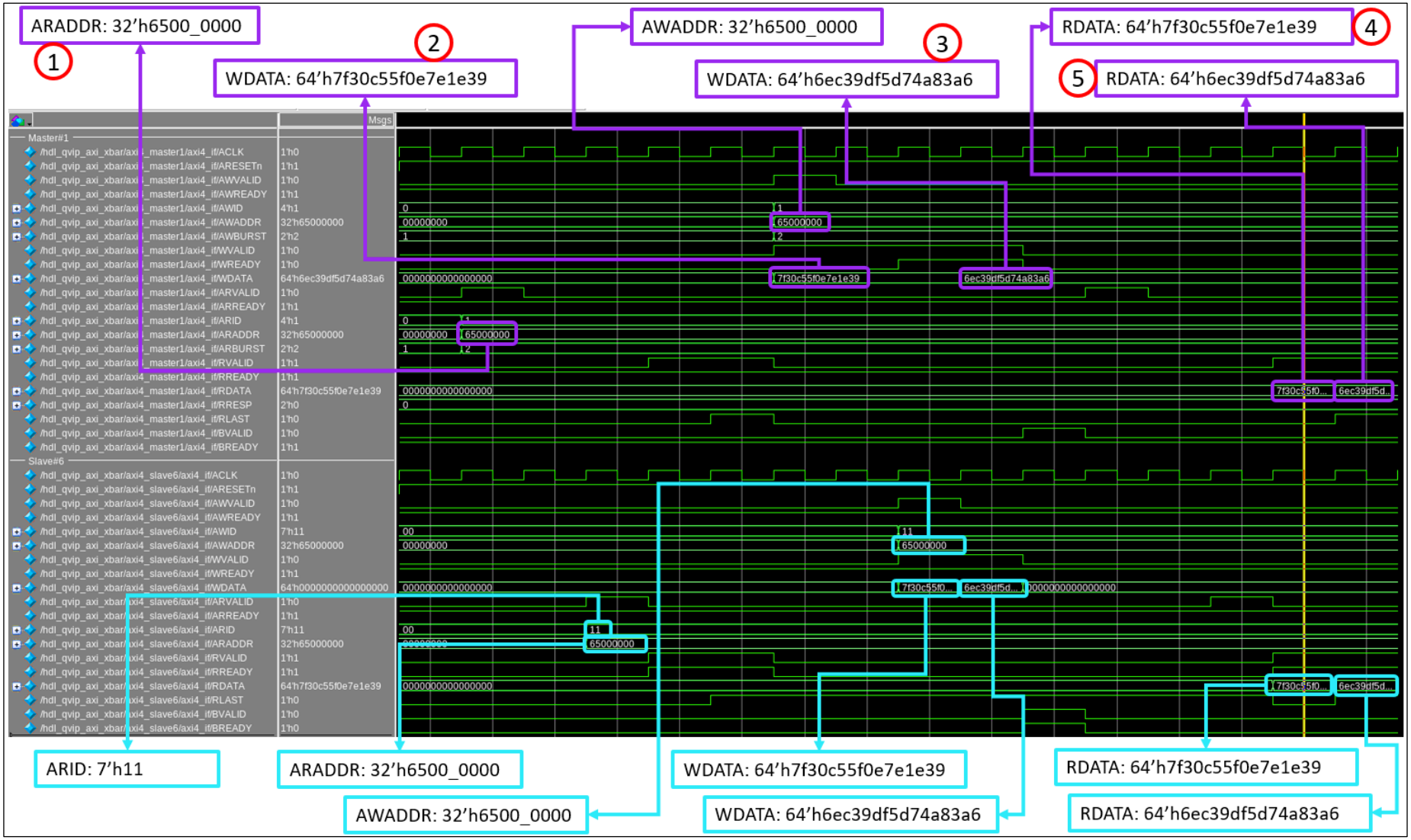

Figura 84. Operaciones de lectura y escritura del maestro 1 en el esclavo 6

7.2.2.3. *Secuencia axi4_sequence_lib_seq*

Cuenta con especial relevancia la comprobación de si un maestro puede enviar varias transacciones a dos esclavos de manera consecutiva, lo cual mostraría el correcto funcionamiento en la gestión de los límites de las direcciones asignadas a un esclavo. Para comprobarlo, primero se enviará una operación de escritura en una dirección de la región del mapa de direcciones del esclavo 2. Seguidamente se leerá dicho registro mediante una operación de lectura. Después, se seguirá exactamente el mismo procedimiento descrito pero en la dirección 32'h4500_0000, correspondiente al esclavo 4.

Tal y como se muestra en la Figura 85, el protocolo *handshake* de las señales *VALID* y *READY* se cumple en todas las transacciones, ya que sólo se envía la dirección y el campo de datos cuando las señales *VALID* y *READY* correspondientes están activas a nivel alto. En primer lugar, se ejecutarán las transacciones de escritura y lectura cuyo destino es el maestro 2. Esto se debe a que a la hora de describir el código de la secuencia *axi4_sequence_lib_seq*, primero se ejecuta la función *start()* que arrancará esta primera transacción. En segundo lugar, se ejecutarán las transacciones de escritura y lectura que accederán a la dirección que se encuentra en la región de memoria del esclavo 4.

El análisis de esta secuencia se ha dividido en dos partes. La primera, representada en la Figura 86, que refleja las operaciones de escritura que se realizan en ambos esclavos. La segunda, mostrada en la Figura 87, que presenta las operaciones de lectura que se ejecutan en la correspondiente dirección donde previamente se ha escrito el dato aleatorio.

En la Tabla 17 se resumen los principales resultados que se obtienen al ejecutar las transacciones que llegan al módulo QVIP esclavo 2, donde el maestro define un parámetro *ID* igual a 4'h2 por lo que el esclavo lo redefinirá con un valor 7'h22, ya que el maestro que ha enviado la transacción es el maestro número 2.

En las marcas 1 y 2 de la Figura 86 se encuentran los valores aleatorios de escritura que se envían al esclavo 2 y, efectivamente, después de atravesar el módulo *crossbar,* a ese esclavo le llegan esos mismos valores como se observa en las marcas 5 y 6 de esa misma figura. Al leer la misma dirección en la transacción de lectura, se obtiene el valor guardado en esa dirección de memoria, como se visualizan en las marcas 2 y 3 de la Figura 87. Al comparar el dato escrito con el dato leído en la correspondiente dirección del esclavo, se confirma el éxito en la operación realizada.

Tabla 17. Resultados obtenidos de la secuencia enviada por el maestro 2 al esclavo 2

Tipo de operación	Señales del maestro 2	Datos obtenidos	Señales del esclavo 2	Datos obtenidos
Transacción de escritura	AWADDR	32'h2500_0000	AWADDR	32'h2500_0000
	AWBURST	2'h1	AWBURST	2'h1
	AWID	4'h2	AWID	7'h22
	WDATA	64'h38C6_8D86_4029_62FA	WDATA	64'h1DEC_55AD_9416_9CA8
Segunda transacción de lectura	ARADDR	32'h2500_0000	ARADDR	32'h2500_0000
	ARBURST	2'h1	ARBURST	2'h1
	ARID	4'h2	ARID	7'h22
	RDATA	64'h38C6_8D86_4029_62FA	RDATA	64'h1DEC_55AD_9416_9CA8

La operación que se llevará a cabo en el esclavo 4 es prácticamente igual a la anterior. La diferencia radica en que la primera operación se ejecuta en modo incremental y, en cambio, la segunda en modo *WRAP*. Los resultados que se obtienen al ejecutar esta operación se presentan en la Tabla 18 y se puede corroborar en la Figura 86 y en la Figura 87 que, tanto la operación de escritura como la de lectura, se han completado con éxito.

Tabla 18. Resultados obtenidos de la secuencia enviada por el maestro 2 al esclavo 4

Tipo de operación	Señales del maestro 2	Datos obtenidos	Señales del esclavo 4	Datos obtenidos
Transacción de escritura	AWADDR	32'h4500_0000	AWADDR	32'h4500_0000
	AWBURST	2'h2	AWBURST	2'h2
	AWID	4'h2	AWID	7'h22
	WDATA	64'h91CC_CC1F_0993_463B	WDATA	64'h1207_10DF_544A_5499
Segunda transacción de lectura	ARADDR	32'h4500_0000	ARADDR	32'h4500_0000
	ARBURST	2'h2	ARBURST	2'h2
	ARID	4'h2	ARID	7'h22
	RDATA	64'h91CC_CC1F_0993_463B	RDATA	64'h1207_10DF_544A_5499

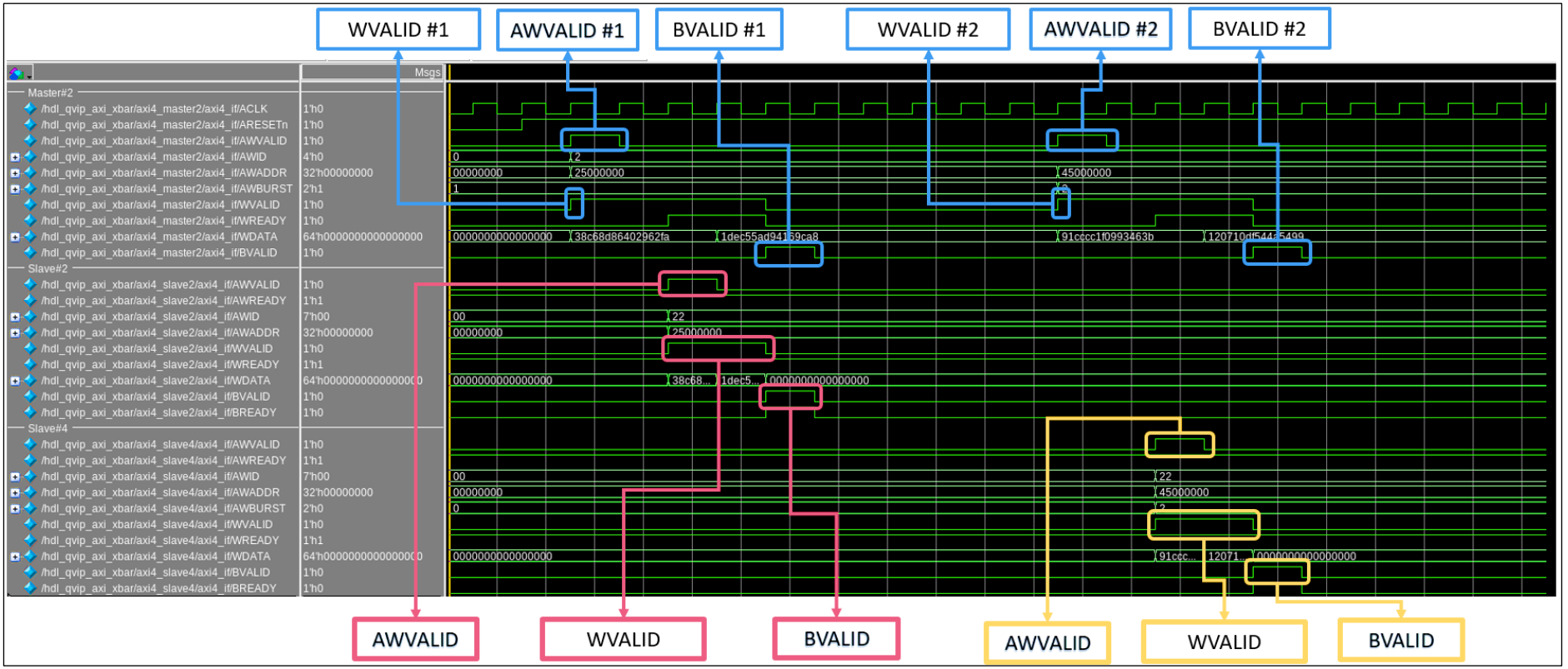

Figura 85. Señales handshake de la secuencia ejecutada por el maestro 2

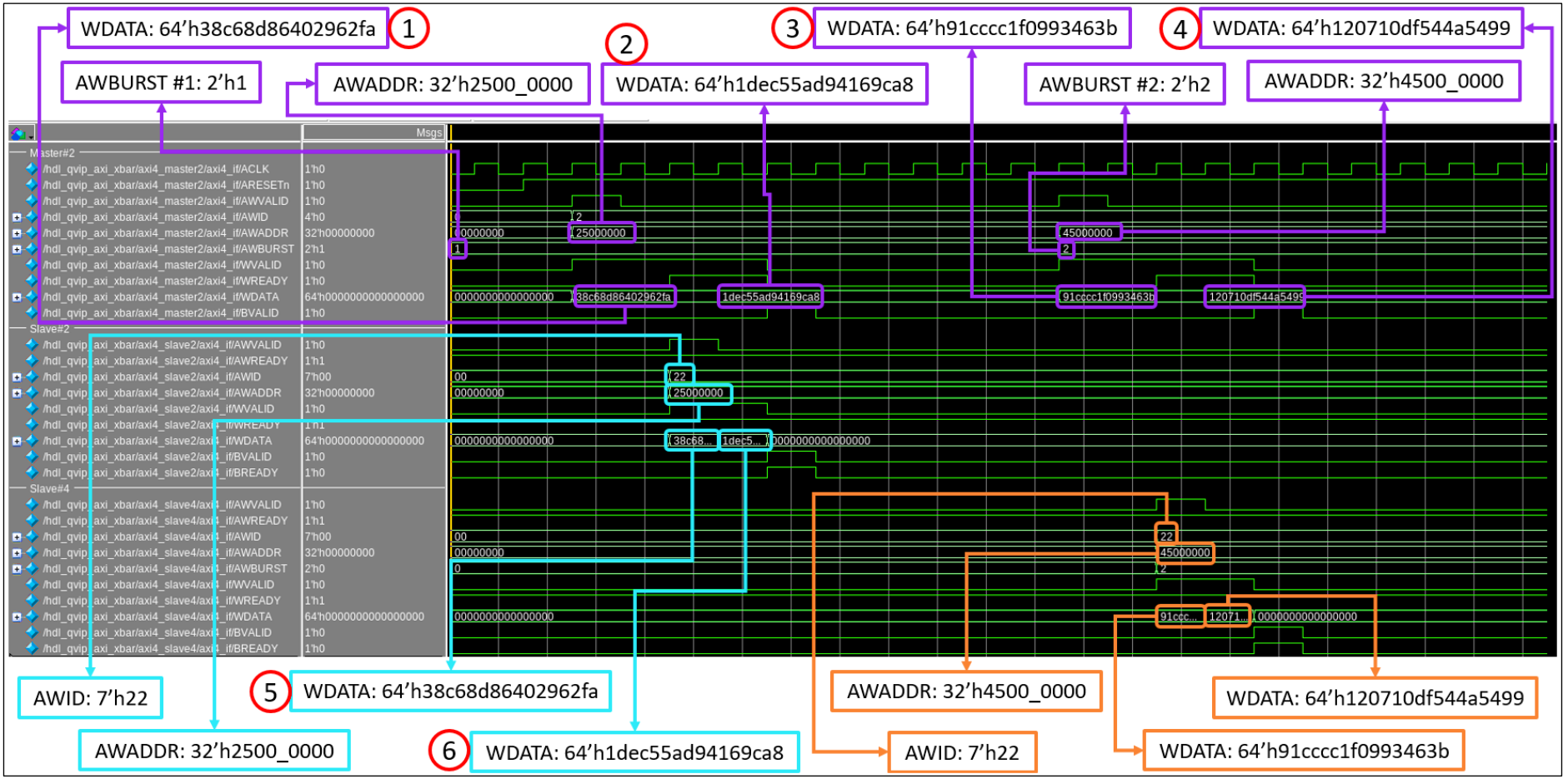

Figura 86. Operación de escritura ejecutada por el maestro 2 en los esclavos 2 y 4

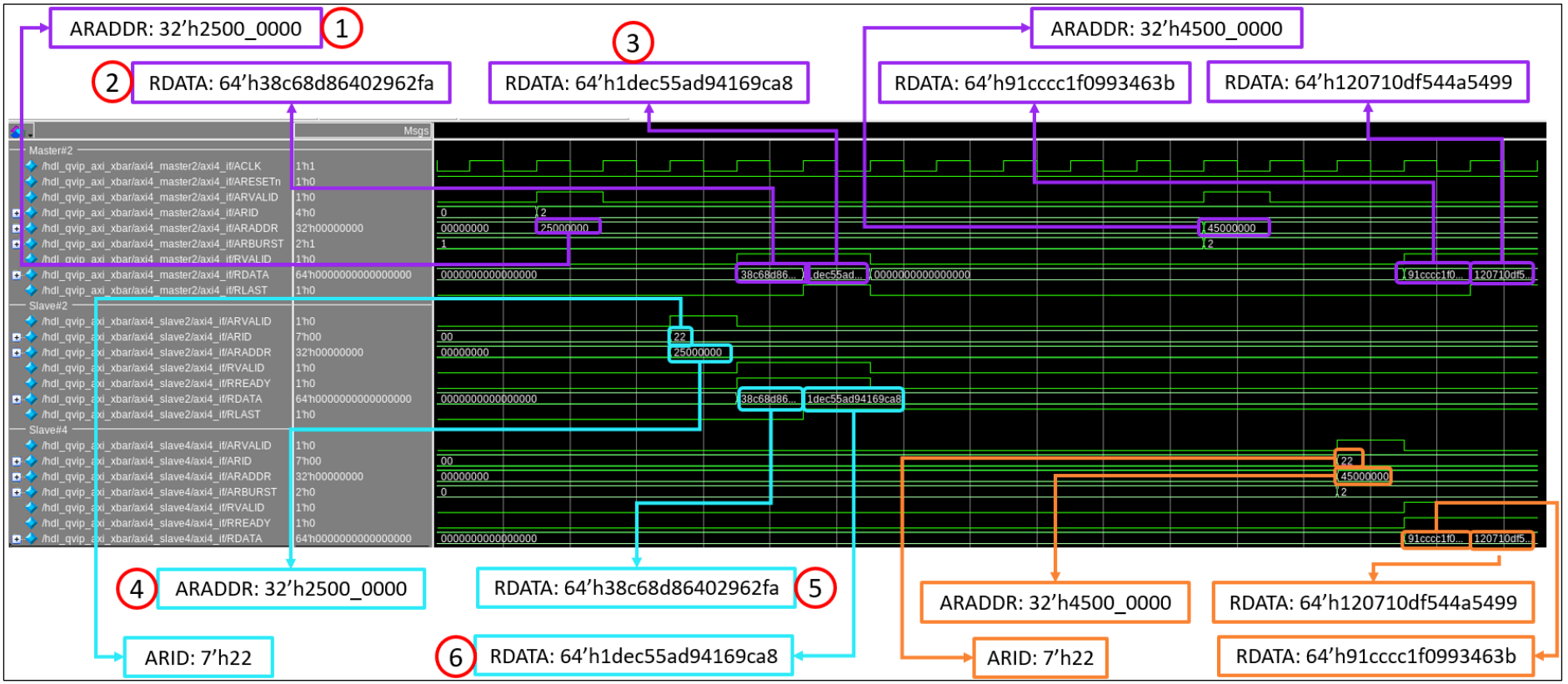

Figura 87. Operación de lectura ejecutada por el maestro 2 en los esclavos 2 y 4

7.2.2.4. Secuencia *axi4_generic_ api_excl_seq*

Esta es la primera secuencia que utiliza las API proporcionadas por la herramienta QVIP *configurator*. Estas API abren multitud de posibilidades a la hora de configurar diferentes parámetros de la secuencia enviada. Con esta secuencia se pretende verificar la escritura y lectura de un acceso exclusivo en una dirección de memoria de un esclavo.

Sólo se van a definir cuatro parámetros de las secuencias que van a ser enviadas por el maestro: el acceso exclusivo (*AxLOCK*), la dirección en la que se va a realizar la correspondiente operación (*AxADDR*), el dato de escritura (*WDATA*) y, por último, el identificador de la transacción (*AxID*). En la Tabla 19, se detallan los valores de los parámetros definidos.

Tabla 19. Configuración de la secuencia ejecutada por el maestro 3

Señales	Valor
AxLOCK	1'h1
AxID	4'h3
AxADDR	32'h1000_2000 (esclavo 1)
WDATA	64'h11

La Figura 88 muestra el resultado de la ejecución de la secuencia *axi4_generic_ api_excl_seq*. En primer lugar, se completa el protocolo de *handshake* de las señales VALID y READY en todos los canales del protocolo. En segundo lugar, se lee la dirección 32'h1000_2000 y se corrobora que está vacía. Después, se escribe en esa misma dirección el dato que se encuentra en la marca 2 de la Figura 88. Por último, se comprueba si la transacción se ha completado con éxito realizando una lectura a la dirección 32'h1000_2000, y como se visualiza en la marca 5 de la figura mencionada, el valor leído es el mismo que el escrito previamente.

Como se ha explicado en ocasiones anteriores, la señal *AxID* del esclavo no concuerda con la definida porque el módulo *crossbar* redefine dicho parámetro internamente, añadiendo 3 *bits* para la codificación del número del maestro que ha realizado la transacción. Por lo tanto, como el maestro 3 ha sido la fuente de la operación, el valor del campo AxID del esclavo es 7'h33.

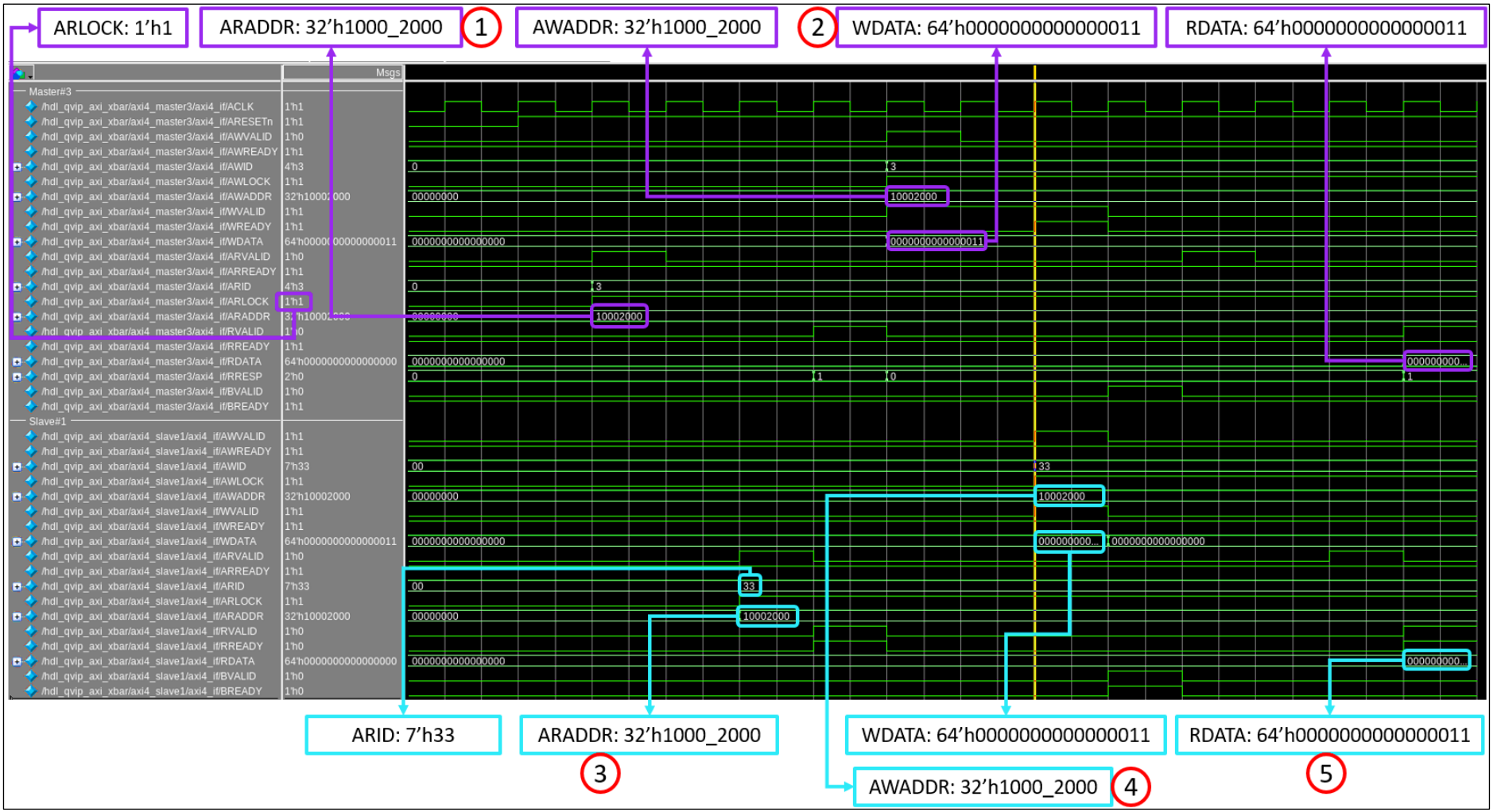

Figura 88. Operaciones de lectura y escritura del maestro 3 en el esclavo 1

7.2.2.5. Secuencia *axi4_generic_ api_normal_seq*

Esta secuencia es similar a la anteriormente analizada. La principal diferencia radica en la definición del parámetro *LOCK*, ya que en este caso no se realiza una escritura en acceso exclusivo, como se explicó en el capítulo 6. Aparte, se definen otros parámetros de manera diferente como los mostrados en la Tabla 20.

Tabla 20. Configuración de la secuencia ejecutada por el maestro 4

Señales	Valor
AxLOCK	1'h0
AxID	4'h4
AxADDR	32'h7000_2000 (esclavo 7)
WDATA	64'h11223344

En la Figura 89 se observa con claridad que se hacen dos operaciones, una de escritura, en los instantes 1 y 2, y otra de lectura, en los instantes 3 y 4. Como en los casos anteriores, se accede a la misma dirección donde se ha escrito el dato para verificar la escritura anterior. Al comparar los datos de la Tabla 20 con los datos de las transacciones mostradas en las formas de onda de la Figura 89, se hace evidente que las transacciones enviadas son correctas.

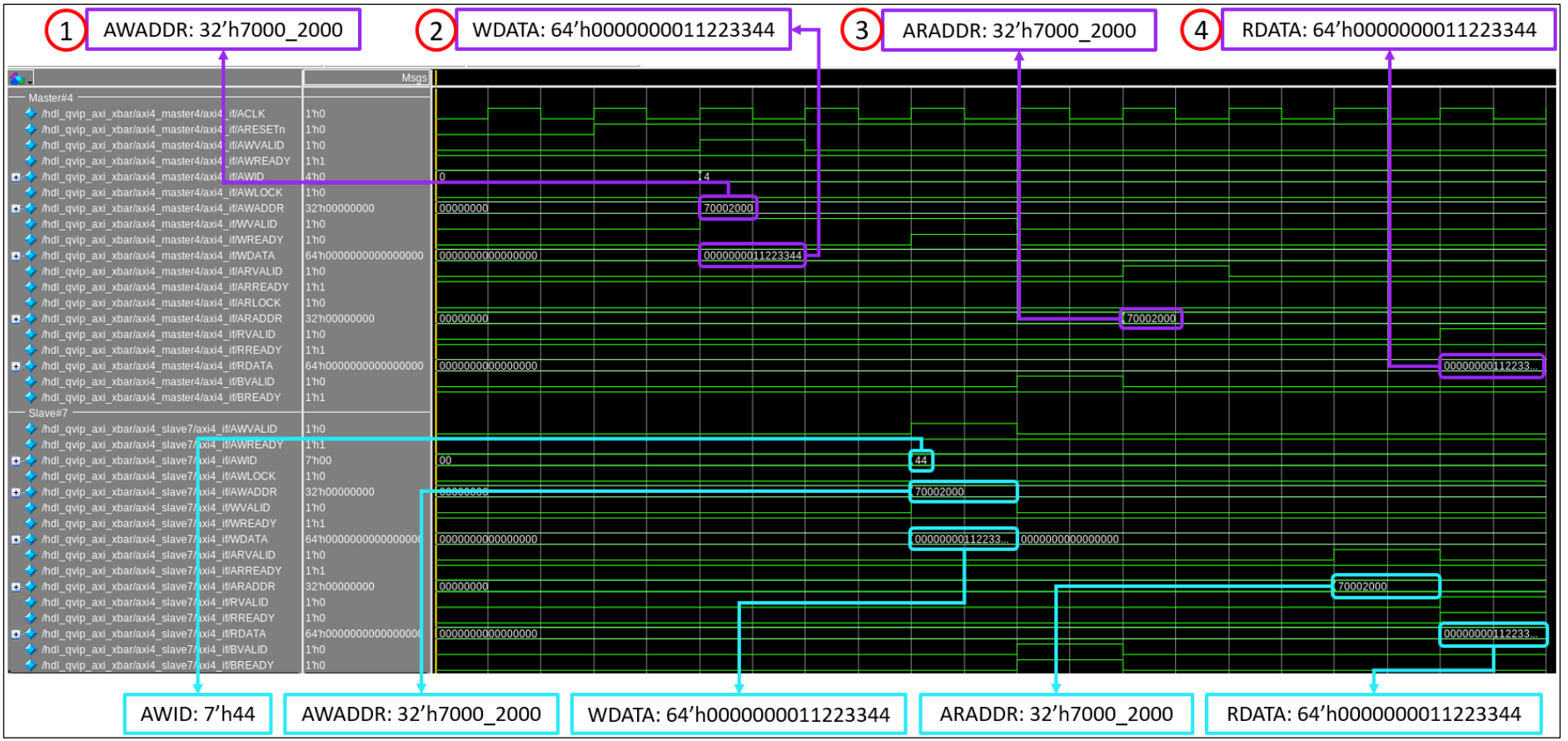

Figura 89. Operaciones de escritura y lectura del maestro 4 en el esclavo 7

7.2.2.6. Secuencia *axi4_generic_ api_wrap_seq*

La última secuencia descrita comprueba qué sucede en caso de enviar un campo de datos con más *bits* que el ancho del canal. En este caso, se van a enviar dos operaciones de escritura y otras dos de lectura. En la Tabla 21 se detallan los parámetros definidos del maestro 5 en cada una de las transacciones.

Tabla 21. Configuración de la secuencia ejecutada por el maestro 5

Transacción	Señal	Valor
Primera transacción (escritura y lectura)	AxBURST	2'h2
	AxSIZE	3'h11
	AxID	4'h5
	AxADDR	32'h3000_1000
	WDATA	64'h1122334455667788
Segunda transacción (escritura y lectura)	AxBURST	2'h1
	AxID	4'h5
	AxADDR	32'h5000_5000
	WDATA	128'h000102030405060708090A0B0C0D0E0F

Con el parámetro *SIZE* se establece el número de *bytes* por transferencia y, en este caso, serán 8 *bytes* por transferencia, lo que supone que se podrá utilizar todo el ancho del campo de datos del canal. La primera transacción se enviará al esclavo 3 y la segunda al esclavo 5. De esta manera, se habrá comprobado el envío de una transacción a todos y cada uno de los esclavos que forman parte del sistema. Además, todos los maestros habrán enviado al menos una secuencia.

En la Figura 96 se muestran las dos operaciones de escritura y en la Figura 97 las de lectura. El protocolo de *handshake* se cumple en ambos casos correctamente. A la hora de enviar la primera transacción de escritura, el campo de datos se escribe en cuatro fragmentos (marca 3 de la Figura 96), ya que de manera inesperada el sistema define para la primera operación de escritura el parámetro *SIZE* igual a 1, lo que supone que sólo se pueden enviar dos *bytes* por transferencia. Por lo tanto, se ha encontrado un funcionamiento no esperado en el sistema a verificar. No se ha profundizado en dicho comportamiento debido a que no es competencia del presente Trabajo Fin de Grado. Para mayor comprensión se explicará brevemente cómo se ejecuta la mencionada transacción.

El parámetro *SIZE* define a su vez los *bits strobes* (marca 4 de la Figura 96), los cuales indican los *bytes* válidos en el *bus* de datos *WDATA*. Como sólo se pueden enviar

dos *bytes* por transferencia, la transacción se ha enviado en cuatro transferencias. En la Figura 90, Figura 91, Figura 92 y Figura 93 se visualiza con mayor nivel de detalle cómo se ha efectuado esta transacción. En la Figura 90 se describe la primera transacción de escritura donde el campo *Master5_WDATA* contiene los 64 *bits* (8 *bytes*) presentes en el *bus* de datos. El campo *WSTRB* contiene el valor 8'h03, lo que indica que, de los 8 *bytes* presentes en el *bus WDATA*, únicamente se validarán los *bytes* 0 y 1, estos son los *bytes* 0x88 y 0x77, respectivamente.

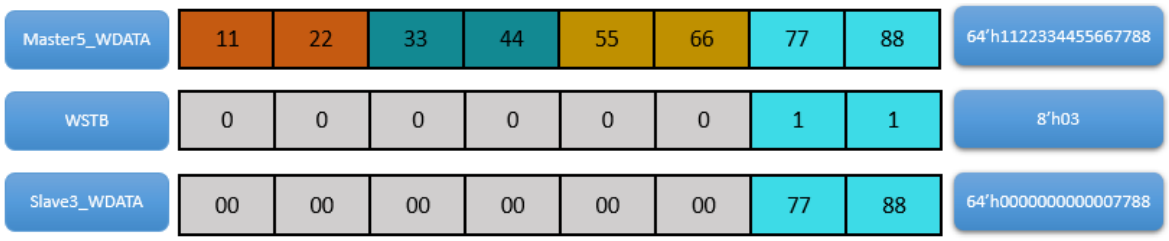

Figura 90. Primera transacción de escritura de los datos 64'h1122334455667788

En la Figura 91 el valor del campo *WSTRB* cambia y pasa a tener el valor 8'h0C, por lo que, en este caso, se validarán los *bytes* 2 y 3 del *bus WDATA*.

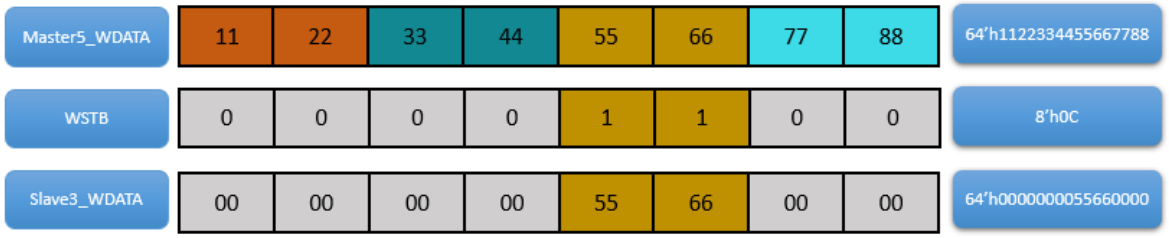

Figura 91. Segunda transacción de escritura de los datos 64'h1122334455667788

A continuación, se completaría la tercera transacción de escritura donde se validarán los *bytes* 0x44 y 0x33 debido a que los *bits strobes* toman el valor 8'h30. En la Figura 92 se muestra el procedimiento descrito de manera gráfica.

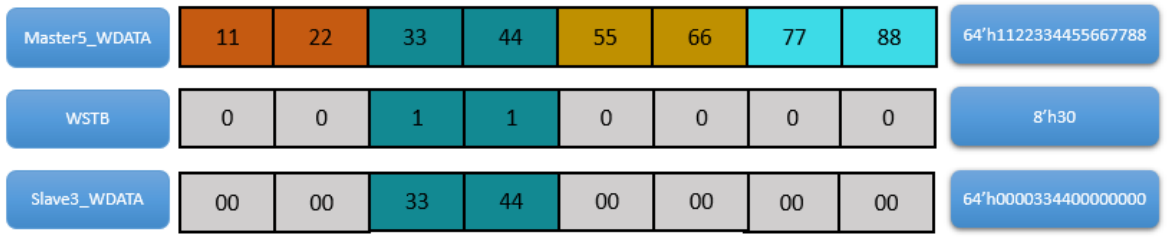

Figura 92. Tercera transacción de escritura de los datos 64'h1122334455667788

Por último, como se presenta en la Figura 93, para completar la transacción de escritura se validan los *bytes* más significativos, 0x22 y 0x11, ya que el campo *WSTRB* tiene el valor 8'hC0 en esta cuarta transferencia. Por lo tanto, como se ha explicado, la transacción de escritura que estaba prevista que se completase en una única transacción, ha finalizado en cuatro transacciones.

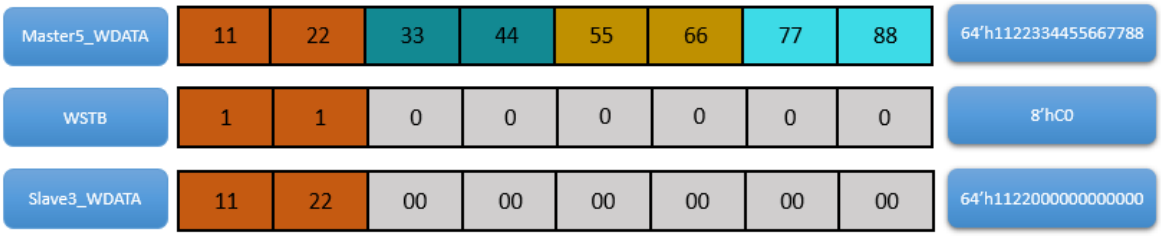

Figura 93. Cuarta transacción de escritura de los datos 64'h1122334455667788

Como se observa en la Figura 96, la escritura se realiza en dos direcciones diferentes (marcas 1 y 2). Esto se debe a que la secuencia está configurada en modo *WRAP,* por lo que el acceso a la siguiente dirección se incrementa 4. Por esta razón, los datos 64'h1122334400000000 se escriben en la dirección 32'h3000_1004.

Respecto a la segunda transacción de escritura, el campo de datos se envía dividido en dos, como se observa en la marca 5, ya que son 128 *bits* y el canal sólo soporta 64 *bits*. En este caso, el parámetro *SIZE* es igual a 3 y la señal *WSTRB* (marca 6 de la Figura 96) valida los 8 *bytes*. La transacción se completa con éxito ya que se envían dos transacciones, cada una de ellas con un tamaño de 8 *bytes*. Para mayor facilidad, se adjuntan la Figura 94 y la Figura 95, donde se explica visualmente la escritura de la dirección de 128 *bits*.

En la Figura 94 se representa la primera transacción de escritura del campo de datos, donde se escribe en la dirección de memoria del esclavo 5 el siguiente dato de 64 *bits*: 64'h0706050403020100. En esta caso, todos los *bits strobes* se encuentra a 1, por lo que validan los 8 *bytes*.

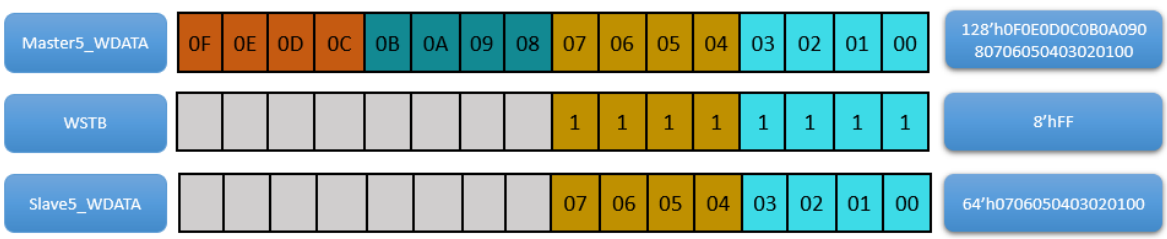

Figura 94. Primera transacción de escritura del campo de datos de 128 bits

A continuación, se lleva a cabo la segunda parte de la transacción de escritura, validando los 8 *bytes* restantes como se observa en la Figura 95. De esta manera, finaliza la operación de escritura.

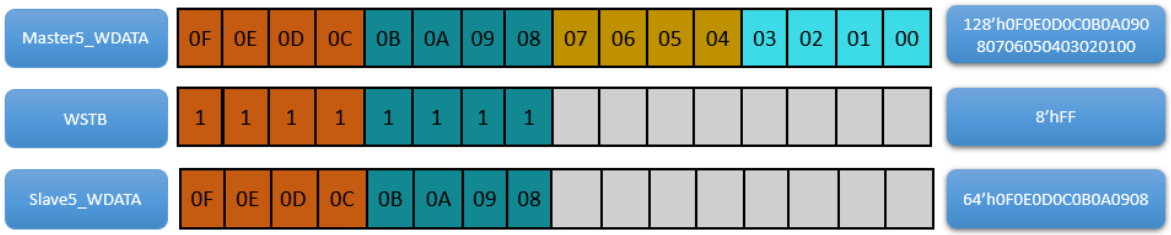

Figura 95. Segunda transacción de escritura del campo de datos de 128 bits

En relación a las operaciones de lectura, representadas en la Figura 97, ocurre lo mismo que en las de escritura, ya que se accede a las direcciones 32'h3000_1000 y 32'h3000_1004 (marcas 1 y 2 de esta figura) por las mismas circunstancias anteriormente analizadas. En este caso, al ser operaciones de lectura, los datos leídos corresponden con los datos enviados.

Por las razones expuestas, en las primeras operaciones de escritura y de lectura se ha detectado el mismo comportamiento a la hora de definir el parámetro *SIZE* de dichas operaciones. Pero, por otro lado, también se puede afirmar que las segundas operaciones de escritura y de lectura se completaron de manera satisfactoria.

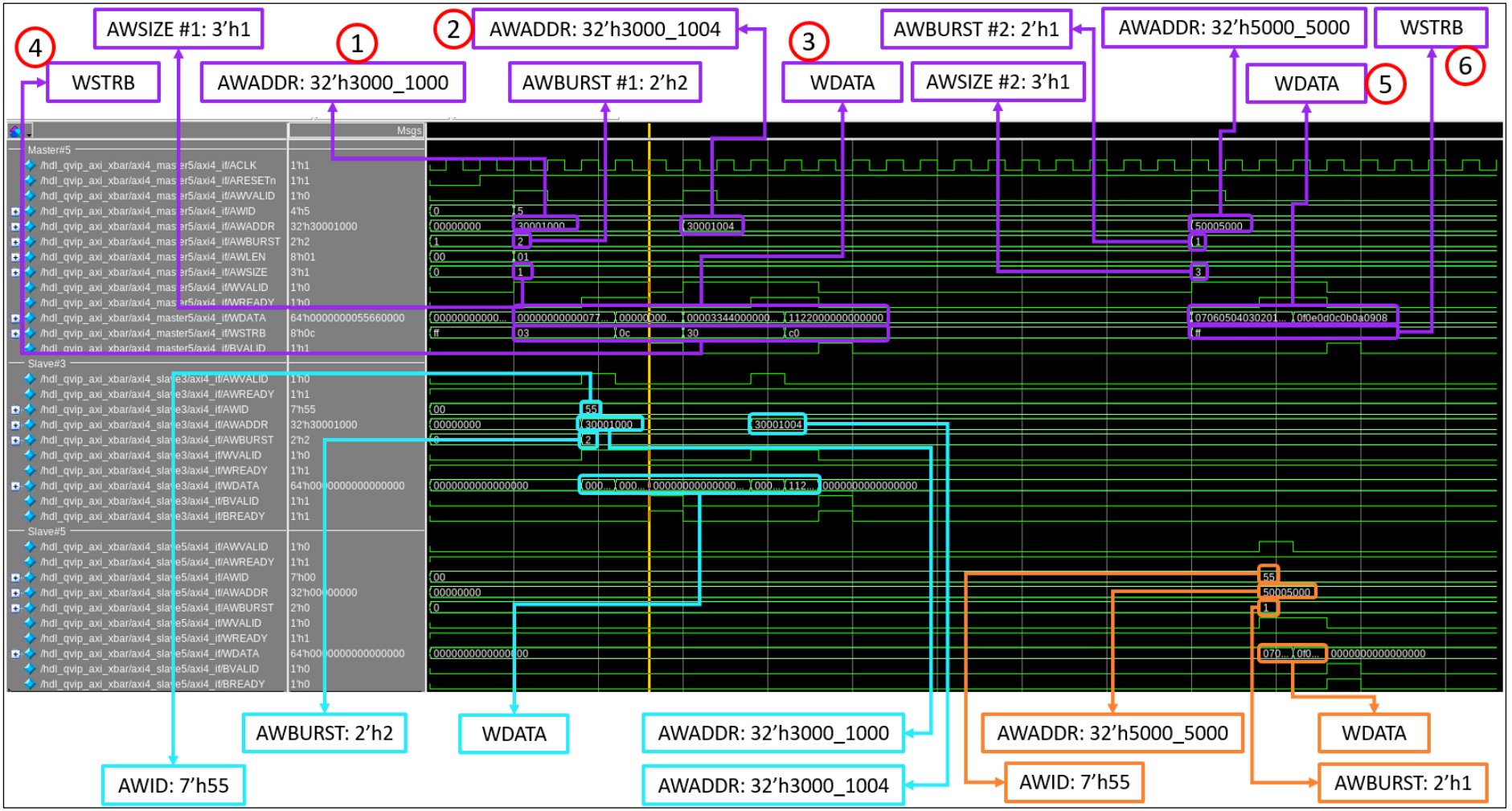

Figura 96. Operaciones de escritura del maestro 5 en los esclavos 3 y 5

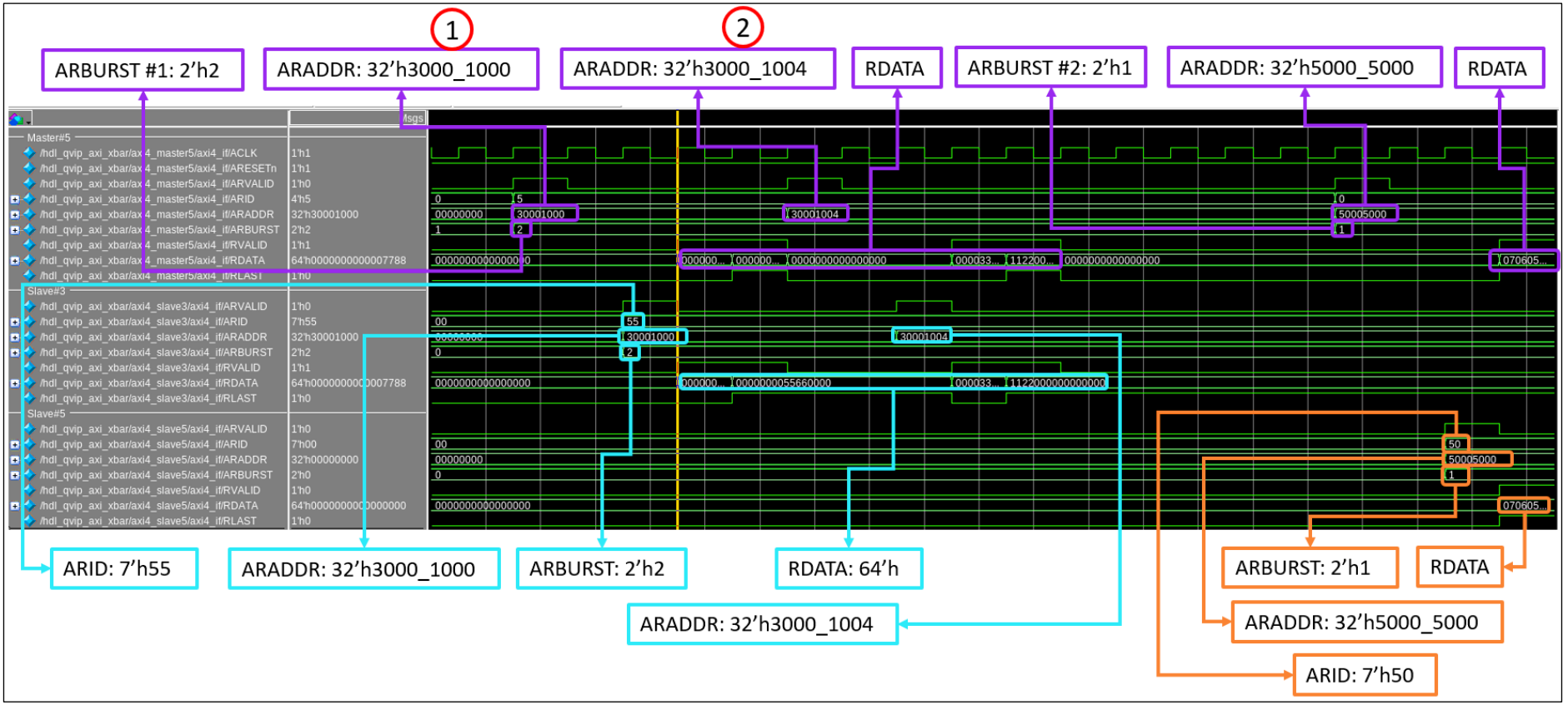

Figura 97. Operaciones de lectura del maestro 5 en los esclavos 3 y 5

Capítulo 8. Conclusiones y líneas futuras

8.1. Conclusiones

Una vez finalizado el Trabajo Fin de Grado denominado “Desarrollo de un *testbench* UVM integrando IP de verificación de *Mentor Graphics* (QVIP)”, se comprueba que se han cumplido los objetivos propuestos inicialmente. El objetivo principal de este Trabajo Fin de Grado consiste en el diseño de un *testbench* UVM usando los módulos QVIP de verificación proporcionados por *Mentor Graphics* y utilizando un entorno de verificación UVM que mejora la reutilización del código de los componentes programados.

Respecto al DUV (*Device Under Verification)*, cuyo comportamiento se verificó a lo largo del desarrollo del presente TFG, se ha optado por elegir un módulo proporcionado por un tercero. Finalmente, el sistema escogido ha sido un componente de interconexión utilizado ampliamente en los protocolos de comunicación, denominado *crossbar*. La principal función de este sistema es la conexión simultánea de todos los puertos de entrada y salida, siempre que no se intente acceder al mismo tiempo a la misma salida. El procedimiento seguido se basó en la estimulación de las entradas del DUV y en el exhaustivo análisis de las interfaces de salida. De esta manera, se comprobó el comportamiento del dispositivo ante unos estímulos controlados.

La verificación del comportamiento del módulo *crossbar* se llevó a cabo siguiendo un enfoque *black-box*, por lo que no importó el contenido del módulo ni su implementación. Como se explicó en el capítulo 4, no se siguió una filosofía *black-box* estricta porque fue

necesario consultar algunos parámetros del módulo *crossbar* como, por ejemplo, los parámetros internos que configuran el dispositivo y el orden de las señales propias del protocolo AXI4. Este módulo posee una interfaz compatible con el protocolo de comunicación AXI4, pero no tiene una interfaz AXI4 propiamente dicha. Debido a estos inconvenientes y al deseo de no modificar el código original del módulo *crossbar* proporcionado por Andreas Kurth, se creó un módulo intermedio que, en última instancia, será el que esté en contacto directo con el entorno de verificación UVM. Este módulo se denominó *wrapper* y fue crucial para la correcta adaptación del módulo *crossbar* con el entorno de verificación UVM generado por la herramienta QVIP *configurator*.

Una vez finalizado el estudio del módulo a verificar, se profundizó en los conceptos de la herramienta QVIP *configurator*. Esta herramienta de configuración es la encargada de generar el entorno de verificación UVM de manera automática. Para ello, en primer lugar, se ha importado el DUV, se han creado los módulos QVIP que se comportan como componentes UVM *Agent*, se ha realizado el conexionado y, finalmente, se ha configurado el mapa de direcciones de los módulos QVIP esclavos.

El *testbench* diseñado cuenta con una pila de secuencias que estimulan de manera eficaz las entradas del módulo *crossbar*, objeto de verificación a lo largo de este TFG. Estas secuencias no cuentan con una gran complejidad y no verifican todos los posibles escenarios de actuación del DUV en cuestión. Esto se ha debido al tiempo limitado asociado a la realización de este Trabajo Fin de Grado y al esfuerzo relacionado con la creación del entorno de verificación UVM. Esto se ha unido a la dificultad innata de aprendizaje de la metodología UVM, el aprendizaje del lenguaje *SystemVerilog* y el tiempo invertido en el estudio de la herramienta QVIP *configurator*. Las secuencias utilizadas se programaron a partir de la biblioteca de secuencias básicas y API proporcionadas por la herramienta QVIP *configurator*, las cuales facilitaron enormemente el desarrollo de las mismas.

A la vista de los resultados obtenidos, todos los objetivos propuestos se han cumplido de manera exitosa, por lo que el planteamiento seguido a lo largo del desarrollo de este Trabajo Fin de Grado ha sido correcto, gracias a las indicaciones proporcionadas por los tutores de este TFG.

8.2. Líneas futuras

En relación a las líneas futuras que se proponen para futuros Trabajos Fin de Grado o Trabajos Fin de Máster, se consideran varias vías. La primera de ellas es la realización

de un Plan de Verificación en el que se detalle un conjunto completo de *test* para la verificación del módulo *crossbar* mediante el desarrollo de una pila de secuencias de mayor complejidad que abarquen todos los escenarios posibles.

Por otro lado, es necesario incluir métricas de cobertura (*coverage*) en el entorno de verificación UVM que mejoren notablemente la efectividad de la verificación aplicada sobre el DUV en cuestión. Otra posible línea futura de este Trabajo Fin de Grado es el cambio del protocolo de comunicación por uno más complejo como, por ejemplo, el PCIe (*Peripheral Component Interconnect Express*) que cuenta con un número mayor de señales y posibilidades. Para ello, sería necesario utilizar la herramienta QVIP *configurator* para la creación del entorno de verificación UVM compatible con el protocolo elegido y, además, sería necesaria la modificación del módulo *wrapper* creado, añadiendo diferentes módulos convertidores de PCIe a AXI4.

Bibliografía

[1] “Part 8: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons.” https://blogs.sw.siemens.com/verificationhorizons/2021/01/06/part-8-the-2020-wilson-research-group-functional-verification-study/ (accessed Jun. 24, 2021).

[2] “Part 6: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons.” https://blogs.sw.siemens.com/verificationhorizons/2020/12/16/part-6-the-2020-wilson-research-group-functional-verification-study/ (accessed Jun. 24, 2021).

[3] V. R. Cooper, *Getting Started with UVM: A Beginner’s Guide*. Verilag Publishing, 2013, 110 págs., ISBN: 978-0-615-81997-6.

[4] “Questa Verification IP | Siemens Digital Industries Software.” https://eda.sw.siemens.com/en-US/ic/questa/verification-ip/ (accessed Jun. 24, 2021).

[5] J. Pavón Luque, “"Movimiento eficiente de datos para un SoC basado en Zynq.”.” https://ebuah.uah.es/dspace/bitstream/handle/10017/24163/TFG Pavón Luque 2015.pdf?sequence=1&isAllowed=y.

[6] “AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite ACE and ACE-Lite.” https://developer.arm.com/documentation/ihi0022/e/.

[7] “What is the difference between white box, black box, and gray box testing? - Quora.” https://www.quora.com/What-is-the-difference-between-white-box-black-box-and-gray-box-testing (accessed Jun. 24, 2021).

[8] “Part 10: The 2020 Wilson Research Group Functional Verification Study | Verification Horizons.” https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020-wilson-research-group-functional-verification-study/ (accessed Jun. 24, 2021).

[9] “Introduction To System Verilog | System Verilog Tutorial | System Verilog.” http://www.asicguru.com/system-verilog/tutorial/introduction/1/ (accessed Jun. 24, 2021).

[10] A. R. Mederos, “‘Desarrollo de la capa RAL en un Entorno UVM para la verificación funcional de un IP multi-interfaz orientado a la compresión de imágenes,’” Universidad de Las Palmas de Gran Canaria, 2018.

[11] “Universal Verification Methodology (UVM) 1.2 User’s Guide,” 2015. Accessed: Jun. 24, 2021. [Online]. Available:

https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[12] “UVM Phasing - The Art of Verification.” https://www.theartofverification.com/uvm-phasing/ (accessed Jun. 24, 2021).

[13] “SystemC TLM.” https://www.accellera.org/community/systemc/about-systemc-tlm (accessed Jul. 03, 2021).

[14] “UVM Introduction.” https://www.chipverify.com/uvm/uvm-introduction (accessed Jun. 24, 2021).

[15] “Verilab Uvm Tutorial | Class (Computer Programming) | Application Programming Interface.” https://es.scribd.com/document/172143106/Verilab-Uvm-Tutorial.

[16] “UVM Driver and Sequencer Communication | Universal Verification Methodology.” https://www.learnuvmverification.com/index.php/2015/07/07/uvm-driver-and-sequencer-communication/ (accessed Jun. 24, 2021).

[17] “UVM Scoreboard.” https://www.chipverify.com/uvm/uvm-scoreboard (accessed Jun. 24, 2021).

[18] “Kurth, Andreas | ETH Zurich.” https://ee.ethz.ch/the-department/people-a-z/person-detail.MTgwNzQz.TGlzdC8zMjc5LC0xNjUwNTg5ODIw.html (accessed Jun. 25, 2021).

[19] “GitHub - pulp-platform/axi: AXI SystemVerilog synthesizable IP modules and verification infrastructure for high-performance on-chip communication.” https://github.com/pulp-platform/axi (accessed Jun. 25, 2021).

[20] A. Kurth, “AXI4+ATOP Fully-Connected Crossbar.” https://github.com/pulp-platform/axi/blob/master/doc/axi_xbar.md (accessed Jul. 03, 2021).

[21] “axi/tb_axi_xbar.sv at master · pulp-platform/axi · GitHub,” 2013. https://github.com/pulp-platform/axi/blob/master/test/tb_axi_xbar.sv (accessed Jul. 20, 2021).

[22] M. G. Corporation, “Questa Verification IP User Guide for the ARM AMBA 4 AXI Protocol,” 2019.

[23] “Getting Started with Questa Verification IP for Protocols | Verification Horizons.” https://blogs.sw.siemens.com/verificationhorizons/2021/02/18/getting-started-with-questa-verification-ip-for-protocols/ (accessed Jul. 11, 2021).

[24] M. G. Corporation, “Questa Verification IP Configurator User Guide,” 2019.

[25] “Questa Advanced Simulator | Siemens Digital Industries Software.” https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/ (accessed Jul. 16, 2021).

Pliego de condiciones

En este apartado se detallarán las condiciones a partir de las cuales se ha desarrollado el presente Trabajo Fin de Grado. Se analizarán tanto las condiciones *hardware* como las condiciones *software*, por lo tanto, se expondrá un despliegue de todos los medios con sus respectivas propiedades.

PC.1 Condiciones *hardware*

En la Tabla 22 se presentan los dispositivos *hardware* necesarios para el desarrollo de este Trabajo Fin de Grado.

Tabla 22. Condiciones hardware

Equipo/dispositivo	Modelo	Fabricante
Ordenador personal	HP *Pavilion* x360 Convertible: • CPU *Intel Core* i5-7500U a 2,5 GHz. • 8 GB RAM. • 512 GB SSD. • 2 puerto USB. • 1 puerto USB 3.0.	*Hewlett-Packard* (HP)
Estación de trabajo	*Acer Predator* G3: • CPU *Intel Core* i7-4790. • 8 GB RAM. • 1 TB HDD. • 4 puertos USB.	*Acer* Inc.

PC.2 Condiciones *software*

Después de haber concretado los dispositivos *hardware* utilizados a lo largo de este Trabajo Fin de Grado, se muestran las herramientas *software* empleadas para su realización en la Tabla 23.

Tabla 23. Condiciones software

Equipo/dispositivo	Modelo	Fabricante
Sistema operativo	*Microsoft Windows* 10 *Home* 64 *bits*	*Microsoft Corporation*
Microsoft Office	2019	*Microsoft Corporation*
Microsoft Project	2018	*Microsoft Corporation*
Visual Studio Code	Versión 1.58.0	*Microsoft Corporation*
Google Chrome	Versión 91.0.4472.124 – 64 *bits*	*Google* LLC
Sistema operativo	*Red Hat Enterprise Linux Server*	*Red Hat* Inc.
Simulador y compilador *QuestaSim*	Versión 2019.04	*Mentor Graphics, a Siemens Business*
Librería UVM	Versión 1.1d	*Mentor Graphics, a Siemens Business*
Librería *Questa* VIP	Versión 2019.1.1	*Mentor Graphics, a Siemens Business*
Foxit Reader	Versión 11.0.0.49893	*Foxit Software*

Presupuesto

Para llevar a cabo el presente Trabajo Fin de Grado, ha sido necesario la utilización de distintos recursos de diferente índole. A partir de todos estos recursos se puede estimar la cuantía de cada uno de ellos en función de su naturaleza y de la aportación que realiza durante el proyecto. Por estas razones, en este capítulo se detallarán todos los recursos materiales y humanos empleados. Por lo tanto, el presupuesto total para la realización de este Trabajo Fin de Grado se desglosará en los siguientes conceptos:

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

P.1. Recursos humanos

Para el cálculo del coste asociado a las horas de trabajo empleadas durante el desarrollo del Trabajo Fin de Grado se utiliza una ecuación recomendada por el Colegio

Oficial de Ingenieros Técnicos de Telecomunicación (COITT), la cual realiza el correspondiente cálculo de los honorarios en función de las horas trabajadas, tanto dentro como fuera de la jornada laboral convencional. Por lo tanto, la ecuación para el cálculo de los costes de los Recursos Humanos (RRHH) es la que se presenta a continuación:

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

Donde las incógnitas corresponden a los siguientes conceptos:

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

Los posibles valores del factor de corrección C_t se encuentran en la Tabla 24, de acuerdo con el procedimiento de cálculo marcado por el COITT:

Tabla 24. Factor de corrección según horas trabajadas

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

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

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

El coste asociado a las horas de trabajo empleadas durante los últimos meses asciende a trece mil cuatrocientos sesenta y ocho euros con cuarenta céntimos (13.478,4 €). Se debe tener en cuenta que a dicha cantidad no se le han aplicado impuestos ni retenciones.

P.2. Recursos materiales

Como se comentó en el Pliego de Condiciones, se han utilizado herramientas *hardware* y herramientas *software* durante la realización de este Trabajo Fin de Grado. A la hora de hacer el cálculo del coste de cada uno de estos recursos se relaciona su coste total con el tiempo útil de estos recursos. Para realizar este cálculo se va a utilizar la siguiente fórmula:

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

Se fija como referencia para el sistema de amortización lineal una vida útil de 3 años de los recursos.

P.2.1. Recursos *hardware*

Es clave tener en cuenta la amortización de los recursos *hardware* que se han utilizado a lo largo del desarrollo de este Trabajo Fin de Grado. En la Tabla 25 se adjuntan los resultados del coste total de los recursos *hardware*.

Tabla 25. Coste total de los recursos hardware

Equipo/dispositivo	Valor de adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Ordenador personal – HP *Pavilion*	650 €	36 meses	18,05 €	6 meses	108,3 €
Estación de trabajo – *Acer Predator*	1000 €	36 meses	27,77 €	6 meses	166,62 €
Coste total					274,92 €

Como se visualiza en la tabla adjuntada, el coste total de los recursos *hardware* es de doscientos setenta y cuatro euros con noventa y dos céntimos (274,92 €).

P2.2. Recursos *software*

En este apartado se calcula el coste total de los recursos *software* utilizados durante el Trabajo Fin de Grado. Cabe destacar que muchas de estas herramientas han sido proporcionadas de forma gratuita por la Universidad de Las Palmas de Gran Canaria (ULPGC). En la Tabla 26 se detallan los diferentes costes de todos estos recursos *softwares*.

Tabla 26. Coste total de los recursos software

Equipo/dispositivo	Valor de adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Sistema operativo *Microsoft Windows* 10 *Home* – 64 *bits*	145 €	36 meses	4,03 €	6 meses	24,18 €
Microsoft Office	0,00 € (licencia ULPGC)	12 meses	0,00 €	6 meses	0,00 €
Microsoft Project	0,00 € (licencia ULPGC)	12 meses	0,00 €	6 meses	0,00 €
Visual Studio Code	0,00 € (*software* libre)	12 meses	0,00 €	6 meses	0,00 €
Google Chrome	0,00 € (*software* libre)	12 meses	0,00 €	6 meses	0,00 €
Sistema operativo	0,00 € (*software* libre)	12 meses	0,00 €	6 meses	0,00 €
Simulador y compilador *QuestaSim*	2.000,00 €	12 meses	166,66 €	6 meses	999,99 €
Librería UVM	0,00 € (librería de libre distribución)	12 meses	0,00 €	6 meses	0,00 €
Librería *Questa* VIP	0,00 € (licencia IUMA, ULPGC)	12 meses	0,00 €	6 meses	0,00 €
Foxit Reader	0,00 € (*software* libre)	12 meses	0,00 €	6 meses	0,00 €
Coste total					1.024,17 €

Por lo tanto, el coste total de los recursos *softwares* se eleva a mil veinticuatro euros con diecisiete céntimos (1.024,17 €).

P.3. Material fungible

Los costes que derivan del material fungible utilizado en este Trabajo Fin de Grado se presentan en la Tabla 27:

Tabla 27. Costes del material fungible

Concepto	Importe
Folios	10,00 €
Tres CD-ROM	4,00 €
Coste total	14,00 €

Después de realizar los cálculos, se confirma que los gastos de los materiales fungibles son de catorce euros (14 €).

P.4. Redacción de la Memoria

También se deben calcular los gastos asociados a la redacción de la memoria de este Trabajo Fin de Grado a partir de la siguiente fórmula:

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

Donde las incógnitas representan los siguientes conceptos:

- El término *R* se refiere a los honorarios derivados de la redacción del trabajo.
- El parámetro *P* hace referencia al presupuesto del proyecto.
- La incógnita C_n representa el coeficiente de corrección en función del presupuesto calculado.

El valor del parámetro *P* se obtiene mediante la suma del coste relativo de los RRHH y los recursos materiales, tanto *hardware* como *software*, como se presenta a continuación:

$$P = 13.478{,}4 + 274{,}92 + 1024{,}17 = 14.777{,}49\ €$$

Según el Colegio Oficial de Ingenieros Técnicos de Telecomunicación, el coeficiente de corrección C_n tendrá por valor la unidad si el parámetro *P* tiene un valor inferior a

los 30.050 €. Por lo tanto, la ecuación de los gastos derivados de la redacción de la memoria se muestra a continuación:

$$R = 0{,}07 * 14.777{,}49 * 1 = \ 1.034{,}42 \ €$$

Por lo tanto, los gastos relativos a la redacción de la memoria del Trabajo Fin de Grado son de mil treinta y cuatro euros con cuarenta y dos céntimos (1.034,42 €).

P.5. Derechos de visado del COITT

El Colegio Oficial de Ingenieros Técnicos de Telecomunicación, anualmente, proporciona valor a los costes derivados de los derechos de visado por la ejecución de proyectos técnicas de carácter general. De esta forma, para el año 2021 se calcula de la siguiente manera:

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

donde estas incógnitas significan lo siguiente:

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

En el caso particular de este Trabajo Fin de Grado, se puede determinar que el coste derivado del parámetro P_2 es igual a 0,00 €, por lo que no será necesario aplicar el coeficiente C_2. En relación al parámetro C_1, como el presupuesto P_1 no supera los 30.050 €, tiene un valor igual a 1. En este caso se suma al presupuesto P_1 el coste asociado a la redacción del trabajo, por lo que se obtiene 15.811,91 €. Por lo tanto, la fórmula que calcula los costes de los derechos de visado queda como:

$$V = 0{,}006 * 15.811{,}91 * 1 = 94{,}87 \ €$$

Después de realizar este cálculo se puede concluir que el coste asociado a los derechos de visado del presupuesto se eleva a los noventa y cuatro euros con ochenta y siete céntimos (94,87 €).

P.6. Gastos de tramitación y envío

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

P.7. Coste total del proyecto

Después de obtener todos los costes de los diferentes conceptos y actividades que forman parte del desarrollo del Trabajo Fin de Grado, se va a proceder al cálculo del coste total del proyecto. Hay que añadir el valor de los impuestos a pagar, que en este caso se le aplica el Impuesto General Indirecto Canario (IGIC), del siete por ciento (7%). El coste total del proyecto se muestra en la Tabla 28.

Tabla 28. Coste total del Trabajo Fin de Grado

Concepto	Importe
Honorarios por tiempo empleado	13.478,4 €
Recursos *hardware*	274,92 €
Recursos *software*	1.024,17 €
Material fungible	14 €
Costes de redacción	1.034,42 €
Derechos del visado del COITT	94,87 €
Costes de tramitación y envío	6 €
Subtotal	15.926,78 €
IGIC (7%)	1.114,87 €
Coste total	17.041,65 €

Como se ha calculado, el importe total al que se eleva el presupuesto del Trabajo Fin de Grado “Desarrollo de un *testbench* UVM integrando IP de verificación de *Mentor Graphics* (QVIP)” es de diecisiete mil cuarenta y un euros con sesenta cinco céntimos (17.041,65 €).

Fdo.: D. Álvaro José Moreno Florido

En Las Palmas de Gran Canaria, a 21 de julio de 2021