

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

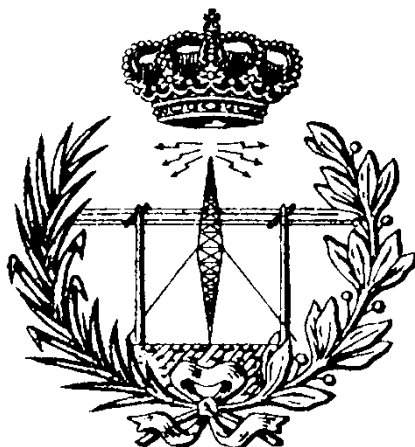


TRABAJO FIN DE MÁSTER

**Entorno UVM para la verificación funcional de un
IP multi-interfaz orientado a la compresión de
imágenes**

Titulación: Máster Universitario en Ingeniería de
Telecomunicación
Autor: D. Samuel Rodríguez Rodríguez
Tutores: Dr. D. Valentín De Armas Sosa
Dr. D. Félix B. Tobajas Guerrero
Fecha: Febrero de 2018

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE MÁSTER

**Entorno UVM para la verificación funcional de un
IP multi-interfaz orientado a la compresión de
imágenes**

HOJA DE FIRMAS

Alumno

Fdo.: D. Samuel Rodríguez Rodríguez

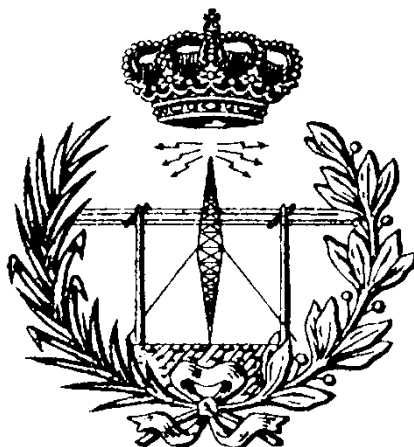
Tutor/a

Tutor/a

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

Fecha: Febrero de 2018

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA



TRABAJO FIN DE MÁSTER

**Entorno UVM para la verificación funcional de un
IP multi-interfaz orientado a la compresión de
imágenes**

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal

Secretario/a

Fdo.:

Fdo.:

Fecha: Febrero de 2018

RESUMEN

Actualmente, los procesos de validación y verificación son extremadamente importantes en el diseño de sistemas hardware digitales y abarcan la mayor parte del tiempo de desarrollo de un producto. Es por ello que en este Trabajo Fin de Máster (TFM) se hace uso de la metodología UVM (*Universal Verification Methodology*) con el fin de reducir el impacto de la verificación funcional en el flujo de diseño de sistemas hardware. Se trata de una metodología de verificación reciente soportada por los principales desarrolladores de herramientas EDA (*Electronic Design Automation*). El objetivo fundamental de este TFM consiste en desarrollar un entorno de verificación basado en UVM para un módulo IP (*Intellectual Property*) específico, partiendo de un *test bench ad-hoc* de referencia ya existente. El entorno UVM será, no solamente reusable, sino que también estará mejor estructurado y resultará más sencillo de modificar que el *test bench* original. En primer lugar, se estudia la metodología UVM a partir de la implementación de un entorno de verificación UVM genérico, para a continuación analizar en detalle el módulo IP a verificar y su *test bench* original, con el fin de desarrollar el entorno de verificación basado en UVM y validar su funcionamiento. Además, se evidencian las mejores prestaciones en la arquitectura del entorno de verificación desarrollado en comparación con el *test bench* de referencia.

ABSTRACT

Validation and verification processes are extremely important in current designs of digital hardware systems and cover most of the development time of a final product. Therefore, in this project it is used UVM (Universal Verification Methodology), which is the most recent verification methodology and is supported by the main EDA (Electronic Design Automation) tools developers. In this case, the main objective consists of rebuilding a traditional test bench focused on a complex IP (Intellectual Property) module by using this methodology. In addition, that environment will be not only reusable, but also better structured and easier to modify (for instance, adding functionalities) than the traditional test bench. Firstly, the methodology was studied (creating generic UVM environment), as well as the specific IP the project deals with and its original test bench. Finally, the final UVM environment was developed and its validity proved. Moreover, it has been verified the more flexible and powerful architecture of the new verification environment in comparison with the original test bench.

ÍNDICE DE CONTENIDOS

Índice de Figuras	V
Índice de Códigos	VII
Índice de Tablas.....	IX
Acrónimos	XI

MEMORIA

Capítulo 1. Introducción	1
1.1 Antecedentes	1
1.2 Objetivos	3
1.3 Peticionario	4
1.4 Estructura del documento	4
Capítulo 2. Universal Verification Methodology.....	5
2.1 Introducción a la verificación	5
2.2 Verificación funcional de sistemas hardware digitales.....	6
2.3 SystemVerilog.....	8
2.4 Introducción a UVM.....	10
2.5 Fundamentos de UVM.....	12
2.5.1 Biblioteca de clases	12
2.5.2 Factory.....	14
2.5.3 Fases de UVM	15
2.5.4 Mecanismo de configuración	17
2.5.5 Mecanismo de mensajes.....	18
2.5.6 TLM	20
2.6 Estructura de un entorno de verificación UVM	23
2.6.1 Secuencias	24
2.6.2 Componente Agent.....	26
2.6.3 Componente Scoreboard	29
2.6.4 Componente Environment.....	30
2.6.5 Componente Test.....	30
2.6.6 Módulo Top	31
Capítulo 3. Módulo IP multi-interfaz para la compresión de imágenes	33
3.1 Módulo IP.....	33
3.2 Entorno de verificación (test bench) original.....	37

3.2.1	Generación de los estímulos y los archivos de simulación.....	37
3.2.2	Generación de las señales principales	42
3.2.3	Referencia de los componentes del test bench.....	43
3.2.4	Proceso de configuración del IP.....	46
3.2.5	Proceso de transferencia del flujo de datos de entrada	50
3.2.6	Proceso de recepción del flujo de datos de salida y comparación con los valores de referencia.....	58
3.2.7	Ejecución de los test	62
Capítulo 4. Desarrollo del entorno de verificación basado en UVM		65
4.1	Generación de un IP usando HDL Coder	65
4.2	Integración de un DUV descrito en VHDL	73
4.3	Creación de un wrapper como nuevo DUV.....	74
4.4	Paquetes con definición de constantes	75
4.5	Estructura del entorno de verificación del IP121 basado en UVM	78
4.5.1	Interfaces virtuales	80
4.5.2	Módulo Top	81
4.5.3	Transacciones	84
4.5.4	Secuencias	86
4.5.5	Componente Test.....	89
4.5.6	Componente Environment.....	93
4.5.7	Componente Agent de configuración	94
4.5.8	Componente Agent de datos.....	103
4.5.9	Componente Scoreboard	114
4.5.10	Makefile.....	118
4.6	Estructura de directorios	120
4.7	Flujo de ejecución.....	121
Capítulo 5. Resultados.....		123
5.1	Salida de texto por consola	123
5.1.1	Test normal.....	124
5.1.2	Test de Error	127
5.1.3	Test de ForceStop	129
5.1.4	Test de reconfiguración.....	130
5.2	Formas de onda	133
5.3	Tiempo de ejecución.....	134
Capítulo 6. Conclusiones		135
6.1	Conclusiones generales.....	135

6.2 Líneas futuras	137
Referencias.....	139
PLIEGO DE CONDICIONES	
Pliego de condiciones.....	143
PRESUPUESTO	
Presupuesto	147
Recursos humanos	147
Recursos hardware	147
Recursos software.....	148
Material fungible.....	148
Coste total del proyecto	149
ANEXOS	
Anexo – Contenido del CD-ROM.....	153

ÍNDICE DE FIGURAS

<i>Figura 1.1. Evolución de las metodologías previas en las que se basa UVM.....</i>	<i>2</i>
<i>Figura 2.1. Diagrama de técnicas de verificación</i>	<i>6</i>
<i>Figura 2.2. Proceso genérico de diseño y verificación funcional.....</i>	<i>7</i>
<i>Figura 2.3. Tendencias en lenguajes de verificación ASIC/IC.....</i>	<i>8</i>
<i>Figura 2.4. Funcionalidades de SystemVerilog</i>	<i>9</i>
<i>Figura 2.5. Tendencias en metodologías de verificación ASIC/IC</i>	<i>12</i>
<i>Figura 2.6. Jerarquía parcial de la biblioteca de clases de UVM</i>	<i>13</i>
<i>Figura 2.7. Fases de UVM y orden de ejecución</i>	<i>15</i>
<i>Figura 2.8. Clases para la mensajería</i>	<i>19</i>
<i>Figura 2.9. Comunicación TLM port – TLM export.....</i>	<i>22</i>
<i>Figura 2.10. Comunicación TLM analysis port – TLM analysis exports.....</i>	<i>22</i>
<i>Figura 2.11. Modelo básico de entorno de verificación UVM</i>	<i>23</i>
<i>Figura 2.12. Tareas en la ejecución de una secuencia</i>	<i>25</i>
<i>Figura 2.13. Agente UVM.....</i>	<i>26</i>
<i>Figura 2.14. Protocolo de handshake Sequencer-Driver.....</i>	<i>27</i>
<i>Figura 3.1. Diagrama de entradas/salidas del IP.....</i>	<i>34</i>
<i>Figura 3.2. Diagrama de bloques de la estructura del test bench</i>	<i>45</i>
<i>Figura 3.3. Diagrama de la máquina de estados original para el flujo de datos de entrada.....</i>	<i>57</i>
<i>Figura 3.4. Bibliotecas de GRLIB IP correctamente enlazadas a QuestaSim</i>	<i>63</i>
<i>Figura 4.1. Resultado de la ejecución del test bench mlhdlc_sobelfilter_tb.m.....</i>	<i>68</i>
<i>Figura 4.2. Ventana de un proyecto en HDL Coder.....</i>	<i>70</i>
<i>Figura 4.3. Ventana del Workflow Advisor de HDL Coder.....</i>	<i>71</i>
<i>Figura 4.4. Selección del lenguaje para la generación del código HDL</i>	<i>71</i>
<i>Figura 4.5. Formas de onda de la simulación en QuestaSim del IP generado en HDL Coder</i>	<i>73</i>
<i>Figura 4.6. Diagrama de bloques de la entidad VHDL ccstds121_duv.....</i>	<i>74</i>
<i>Figura 4.7. Estructura del entorno UVM específico creado.....</i>	<i>79</i>
<i>Figura 4.8. Diagrama de la nueva máquina de estados para el flujo de datos de entrada</i>	<i>106</i>
<i>Figura 5.1. Formas de onda en el comienzo de la recepción del flujo de datos de salida.....</i>	<i>134</i>

ÍNDICE DE CÓDIGOS

<i>Código 3.1. Entidad VHDL del IP</i>	35
<i>Código 3.2. Ayuda de ejecución del script <code>run_vhdl_tests_121.py</code></i>	38
<i>Código 3.3. Paquete <code>ccsds121_parameters</code> para el Test 25</i>	39
<i>Código 3.4. Paquete <code>ccsds121_tb_parameters</code> para el Test 25</i>	41
<i>Código 3.5. Generación de las señales principales del test bench</i>	43
<i>Código 3.6. Referencia, en el test bench, del IP y los componentes para la comunicación AHB</i>	45
<i>Código 3.7. Proceso general de configuración del IP</i>	47
<i>Código 3.8. Procedimiento de ejecución del test0 (dos compresiones consecutivas)</i>	49
<i>Código 3.9. Proceso síncrono de la máquina de estados para el flujo de datos de entrada</i>	51
<i>Código 3.10. Proceso asíncrono de la máquina de estados para el flujo de datos de entrada</i>	57
<i>Código 3.11. Proceso de recepción del flujo de datos de salida</i>	60
<i>Código 3.12. Procedimiento de comparación de los datos de salida con sus valores de referencia</i>	62
<i>Código 3.13. Comando de ejecución de los test en QuestaSim</i>	63
<i>Código 4.1. Función MATLAB <code>mlhdlc_sobelfilter</code></i>	67
<i>Código 4.2. Archivo <code>mlhdlc_sobelfilter_tb.m</code></i>	68
<i>Código 4.3. Almacenamiento de la imagen <code>mlhdlc_img_yuv.tif</code> como matriz de datos</i>	69
<i>Código 4.4. Modificación del modo de lectura de la imagen en el test bench</i>	69
<i>Código 4.5. Código a eliminar del test bench</i>	70
<i>Código 4.6. Simulación en QuestaSim del código VHDL generado en HDL Coder</i>	72
<i>Código 4.7. Comando para la supresión de warnings en la simulación con un DUV VHDL</i>	73
<i>Código 4.8. Entidad VHDL del nuevo DUV <code>ccsds121_duv</code></i>	75
<i>Código 4.9. Ayuda de ejecución del script <code>gen_config_tests_121.py</code></i>	76
<i>Código 4.10. Paquete SystemVerilog <code>ccsds121_tb_parameters</code> para el Test 25</i>	78
<i>Código 4.11. Interfaz <code>ccsds121_config_if</code></i>	80
<i>Código 4.12. Interfaz <code>ccsds121_data_if</code></i>	81
<i>Código 4.13. Archivo <code>ccsds121_top.sv</code></i>	83
<i>Código 4.14. Archivo <code>ccsds121_pkg.sv</code></i>	84
<i>Código 4.15. Transacción <code>ccsds121_config_packet</code></i>	85
<i>Código 4.16. Transacción <code>ccsds121_data_packet</code></i>	86
<i>Código 4.17. Secuencia <code>ccsds121_config_sequence</code></i>	87
<i>Código 4.18. Secuencia <code>ccsds121_data_sequence</code></i>	89

<i>Código 4.19. Componente <code>ccsds121_base_test</code>.....</i>	<i>91</i>
<i>Código 4.20. Componentes UVM Test asociados a la funcionalidad de cada test</i>	<i>93</i>
<i>Código 4.21. Componente <code>ccsds121_env</code>.....</i>	<i>94</i>
<i>Código 4.22. Componente <code>ccsds121_config_agent</code>.....</i>	<i>95</i>
<i>Código 4.23. Componente <code>ccsds121_config_sequencer</code>.....</i>	<i>95</i>
<i>Código 4.24. Funciones principales del componente <code>ccsds121_config_driver</code>.....</i>	<i>97</i>
<i>Código 4.25. Tarea <code>test</code>.....</i>	<i>99</i>
<i>Código 4.26. Componente <code>ccsds121_config_monitor</code>.....</i>	<i>102</i>
<i>Código 4.27. Componente <code>ccsds121_data_agent</code>.....</i>	<i>103</i>
<i>Código 4.28. Componente <code>ccsds121_data_sequencer</code>.....</i>	<i>104</i>
<i>Código 4.29. Funciones principales del componente <code>ccsds121_data_driver</code>.....</i>	<i>105</i>
<i>Código 4.30. Tarea <code>reset</code>.....</i>	<i>105</i>
<i>Código 4.31. Tarea <code>drive_inputs</code>.....</i>	<i>111</i>
<i>Código 4.32. Componente <code>ccsds121_data_monitor</code>.....</i>	<i>113</i>
<i>Código 4.33. Componente <code>ccsds121_scoreboard</code>.....</i>	<i>117</i>
<i>Código 4.34. Contenido principal del archivo <code>Makefile</code>.....</i>	<i>119</i>
<i>Código 4.35. Generación de los archivos con los parámetros de configuración</i>	<i>121</i>
<i>Código 4.36. Ejecución de dos test sobre el entorno de verificación creado</i>	<i>121</i>
<i>Código 5.1. Mensajes durante la ejecución de la simulación del test <code>ccsds121_normal_test</code></i>	<i>126</i>
<i>Código 5.2. Mensajes recibidos en la fase <code>run_phase</code> del test <code>ccsds121_error_test</code>....</i>	<i>129</i>
<i>Código 5.3. Mensajes recibidos en la fase <code>run_phase</code> del test <code>ccsds121_forcestop_test</code></i>	<i>130</i>
<i>Código 5.4. Mensajes recibidos en la fase <code>run_phase</code> del test <code>ccsds121_reconfig_test</code></i>	<i>133</i>

ÍNDICE DE TABLAS

<i>Tabla 2.1. Acciones por defecto en las macros del mecanismo de mensajes.....</i>	<i>19</i>
<i>Tabla 2.2. Valores de verbosity.....</i>	<i>20</i>
<i>Tabla PL.1. Condiciones hardware.....</i>	<i>143</i>
<i>Tabla PL.2. Condiciones software.....</i>	<i>143</i>
<i>Tabla P.1. Coste de recursos hardware.....</i>	<i>148</i>
<i>Tabla P.2. Coste de recursos software.....</i>	<i>148</i>
<i>Tabla P.3. Coste de material fungible.....</i>	<i>148</i>
<i>Tabla P.4. Coste total.....</i>	<i>149</i>

ACRÓNIMOS

ABV	<i>Assertion-Based Verification</i>
AHB	<i>Advanced High-performance Bus</i>
AMBA	<i>Advanced Microcontroller Bus Architecture</i>
API	<i>Application Programming Interface</i>
ASIC	<i>Application-Specific Integrated Circuit</i>
AVM	<i>Advanced Verification Methodology</i>
CCSDS	<i>Consultative Committee for Space Data Systems</i>
CD-ROM	<i>Compact Disc Read-Only Memory</i>
CPU	<i>Central Processing Unit</i>
CSV	<i>Comma-Separated Values</i>
DPI	<i>Direct Programming Interface</i>
DUT	<i>Device Under Test</i>
DUV	<i>Device Under Verification</i>
ECTS	<i>European Credit Transfer and Accumulation System</i>
EDA	<i>Electronic Design Automation</i>
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
eRM	<i>e Reuse Methodology</i>
ESA	<i>European Space Agency</i>
FIFO	<i>First In, First Out</i>
FPGA	<i>Field-Programmable Gate Array</i>
HDL	<i>Hardware Description Language</i>
HVL	<i>Hardware Verification Language</i>
IC	<i>Integrated Circuit</i>
IGIC	Impuesto General Indirecto Canario
IP	<i>Intellectual Property</i>
IUMA	Instituto Universitario de Microelectrónica Aplicada
MDV	<i>Metric-Driven Verification</i>
MUIT	Máster Universitario en Ingeniería de Telecomunicación

OOP	<i>Object-Oriented Programming</i>
OVM	<i>Open Verification Methodology</i>
PDF	<i>Portable Document Format</i>
RAL	<i>Register Abstraction Layer</i>
RRHH	Recursos Humanos
RTL	<i>Register-Transfer Level</i>
TFM	Trabajo Fin de Máster
TLM	<i>Transaction-Level Modeling</i>
ULPGC	Universidad de Las Palmas de Gran Canaria
UVC	<i>UVM Verification Component</i>
UVM	<i>Universal Verification Methodology</i>
VHDL	<i>VHSIC Hardware Description Language</i>
VHSIC	<i>Very High Speed Integrated Circuit</i>
VMM	<i>Verification Methodology Manual</i>
VMM-RAL	<i>VMM - Register Abstraction Layer</i>
VP	<i>Verification Plan</i>
V&V	<i>Verification and Validation</i>

MEMORIA

Capítulo 1. INTRODUCCIÓN

En este capítulo se detallarán los antecedentes y las necesidades que dan lugar a la realización de este Trabajo Fin de Máster (TFM), así como los objetivos inicialmente planteados y la estructura del presente documento.

1.1 ANTECEDENTES

En la actualidad, la verificación funcional de un sistema hardware digital es un proceso de gran relevancia. Muchos de los sistemas desarrollados hoy en día no son entregados al cliente con evidencias reales de que cumplen con lo dispuesto en el documento de especificaciones funcionales, debido a la dificultad que puede conllevar la realización de un entorno de validación y verificación [1].

Este procedimiento de verificación y validación (V&V, *Verification and Validation*) se utiliza, de modo genérico, para asegurar que un sistema hardware digital satisface las necesidades y especificaciones de diseño, además de no poseer defectos. Por lo general, los sistemas de verificación funcional se dividen en estáticos y dinámicos, siendo estos últimos los más utilizados. Entre las técnicas de verificación dinámica, destaca la simulación, utilizada desde los comienzos del diseño electrónico, y que consiste en la ejecución de diferentes *test* o vectores de estímulos, los cuales pueden seguir patrones totalmente aleatorios o dirigidos [2].

Dada la complejidad de los actuales módulos o bloques funcionales IP (*Intellectual Property*), dicho proceso debe estar presente desde las primeras fases de diseño de un sistema para alcanzar su eficacia óptima, y no relegarse a las últimas etapas de desarrollo [2]. De hecho, la verificación funcional consume más del 60% del tiempo y el esfuerzo en un proyecto de diseño de un sistema hardware digital, por lo que existe una búsqueda constante de nuevas metodologías que proporcionen un mejor rendimiento sobre la verificación funcional [3].

Así, de entre las metodologías existentes para afrontar el proceso de verificación funcional de un componente hardware digital, una de las más recientes es UVM (*Universal Verification Methodology*). El estándar UVM fue publicado en 2011 por *Accellera Systems Initiative* como la primera metodología de verificación promovida por los tres principales desarrolladores de hardware digital en el mercado: *Cadence*, *Synopsys* y *Mentor* (Figura 1.1) [3].

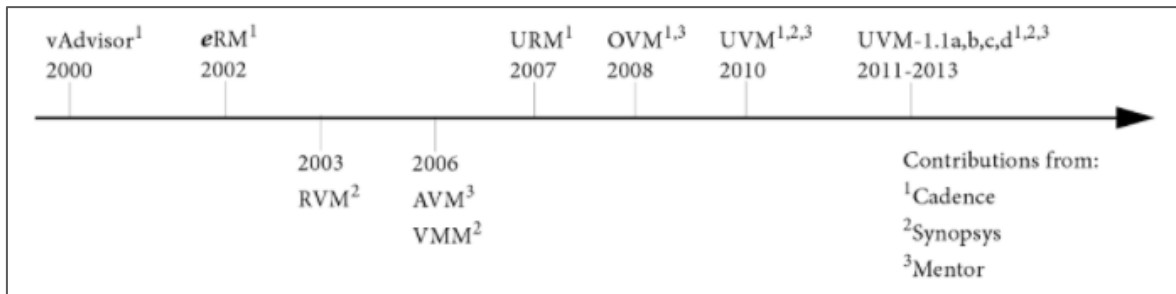


Figura 1.1. Evolución de las metodologías previas en las que se basa UVM

A pesar de su relativa juventud, se trata de una metodología madura, ya que su código está basado en modificaciones efectuadas sobre la biblioteca OVM (*Open Verification Methodology*). Además, UVM es de código abierto, así como compatible y portable a los simuladores comerciales más destacados. Esta metodología se basa en el desarrollo y la integración de componentes independientes de verificación o UVC (*UVM Verification Component*), los cuales son diseñados mediante el lenguaje *SystemVerilog*, y se interconectan entre sí a través del uso de la API TLM (*Transaction-Level Modeling*) con el fin de facilitar su reusabilidad [3].

Las principales ventajas de UVM se fundamentan en la adopción de un flujo de trabajo común y en la reusabilidad, de modo que se incrementa la productividad y la eficiencia. Por otra parte, el entorno de verificación se desarrolla de forma aislada al sistema hardware digital a verificar o DUV (*Device Under Verification*), que en ocasiones también se denominará como DUT (*Device Under Test*). Así, se consigue que los componentes de un entorno de verificación puedan ser reutilizados para validar la funcionalidad de diferentes DUV, o bien que un único entorno permita verificar dos implementaciones de un mismo sistema descrito en diferentes lenguajes HDL (*Hardware Description Language*), únicamente adaptando la interfaz con el DUV [3], [4].

Esto choca con el proceso de verificación funcional tradicional, en el que se sigue una metodología *ad-hoc* basada en un Plan de Verificación (VP, *Verification Plan*) rígido elaborado a partir de las especificaciones funcionales del DUV [5]. Este Plan de Verificación se traduce en entornos de *test* específicos implementados en lenguajes HDL, donde para cada nuevo diseño es necesario implementar un *test bench* o entorno de verificación, prácticamente desde el principio. Por tanto, UVM pretende reducir este esfuerzo a partir de la reutilización de los entornos de verificación desarrollados, permitiendo que sean dinámicos y configurables [6].

Para el caso de bloques IP multi-interfaz, se debe prestar especial atención a la integración y adaptación del DUV al entorno UVM, así como al modo de gestionar las distintas interfaces. Este último aspecto determinará la complejidad del diseño, debido a las dependencias entre las

informaciones transferidas, los diferentes protocolos de comunicaciones, etc. En este TFM en particular se trabajará en esta línea, debido a que se utilizará como DUV un IP específico que integra una interfaz de configuración y otra de datos.

1.2 OBJETIVOS

El objetivo principal que se pretende alcanzar con la realización de este Trabajo Fin de Máster es el de elaborar un entorno basado en UVM para la verificación funcional de un IP multi-interfaz (consta de interfaces independientes de configuración y de datos) cuya funcionalidad consiste en la compresión sin pérdidas de imágenes hiperespectrales y multiespectrales. Este IP fue desarrollado en un proyecto llevado a cabo por el Instituto Universitario de Microelectrónica Aplicada (IUMA), en colaboración con la Agencia Espacial Europea (ESA, *European Space Agency*). Se trata de un IP complejo y altamente configurable, por lo que en el entorno de verificación se requiere automatizar el proceso de ejecución de *test*.

El IP en cuestión está descrito en lenguaje VHDL y posee un entorno de verificación *ad-hoc* tradicional y complejo. El entorno UVM a desarrollar, por su parte, se definirá mediante el lenguaje *SystemVerilog*, por lo que se hace necesaria una adaptación multilenguaje que permita la integración del IP en el mismo. Se persigue que este nuevo entorno UVM sea más fácil de interpretar, además de reutilizable, características inexistentes en el *test bench* de referencia.

Una vez establecido el objetivo principal del TFM, se enumeran a continuación los objetivos operativos que se plantean para su desarrollo:

- Adquirir los conocimientos relativos a la metodología UVM: estructura de un entorno básico, diferentes elementos, componentes e interconexiones, etc.
- Implementar un entorno UVM básico y adaptarlo para comprobar su correcto funcionamiento mediante la verificación de un IP sencillo, independiente del IP específico sobre el que se centra el presente TFM (por ejemplo, un IP generado a partir de una funcionalidad desarrollada por un tercero en *MATLAB*).
- Estudiar y analizar el funcionamiento del IP multi-interfaz orientado a la compresión de archivos de imagen (a nivel del protocolo de comunicación utilizado en sus interfaces de configuración y de datos), así como de su *test bench* original, lo que servirá de referencia para la generación del entorno de verificación basado en UVM.
- Desarrollar un entorno UVM para la verificación del IP multi-interfaz específico. En este caso, se reutilizarán aspectos del entorno de verificación UVM básico previamente

implementado. Una vez finalizado el desarrollo del entorno basado en UVM, se generarán una serie de *test* de verificación de la funcionalidad básica del IP.

1.3 PETICIONARIO

Actúa como peticionario del presente Trabajo Fin de Máster la Escuela de Ingeniería de Telecomunicación y Electrónica (EITE) de la Universidad de Las Palmas de Gran Canaria (ULPGC), con el fin de satisfacer los requisitos de la asignatura Trabajo Fin de Máster en el plan de estudios de la titulación Máster Universitario en Ingeniería de Telecomunicación (MUIT).

1.4 ESTRUCTURA DEL DOCUMENTO

El presente documento se encuentra dividido en cuatro partes claramente diferenciadas: Memoria, Pliego de condiciones, Presupuesto y Anexos. La memoria, a su vez, se estructura en seis capítulos y la bibliografía empleada. El contenido de estos capítulos es el que se resume a continuación:

- *Capítulo 1. Introducción.* Este capítulo recoge los antecedentes que han dado lugar a la realización de este TFM, sus objetivos, el peticionario del mismo, y la estructura del documento.
- *Capítulo 2. Universal Verification Methodology.* Este capítulo presenta una introducción al proceso de verificación funcional, así como los aspectos que han propiciado la creación de la metodología UVM. Además, se introduce esta metodología, incluyendo sus principales características, los fundamentos en los que se sustenta, y los componentes que integra.
- *Capítulo 3. Módulo IP multi-interfaz para la compresión de imágenes.* En este capítulo se realiza una descripción detallada del sistema hardware digital que se pretende verificar haciendo uso de la metodología UVM, así como de su *test bench* original.
- *Capítulo 4. Desarrollo del entorno de verificación basado en UVM.* En este capítulo se presenta el entorno UVM desarrollado para la verificación funcional del IP multi-interfaz descrito en el Capítulo 3, así como el procedimiento llevado a cabo para su implementación.
- *Capítulo 5. Resultados.* Este capítulo muestra los resultados obtenidos tras la ejecución de varios *test* sobre el DUV en el entorno de verificación UVM desarrollado.
- *Capítulo 6. Conclusiones.* Tras haber completado los objetivos establecidos en este TFM, en este capítulo se recogen las conclusiones obtenidas a partir de su realización, así como las posibles ampliaciones futuras que puedan surgir a raíz de la línea de trabajo iniciada.

Capítulo 2. UNIVERSAL VERIFICATION METHODOLOGY

En este capítulo se presentan los principales aspectos de la metodología UVM (*Universal Verification Methodology*), utilizada durante el desarrollo del presente TFM. En primer lugar, se realizará una introducción al proceso de verificación y su importancia en el flujo de diseño de sistemas hardware, centrándose particularmente en la verificación funcional. En relación con UVM, se presentará su arquitectura basada en componentes, los fundamentos en los que se basa su propuesta, el lenguaje de verificación hardware que utiliza, etc. Con ello, se asentarán las bases de la metodología con el fin de comprender el porqué de su concepción y uso actual.

2.1 INTRODUCCIÓN A LA VERIFICACIÓN

La verificación es la actividad que determina el correcto funcionamiento de un sistema. En otras palabras, asegura que un diseño cumple perfectamente con las especificaciones funcionales establecidas. En el ámbito del diseño de sistemas hardware digitales, la verificación se centra, en la mayoría de los casos, en asegurar que las especificaciones de diseño están correctamente implementadas en su descripción a nivel RTL (*Register-Transfer Level*) [7].

Este procedimiento es crucial para determinar si un producto final puede ser utilizado, o no, de modo seguro por sus usuarios potenciales. Si se considerase un escenario en el que un producto fuese comercializado hacia los consumidores finales sin seguridad de que se comporta de forma acorde a sus especificaciones, existiría la posibilidad de que el mismo tuviese un comportamiento erróneo. Ello podría ocasionar pérdidas considerables a la compañía, tanto a nivel económico como a nivel de reputación ya que, como es lógico, un usuario no tiene intención de adquirir productos sin la seguridad de que estos operen correctamente [7].

Es una realidad que, hoy en día, la verificación ha adquirido una importancia muy significativa, hasta el punto de que un equipo de desarrollo típico posee el mismo número de ingenieros de verificación que de diseño. Ello es debido al tamaño y la complejidad de los sistemas hardware actuales, que hacen que el proceso de verificación pueda consumir, aproximadamente, el 70% del esfuerzo de un proyecto (tanto económico como temporal) [8]. Y es que un ingeniero de diseño, por lo general, se ciñe a los casos representativos de las especificaciones, y su trabajo finaliza cuando contempla dichos casos en la implementación. Sin embargo, un ingeniero de verificación debe verificar el diseño bajo todo tipo de casos, los cuales tienden a ser infinitos [7].

Debido a su trascendencia, existen numerosas técnicas de verificación (Figura 2.1) [8], las cuales se dividen en estáticas (verificación formal) y dinámicas (verificación funcional, basada en simulación y emulación).

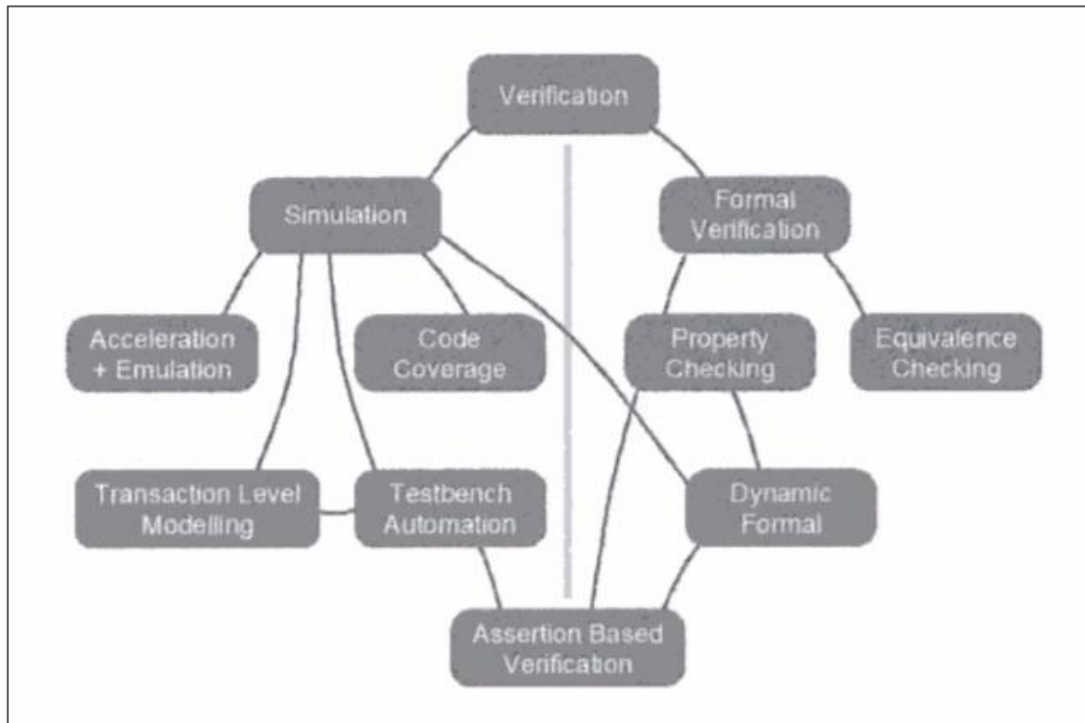


Figura 2.1. Diagrama de técnicas de verificación

Por un lado, la verificación formal consiste en el uso de técnicas matemáticas (no requiere de simulación) para comprobar que un diseño cumple con sus especificaciones funcionales o, al menos, con una parte de ellas. Por otro, la verificación funcional tiene como objetivo asegurar que un diseño cumple con la funcionalidad para la que fue implementado [8], [9]. En el caso particular de este TFM, se hará uso de esta última como ámbito de trabajo.

2.2 VERIFICACIÓN FUNCIONAL DE SISTEMAS HARDWARE DIGITALES

La verificación funcional es un tipo de verificación particular que puede probar la existencia de errores funcionales en un diseño, pero no su ausencia, pues ello depende del nivel de cobertura (el porcentaje de casos de funcionalidad comprobados) definido en el Plan de Verificación.

Aunque la verificación formal tiene hoy en día una tendencia creciente, la verificación funcional, especialmente basada en simulación, sigue siendo la más utilizada en la actualidad, debido a la facilidad que ofrece para situar al sistema en múltiples escenarios o casos de uso, sobre todo en los denominados *corner cases*, los cuales son difíciles de alcanzar en situaciones normales [6].

Para aplicar la verificación funcional, en primer lugar, en la etapa de diseño se genera el código RTL del sistema a partir de sus especificaciones funcionales, para posteriormente comprobar que la funcionalidad de este código cumple con las especificaciones inicialmente establecidas [7]. Una descripción gráfica de este procedimiento se muestra en la Figura 2.2.

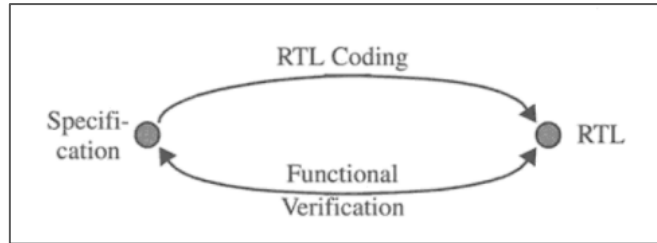


Figura 2.2. Proceso genérico de diseño y verificación funcional

Así, esta verificación se aplica cuando, conociendo los datos de entrada al DUV, se realiza una comparación de las salidas esperadas con las realmente obtenidas desde el mismo. A la hora de implementar la verificación funcional existen tres métodos o enfoques fundamentales: *Black-Box*, *White-Box* y *Grey-Box*, cuyas diferencias principales consisten en el nivel de conocimiento o de abstracción que se tiene del DUV [7], [9].

- **Verificación *Black-Box*:** La verificación funcional se lleva a cabo sin ningún tipo de conocimiento acerca de la implementación del diseño del DUV. Todo el proceso de verificación se realiza a partir de las interfaces disponibles en el DUV, sin acceso directo a sus estados internos, su estructura o su implementación. La decisión de si el resultado de un determinado *test* es correcto, o no, viene dada por la respuesta en las salidas del DUV ante unos determinados estímulos de entrada. La principal ventaja de este modelo es que no depende de ninguna implementación específica del diseño. Sin embargo, en caso de no obtenerse los resultados esperados, es difícil conocer la fuente del problema.
- **Verificación *White-Box*:** En este caso, la verificación funcional permite el acceso y el control absoluto de la estructura interna y la implementación del diseño del DUV. Con ello, se consigue simplificar en gran medida la fase de depuración, permitiendo la posibilidad de localizar el origen de un error con mayor facilidad. Por otra parte, una desventaja de este modelo consiste en la fuerte dependencia con la implementación del diseño, de modo que cualquier modificación interna del DUV llevará asociada una modificación en el *test bench*.
- **Verificación *Grey-Box*:** Este tipo de verificación se encuentra a medio camino entre las dos anteriores, estableciendo un compromiso entre el nivel de abstracción del proceso de

verificación *Black-Box* y el nivel de dependencia con la implementación de la verificación *White-Box*. En otras palabras, la verificación *Grey-Box* únicamente permite el acceso a determinados puntos de la estructura interna y la implementación del diseño del DUV.

Otro aspecto fundamental a tener en cuenta en la verificación funcional es el lenguaje de verificación hardware a emplear (HVL, *Hardware Verification Language*). Según el estudio “*ASIC/IC and FPGA Functional Verification Study*” llevado a cabo por *Wilson Research Group* en el año 2016 [10], en los últimos 10 años el lenguaje *SystemVerilog* ha seguido una tendencia muy positiva, situándose en la actualidad como HVL por excelencia para la verificación ASIC/IC, tal y como se muestra en la Figura 2.3.

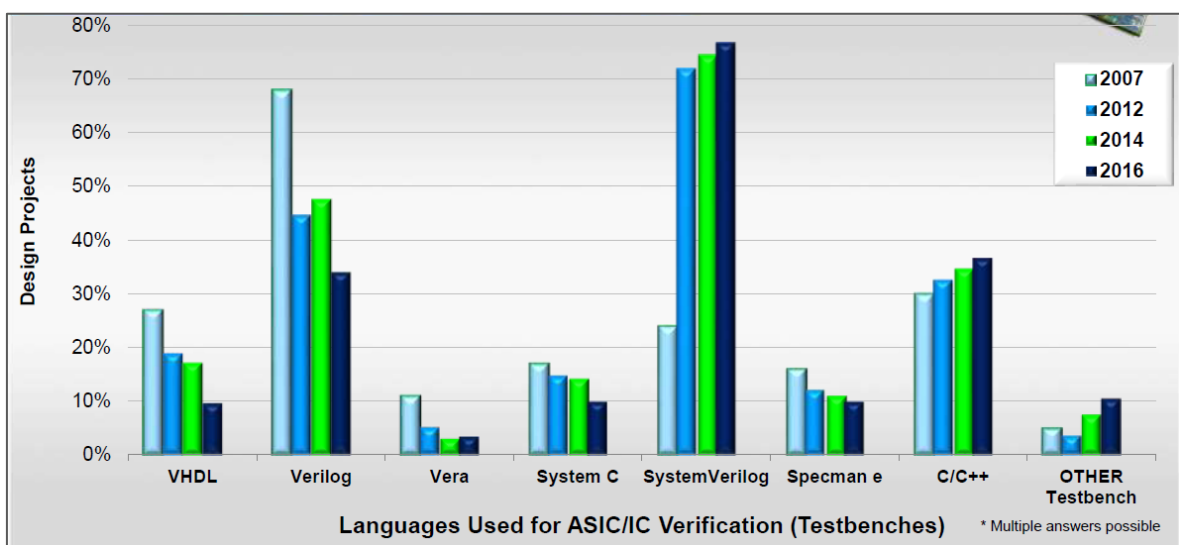


Figura 2.3. Tendencias en lenguajes de verificación ASIC/IC

En el caso particular de los lenguajes utilizados para la verificación FPGA, se observa una tendencia en la misma línea, con *SystemVerilog* imponiéndose a otros lenguajes como VHDL y *Verilog*, los cuales tienen un mayor mercado como lenguajes de diseño hardware. Debido a este hecho, resulta necesario realizar un breve análisis de las características básicas de este lenguaje, que lo han llevado a ser el más utilizado actualmente en el proceso de verificación funcional.

2.3 SYSTEMVERILOG

SystemVerilog es un lenguaje de descripción hardware (HDL), así como un lenguaje de verificación hardware (HVL), que surge desde la necesidad de disponer de un único lenguaje a aplicar, tanto en la etapa de diseño como en la de verificación de un sistema hardware digital. Se trata de una extensión de Verilog-2001, pero que también incluye características propias de otros lenguajes

como VHDL, C y C++ [11]. En la Figura 2.4 se proporciona un desglose de las propiedades más representativas presentes en el lenguaje *SystemVerilog*.

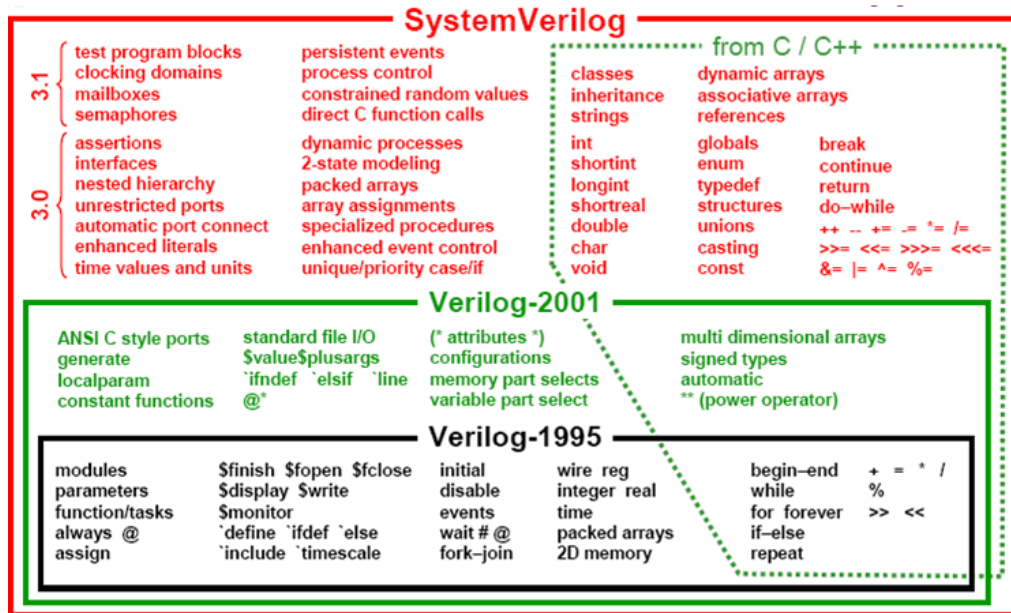


Figura 2.4. Funcionalidades de *SystemVerilog*

Al haber evolucionado desde el lenguaje de descripción hardware *Verilog*, *SystemVerilog* mantiene las características básicas para la implementación de diseños electrónicos. Sin embargo, el verdadero propósito de este lenguaje se encuentra directamente ligado a la verificación [11], como ya pudo observarse en la Figura 2.3. Las principales razones que justifican este hecho son las siguientes [8]:

- El uso de clases, que permite aplicar las técnicas de la programación orientada a objetos (OOP, *Object-Oriented Programming*), una característica muy importante para el desarrollo de entornos de verificación complejos. Una clase es un conjunto de variables y rutinas que definen las características y el comportamiento del objeto a referenciar y, en *SystemVerilog*, son elementos dinámicos que permiten el modelo de herencia simple y que pueden ser parametrizadas (función básica de C++). La herencia de clases permite reutilizar código existente, de modo que una subclase solamente requiera implementar ciertas características adicionales.
- La posibilidad del uso de *assertions* y medidas de cobertura. Un *assertion* no es más que una propiedad que se describe dentro del mismo código del diseño y que se comprueba automáticamente durante toda la fase de simulación. *SystemVerilog* soporta la verificación ABV (*Assertion-Based Verification*), además de definir un mecanismo de cobertura y de adquisición de datos flexible. Esto puede utilizarse, por ejemplo, para comprobar que los

assertions se han cumplido durante la ejecución de *test benches* automáticos utilizando una generación de estímulos aleatoria con restricciones.

- La aleatorización de los datos. Como se ha dejado entrever en el punto anterior, *SystemVerilog* permite la generación de estímulos de forma puramente aleatoria, o bien aleatoria con restricciones.
- La utilización de diferentes tipos de datos, tanto estáticos como dinámicos.
- La posibilidad de usar interfaces para encapsular las comunicaciones. Las interfaces, además de ser un conjunto ordenado de datos, pueden contener comportamientos y utilizarse para describir modelos funcionales de un *bus*. En este sentido, *SystemVerilog* soporta el uso de comunicaciones TLM, lo que también ofrece la posibilidad de reutilización de entornos de verificación en diferentes niveles de abstracción.
- La interfaz DPI (*Direct Programming Interface*), que permite referenciar funciones descritas en C de forma directa en el código *SystemVerilog*.
- El uso de paquetes (*packages*) para compartir código entre distintos módulos. Un paquete incluye declaraciones y definiciones que se agrupan bajo un nombre común, el del propio paquete. Estas declaraciones pueden incluir tipos, constantes, funciones, tareas, clases...

Es por todas estas características que *SystemVerilog* se ha consolidado hoy en día como lenguaje de verificación hardware por excelencia. De ahí que la metodología UVM haga uso del mismo como HVL para la implementación de entornos de verificación.

2.4 INTRODUCCIÓN A UVM

A lo largo de este capítulo se han ido indicando las diferentes necesidades que han surgido en la verificación funcional con la evolución de la complejidad de los sistemas hardware, hasta el punto del establecimiento de *SystemVerilog* como HVL de referencia. Todo ello invita a pensar en la necesidad de disponer de una metodología estándar que proporcione una guía para la aplicación de diversas técnicas de verificación, de modo que se consiga la máxima eficiencia y eficacia posible en el desarrollo de un diseño electrónico.

Es por ello que surge UVM (*Universal Verification Methodology*), un estándar de *Accellera Systems Initiative* que fue desarrollado mediante el trabajo cooperativo de los principales fabricantes y consumidores de herramientas EDA (*Electronic Design Automation*). Todo ello fue posible gracias a la sólida base ya existente en OVM (*Open Verification Methodology*) y a las contribuciones incluidas desde VMM (*Verification Methodology Manual*) [12].

Entrando más en detalle, UVM es un híbrido de tecnologías de diferentes desarrolladores, tomando de cada una de ellas sus características más destacadas [12]:

- AVM (*Advanced Verification Methodology*) de *Mentor*.
- OVM de *Mentor* y *Cadence*.
- eRM (*e Reuse Methodology*) de *Verisity*.
- VMM-RAL (*Verification Methodology Manual - Register Abstraction Layer*) de *Synopsys*.
- *Resources*, *TLM2* y *Phasing*, desarrolladas por *Mentor* específicamente para UVM.

En conjunto, se obtiene una metodología potente y flexible con la que es posible implementar entornos de verificación escalables, reusables e interoperables, cuyas características fundamentales son [3]:

- A través de la metodología y una biblioteca de código, ofrece la posibilidad de dividir de manera limpia el entorno de verificación en un conjunto de ítems de datos (estímulos y respuestas) y componentes de verificación (UVC). Con esto se consigue estructurar y organizar los objetos y las funcionalidades del entorno de verificación de forma más sencilla, así como incrementar su reusabilidad.
- Posee clases e infraestructuras para permitir el control minucioso de los flujos de datos secuenciales a enviar como estímulos al DUV. Además, ofrece un mecanismo de generación de estímulos que puede ser adaptado para incluir transacciones jerárquicas definidas por el usuario, o crear flujos de transacciones.
- Utiliza TLM para facilitar la comunicación entre componentes de verificación descritos en diferentes lenguajes como, por ejemplo, VHDL, *SystemVerilog* o *SystemC*, entre otros.
- Debido a su estructura de clases y su característica de herencia, un buen flujo de trabajo a partir de clases base permite desarrollar entornos de verificación automatizados con una topología jerárquica y reutilizable.
- Proporciona los mecanismos necesarios para llevar a cabo una verificación dirigida por métricas de cobertura (MDV, *Metric-Driven Verification*) sobre los UVC reusables.
- Ofrece capacidades de análisis y depuración a través de mecanismos como el informe de errores, el registro de transacciones, el rastreo de secuencias, etc.

Todas estas características han hecho que, según el estudio [10], y tal y como puede verse en la Figura 2.5, UVM se haya consolidado como la metodología de verificación ASIC/IC más utilizada desde su creación. Al igual que en el caso del lenguaje *SystemVerilog* como HVL, los resultados del

estudio para la verificación FPGA muestran igualmente cómo UVM sigue destacando frente al resto de metodologías del mercado.

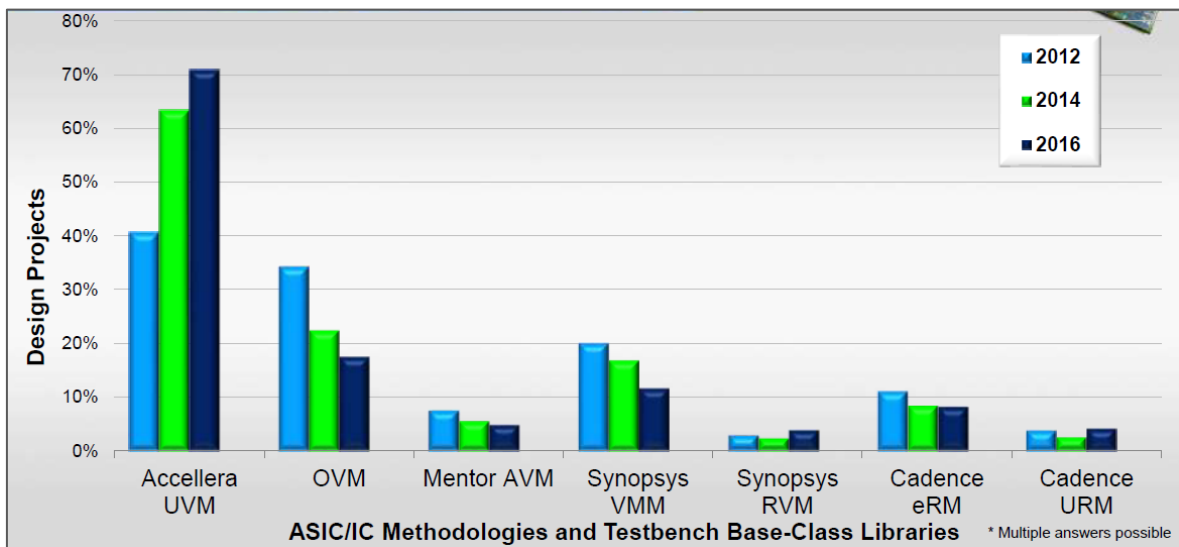


Figura 2.5. Tendencias en metodologías de verificación ASIC/IC

Una vez definidas las principales características que hacen que UVM sea la metodología de verificación más adoptada por los desarrolladores de hardware digital, se procede a continuación a estudiar en detalle los fundamentos en los que se basa la misma, así como los componentes y las funcionalidades que comprende un entorno de verificación UVM.

2.5 FUNDAMENTOS DE UVM

2.5.1 BIBLIOTECA DE CLASES

Desde un punto de vista general, UVM consiste en una biblioteca de clases base, utilidades y macros descritas en *SystemVerilog* que facilita la creación de entornos de verificación estructurados. Estos componentes pueden encapsularse e instanciarse jerárquicamente, dando lugar a un *test bench*, y son controlados a través de un conjunto de fases (*phases*) para inicializar, ejecutar y completar cada uno de los *test* [3]. Todas estas fases, que se definirán posteriormente, se encuentran integradas en la clase base de la biblioteca (`uvm_object`), pero puede extenderse por herencia para ajustarse a los requisitos de cada entorno de verificación.

De entre las diferentes clases incluidas en el paquete UVM pueden distinguirse tres grupos fundamentales, los cuales se describen a continuación [13]. Seguidamente, en la Figura 2.6 se incluye un diagrama de bloques que recoge la jerarquía de los grupos de componentes a describir.

- **uvm_object.** Se trata de la clase base o padre en la jerarquía de clases de UVM, de la cual hereda el resto de clases. Define y automatiza una serie de métodos para la implementación de funcionalidades comunes a todas las clases, así como atributos de identificación de un objeto.
- **uvm_component.** Es la clase padre de todos los UVC a integrar en un entorno de verificación UVM. Es una clase bastante importante, pues incluye todas las funcionalidades de la metodología comunes a los UVC (*Factory*, fases, etc.).
- **uvm_sequence_item** y **uvm_sequence.** Son las clases utilizadas para la generación de estímulos y para agrupar las respuestas del DUV que se desea verificar.

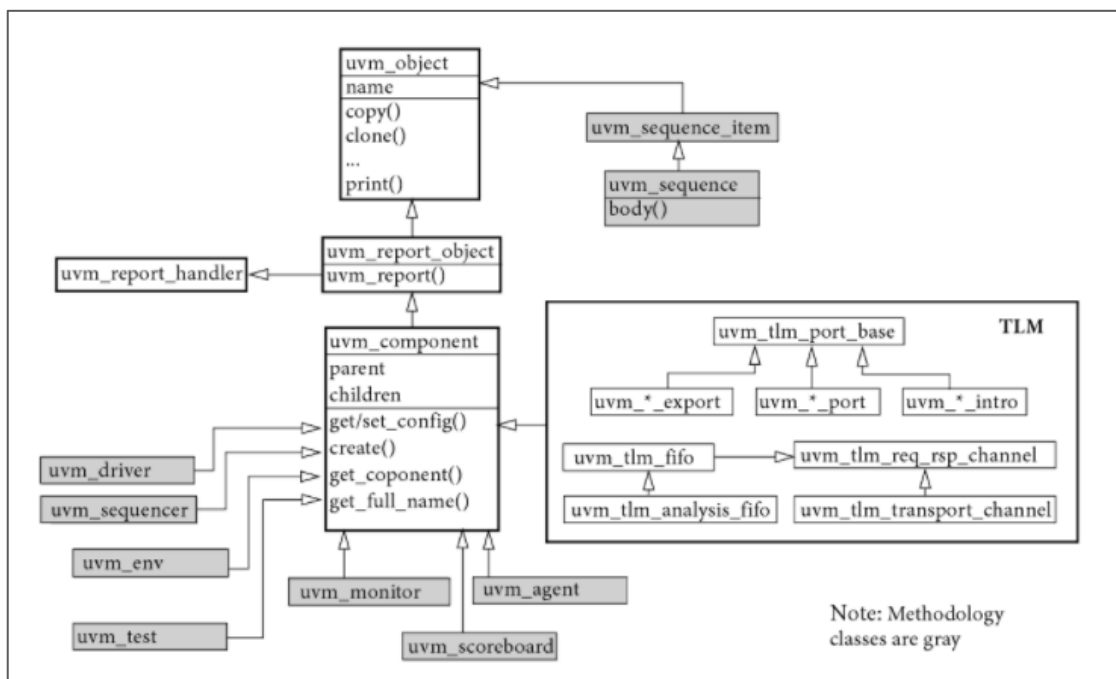


Figura 2.6. Jerarquía parcial de la biblioteca de clases de UVM

Los objetos pertenecientes a las subclases de `uvm_component` son estáticos y forman parte de la jerarquía del *test bench* durante toda la simulación. Los objetos instanciados de las clases `uvm_sequence_item` y `uvm_sequence`, por su parte, son dinámicos y están relacionados con los datos de entrada y de salida del DUV, por lo que se crean, se utilizan, y se descartan en tiempo de ejecución [12].

Tal y como se especifica en la Figura 2.6, las clases de la metodología a las que da acceso UVM son las que se destacan en gris (que serán estudiadas en detalle más adelante), de modo que cada una de ellas ya incluye las funcionalidades básicas propias de cada componente o elemento. De esta manera se aumenta la reusabilidad y se facilita la labor del ingeniero a la hora de crear un entorno

de verificación. Ello es debido a que se evita tener que crear los atributos y las funciones genéricas de cada uno de estos componentes.

Además, UVM también define un paquete para gestionar la comunicación TLM, en el cual se encuentran los diferentes puertos e interfaces para efectuar las comunicaciones entre componentes. Como ya se ha comentado, estas comunicaciones permiten separar el código en componentes, lo que aporta reusabilidad, interoperabilidad y modularidad al diseño del entorno de verificación [3].

La biblioteca de UVM se encuentra contenida en el paquete `uvm_pkg`, que agrupa todas las clases ya citadas. Por tanto, para poder hacer uso de las mismas, este paquete debe ser importado en el módulo *Top* que, como se verá posteriormente, actúa de enlace entre el DUV y el entorno de verificación UVM [14].

2.5.2 FACTORY

De cara a proporcionar una configuración dinámica y flexible, los diferentes componentes que integran un entorno de verificación UVM son registrados en la *Factory*. Este mecanismo se utiliza para crear dichos objetos y componentes, posibilitando jerarquías de componentes dinámicamente configurables, así como sustitución de objetos sin necesidad de modificar el código [14].

La operación de registrar objetos y componentes en la *Factory* de UVM puede efectuarse de diferentes maneras. Sin embargo, la más común consiste en hacer uso de dos macros parametrizables con el nombre del objeto o componente a registrar. Estas macros son ``uvm_object_utils` y ``uvm_component_utils` (en el caso de que el objeto o componente a registrar sea de un tipo parametrizable, se sustituyen por ``uvm_object_param_utils` y ``uvm_component_param_utils`, respectivamente) [15].

Estas macros proporcionan la implementación del método virtual `create()` como nuevo constructor, que reemplaza al tradicional método `new()`. Así, evita crear componentes o transacciones de tipo fijo, sino que los referencia en la *Factory* y devuelve un identificador (*handler*). Con ello, se posibilitan las sustituciones de tipo en tiempo de ejecución, sin tener que acceder internamente a la clase [15]. Otro método importante que proporcionan dichas macros es `get_type_name()`, que retorna el identificador de un objeto o componente y resulta bastante útil en tareas de depuración [14].

En cualquier caso, la definición del constructor `new()` en UVM es obligatoria, al contrario que el uso de la *Factory*. Aunque, como se ha comentado, el uso de la *Factory* y sus macros ofrece unas prestaciones únicas que aportan dinamismo al proceso de verificación.

2.5.3 FASES DE UVM

Una simulación en un entorno de verificación UVM consiste en la ejecución en secuencia de una serie de fases. En *SystemVerilog*, las clases se referencian durante la simulación, por lo que es necesario asegurarse de que el entorno haya sido creado en su totalidad previamente a comenzar con la ejecución de un *test*, o que los diferentes elementos hayan sido contruidos antes de intentar conectarlos. Para ello, en UVM se definen las fases que se muestran en la Figura 2.7, las cuales se dividen en tres grandes grupos y se ejecutan de forma secuencial siguiendo el orden especificado.

La ejecución de estas fases comienza tras la llamada a la función `run_test()`, que inicia la simulación de un determinado *test*, el cual se especifica, bien como parámetro de dicha función, o bien desde la línea de comandos a través de la variable global del entorno `UVM_TESTNAME` [12].

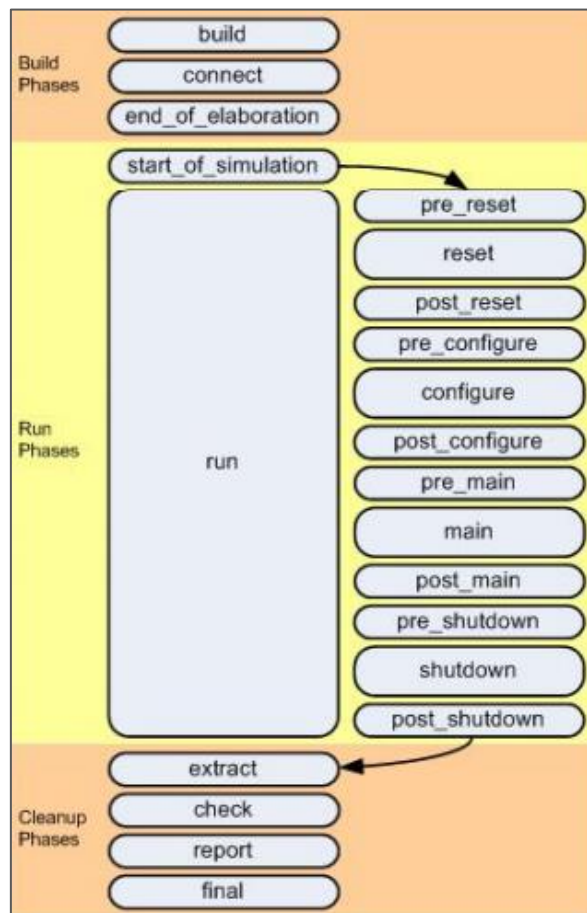


Figura 2.7. Fases de UVM y orden de ejecución

Las fases de construcción o de creación (*build phases*) se ejecutan al comienzo de la simulación UVM y su propósito principal es el de construir, configurar y conectar la jerarquía de componentes del entorno de verificación. Todas estas fases son funciones que se ejecutan en tiempo cero de simulación, es decir, no consumen tiempo [12].

Durante las fases de ejecución (*run phases*) se generan los estímulos y se suceden las comunicaciones de transacciones entre los componentes del entorno, así como la interacción con el DUV. Dentro de este conjunto se encuentra la única fase que está definida como tarea y que consume tiempo de ejecución, la fase `run_phase` [12].

Finalmente, las fases de procesamiento (*clean up phases*) se encargan de extraer la información generada por ciertos componentes, de modo que determinan si la ejecución del *test* ha alcanzado o no los objetivos planteados. Estas fases también son funciones que no consumen tiempo de ejecución [12].

Cabe destacar que la clase `uvm_component` hereda los métodos y atributos de la clase `uvm_report_object`, que define un método virtual para cada una de las fases de UVM. Por lo tanto, si alguna de estas fases no está incluida en la clase base de un componente determinado, este no participará durante la ejecución de la misma. De esta forma, se consigue reducir el tiempo y la carga de procesamiento en la CPU (*Central Processing Unit*).

En un entorno de verificación común, las fases más importantes y utilizadas son las siguientes [14]:

- **`build_phase()`**. En esta fase se construye la jerarquía de componentes del entorno de verificación siguiendo una filosofía *top-down*, desde el nivel más alto de la jerarquía hasta los más bajos.
- **`connect_phase()`**. En esta fase se realiza el conexionado de los componentes incluidos en el nivel de jerarquía actual. En este caso, se sigue una filosofía *bottom-up*, desde los componentes de los niveles más bajos de la jerarquía hasta el nivel más alto.
- **`run_phase()`**. En esta fase se describe el comportamiento de cada uno de los componentes que constituyen el entorno de verificación. Todos los componentes implementan esta fase de forma paralela en el tiempo. Como puede verse en la Figura 2.7, esta fase está compuesta por múltiples sub-fases, si bien para un *test* de complejidad baja o media estas sub-fases no resultan de utilidad, por lo que el código correspondiente se escribe directamente bajo la denominación de `run_phase`. Como se trata de la única fase que consume tiempo de ejecución, resulta necesario determinar cuándo ha finalizado, para lo que se utiliza el mecanismo de *objections*. Este mecanismo contabiliza el número de

componentes y secuencias que aún se encuentran participando en el *test bench*, para lo que define tres métodos:

- **raise_objection()**. Señaliza el momento en el que se inicia la ejecución de la fase `run_phase` en un componente.
- **drop_objection()**. Señaliza el momento en el que finaliza la fase `run_phase` en un componente. Una vez invocado este método en todos los componentes que hayan utilizado anteriormente el método `raise_objection`, se da por finalizada la simulación y se prepara la ejecución de las fases de procesamiento.
- **set_drain_time()**. Establece un intervalo de tiempo a esperar de cara a finalizar la simulación una vez todos los componentes hayan efectuado la llamada al método `drop_objection`.

Otras fases de gran utilidad pueden ser las siguientes:

- **end_of_elaboration_phase() / start_of_simulation_phase()**. En estas fases se puede mostrar en pantalla ciertos aspectos tras las fases de creación del *test bench*, como puede ser información acerca de la topología final de la jerarquía del entorno o información de configuración.
- **report_phase()**. Esta fase es utilizada para mostrar en consola los resultados de la simulación, o bien escribirlos en un archivo externo.

2.5.4 MECANISMO DE CONFIGURACIÓN

UVM proporciona un mecanismo de configuración flexible a través de una base de datos con el fin de permitir la configuración de ciertas características de un componente sin necesidad de hacerlo de forma estática, o bien utilizar la *Factory* como intermediario. Ello otorga reusabilidad a los componentes de verificación o UVC, ya que pueden configurarse componentes genéricos para un modo de operación específico en tiempo de ejecución.

Este mecanismo se basa en la clase `uvm_config_db`, que contiene una interfaz simplificada de acceso a los objetos de configuración que han sido creados en la base de datos. Haciendo uso de los métodos `set()` y `get()`, es posible leer o escribir los parámetros de configuración que se deseen para determinados objetos. Estas funciones son estáticas, por lo que deben invocarse mediante el operador “::” [15]. La definición de estos métodos es la siguiente [14]:

- `uvm_config_db#(T)::set(context, inst_name, field_name, value)`. Permite incluir un nuevo elemento en la base de datos, o bien actualizar uno ya existente.
- `uvm_config_db#(T)::get(context, inst_name, field_name, value)`. Permite recuperar un elemento de la base de datos, siempre y cuando dicho elemento se encuentre en ella.

En ambos casos, el argumento `T` indica el tipo de dato a incluir o recuperar, pudiendo utilizar incluso tipos definidos por el usuario. En cuanto a los argumentos de ambos métodos [13], [14]:

- El contexto (`context`) indica el punto de entrada de la jerarquía desde el cual es accesible el objeto de la base de datos. En caso de utilizarse el valor `this`, hace referencia al componente actual en el que se ejecuta el método. Si el contexto es `null` o `uvm_rot::get()`, es aplicable a todo el entorno.
- El nombre del componente (`inst_name`) es un *String* que indica la ruta jerárquica desde el contexto hasta el nombre que se le dio al componente sobre el que se desea actuar.
- El nombre del campo (`field_name`) es un *String* que actúa como identificador o etiqueta del objeto en la base de datos.
- El campo valor (`value`) hace referencia al valor que se desea aplicar en el caso del método `set()`, o a la variable en la que se desea recibir el valor, en el caso del método `get()`.

2.5.5 MECANISMO DE MENSAJES

Un aspecto básico de un entorno de verificación consiste en poder hacer uso de un mecanismo de mensajes de información o de errores. Como se observa en la Figura 2.8, todos los componentes heredan de la clase `uvm_report_object`, la cual ofrece una interfaz que posibilita el uso de este tipo de mensajería.

Esta interfaz se basa en delegar las tareas de comunicación en una clase interna, la clase `uvm_report_handler`. Dicha clase almacena la configuración del mecanismo de mensajes y, en base a ello, decide qué mensajes mostrar al usuario y cuáles no. En caso de que un mensaje deba ser mostrado, esa tarea se delega, a su vez, en la clase `uvm_report_server`, que se encarga de generar los mensajes conforme a un formato dado [14].

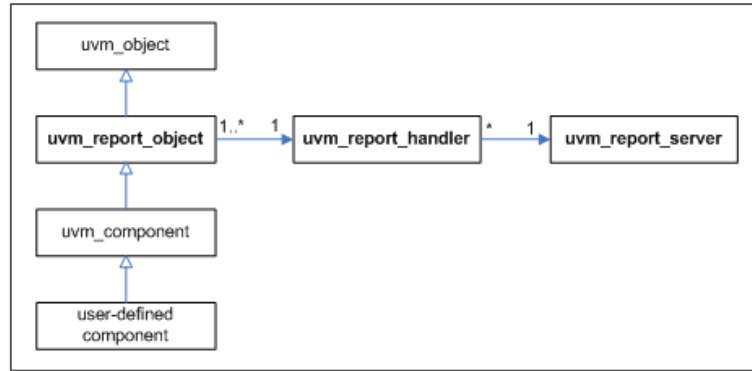


Figura 2.8. Clases para la mensajería

Este mecanismo de mensajes contiene cuatro macros, a partir de las cuales se puede controlar el entorno de verificación. Estas macros son las que se listan a continuación. Además, en la Tabla 2.1 se detallan las acciones que implica cada una de ellas [3], [14].

- ``uvm_info(string id, string message, int verbosity)`.
- ``uvm_warning(string id, string message)`.
- ``uvm_error(string id, string message)`.
- ``uvm_fatal(string id, string message)`.

Tabla 2.1. Acciones por defecto en las macros del mecanismo de mensajes

TIPO	ACCIÓN POR DEFECTO	DESCRIPCIÓN
UVM_INFO	UVM_DISPLAY	Envía un mensaje por la salida estándar (consola).
UVM_WARNING	UVM_DISPLAY	Envía un mensaje por la salida estándar (consola).
UVM_ERROR	UVM_DISPLAY UVM_COUNT	Envía un mensaje por la salida estándar (consola). Finaliza la simulación si el número de mensajes de error supera un número especificado de errores (el valor por defecto es cero).
UVM_FATAL	UVM_DISPLAY UVM_EXIT	Envía un mensaje por la salida estándar (consola). Finaliza la simulación inmediatamente.

En cuanto a la estructura de las macros, el primer argumento es una etiqueta que suele utilizarse como filtro, mientras que el segundo se corresponde con el mensaje de información que se desea imprimir por la salida estándar (consola). Finalmente, en el caso de la macro ``uvm_info`, se debe indicar un valor de *verbosity* como último argumento.

El argumento *verbosity* indica el nivel de importancia del mensaje y, dependiendo del valor que se haya asignado a la variable global del entorno `UVM_VERBOSITY`, el mensaje se mostrará o no.

Este nivel de importancia es de carácter inverso al valor de esta variable, es decir, que será mayor cuanto menor sea el valor de *verbosity*. UVM discretiza los valores de *verbosity* que se pueden tomar, los cuales se recogen en la Tabla 2.2 [3].

Tabla 2.2. Valores de *verbosity*

VERBOSITY	VALOR
UVM_NONE	0
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

De este modo, si la variable global `UVM_VERBOSITY` toma el valor `UVM_DEBUG`, se mostrarán todos los mensajes ``uvm_info` introducidos en el entorno. Por otra parte, si a dicha variable se le asigna el valor `UVM_NONE`, solamente se mostrarán los mensajes ``uvm_info` cuyo argumento *verbosity* sea `UVM_NONE`. Al igual que el *test* a ejecutar en el entorno de verificación UVM, el valor de esta variable puede especificarse, bien localmente, a través de la función `set_report_verbosity_level()`, o bien desde línea de comandos.

2.5.6 TLM

Una de las claves para la productividad en la verificación es conseguir enfocar los problemas desde un nivel de abstracción apropiado. La interfaz con el DUV se representa a nivel de señales, pero en el caso de un entorno de verificación complejo, las transferencias de datos que requiere el diseño son muy diversas y complejas. Por lo tanto, el nivel de abstracción debe ser bastante elevado, a nivel de transacción. Para ello, UVM proporciona un conjunto de canales e interfaces de comunicación TLM (*Transaction-Level Modeling*), que permiten conectar los diferentes componentes del entorno de verificación a nivel de transacciones [15].

En UVM, TLM permite comunicar componentes descritos a diferentes niveles de abstracción y que implementen una misma interfaz. Como pudo verse en la Figura 2.6, la biblioteca de clases de UVM describe, en su jerarquía, un subconjunto para TLM en el que se incluyen una serie de interfaces y puertos con el objetivo de posibilitar dichas comunicaciones. Cada interfaz TLM consiste en uno o varios métodos que se utilizan para intercambiar datos, normalmente transacciones.

El uso de TLM proporciona un conjunto de beneficios con respecto a trabajar con comunicaciones a nivel de señales [3]:

- Los modelos TLM son más concisos y rápidos de simular que los modelos RTL.
- Los modelos TLM se sitúan en un mayor nivel de abstracción, que se acerca mucho más al nivel de abstracción deseado por los ingenieros de diseño o de verificación. De este modo, los modelos son más fáciles de escribir y de entender para desarrollos en equipo.
- Los modelos TLM tienden a ser más reusables ya que, además de ofrecer el uso de técnicas de la programación orientada a objetos, se extraen los detalles que dificultan la reusabilidad fuera de estos modelos.

2.5.6.1 TLM PORTS Y TLM EXPORTS

De cara a enviar y recibir transacciones de datos, UVM hace uso de *TLM ports* y *TLM exports*. Los primeros se encargan de especificar el conjunto de métodos que pueden ser utilizados y de iniciar las peticiones de transacción, mientras que los segundos proporcionan la implementación de dichos métodos. Estos puertos deben conectarse a un único puerto en la fase de construcción del entorno, normalmente en la fase `connect_phase`, tras la creación de los componentes y previamente a la simulación. Dicha conexión se implementa a través del método `connect()`, que es invocado por los *TLM ports* [3].

Atendiendo a su representación gráfica, TLM incluye una notación para diferenciar los tipos de comunicación. Así, los *TLM ports* se representan gráficamente con un cuadrado en la comunicación, mientras que los *TLM exports* se identifican mediante un círculo.

La Figura 2.9 muestra un ejemplo de comunicación de transacciones desde un componente de tipo *Producer*, a otro de tipo *Consumer*, mediante un *TLM port* y un *TLM export*. En el ejemplo a), el *Producer* inicia el envío de transacciones haciendo uso de un método `put()`, mientras que en el ejemplo b), el *Consumer* solicita la recepción de transacciones mediante un método `get()` [3], [15].

Además, también sería posible incluir una cola entre ambos componentes, por ejemplo, de tipo FIFO (*First In, First Out*). De este modo, la cola implementaría los *TLM exports* y tanto *Producer* como *Consumer* podrían hacer uso de los métodos `put()` y `get()`, desde sus *TLM ports*, sobre la FIFO, que es lo que sucede en el ejemplo c).

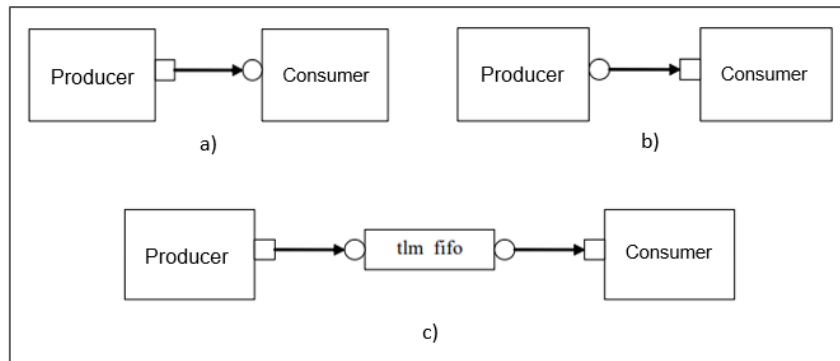


Figura 2.9. Comunicación TLM port – TLM export

2.5.6.2 TLM ANALYSIS PORTS Y TLM ANALYSIS EXPORTS

En ocasiones, la complejidad del entorno de verificación hace que cierta información deba ser distribuida al resto de componentes. Por este motivo surgen los *TLM analysis ports* y los *TLM analysis exports*. A diferencia de los *TLM ports* y los *TLM exports*, en los que la comunicación se establece forzosamente entre dos componentes, este nuevo tipo de puertos permite que un *TLM analysis port* tenga varios *TLM analysis exports* conectados al mismo, o incluso ninguno. En este caso, únicamente se implementa un método `write()`, que consiste en efectuar una transmisión *broadcast* a todos los *TLM analysis exports* que hayan decidido conectarse (suscriptores) [3], [12], [15].

Como puede verse en la Figura 2.10, el modo de representar gráficamente un *TLM analysis port* es mediante un rombo. En dicha figura, se encuentra un componente de tipo *Consumer* que retransmite las transacciones recibidas a través de su *TLM analysis port*, al que se encuentran conectados dos componentes de tipo *Subscriber*.

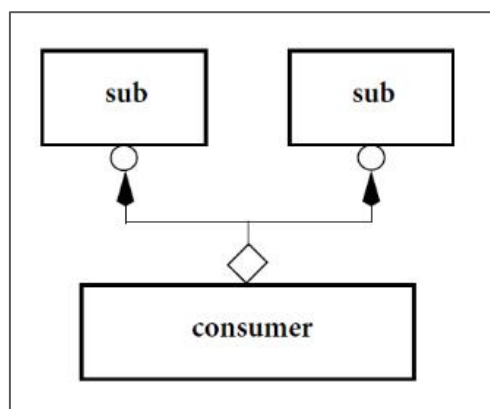


Figura 2.10. Comunicación TLM analysis port – TLM analysis exports

2.6 ESTRUCTURA DE UN ENTORNO DE VERIFICACIÓN UVM

Un *test bench* UVM sigue el planteamiento introducido acerca de la verificación funcional, pero ofrece la posibilidad de tener un mayor control sobre los siguientes procesos:

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

Un entorno de verificación UVM se crea a partir de la instanciación de componentes de todas y cada una de las clases que heredan de `uvm_component`, las cuales se presentaron en el apartado 2.5.1 Biblioteca de clases. La jerarquía se determina a partir de la relación entre estos componentes, de forma que unos son instanciados dentro de otros. Esta jerarquía puede observarse en la Figura 2.11, con un modelo de referencia básico de lo que sería un entorno de verificación de un IP mediante la metodología UVM.

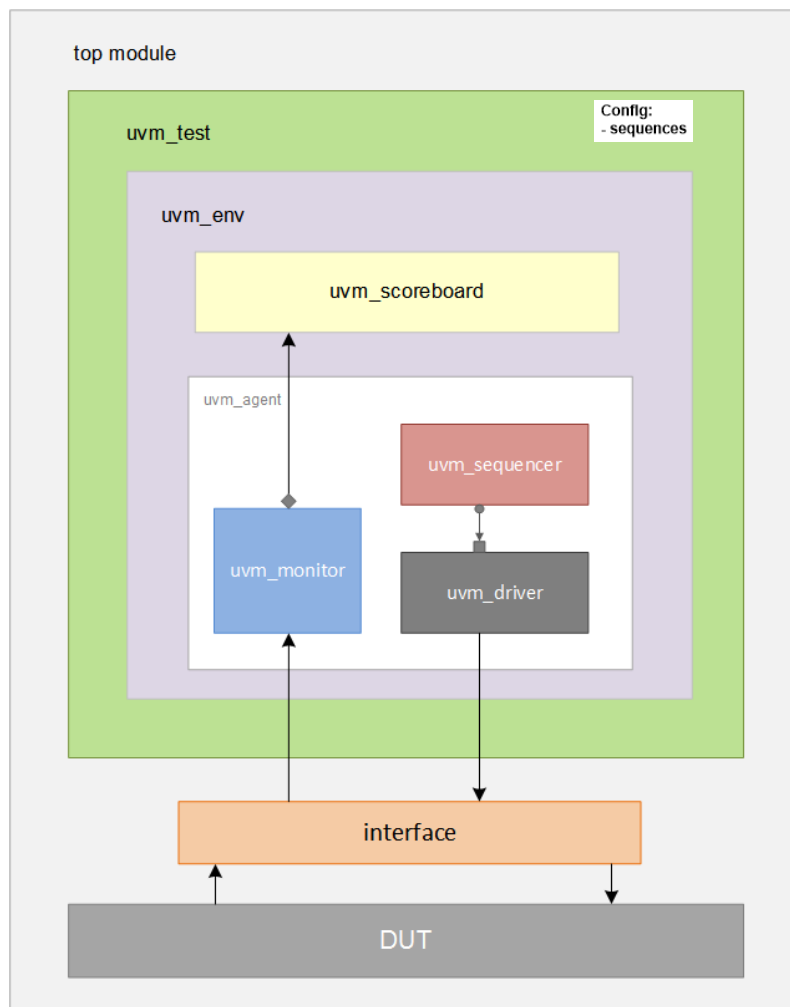


Figura 2.11. Modelo básico de entorno de verificación UVM

En el diagrama representado en la Figura 2.11, la parte correspondiente al entorno UVM es la que engloba el módulo *test* (*uvm_test*), que es el mayor nivel en la jerarquía. Este módulo puede incluir uno o varios componentes *Environment* (*uvm_env*), que no es más que un contenedor para integrar los diferentes UVC, los cuales implementarán el desarrollo de las funcionalidades del *test bench*. El componente *test* también es el encargado de inicializar los estímulos.

Además, para comunicar el entorno de verificación con el DUV (o DUT), se hace uso de un módulo *top*, que instancia el DUV y conecta sus puertos a una interfaz virtual (*interface*) accesible desde el *test bench*. Desde este módulo también se lanza la ejecución del *test*.

Llegados a este punto, se va a proceder a describir con mayor detalle cada uno de los componentes que pueden formar parte de un entorno de verificación basado en UVM.

2.6.1 SECUENCIAS

Una Secuencia UVM (*UVM Sequence*) es un objeto que hereda de `uvm_sequence` y contiene el comportamiento para generar estímulos en forma de transacciones TLM durante el tiempo de simulación (*run_phase*). Por lo tanto, cada secuencia estará parametrizada con el tipo de transacción que va a generar. Las secuencias no forman parte de la jerarquía de componentes, sino que se trata de objetos dinámicos. Estos objetos, además, pueden tener su propia jerarquía, incluyendo secuencias dentro de otras secuencias [15].

Al igual que para los componentes de la jerarquía en un entorno de verificación UVM, cuyas fases determinan la ejecución en la simulación, las secuencias UVM también siguen un patrón secuencial de tareas a ejecutarse, las cuales son exclusivas de la clase `uvm_sequence`. Este patrón se encuentra representado en la Figura 2.12.

La tarea principal es la tarea `body()`, que es la responsable de la generación de transacciones. Otras tareas que podrían utilizarse son `pre_body()` y `post_body()`, que permiten realizar acciones de inicialización o de sincronismo con otros eventos, antes o después de la ejecución de la tarea `body()`, respectivamente.

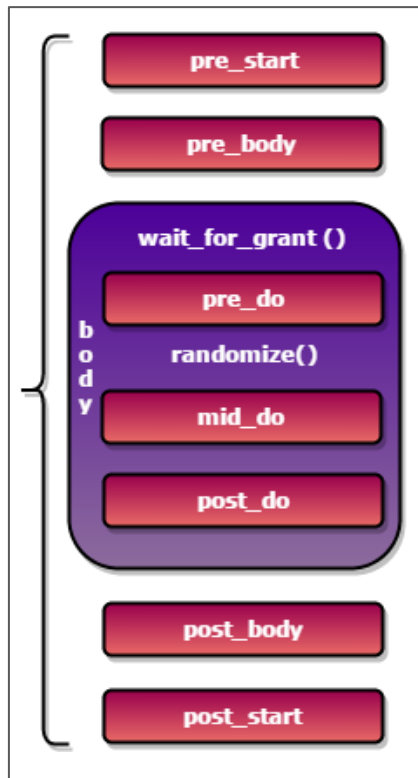


Figura 2.12. Tareas en la ejecución de una secuencia

UVM proporciona dos macros para crear, aleatorizar y enviar las transacciones de una secuencia, que son ``uvm_do` y ``uvm_do_with`. La principal diferencia entre ambas consiste en que la segunda ofrece la posibilidad de incluir restricciones en la aleatorización de los campos de una transacción [14]. Sin embargo, se recomienda hacer uso directamente de las funciones `start_item()` y `finish_item()`, las cuales son invocadas implícitamente por las dos macros comentadas. Con la primera, se bloquea la ejecución hasta que el componente *Sequencer* (componente encargado de enviar la transacción) esté listo para enviar la transacción de estímulos, mientras que la segunda hace lo propio hasta que la transacción sea enviada [12]. Entre estas dos funciones, se aleatorizan los campos de la transacción mediante la ejecución del método `randomize()`, que también permite la opción `with{}` para la inclusión de restricciones.

2.6.1.1 TRANSACCIONES

Una transacción UVM o ítem de datos es un objeto que hereda de la clase `uvm_sequence_item`. Este tipo de objetos consiste en una serie de campos de datos requeridos para la generación de estímulos. Dichos campos de datos suelen ser aleatorios para luego incluir restricciones en los mismos y, por lo general, coinciden con las señales específicas a enviar o a leer desde los puertos del DUV [13].

Como ya se ha comentado, las transacciones en UVM se generan en las secuencias, que se encargan de darles valores a sus campos, bien de forma aleatoria, o bien con valores específicos indicando restricciones en el código. Además, las mismas se transmiten entre los diferentes componentes que las usan haciendo uso de los puertos TLM.

2.6.2 COMPONENTE *AGENT*

Un *UVM Agent* es un componente de la jerarquía que se deriva de la clase `uvm_agent` y que agrupa y estructura otros UVC, los cuales interactúan con una interfaz específica del DUV. Un componente *UVM Agent* genérico incluye, a fin de crear un nivel de abstracción mayor, un componente *UVM Sequencer*, un componente *UVM Driver* y un componente *UVM Monitor*. De este modo el componente *Sequencer* se encargará de controlar el flujo de estímulos (los cuales le llegan en forma de secuencia), el componente *Driver* de aplicar dichos estímulos sobre la interfaz con el DUV, y el componente *Monitor*, como su propio nombre indica, de monitorizar la interfaz con el DUV [15].

En la Figura 2.13 se muestra una representación gráfica de un componente *UVM Agent* genérico.

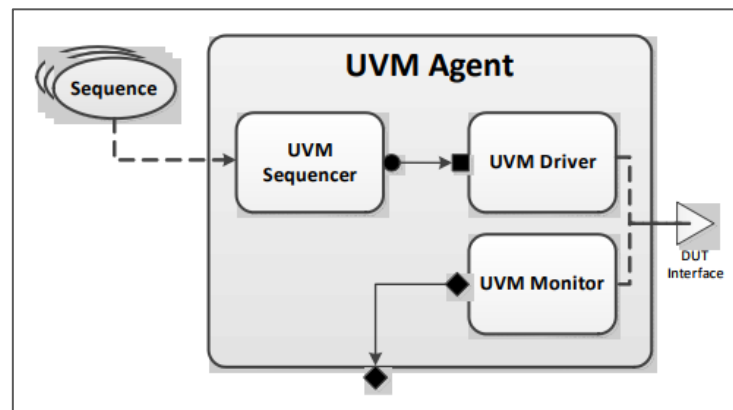


Figura 2.13. Agente UVM

Además, cabe comentar que los componentes *UVM Agent* pueden ser configurados bien como activos, o bien como pasivos. En el primer caso, su funcionalidad es la de captar estímulos y conducirlos hacia el DUV, mientras en el segundo, solamente se encarga de muestrear las señales en la interfaz con el DUV, sin controlarla. Por tanto, un componente *Agent* pasivo no necesita componentes *Sequencer* ni *Driver*, sino únicamente un componente *Monitor* para acceder a dicha interfaz [3], [13].

Además de los UVC ya citados que se integran en un componente *UVM Agent*, el mismo también puede incluir otros componentes. Estos componentes pueden ser: modelos TLM, colectores de

cobertura, comprobadores de protocolo, etc. [15]. Sin embargo, en el caso particular de este TFM, únicamente será necesario hacer uso de los tres componentes principales ya mencionados, los cuales se van a describir con mayor detalle a continuación.

2.6.2.1 COMPONENTE SEQUENCER

Un *UVM Sequencer* es un componente que se deriva de la clase `uvm_sequencer` y que actúa como un medio de arbitraje para controlar el flujo de transacciones con protocolo *request* y *response* entre las secuencias y el componente *Driver* (comunicación bidireccional). Esta comunicación entre el componente *Sequencer* y el componente *Driver* se efectúa a través de puertos TLM (cuyos puertos se heredan directamente de las clases `uvm_sequencer` y `uvm_driver`) y sigue un protocolo de *handshake* [13], [15].

Este protocolo de comunicación se muestra en la Figura 2.14, donde en primer lugar, la secuencia informa al componente *Sequencer* sobre una transacción disponible para el componente *Driver* (`start_item()`). Una vez que el componente *Driver* haga una petición de recepción de una nueva transacción al componente *Sequencer* (función bloqueante `get_next_item()` sobre su puerto TLM), la secuencia es informada y se envía dicha transacción al componente *Driver* (`finish_item()`). Finalmente, el componente *Driver* informa a la secuencia de que ya ha hecho uso de la transacción mediante la función `item_done()` sobre su puerto TLM, con la que también puede enviarse información adicional a la secuencia.

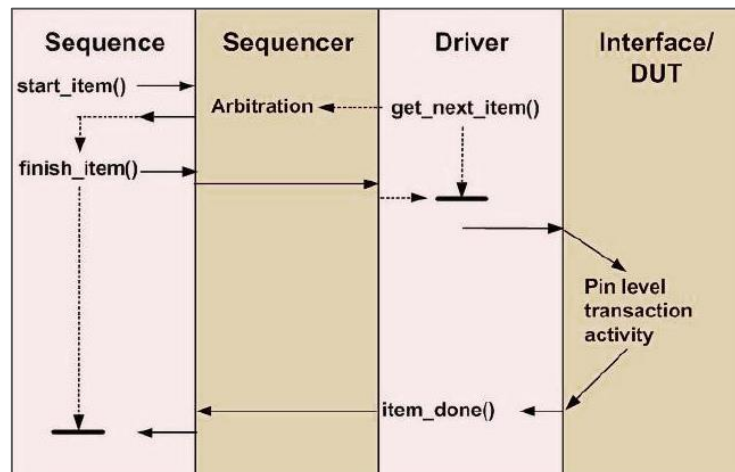


Figura 2.14. Protocolo de *handshake* Sequencer-Driver

En definitiva, un componente *Sequencer* se puede entender como un generador avanzado de estímulos que controla las transacciones que se envían desde una o más secuencias hacia el

componente *Driver* [3]. Estos estímulos son las secuencias que dicho componente *Sequencer* tiene asignadas. En general, todos los componentes del componente *Agent* están parametrizados con el tipo de transacción que van a utilizar.

2.6.2.2 COMPONENTE DRIVER

Un *UVM Driver* es un componente que se deriva de la clase `uvm_driver` y cuya funcionalidad consiste en recibir transacciones individuales desde el componente *Sequencer* a través de un puerto TLM, y transformarlas en señales RTL para aplicarlas sobre la interfaz del DUV. Por tanto, elimina los niveles de abstracción del entorno UVM, convirtiendo los estímulos a nivel de transacciones en estímulos a nivel de pines del DUV [15].

Además de las funciones `get_next_item()` e `item_done()` que fueron mostradas en la Figura 2.14, el componente *Driver* permite hacer uso de otras en su lugar. Una de ellas, `try_next_item()`, es una variante no bloqueante de la función `get_next_item()` y devuelve un puntero nulo en caso de no existir transacciones disponibles en el componente *Sequencer*. También se tiene la función `get()`, que engloba directamente el procedimiento de la dupla `get_next_item()-item_done()`. Finalmente, la función `put()` puede utilizarse de modo excepcional para comunicarse con la secuencia a través del componente *Sequencer* [14].

En cuanto a la interfaz de comunicación con el DUV, en la Figura 2.11 pudo observarse que la misma se efectúa a través de un objeto del tipo *interface* en *SystemVerilog*. En este caso no se trata de una interfaz física, sino de una interfaz virtual (palabra clave `virtual` en *SystemVerilog*), debido a que una interfaz no puede ser instanciada dentro de una clase. Como se verá posteriormente, esta interfaz es incluida en la base de datos en el módulo *top* previamente al comienzo del *test*, de modo que los componentes *Driver* y *Monitor* pueden extraerla desde la misma.

2.6.2.3 COMPONENTE MONITOR

Un *UVM Monitor* es un componente que se deriva de la clase `uvm_monitor` y se encarga de muestrear las señales de la interfaz con el DUV. Inversamente al componente *Driver*, el componente *Monitor* utiliza la información recabada desde el DUV y elabora transacciones TLM que pone a disposición del resto del entorno de verificación. Para ello, hace uso de la interfaz virtual con el DUV y de un *TLM analysis port*, respectivamente [15].

Además de esta función, el componente *Monitor* puede efectuar, internamente, algún procesamiento sobre las transacciones generadas. Estas acciones incluyen, por ejemplo, tareas de comprobación y cobertura, que permiten chequear si se cumple el Plan de Verificación establecido, o bien acciones de mensajería en consola, almacenamiento en archivos, etc. [15]. Sin embargo, lo más común es delegar todo este tipo de actividades en componentes dedicados que se conecten a su puerto TLM.

En determinados casos, como en aquellos en los que el componente *Monitor* incluya funcionalidades complejas, se puede requerir la completa abstracción desde el nivel de señales, por lo que se separa el muestreo de señales del resto de funcionalidades. Para ello, se incluye un nuevo componente, denominado *UVM Collector*, que se encargará de captar la información de la interfaz con el DUV y generar transacciones para el componente *Monitor* [3]. Este nuevo componente no tiene una clase base directa, por lo que a la hora de implementarlo debe derivar directamente de la clase `uvm_component`.

2.6.3 COMPONENTE SCOREBOARD

Un *UVM Scoreboard* es un componente que se deriva de la clase `uvm_scoreboard` y tiene como objetivo verificar la validez del DUV mediante la comparación de las salidas del mismo con los valores esperados [3], [15]. Este componente puede ubicarse en diferentes niveles de la jerarquía, por lo que su descripción y tareas dependerán de dicha localización.

La obtención de los valores esperados puede realizarse, principalmente, de dos formas: bien mediante la ejecución de un modelo de referencia con las mismas entradas que el DUV, o bien mediante la lectura de un fichero en el que se encuentren almacenados estos valores de referencia [13].

Las salidas del DUV, en cambio, las obtiene mediante la suscripción al *TLM analysis port* de uno o más componentes *Monitor*, recibiendo transacciones desde los mismos. A diferencia de la comunicación *secuencia-Driver*, en este caso no se tienen ningún mecanismo de arbitraje, por lo que se suele utilizar una cola FIFO en el componente *Scoreboard*. De este modo, las transacciones enviadas por el componente *Monitor* se almacenarán en la FIFO, mientras que el componente *Scoreboard* las consumirá desde la misma, asegurando que no se pierdan transacciones. Para implementar la FIFO, UVM ofrece las clases `uvm_tlm_fifo` y `uvm_tlm_analysis_fifo` (cada una de ellas incluye un puerto *TLM export* para la conexión). Según el tipo de puerto TLM del componente *Monitor*, en este caso se hará uso de la segunda clase.

2.6.4 COMPONENTE *ENVIRONMENT*

Un *UVM Environment* es un componente que se deriva de la clase `uvm_env` y actúa como un contenedor. Se trata del componente de segundo mayor nivel en la jerarquía de UVC y su funcionalidad consiste en incluir e interconectar otros UVC, como pueden ser componentes *Agent*, *Scoreboard*, *Monitor* de alto nivel, e incluso otros *UVM Environments*. [13]. En este componente es donde realmente se aprecia el nivel de reusabilidad, ya que el componente *Environment* posee propiedades de configuración que permiten personalizar su topología y el comportamiento de los UVC que integra [3], pudiendo enfocarse a diversas tareas de verificación.

2.6.5 COMPONENTE *TEST*

Un *UVM Test* es un componente que se deriva de la clase `uvm_test` y se encarga de definir el escenario de verificación del *test bench*. El proceso de creación de un entorno de verificación UVM comienza en este componente, y luego va descendiendo por toda la jerarquía. Las funciones que realiza son, fundamentalmente, tres [15]:

- Referenciar al componente *Environment* de más alto nivel en la jerarquía y, a partir del mismo, el resto del entorno de verificación.
- Configurar el entorno de verificación, bien a través de la *Factory* (sobrescribiendo elementos ya existentes o creando nuevos), o bien mediante el uso de la base de datos de configuración.
- Aplicar la generación de estímulos, invocando la ejecución de secuencias UVM en los componentes *Sequencer* del *test bench*. Según las características del DUV que se esté verificando, se asignarán unas secuencias u otras. Debido a la complejidad de los sistemas actuales, se aconseja disponer de una biblioteca de secuencias que cubran todos los casos establecidos en el Plan de Verificación.

Típicamente, se suele crear un *test UVM base* con la instanciación del componente *Environment* principal y otros elementos comunes. A partir de este, se extienden otros *test* individuales, los cuales configuran el entorno de verificación de un modo distinto, o bien seleccionan diferentes secuencias a ejecutar en los componentes *Sequencer*, etc. [15].

2.6.6 MÓDULO *TOP*

El módulo *Top* (también conocido como *UVM Testbench*) es un componente instancia *SystemVerilog* de tipo `module`, que típicamente se encarga de referenciar y configurar la conexión entre el DUV y el entorno de verificación UVM. Todo ello se efectúa en tiempo de ejecución, con lo que se permite que, con una única compilación, puedan ejecutarse varios *test* [15].

Además de esta funcionalidad principal, el módulo *Top* incluye otras acciones necesarias para la correcta ejecución de los *test*. Entre dichas acciones destacan [13]:

- La creación e instanciación de una o varias interfaces virtuales, que servirán de nexo entre el DUV (modelo físico) y el entorno de verificación (modelo basado en clases). Estas interfaces definirán y agruparán cada una de las señales del DUV a las que se desea tener acceso desde el entorno UVM durante la verificación.
- La ya comentada instanciación del DUV, conectando físicamente las señales de sus puertos con sus homólogas en las interfaces virtuales.
- La generación de las señales de reloj y de *reset*, así como de cualquier señal que sea común a todo el sistema.
- La inclusión de las interfaces virtuales en la base de datos, de modo que los diferentes componentes *Driver* y *Monitor* puedan tener acceso a las mismas para introducir estímulos y monitorizar salidas del DUV.
- Por último, la llamada al método `run_test()`, que ya fue citada en el apartado 2.5.3 Fases de UVM. Este método inicia la ejecución de las diferentes fases por las que pasa una simulación UVM, de modo que indica el comienzo del *test*.

Capítulo 3. MÓDULO IP MULTI-INTERFAZ PARA LA COMPRESIÓN DE IMÁGENES

Una vez presentada la metodología UVM, que será utilizada para implementar el entorno de verificación propuesto en este TFM, en este capítulo se va a describir el sistema hardware digital correspondiente al DUV sobre el que se aplicará el entorno de verificación UVM. Este sistema es un módulo IP que ha sido desarrollado por el Instituto Universitario de Microelectrónica Aplicada (IUMA) de la ULPGC en un proyecto realizado en colaboración con la Agencia Espacial Europea (ESA, *European Space Agency*) y, por tanto, incluye su propio entorno de verificación (*test bench*). Sin embargo, el entorno de verificación original es un entorno de verificación *ad-hoc* tradicional, complejo y no reutilizable. Estos aspectos se tratan, con más detalle, a lo largo de este capítulo.

3.1 MÓDULO IP

El sistema hardware digital a verificar mediante el entorno de verificación UVM desarrollado en este TFM se corresponde con un módulo IP multi-interfaz que fue diseñado, principalmente, para la compresión de datos obtenidos desde otro codificador implementado en VHDL, que actuaría como bloque preprocesador. Dicho codificador se encarga de la compresión sin pérdidas de imágenes multispectrales e hiperespectrales. Sin embargo, también sería posible hacer uso del mismo para realizar la compresión de cualquier cadena de datos binarios, independientemente del tipo de archivo de entrada.

El nombre de este IP es *SHyLoC-121* (en adelante “IP121”) y el nombre de la entidad VHDL que lo implementa es `ccsds121_shyloc_top`. Se trata de un compresor universal sin pérdidas para aplicaciones espaciales basado en el estándar *CCSDS-121* del CCSDS (*Consultative Committee for Space Data Systems*) [16]. Este estándar atiende a la compresión de datos en satélites y surge debido a las limitaciones en el almacenamiento y ancho de banda necesario para su transmisión, que obligan a reducir la cantidad de datos a bordo [17].

Aunque el estándar también posibilita la compresión con pérdidas, en este caso, y como ya se ha comentado, se hace uso de una compresión sin pérdidas. Este tipo de compresión conserva la precisión de los datos fuente, eliminando únicamente la redundancia, si bien es cierto que obtiene tasas de compresión menores [17].

Este IP consta, fundamentalmente, de dos interfaces diferenciadas: una interfaz de configuración y una interfaz de datos, además de las entradas básicas para las señales de reloj y de *reset* externas. Atendiendo a la configuración, como se ha comentado en el apartado 1.2 Objetivos, se trata de un IP altamente configurable. En este sentido, el usuario puede ajustar los parámetros de configuración para lograr el compromiso entre complejidad y eficiencia de compresión que mejor se adapte a la misión [18]. Así, existen dos modos principales de configuración [18]:

- Configuración general del sistema (IP) en tiempo de compilación mediante el uso de constantes que determinan parámetros como, por ejemplo, el tamaño de los *buffers* de entrada y de salida de datos.
- Configuración de los parámetros de compresión en tiempo de ejecución mediante una interfaz AHB (*Advanced High-performance Bus*) y registros localizados en memoria. La posibilidad de esta configuración puede estar, o no, activa en función de las constantes generadas en tiempo de compilación. En el caso de no encontrarse activa, se establece una configuración por defecto en la compilación.

Acerca del modo de funcionamiento del IP121, para el desarrollo de este TFM y, en general, para la implementación de su entorno de verificación de tipo *Black-Box*, no resulta necesario entrar en más detalles. De este modo, solamente es importante conocer sus interfaces de entradas y salidas, así como los protocolos de comunicación que estas implementan. La Figura 3.1 muestra un diagrama con las entradas y salidas del IP121, donde los puertos de la mitad superior corresponden a la interfaz de configuración y control, mientras que los puertos de la mitad inferior pertenecen a la interfaz de datos.

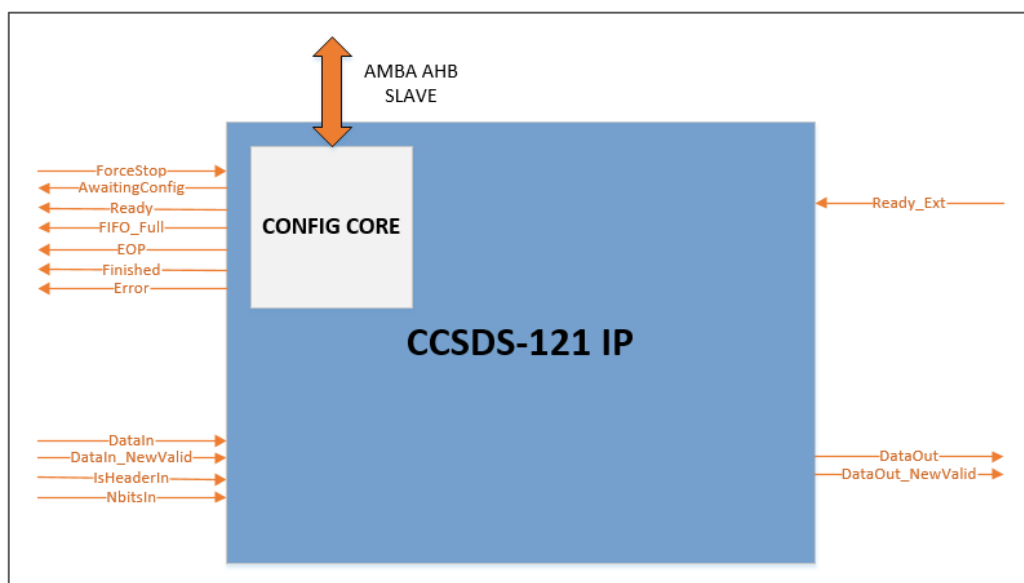


Figura 3.1. Diagrama de entradas/salidas del IP

El Código 3.1 presenta la definición VHDL original de la entidad del IP121, incluyendo sus interfaces de entrada y de salida. A continuación, se detalla el uso y las características de todos y cada uno de estos puertos.

```

entity ccsds121_shyloc_top is
  port (
    -- System Interface
    Clk_S: in std_logic;
    Rst_N: in std_logic;

    -- AMBA Interface
    Clk_AHB      : in  std_logic;
    Reset_AHB    : in  std_logic;
    AHBSlave121_In : in  ahb_slv_in_type;
    AHBSlave121_Out: out ahb_slv_out_type;

    -- Data Input Interface
    DataIn      : in std_logic_vector (D_GEN-1 downto 0);
    DataIn_NewValid: in std_logic;
    IsHeaderIn  : in std_logic;
    NbitsIn     : in Std_Logic_Vector (5 downto 0);

    -- Data Output Interface
    DataOut      : out std_logic_vector (W_BUFFER_GEN-1 downto 0);
    DataOut_NewValid: out std_logic;

    -- Control Interface
    ForceStop    : in  std_logic;
    Ready_Ext    : in  std_logic;
    AwaitingConfig: out std_logic;
    Ready        : out std_logic;
    FIFO_Full    : out std_logic;
    EOP          : out std_logic;
    Finished     : out std_logic;
    Error        : out std_logic;
  );
end ccsds121_shyloc_top;

```

Código 3.1. Entidad VHDL del IP

- **Clk_S** y **Clk_AHB**. Entradas de la señal de reloj del IP121 y de su interfaz *AMBA (Advanced Microcontroller Bus Architecture) AHB Slave*, respectivamente.
- **Rst_N** y **Reset_AHB**. Entradas de la señal de *reset* del IP121 y de su interfaz *AMBA AHB Slave*, respectivamente. Ambas señales son activas a nivel bajo.
- **AHBSlave121_In**. Puerto de tipo `ahb_slv_in_type` que engloba todas las señales correspondientes a un puerto de entrada de tipo *AMBA AHB Slave*.
- **AHBSlave121_Out**. Puerto de tipo `ahb_slv_out_type` que engloba todas las señales correspondientes a un puerto de salida de tipo *AMBA AHB Slave*.

- **DataIN.** Valor de la muestra de entrada del flujo de datos a comprimir. Su tamaño es variable y se determina en tiempo de compilación mediante el uso de parámetros.
- **DataIn_NewValid.** Señal de validación de datos de entrada. Cuando esta señal se encuentra activa a nivel alto, la muestra de datos presente en el puerto `DataIN` es válida para la compresión.
- **IsHeaderIn.** Señal que, a nivel alto, indica que la muestra de datos de entrada disponible en el puerto `DataIN` se corresponde con la cabecera de un bloque preprocesador (por ejemplo, otro codificador previo como el citado en la introducción de este apartado).
- **NbitsIn.** Puerto de 6 bits que, en el caso de que la señal `IsHeaderIn` se encuentre activa a nivel alto, indica el número de bits válidos en la cabecera de entrada disponible en el puerto `DataIN`.
- **DataOut.** Valor de muestra de salida del flujo de datos comprimido. Su tamaño es variable y se determina en tiempo de compilación mediante el uso de parámetros.
- **DataOut_NewValid.** Señal de validación de datos de salida. Cuando esta señal se encuentra activa a nivel alto, la muestra de datos presente en el puerto `DataOut` es una muestra comprimida válida.
- **ForceStop.** Señal de entrada que fuerza la parada y finalización de cualquier compresión que esté ejecutándose en ese momento.
- **Ready_Ext.** Señal de entrada que indica al IP que el receptor externo no está preparado, por lo que no se podrán validar datos de salida.
- **AwaitingConfig.** Señal de salida que indica que el IP está esperando para recibir los parámetros de configuración para la compresión.
- **Ready.** Señal de salida que indica que la configuración se ha recibido correctamente y que, por tanto, el IP se encuentra preparado para recibir el flujo de datos de entrada a comprimir.
- **FIFO_Full.** Señal de salida que indica que la FIFO de datos de entrada está llena, por lo que no se aceptarán más muestras de datos de entrada.
- **EOP. (End Of Package)** Señal de salida que indica que ha comenzado la compresión de la última muestra de datos.
- **Finished.** Señal de salida que indica que el IP121 ha finalizado la compresión del flujo de datos de entrada.
- **Error.** Señal de salida que indica que ha habido un error, bien en la recepción de los parámetros de configuración, o bien durante las tareas de compresión.

Tras la definición de cada uno de los puertos de entrada/salida del IP, cabe destacar que, exceptuando las señales de *reset* que son activas a nivel bajo, el resto de señales de control son activas a nivel alto. Por otra parte, `ahb_slv_in_type` y `ahb_slv_out_type` son tipos VHDL que están definidos en el paquete AMBA de la biblioteca *GRLIB IP*. Esta biblioteca ha sido utilizada en la implementación del módulo original y ofrece múltiples implementaciones VHDL de gran utilidad para los *buses on-chip* más comunes, entre los que se encuentra AMBA AHB [19]. Como se verá posteriormente, varias de estas implementaciones son necesarias a la hora de gestionar el entorno de verificación.

3.2 ENTORNO DE VERIFICACIÓN (*TEST BENCH*) ORIGINAL

En este apartado se detalla el entorno de verificación o *test bench* VHDL original utilizado para la verificación del IP121. Se trata de un entorno de verificación *ad-hoc* tradicional con un único archivo que comprende más de 1200 líneas de código VHDL, en el cual se realizan todos los procedimientos relativos a la verificación funcional del IP121: la referencia al DUV, la generación y envío de los estímulos hacia este (tanto parámetros de configuración como datos), la recepción de los resultados, y la comparación de los mismos con los datos de referencia. Este fichero se encuentra en la ruta `/src/tb` desde el directorio principal del IP, y su nombre es `ccsds121_tb_v3.vhd`.

El entorno original, sin embargo, mantiene las constantes relativas a los parámetros de compilación y de configuración en otros archivos VHDL. En este caso, se tienen dos archivos VHDL por cada vector de estímulos a ejecutar (uno para la compilación del IP y otro para la ejecución del *test* en el entorno de verificación). Al existir un elevado número de *test*, todos estos archivos se generan automáticamente haciendo uso del lenguaje de programación *Python* (específicamente la versión *Python 2.7*), así como de un fichero CSV (*Comma-Separated Values*) que contiene los valores de las diferentes constantes necesarias para caracterizar cada *test*.

A continuación, se abordan con mayor detalle los aspectos más significativos del entorno de verificación original.

3.2.1 GENERACIÓN DE LOS ESTÍMULOS Y LOS ARCHIVOS DE SIMULACIÓN

Como ya se ha comentado, para generar los archivos VHDL con las constantes relativas a los estímulos de cada *test* y a la configuración del IP121, de forma automatizada, se ha hecho uso de *Python 2.7*. En este caso, se tiene un *script* en *Python* denominado `run_vhdl_tests_121.py`,

que además de la generación de estos archivos VHDL, también se encarga de crear una serie de archivos con extensión `do` para su ejecución en la herramienta *QuestaSim*. Dichos archivos contienen los comandos de compilación del IP y de ejecución de los diferentes *test* sobre el entorno de verificación.

El *script* en *Python* se encuentra en la carpeta *verification_scripts*, y está en el directorio principal del IP121. Para lograr los objetivos planteados en este TFM, el contenido de este *script* no es relevante, puesto que solamente se necesita conocer su funcionalidad y el modo de ejecutarlo, incluyendo sus parámetros. Para facilitar este proceso, se dispone de una ayuda en la cabecera del archivo, la cual se muestra en el Código 3.2.

```
# Some command line examples:
# run_vhdl_tests_121.py testcases_121.csv ../images/raw
../images/compressed ../images/reference ../ modelsim
# run_vhdl_tests_121.py synthesis_params_121.csv ../images/raw
../images/compressed ../images/reference ../ synplify
# run_vhdl_tests_121.py synthesis_params_121.csv ../images/raw
../images/compressed ../images/reference ../ ise

#How to run this script
# run_vhdl_tests_121.py
#(0) *.csv file containing the desired parameters
#(1) path to the folder containing the raw images that will be
compressed during the simulation
#(2) path to the folder where the compressed images will be stored
#(3) path to the folder containing the reference compressed images
#(4) IP core database folder
#(5) select one among the possible options:
#modelsim --> generates TCL scripts for simulations
#synplify --> generates TCL scripts for synthesis with synplify
#ise --> generates TCL scripts for synthesis with ise
```

Código 3.2. Ayuda de ejecución del *script* `run_vhdl_tests_121.py`

En este caso, como se pretende generar los archivos relativos a la simulación en la herramienta *QuestaSim*, se ejecuta desde la consola de *Python* el primero de los ejemplos que se muestra en el Código 3.2, cuyo último parámetro es `modelsim`. Además, se puede observar que el primer parámetro de entrada en el comando de ejecución del *script* es el archivo CSV en el que se encuentran los valores que tomarán los diferentes parámetros de configuración, cuyo nombre es `testcases_121.csv`.

Con la ejecución del *script*, ya se han generado todos y cada uno de los archivos comentados anteriormente. De entre ellos, destacan los ficheros VHDL con la definición de las constantes necesarias para la compilación del IP121 y los estímulos para la ejecución de los *test*. Ambos

ficheros contienen paquetes VHDL, cuyos nombres son `ccsds121_parameters` (constantes de compilación del IP) y `ccsds121_tb_parameters` (parámetros para la configuración del *test bench* y los estímulos para la ejecución de los *test*). Un ejemplo de estos archivos, en concreto los relativos al *Test 25*, se muestra en el Código 3.3 y el Código 3.4, respectivamente.

```

--! Use standard library
library ieee;
--! Use logic elements
use ieee.std_logic_1164.all;
--! Use numeric elements
use ieee.numeric_std.all;

--! ccsds121_parameters package
package ccsds121_parameters is

-- TEST: 25_Test
constant EN_RUNCFG: integer := 1; --! (0) Disables runtime
configuration; (1) Enables runtime configuration.
constant RESET_TYPE: integer := 1; --! (0) Asynchronous reset; (1)
Synchronous reset.
constant HSINDEX_121: integer := 3; --! AHB slave index.
constant HSCONFIGADDR_121: integer := 16#100#; --! ADDR field of
the AHB Slave.
constant HSADDRMASK_121: integer := 16#FFF#; --! MASK field of the
AHB slave.
constant EDAC: integer := 0; --! (0) Inhibits EDAC implementation;
(1) EDAC is implemented.
constant Nx_GEN: integer := 1024; --! Maximum allowed number of
samples in a line.
constant Ny_GEN: integer := 1024; --! Maximum allowed number of
samples in a row.
constant Nz_GEN: integer := 1024; --! Maximum allowed number of
bands.
constant D_GEN: integer := 16; --! Maximum dynamic range of the
input samples.
constant ENDIANESS_GEN: integer := 0; --! (0) Little-Endian; (1)
Big-Endian.
constant J_GEN: integer := 64; --! Block Size.
constant REF_SAMPLE_GEN: integer := 4096; --! Reference Sample
Interval.
constant CODESET_GEN: integer := 0; --! Code Option.
constant W_BUFFER_GEN: integer := 32; --! Bit width of the output
buffer.
constant PREPROCESSOR_GEN: integer := 0; --! (0) Preprocessor is
not present; (1) CCSDS123 preprocessor is present; (2) Any-other
preprocessor is present.
constant DISABLE_HEADER_GEN: integer := 0; --! Selects whether to
disable (1) or not (0) the header.

end ccsds121_parameters;

```

Código 3.3. Paquete `ccsds121_parameters` para el *Test 25*

```

--! Use standard library
library ieee;
--! Use logic elements
use ieee.std_logic_1164.all;
--! Use shyloc_121 library
library shyloc_121;
--! Use shyloc_121 parameters
use shyloc_121.ccsds121_parameters.all;
--! Use shyloc_utils library
library shyloc_utils;
--! Use shyloc_utils functions
use shyloc_utils.shyloc_functions.all;

--! ccsds121_tb_parameters package. Package with configuration values
used by the test.
package ccsds121_tb_parameters is

-- TEST: 25_Test
--! Test to perform (0) regular; (4) Force Stop; (7) Configuration
Error; (14) Attempt to send new configuration
constant test_id: integer := 0; --! Indicates the test to perform

constant EN_RUNCFG_G: integer :=
    shyloc_121.ccsds121_parameters.EN_RUNCFG; --! (0) Disables
    runtime configuration; (1) Enables runtime configuration.
constant RESET_TYPE: integer :=
    shyloc_121.ccsds121_parameters.RESET_TYPE; --! (0) Asynchronous
    reset; (1) Synchronous reset.

constant HSINDEX_121: integer :=
    shyloc_121.ccsds121_parameters.HSINDEX_121; --! AHB slave index.
constant HSCONFIGADDR_121: integer :=
    shyloc_121.ccsds121_parameters.HSCONFIGADDR_121; --! ADDR field
    of the AHB Slave.
constant HSADDRMASK_121: integer :=
    shyloc_121.ccsds121_parameters.HSADDRMASK_121; --! MASK field of
    the AHB slave.

constant Nx_tb: integer := 256; --! Number of columns.
constant Ny_tb: integer := 1; --! Number of lines.
constant Nz_tb: integer := 1; --! Number of bands.
constant D_tb: integer := 7; --! Dynamic range of the input
    samples.
constant ENDIANESS_tb: integer := 0; --! (0) Little-Endian; (1)
    Big-Endian.
constant J_tb: integer := 64; --! Block Size.
constant REF_SAMPLE_tb: integer := 4096; --! Reference Sample
    Interval (Determine how often to insert references not coded).
constant CODESET_tb: integer := 0; --! Code Option (If specified
    and the dynamic range D is <= 4, the restricted mode will be
    used).
constant W_BUFFER_tb: integer := 32; --! Output word size.
constant BYPASS_tb: integer := 0; --! (0) Compression; (1)
    Bypass Compression.
constant PREPROCESSOR_tb: integer := 0; --! (0) Preprocessor is not
    present; (1) CCSDS123 preprocessor is present; (2) Any-other
    preprocessor is present.
constant DISABLE_HEADER_tb: integer := 1; --! Selects whether to
    disable (1) or not (0) the header generation.

```



```

--! Stimuli, reference and output files.
constant stim_file: string :=
"/home/users/master/divdsi/srodriguez/Compresion_IUMA/SHyLoC-
PDR/CCSDS121IP-VHDL/images/raw/test_p256n07.dat";
constant ref_file: string :=
"/home/users/master/divdsi/srodriguez/Compresion_IUMA/SHyLoC-
PDR/CCSDS121IP-VHDL/images/reference/comp_25.dat";
constant out_file: string :=
"/home/users/master/divdsi/srodriguez/Compresion_IUMA/SHyLoC-
PDR/CCSDS121IP-VHDL/images/compressed/comp_25.dat.vhd";

--! Some other necessary parameters.
constant D_G_tb : integer := shyloc_121.ccsds121_parameters.D_GEN;
constant W_BUFFER_G_tb : integer :=
shyloc_121.ccsds121_parameters.W_BUFFER_GEN;
constant N_SAMPLES_G_tb : integer := (Nx_tb*Ny_tb*Nz_tb);
constant W_N_SAMPLES_G_tb : integer := log2(N_SAMPLES_G_tb);
constant CODESET_G_tb : integer :=
shyloc_121.ccsds121_parameters.CODESET_GEN;
constant N_K_G_tb: integer := get_n_k_options (D_G_tb,
CODESET_G_tb);
constant W_K_G_tb: integer := maximum(3,log2(N_K_G_tb)+1);
constant W_NBITS_K_G_tb: integer :=
get_k_bits_option(W_BUFFER_G_tb, CODESET_G_tb, W_K_G_tb);

end ccsds121_tb_parameters;

```

Código 3.4. Paquete `ccsds121_tb_parameters` para el Test 25

A continuación, se definen algunos de los parámetros más significativos de estos archivos, destacándose que cada parámetro relativo a la configuración y compilación del IP121 tiene su parámetro homólogo en el archivo de parámetros para la configuración del *test bench*.

- **EN_RUNCFG.** Especifica al IP121 si la posibilidad de configuración en tiempo de ejecución se encuentra, o no, activada.
- **HSCONFIGADDR_121.** Indica al IP121 la dirección base de memoria en la que se encuentran sus registros de configuración.
- **D_GEN.** Configura el tamaño del *buffer* de muestras de datos de entrada del IP121.
- **W_BUFFER_GEN.** Configura el tamaño del *buffer* de muestras de datos de salida del IP121 tras la compresión.
- **test_id.** Configura el *test bench* en función del tipo de procedimiento de *test* que se va a efectuar. Esto se verá con mayor detalle en el apartado 3.2.4 Proceso de configuración del IP.
- **stim_file, ref_file y out_file.** *String* con las rutas a los archivos de estímulos, datos de referencia y datos de salida del DUV, respectivamente, para que el *test bench* pueda acceder a los mismos.

De este modo, el paquete `ccsds121_parameters` se incluye en el código VHDL del IP121, lo que hace que los valores de todas las constantes definidas sean accesibles desde el mismo en el momento de compilación. El paquete `ccsds121_tb_parameters`, por su parte, se incluye en el código del *test bench*, haciendo visibles en el mismo todas las constantes definidas relativas a los estímulos.

3.2.2 GENERACIÓN DE LAS SEÑALES PRINCIPALES

A la hora de generar el código del entorno de verificación o *test bench*, las primeras acciones básicas a realizar consisten en la generación de las principales señales que gobernarán el DUV y el entorno de verificación, así como la referencia de los módulos necesarios para la ejecución de los *test* (entre los que se encuentra el IP121 a verificar). Aparte de esto, es necesario realizar la inclusión, al comienzo del archivo, de las bibliotecas y los paquetes que se vayan a utilizar, así como llevar a cabo la creación de todas las señales internas que se precisen para el conexionado de los módulos y la ejecución de los diferentes procesos. Estos últimos procesos se van a obviar en este documento debido a que se considera que no aportan información significativa.

Este punto se va a centrar en el análisis de la generación de las principales señales que, por lo general, son las señales de reloj y de *reset*. El Código 3.5 muestra la generación de estas señales, tanto para el IP como para la interfaz *AMBA AHB Slave*, cuyas características se describen a continuación

```
-----  
--! Clock generation  
-----  
gen_clk: process  
begin  
    clk <= '1';  
    wait for 100 ns;  
    clk <= '0';  
    wait for 100 ns;  
end process;  
  
-----  
--! AMBA Clock generation  
-----  
gen_clk_amba: process  
begin  
    amba_clk <= '1';  
    wait for 70 ns;  
    amba_clk <= '0';  
    wait for 70 ns;  
end process;
```

```

-----
--! Reset generation
-----
gen_rst: process
begin
    rst_n <= '0';
    wait for 400 ns;
    rst_n <= '1';
    wait for 200 ns;
    rst_n <= '1';
    wait;
end process;
amba_reset <= rst_n;

-----
--! ReadyExt generation
-----
gen_ReadyExt: process
begin
    ReadyExt <= '1';
    -- Minimum must wait for s3 state, which handles properly the
       read of the file in case Ready is not asserted
    wait for 9000 ns;
    if (test_id = 5) then
        ReadyExt <= '0' after 1 ns;
    end if;
    wait for 400 ns;
    ReadyExt <= '1' after 1 ns;
    wait;
end process;

```

Código 3.5. Generación de las señales principales del *test bench*

- **gen_clk** y **gen_clk_amba**. Genera una señal de reloj para el IP121 que tiene un periodo de 200 ns, o lo que es lo mismo, una frecuencia de 5 MHz. La señal de reloj que gobernará la interfaz AHB, por su parte, posee un periodo de 140 ns.
- **gen_rst**. Genera una señal de *reset* común para el IP y su interfaz *AMBA AHB Slave*, con un estado de *reset* inicial activo a nivel bajo durante 400 ns, permaneciendo inactivo desde ese momento hasta el final de la ejecución del *test*.
- **gen_ReadyExt**. Adicionalmente, también se incluye una señal para el puerto `Ready_Ext` del IP. En este caso, esta señal se encontrará siempre activa a nivel alto, excepto en el caso particular de un determinado *test*, que se pondrá a nivel bajo durante 400 ns en la ejecución de la simulación. Este *test* es muy particular y no se ha contemplado.

3.2.3 REFERENCIA DE LOS COMPONENTES DEL *TEST BENCH*

El componente principal a referenciar en el *test bench* es el IP a verificar. Sin embargo, al incluirse una interfaz AHB para recibir los parámetros de configuración para la compresión, cuyos puertos

están definidos en la biblioteca *GRLIB IP*, se hace necesario el uso de otros componentes con el propósito de comunicarse de forma efectiva con esta interfaz.

En lo referente a la comunicación AHB, el DUV actúa como esclavo (*Slave*) de la misma, por lo que también se deben referenciar los módulos maestro (*Master*) y decodificador (*Decoder*), efectuando el conexionado necesario entre los mismos. El código VHDL relativo a estas referencias y conexiones se muestra en el Código 3.6, donde, tanto los componentes maestro y decodificador, como los tipos de la mayoría de las señales involucradas en la comunicación, se encuentran definidos en la biblioteca *GRLIB IP*.

```

-----
--! AMBA master
-----
ahbtbm0: ahbtbm
generic map(hindex => 0)
port map(amba_reset, amba_clk, ctrl.i, ctrl.o, ahbmi, ahbmo);

-----
--! AMBA decoder
-----
ahbtbctrl: ahbctrl
generic map (nahbm => 1, nahbs => 1)
port map(amba_reset, amba_clk, ahbmi, msto, ahbsi, slvo);

-----
--! Assignments
-----
msto(0) <= ahbmo;

AHBSlave121_In.HSEL <= ahbsi.hsel(0);
AHBSlave121_In.HADDR <= ahbsi.haddr;
AHBSlave121_In.HWRITE <= ahbsi.hwrite;
AHBSlave121_In.HTRANS <= ahbsi.htrans;
AHBSlave121_In.HSIZE <= ahbsi.hsize;
AHBSlave121_In.HBURST <= ahbsi.hburst;
AHBSlave121_In.HWDATA <= ahbsi.hwdata;
AHBSlave121_In.HPROT <= ahbsi.hprot;
AHBSlave121_In.HREADY <= ahbsi.hready;
AHBSlave121_In.HMASTER <= ahbsi.hmaster;
AHBSlave121_In.HMASTLOCK <= ahbsi.hmastlock;

slvo(0).hready <= AHBSlave121_Out.HREADY;
slvo(0).hresp <= AHBSlave121_Out.HRESP;
slvo(0).hrdata <= AHBSlave121_Out.HRDATA;
slvo(0).hsplit <= AHBSlave121_Out.HSPLIT;
slvo(0).hindex <= HSINDEX_121;
slvo(0).hconfig <= (0 => zero32, 4 => ahb_membar(HSCONFIGADDR_121,
    '1', '1', HSADDRMASK_121), others => zero32);

-----
--! AMBA masters vector
-----
masters: for i in 1 to NAHBMST-1 generate
    msto(i).hconfig <= (others => (others => '0'));
end generate;

```

```

-----
--! AMBA slaves vector
-----
slaves: for i in 1 to NAHBSLV-1 generate
    slvo(i).hconfig <= (others => (others => '0'));
end generate;

-----
--! CCSDS-121 IP Core
-----
uut_shyloc: entity shyloc_121.ccsds121_shyloc_top(arch)
port map (
    Clk_S => clk,
    Rst_N => rst_n,

    AHBSlave121_In => AHBSlave121_In,
    AHBSlave121_Out => AHBSlave121_Out,
    Clk_AHB => amba_clk,
    Reset_AHB => amba_reset,

    DataIn_NewValid => DataIn_Valid,
    DataIn => DataIn,
    NBitsIn => NBitsIn,
    DataOut => buff_out,
    DataOut_NewValid => buff_full,
    ForceStop => ForceStop,
    IsHeaderIn => IsHeaderIn,
    AwaitingConfig => AwaitingConfig,
    Ready => Ready,
    FIFO_Full => FIFO_Full,
    EOP => EOP,
    Finished => Finished,
    Error => Error_s,
    Ready_Ext => ReadyExt
);

```

Código 3.6. Referencia, en el *test bench*, del IP y los componentes para la comunicación AHB

Llegados a este punto, la configuración del entorno de verificación queda como se muestra de forma gráfica en la Figura 3.2. Así, para enviar los parámetros de configuración al IP se debe interactuar con el módulo *Master* *ahbtbm* mediante unas funciones descritas en la biblioteca *GRLIB IP* y una señal de control, denominada *ctrl*, cuyo tipo también procede de dicha biblioteca. Tras ello, el módulo *ahbtbm* se encargará de hacer llegar la configuración al IP o DUV, con el formato correcto, a través del módulo *Decoder* *ahbctrl*.

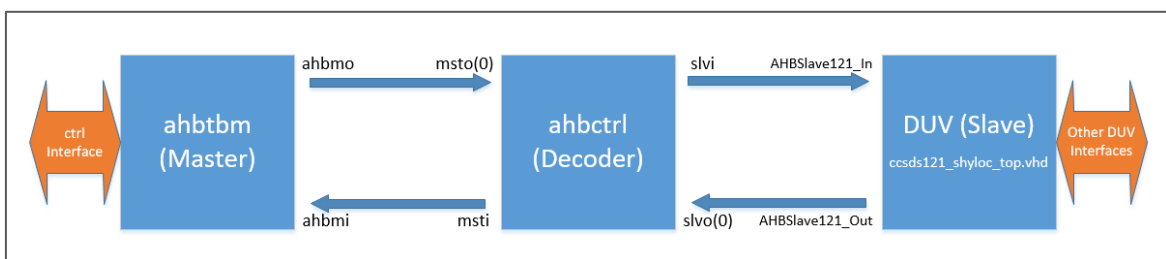


Figura 3.2. Diagrama de bloques de la estructura del *test bench*

3.2.4 PROCESO DE CONFIGURACIÓN DEL IP

El proceso de configuración del IP es uno de los tres grandes procesos independientes a estudiar en el *test bench* original, conjuntamente con el proceso de introducción del flujo de datos de entrada y el de recepción del flujo de datos de salida. Este proceso en particular se encarga de enviar al IP121 los parámetros correspondientes a la configuración de la siguiente compresión a realizar. En este caso, y haciendo referencia al esquemático representado en la Figura 3.2, el procedimiento a seguir consistirá en modificar los valores de la señal `ctrl`, la cual se encuentra conectada al módulo maestro `ahbtbm`, para que este envíe los parámetros de configuración al DUV.

El código principal relativo al proceso de configuración del IP121 se presenta en el Código 3.7. En el mismo, únicamente se espera hasta que el IP o DUV se encuentre disponible para su configuración mediante la monitorización de su puerto de salida `AwaitingConfig`. Cuando este puerto se activa, se ejecutará uno de los 5 procedimientos de *test* existentes, en función de un identificador que se encuentra en el fichero que contiene los parámetros de configuración del *test bench* `ccsds121_tb_parameters.vhd`. Estos procedimientos son los siguientes:

- **test0.** Ejecución de dos compresiones consecutivas.
- **test2.** Intento de reconfigurar el IP121 mientras se está realizando una compresión.
- **test4.** Detención de la ejecución de una compresión mediante la señal `ForceStop`.
- **test5.** Ejecución de dos compresiones consecutivas, imponiendo una configuración distinta para la segunda compresión.
- **test7.** Envío de una configuración inválida y, tras la respuesta de error por parte del IP121, envío de una configuración válida.

Como puede deducirse de su descripción, para varios de los casos es necesario que la configuración en tiempo de ejecución esté habilitada de cara a su correcto funcionamiento. Es por ello que solamente existe un *test* para la ejecución de cada uno de los 4 últimos procedimientos, mientras que el resto de *test* harán uso del procedimiento básico *test0*. Este procedimiento básico permite ambas posibilidades en cuanto a los permisos de configuración en tiempo de ejecución.

La definición de posibilidad de configuración en tiempo de ejecución, como ya se ha comentado, viene determinada por el parámetro de configuración `EN_RUNCFG_G`. Un valor 1 en este parámetro activa la capacidad de configuración en tiempo de ejecución, mientras que el valor 0 la mantiene inhabilitada.

```

-----
--! Control the test to perform and check signals
-----
process
begin
    D_conf_test <= D_tb;
    ForceStop <= '0';
    while (AwaitingConfig = '0') loop
    end loop;
    wait for (1800ns);
    config_reg(0) <= (others => '0');
    if (EN_RUNCFG_G = 1) then
        ahbtbminit(ctrl);
    end if;
    if (test_id = 4) then
        test4;
    elsif (test_id = 7) then
        test7;
    elsif (test_id = 2) then
        test2;
    elsif (test_id = 5) then
        test5;
    else
        test0;
    end if;
    wait until clk'event and clk = '1';
    if (EN_RUNCFG_G = 1) then
        ahbtbmdone(1, ctrl);
    end if;
    print("*****");
    print("          CCSDS121      Testbench Done");
    print("*****");
    assert false report "**** CCSDS-121 Testbench done ****"
        severity note;
    stop(0);
end process;

```

Código 3.7. Proceso general de configuración del IP

El Código 3.8 muestra la implementación del procedimiento *test0*. En este procedimiento se envían, mediante la señal *ctrl* y en el momento adecuado, los parámetros y las direcciones de los registros de configuración al maestro *ahbtbm*, encargado de gestionar la comunicación AHB. Además, se comprueba el correcto funcionamiento de las señales de salida correspondientes a la interfaz de control (*AwaitingConfig*, *Ready*, *Finished*, etc.), notificando valores anómalos en las mismas.

```

-----
--! Regular test (2 consecutive compressions)
-----
procedure test0 is
    variable address: std_logic_vector(31 downto 0);
begin
    assert (Finished /= '1') report "Finished started with a high
        value" severity warning;
    assert (Ready /= '1') report "Ready started with a high value"
        severity warning;

```

```

assert (AwaitingConfig /= '0') report "AwaitingConfig started with
a low value" severity warning;
if (EN_RUNCFG_G = 1) then
  assert false report "Sending a new configuration..." severity
  note;
  config_reg(1)(31 downto 16) <=
    std_logic_vector(to_unsigned(Nx_tb, 16));
  config_reg(1)(15 downto 15) <=
    std_logic_vector(to_unsigned(CODESET_tb, 1));
  config_reg(1)(14 downto 14) <=
    std_logic_vector(to_unsigned(DISABLE_HEADER_tb, 1));
  config_reg(1)(13 downto 7) <=
    std_logic_vector(to_unsigned(J_tb, 7));
  config_reg(1)(6 downto 0) <=
    std_logic_vector(to_unsigned(W_BUFFER_tb, 7));

  config_reg(2)(31 downto 16) <=
    std_logic_vector(to_unsigned(Ny_tb, 16));
  config_reg(2)(15 downto 3) <=
    std_logic_vector(to_unsigned(REF_SAMPLE_tb, 13));
  config_reg(2)(2 downto 0) <= (others => '0');

  config_reg(3)(31 downto 16) <=
    std_logic_vector(to_unsigned(Nz_tb, 16));
  config_reg(3)(15 downto 10) <=
    std_logic_vector(to_unsigned(D_conf_test, 6));
  config_reg(3)(8 downto 8) <=
    std_logic_vector(to_unsigned(ENDIANESS_tb, 1));
  config_reg(3)(7 downto 6) <=
    std_logic_vector(to_unsigned(PREPROCESSOR_tb, 2));
  config_reg(3)(5 downto 5) <=
    std_logic_vector(to_unsigned(BYPASS_tb, 1));
  config_reg(3)(4 downto 0) <= (others => '0');
  config_reg(0)(0) <= '0';
  address := x"10000000";
  wait until clk'event and clk = '1';
  ahbwrite(x"10000000", config_reg, "10", 4, 2, ctrl);
  -- single write of valid
  config_reg(0)(0) <= '1';
  ahbwrite(address, config_reg(0), "10", "10", '1', 2, true,
  ctrl);
end if;
wait until AwaitingConfig = '0';
assert false report "AwaitingConfig lowered correctly when
configuration was received" severity note;
assert (Ready = '1') report "Ready not asserted correctly when IP
core has been configured" severity warning;
assert false report "Ready asserted correctly when IP core is ready
to receive new samples" severity note;
while Finished = '0' loop
  assert Error_s = '0' report "Unexpected IP core error during
compression" severity error;
  wait until clk'event and clk = '1';
end loop;
assert false report "Finished correctly activated when compression
finished" severity note;
assert (Error_s = '0') report "Unexpected IP core error after
compression" severity error;
wait until AwaitingConfig = '1';
assert false report "AwaitingConfig correctly activated after
compression finished" severity note;
wait until clk'event and clk = '1';

```



```

if (EN_RUNCFG_G = 1) then
  address := x"10000000";
  assert false report "Sending a new configuration..." severity
    note;
  config_reg(0)(0) <= '0';
  ahbwrite(address, config_reg(0), "10", "10", '1', 2, true,
    ctrl);
  ahbwrite(x"10000000", config_reg, "10", 4, 2, ctrl);
  config_reg(0)(0) <= '1';
  ahbwrite(address, config_reg(0), "10", "10", '1', 2, true,
    ctrl);
end if;
while Awaitingconfig = '1' loop
  assert Finished = '1' report "Error between sequential
    compressions, value of Finished shall be kept high" severity
    error;
  wait until clk'event and clk = '1';
end loop;
assert false report "AwaitingConfig lowered correctly when
  configuration was received" severity note;
assert (Ready = '1') report "Ready not asserted correctly when IP
  core has been configured" severity warning;
assert false report "Ready asserted correctly when IP core is ready
  to receive new samples" severity note;
assert Finished = '0' report "Error for sequential compressions,
  Finished shall be de-asserted with AwaitingConfig" severity
  error;
while Finished = '0' loop
  assert Error_s = '0' report "Unexpected IP core error during
  compression" severity error;
  wait until clk'event and clk = '1';
end loop;
assert false report "Finished correctly activated when compression
  finished" severity note;
assert Error_s = '0' report "Unexpected IP core error after
  compression" severity error;
wait until AwaitingConfig = '1';
assert false report "AwaitingConfig correctly activated after
  compression finished" severity note;
assert false report "Two sequential compressions test performed"
  severity note;
end test0;

```

Código 3.8. Procedimiento de ejecución del test0 (dos compresiones consecutivas)

En este proceso, se puede observar el uso de funciones específicas de la biblioteca *GRLIB IP* para modificar la señal `ctrl` que interactúa con el maestro `ahbtbm`. Estas funciones son `ahbtbminit`, `ahbtbmdone` y `ahbwrite`, esta última con dos implementaciones distintas, diferenciadas por sus parámetros (cabe destacar que una de estas implementaciones fue redefinida originalmente para ajustarla a los requerimientos específicos del *test bench*). Como puede observarse, dichas funciones solamente se ejecutan cuando la configuración en tiempo de ejecución se encuentra habilitada (parámetro `EN_RUNCFG_G` con valor 1). Sus funcionalidades son las siguientes:

- **ahbtbminit.** Inicializa la señal de control `ctrl` que recibe como parámetro.
- **ahbtbmdone.** Finaliza la simulación de la señal de control `ctrl` que recibe como parámetro.
- **ahbwrite.** Envía la orden de escritura de un determinado dato en una determina dirección de memoria del esclavo de la comunicación AMBA AHB (en este caso, el DUV). En función de su implementación, puede realizar escrituras normales, en modo ráfaga, etc. También permite incluir ciertos parámetros de configuración para la comunicación mediante el protocolo AHB.

En cuanto al resto de procedimientos de *test*, estos son similares al caso de *test0*, incluyendo las diferencias referentes a su funcionalidad. Sin embargo, comparando todos estos procedimientos, se observa bastante repetición de código, debido a que el proceso de configuración correcto sigue, de forma genérica, unos patrones fijos de observación en las señales de la interfaz de control. Además, como ya se ha comentado, los parámetros de configuración vienen dados por constantes, cuyos valores son modificados directamente entre diferentes *test*, obtenidos del fichero de parámetros `ccsds121_tb_parameters.vhd`.

3.2.5 PROCESO DE TRANSFERENCIA DEL FLUJO DE DATOS DE ENTRADA

A la hora de analizar el proceso de transferencia del flujo de datos de entrada en el *test bench* original, se observa la existencia de una máquina de estados codificada en VHDL. Esta máquina de estados se divide en dos procesos: uno síncrono para asignar los datos de entrada en cada flanco de reloj (si bien es cierto que el *reset*, en este caso, es asíncrono), y otro asíncrono para dar valores actualizados a las variables con los datos correspondientes a las muestras a introducir en el DUV por parte del proceso síncrono en el siguiente ciclo de reloj. Es por este hecho que cada una de las señales de la interfaz de datos de entrada al IP121, así como también otras señales importantes en estos procesos, tienen su señal homóloga asociada, cuyo nombre es el mismo, incluyendo el sufijo `_cmb`.

El Código 3.9 contiene la implementación del proceso síncrono, con simples asignaciones a las señales de la interfaz de datos de entrada del DUV, así como la actualización del estado actual de la máquina de estados. Posteriormente, el Código 3.10 muestra la implementación de la máquina de estados, un código bastante largo y repetitivo.

```

-----
--! Process to provide input preprocessing header words and input
    samples and control
-----
process (clk, rst_n)
begin
    if (rst_n = '0') then
        counter <= (others => '0');
        words_prep_header <= 0;
        state_reg <= idle;
        DataIn <= (others => '0');
        DataIn_Valid <= '0';
        IsHeaderIn <= '0';
        NBitsIn <= (others => '0');
        counter_clks <= (others => '0');
    elsif (clk'event and clk = '1') then
        state_reg <= state_next;
        DataIn <= DataIn_cmb;
        DataIn_Valid <= DataIn_Valid_cmb;
        if (Finished = '1') then
            counter <= (others => '0');
            words_prep_header <= 0;
        else
            counter <= counter_cmb;
            words_prep_header <= words_prep_header_cmb;
        end if;
        counter_clks <= counter_clks + 1;
        IsHeaderIn <= IsHeaderIn_cmb;
        NBitsIn <= NBitsIn_cmb;
    end if;
end process;

```

Código 3.9. Proceso síncrono de la máquina de estados para el flujo de datos de entrada

```

-----
--! Process to provide input preprocessing header words and input
    samples and control
-----
process (Ready, state_reg, counter, counter_clks, DataIn, Finished,
        DataIn_Valid, NBitsIn, IsHeaderIn, words_prep_header,
        AwaitingConfig)
    -----
    --! Simulates an input preprocessing header
    -----
    procedure get_prep_header_word (word: integer) is
    begin
        DataIn_cmb <=
            std_logic_vector(to_unsigned(words_prep_header_cmb,
                DataIn_cmb'LENGTH));
        NBitsIn_cmb <= std_logic_vector(to_unsigned(8,
            NBitsIn_cmb'length));
        words_prep_header_cmb <= words_prep_header + 1;
        DataIn_Valid_cmb <= '1';
        IsHeaderIn_cmb <= '1';
    end get_prep_header_word;
    variable dif_comp: integer := 1;
begin

```

```

counter_cmb <= counter;
words_prep_header_cmb <= words_prep_header;
DataIn_cmb <= DataIn;
DataIn_Valid_cmb <= '0';
block_avl <= '0';
IsHeaderIn_cmb <= '0';
NBitsIn_cmb <= (others => '0');
case state_reg is
  when idle =>
    if (rst_n = '0') then
      if (ini = 0) then
        file_open(stim, stim_file, read_mode);
        words_prep_header_cmb <= 0;
        state_next <= s0_0;
        ini := 1;
      end if;
    else
      state_next <= s0_0;
    end if;
  when s0_0 =>
    if (AwaitingConfig /= '1') then
      if (Ready = '1') then
        if ((D_conf_test <= 8 and EN_RUNCFG_G = 1) or (D_G_tb <= 8
          and EN_RUNCFG_G = 0)) then
          word := 1;
        else
          word := 2;
        end if;
        if ((EN_RUNCFG_G = 1 and DISABLE_HEADER_tb = 0 and
          PREPROCESSOR_tb /= 0) or (EN_RUNCFG_G = 0 and
          DISABLE_HEADER_tb = 0 and PREPROCESSOR_tb /= 0)) then
          prep_header := 17;
        end if;
        if (words_prep_header < prep_header) then
          get_prep_header_word(word);
        else
          state_next <= s1;
          words_prep_header_cmb <= 0;
        end if;
      end if;
      if (Error_s = '1') then
        counter_iter := counter_iter + 1;
        state_next <= s0_0;
        counter_cmb <= (others => '0');
        file_close(stim);
        file_open(stim, stim_file, read_mode);
        ini := 1;
      end if;
    end if;
  when s1 =>
    if (ForceStop='1') then
      counter_iter := counter_iter + 1;
      state_next <= idle;
      counter_cmb <= (others => '0');
      file_close(stim);
      file_open(stim, stim_file, read_mode);
    end if;

```

```

else
  -- It always reads 32 bits, so we have to decide which
  -- part to get every moment (we can use any length from 2 to
  -- 16 at the time), and to consider the endianness as well
  if (Ready = '1') then
    if (counter rem (4/word) = 1) then
      if (EN_RUNCFG_G = 1) then
        --if (ENDIANESS_tb = 0 and word = 2) then
        if (word = 2) then
          DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*2-8-1
            downto word*8) & varaux(word*8*2-1 downto word*8*2-8);
        else
          DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*2-1
            downto word*8);
        end if;
      else
        if (ENDIANESS_tb = 0 and word = 2) then
          DataIn_cmb(D_G_tb-1 downto 0) <= varaux(23 downto 16) &
            varaux(24+(D_G_tb-9) downto 24);
        else
          DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8+D_G_tb-1
            downto word*8);
        end if;
      end if;
      DataIn_Valid_cmb <= '1';
      counter_samples := counter_samples +1;
    elsif (counter rem (4/word) = 2) then
      if (EN_RUNCFG_G = 1) then
        DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*3-1 downto
          word*8*2);
      else
        DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8*2+D_G_tb-1
          downto word*8*2);
      end if;
      DataIn_Valid_cmb <= '1';
      counter_samples := counter_samples +1;
    elsif (counter rem (4/word) = 3) then
      if (EN_RUNCFG_G = 1) then
        DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*4-1 downto
          word*8*3);
      else
        DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8*3+D_G_tb-1
          downto word*8*3);
      end if;
      DataIn_Valid_cmb <= '1';
      counter_samples := counter_samples +1;
    else
      if (not endfile(stim)) then
        read(stim, dataread_stim);
        varaux :=
          std_logic_vector(unsigned(to_signed(dataread_stim,
            varaux'length)));
        if (EN_RUNCFG_G = 1) then
          --if (ENDIANESS_tb = 0 and word = 2) then
          if (word = 2) then
            DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-9
              downto 0) & varaux(word*8-1 downto 8);
          else
            DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-1
              downto 0);
          end if;
        end if;
      end if;
    end if;
  end if;
end if;

```

```

        else
            if (ENDIANESS_tb = 0 and word = 2) then
                DataIn_cmb(D_G_tb-1 downto 0) <= varaux(7 downto
                    0) & varaux(D_G_tb-1 downto 8);
            else
                DataIn_cmb(D_G_tb-1 downto 0) <= varaux(D_G_tb-1
                    downto 0);
            end if;
        end if;
        DataIn_Valid_cmb <= '1';
        counter_samples := counter_samples + 1;
    end if;
end if;
if (counter = to_unsigned(J_tb-1, counter'length)) then
    counter_cmb <= (others => '0');
    state_next <= s3;
else
    counter_cmb <= counter + 1;
    state_next <= s1;
end if;
block_avl <= '1';
end if;
when s2 =>
    block_avl <= '0';
    if (Finished = '0') then
        if (counter = to_unsigned(J_tb-1, counter'length)) then
            counter_cmb <= (others => '0');
            state_next <= s3;
        else
            counter_cmb <= counter + 1;
            state_next <= s2;
        end if;
    else
        counter_iter := counter_iter + 1;
        state_next <= idle;
        counter_cmb <= (others => '0');
        file_close(stim);
        -- Only 05_test uses a different stimulus file
        if (test_id = 5) then
            file_open(stim, sec_stim_file, read_mode);
        else
            file_open(stim, stim_file, read_mode);
        end if;
    end if;
when s3 =>
    if (Finished = '0') then
        if (Ready = '1' and AwaitingConfig /= '1') then
            if (counter rem (4/word) = 1) then
                new_read_req := true;
                if (EN_RUNCFG_G = 1) then
                    --if (ENDIANESS_tb = 0 and word = 2) then
                    if (word = 2) then
                        DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*2-8-1
                            downto word*8) & varaux(word*8*2-1 downto
                                word*8*2-8);
                    else
                        DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*2-1
                            downto word*8);
                    end if;
                end if;
            end if;
        end if;
    end if;
end if;

```

```

else
  if (ENDIANESS_tb = 0 and word = 2) then
    DataIn_cmb(D_G_tb-1 downto 0) <= varaux(23 downto 16) &
      varaux(24+(D_G_tb-9) downto 24);
  else
    DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8+D_G_tb-1
      downto word*8);
  end if;
end if;
DataIn_Valid_cmb <= '1';
counter_samples := counter_samples + 1;
if (counter = to_unsigned(J_tb-1, counter'length)) then
  counter_cmb <= (others => '0');
else
  counter_cmb <= counter + 1;
end if;
block_avl <= '1';
elsif (counter rem (4/word) = 2) then
  if (EN_RUNCFG_G = 1) then
    DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*3-1 downto
      word*8*2);
  else
    DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8*2+D_G_tb-1 downto
      word*8*2);
  end if;
  DataIn_Valid_cmb <= '1';
  counter_samples := counter_samples + 1;
  if (counter = to_unsigned(J_tb-1, counter'length)) then
    counter_cmb <= (others => '0');
  else
    counter_cmb <= counter + 1;
  end if;
  block_avl <= '1';
elsif (counter rem (4/word) = 3) then
  if (EN_RUNCFG_G = 1) then
    DataIn_cmb(word*8-1 downto 0) <= varaux(word*8*4-1 downto
      word*8*3);
  else
    DataIn_cmb(D_G_tb-1 downto 0) <= varaux(word*8*3+D_G_tb-1 downto
      word*8*3);
  end if;
  DataIn_Valid_cmb <= '1';
  counter_samples := counter_samples + 1;
  if (counter = to_unsigned(J_tb-1, counter'length)) then
    counter_cmb <= (others => '0');
  else
    counter_cmb <= counter + 1;
  end if;
  block_avl <= '1';
else
  if (not endfile(stim) and new_read_req) then
    new_read_req := false;
    read(stim, dataread_stim);
    varaux := std_logic_vector(unsigned(to_signed(dataread_stim,
      varaux'length)));
    if (EN_RUNCFG_G = 1) then
      --if (ENDIANESS_tb = 0 and word = 2) then
      if (word = 2) then
        DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-9 downto 0)
          & varaux(word*8-1 downto 8);
      end if;
    end if;
  end if;
end if;

```

```

        else
            DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-1
                downto 0);
        end if;
    else
        if (ENDIANESS_tb = 0 and word = 2) then
            DataIn_cmb(D_G_tb-1 downto 0) <= varaux(7 downto 0)
                & varaux(D_G_tb-1 downto 8);
        else
            DataIn_cmb(D_G_tb-1 downto 0) <= varaux(D_G_tb-1
                downto 0);
        end if;
    end if;
    DataIn_Valid_cmb <= '1';
    counter_samples := counter_samples + 1;
    if (counter = to_unsigned(J_tb-1, counter'length)) then
        counter_cmb <= (others => '0');
    else
        counter_cmb <= counter + 1;
    end if;
    block_avl <= '1';
else
    if (EN_RUNCFG_G = 1) then
        --if (ENDIANESS_tb = 0 and word = 2) then
        if (word = 2) then
            DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-9
                downto 0) & varaux(word*8-1 downto 8);
        else
            DataIn_cmb(word*8-1 downto 0) <= varaux(word*8-1
                downto 0);
        end if;
    else
        if (ENDIANESS_tb = 0 and word = 2) then
            DataIn_cmb(D_G_tb-1 downto 0) <= varaux(7 downto 0)
                & varaux(D_G_tb-1 downto 8);
        else
            DataIn_cmb(D_G_tb-1 downto 0) <= varaux(D_G_tb-1
                downto 0);
        end if;
    end if;
    DataIn_Valid_cmb <= '1';
    counter_samples := counter_samples + 1;
    if (counter = to_unsigned(J_tb-1, counter'length)) then
        counter_cmb <= (others => '0');
    else
        counter_cmb <= counter + 1;
    end if;
    block_avl <= '1';
end if;
end if;
else
    counter_iter := counter_iter + 1;
    state_next <= s0_0;
    counter_cmb <= (others => '0');
    file_close(stim);

```



```

-- Only 05_test uses a different stimulus file
if (test_id = 5) then
  DataIn_cmb(D_G_tb-1 downto 0) <= (others => '0');
  file_open(stim, sec_stim_file, read_mode);
else
  file_open(stim, stim_file, read_mode);
end if;
ini := 1;
end if;
end case;
end process;

```

Código 3.10. Proceso asíncrono de la máquina de estados para el flujo de datos de entrada

Como se muestra en este código, se trata de una máquina de 5 estados con un código extenso y bastante repetitivo en algunas ocasiones, algo que podría optimizarse. En general, se realiza la lectura de los datos desde un fichero básico (tipo `dat`) y, al igual que en el proceso de configuración, se introduce el flujo de muestras en el IP121 en función de los valores que presenten las señales de la interfaz de control, así como de las señales de reloj y `reset` que gobiernan el sistema. Además, cabe destacar que se hace uso de un procedimiento VHDL, denominado `get_prep_header_word`, cuya funcionalidad consiste en simular la generación de cabeceras procedentes de un bloque preprocesador de cara a su uso como datos de entrada al DUV, y que también se tienen en cuenta determinados valores de los parámetros de configuración para gestionar el formato del flujo de datos de entrada.

Con el objetivo de ofrecer una mejor visualización de la máquina de estados, la Figura 3.3 muestra un diagrama de la misma. Posteriormente, se describe brevemente la funcionalidad de cada uno de los estados.

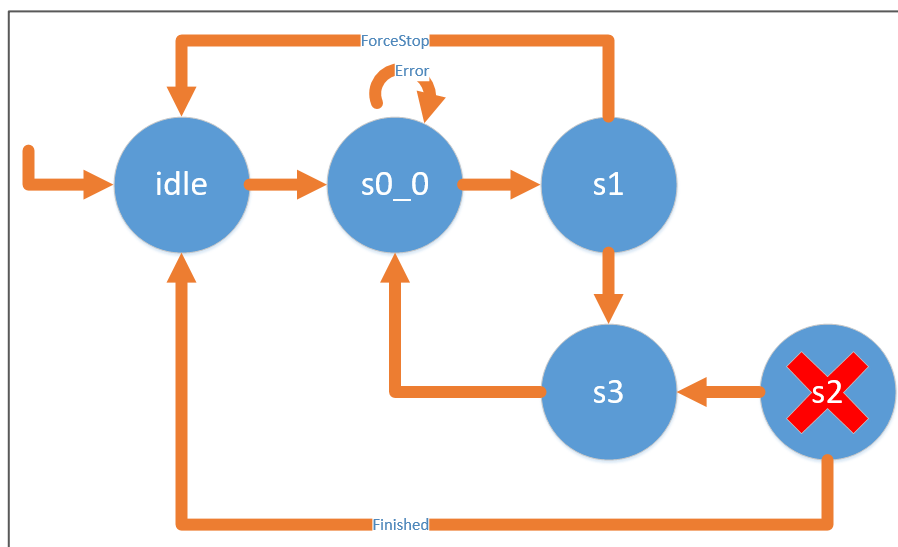


Figura 3.3. Diagrama de la máquina de estados original para el flujo de datos de entrada

- **idle**. Estado inicial en el que se abre el archivo que contiene los estímulos de entrada, se inicializan ciertos parámetros, y se pasa directamente al estado *s0_0*.
- **s0_0**. Estado que espera a que el IP121 haya sido configurado correctamente y se encuentre listo para recibir muestras de datos de entrada. Cuando esto suceda, se pasa al estado *s1*.
- **s1**. Estado que envía muestras de datos de entrada al IP121. Cuando un contador determinado alcanza un valor específico, se pasa al estado *s3*. En caso de observarse la señal `ForceStop` activa, se pasa al estado *idle*.
- **s2**. En el análisis de la máquina de estados se observa que nunca se accede a este estado, por lo que su implementación es innecesaria.
- **s3**. Estado que envía muestras de datos de entrada al IP121. Cuando se observa la señal `Finished` activa, se pasa al estado *s0_0*.

3.2.6 PROCESO DE RECEPCIÓN DEL FLUJO DE DATOS DE SALIDA Y COMPARACIÓN CON LOS VALORES DE REFERENCIA

El último de los grandes procesos que integran el entorno de verificación es el relativo a la recepción del flujo de datos de salida, así como la comparación de estos con los valores obtenidos de la ejecución de los mismos vectores de estímulos sobre el modelo de referencia. Estos últimos datos se encuentran almacenados en archivos básicos (tipo `dat`), al igual que los estímulos de entrada.

La implementación VHDL de este proceso se muestra en el Código 3.11. En el mismo, para cada flanco de bajada en la señal de reloj del sistema, se evalúan los valores de las señales de `reset`, `ForceStop` y `Error`, pues si alguna de ellas se encontrase activa, la compresión se interrumpiría antes de finalizar el proceso y sin haberse recibido un flujo completo de datos de salida. Por lo tanto, no sería posible establecer una comparación válida con los valores de referencia con el fin de verificar la funcionalidad del DUV. En caso contrario, se reciben los datos de salida y se almacenan en un fichero externo.

```

-----
--! Process to store output words after some control, and to compare
output file generated by the IP Core and the reference file provided
-----
process (clk)
begin
  if (clk'event and clk = '0') then
    if (rst_n = '0') then
      ini := 0;
      fin := 0;
      sec := 0;
      sim_successful <= false;
    elsif (ForceStop = '1') then
      assert false report "Comparison not possible because there
        has been a ForceStop assertion" severity note;
      file_close(output);
      ini := 0;
      fin := 0;
      error_f := 0;
      sim_successful <= false;
    elsif (Error_s = '1') then
      if (error_f = 1) then
        assert false report "Comparison not possible because there
          has not been compression performed (configuration error)"
          severity note;
        file_close(output);
        ini := 0;
        fin := 0;
        error_f := 0;
      end if;
      sim_successful <= false;
    else
      if (buff_full = '1' and (AwaitingConfig = '0')) then
        if (ini = 0) then
          file_open(output, out_file, write_mode);
          ini := 1;
          fin := 1;
        end if;
        sim_successful <= false;
        if (EN_RUNCFG_G = 1) then
          size := W_BUFFER_tb;
        else
          size := W_BUFFER_G_tb;
        end if;
        for i in 0 to (size/8) - 1 loop
          probe:= buff_out((((size/8) -1-i)+1)*8-1 downto
            ((size/8) -1-i)*8);
          uns := unsigned(probe);
          int := to_integer(uns);
          pixel_file:= character'val(int);
          write(output,pixel_file);
        end loop;
      end if;
    end if;
  end if;
end process;

```

```

    if (Finished = '1') then
        if (fin = 1) then
            file_close(output);
            ini := 0;
            fin := 0;
            error_f := 0;
            compare_files(sec);
            if (test_id = 5) then
                sec := 1;
            end if;
        end if;
    end if;
end if;
end if;
end process;

```

Código 3.11. Proceso de recepción del flujo de datos de salida

Finalmente, cuando se observa que la señal `Finished` se encuentra activa, lo que indica que el proceso de compresión ha finalizado y el flujo de datos de salida ha sido recibido y almacenado en su totalidad, se ejecuta el procedimiento `compare_files`. La descripción VHDL de este procedimiento se muestra en el Código 3.12, el cual, en determinadas partes, también posee ciertas características de código repetitivo.

En dicho código, se abren ambos archivos (el que almacena los datos de salida y el que contiene los valores de referencia) y, tras su apertura, se efectúa una comparación *byte a byte* de su contenido. En dicha comparación se tiene en cuenta la posible existencia de cabeceras de un bloque preprocesador, así como de anulación de residuos, o que alguno de los ficheros contenga un mayor número de muestras de datos. En caso de existir errores de cualquier índole, se notificaría este hecho por consola.

```

procedure compare_files (sec: integer) is
    -- sec: second file flag for 05_test
    function ceil (A: integer; B: integer) return integer is
        variable q: integer := 0;
        variable r: integer := 0;
    begin
        q := A/B;
        r := A-q*B;
        if (r > 0) then
            q := q+1;
        end if;
        return q;
    end function;
begin
    -- Opening both files
    file_open(output, out_file, read_mode);

```

```

if (sec = 0) then
    file_open(reference, ref_file, read_mode);
elsif (sec = 1) then
    file_open(reference, sec_ref_file, read_mode);
end if;
-- Checking if it's necessary to bypass some preprocessing-header
if (EN_RUNCFG_G = 1) then
    if (DISABLE_HEADER_tb = 0) then
        if (PREPROCESSOR_tb /= 0) then
            number_prep_header_words := 17;
            for i in 0 to number_prep_header_words-1 loop
                read(output, output_byte);
                pixel_counter := pixel_counter + 1;
            end loop;
        end if;
    end if;
elsif (EN_RUNCFG_G = 0) then
    if (DISABLE_HEADER_tb = 0) then
        if (PREPROCESSOR_tb /= 0) then
            number_prep_header_words := 17;
            for i in 0 to number_prep_header_words-1 loop
                read(output, output_byte);
                pixel_counter := pixel_counter + 1;
            end loop;
        end if;
    end if;
end if;
-- Read the output words and comparison part (byte per byte)
-- (considering certain circumstances)
if ((EN_RUNCFG_G = 0 and BYPASS_tb = 0) or (EN_RUNCFG_G = 1 and
BYPASS_tb = 0)) then
    while (not endfile(reference) and (not endfile(output))) loop
        read(output, output_byte);
        read(reference, ref_byte);
        pixel_counter := pixel_counter +1;
        if ref_byte /= output_byte then
            assert false report "Problems in final stream" severity
                error;
        end if;
    end loop;
    while (not endfile(reference)) loop
        read(reference, ref_byte);
        if (ref_byte /= character'val(0)) then
            assert false report "Reference file has more samples"
                severity error;
        end if;
    end loop;
    while (not endfile(output)) loop
        read(output, output_byte);
        if (output_byte /= character'val(0)) then
            assert false report "Output file has more samples"
                severity error;
        end if;
    end loop;
-- We need to check the bypassed residuals here
else
    if (EN_RUNCFG_G = 0) then
        bytes_discard_top := (W_BUFFER_G_tb/8) - ceil(D_G_tb,8);
    else
        bytes_discard_top := (W_BUFFER_tb/8) - ceil(D_tb,8);
    end if;
end if;

```

```

while (not endfile(reference) and (not endfile(output))) loop
  for bytes_discard in 0 to bytes_discard_top-1 loop
    read(output, output_byte);
  end loop;
  for bytes_2comp in 0 to (W_BUFFER_G_tb/8) -
  bytes_discard_top-1 loop
    read(output, output_byte);
    read(reference, ref_byte);
    if ref_byte /= output_byte then
      assert false report "Problems in final stream" severity
      error;
    end if;
  end loop;
end loop;
end if;
assert false report "Comparison was successful!" severity note;
sim successful <= true;
file_close(output);
file_close(reference);
end compare_files;

```

Código 3.12. Procedimiento de comparación de los datos de salida con sus valores de referencia

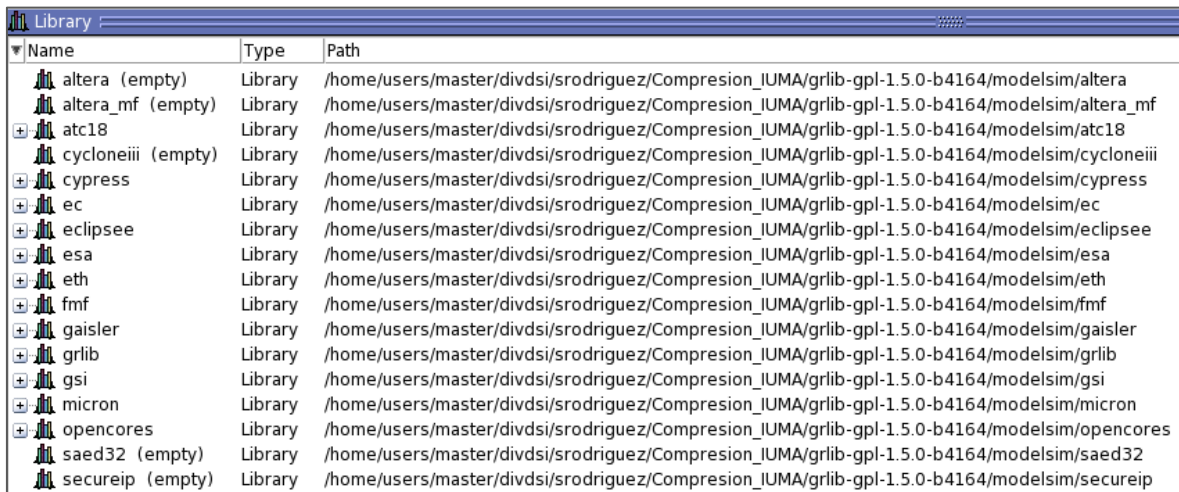
3.2.7 EJECUCIÓN DE LOS TEST

Tras haber generado los archivos VHDL con los estímulos de configuración del entorno de verificación, y haber descrito el *test bench* original de referencia, se va a describir el procedimiento a seguir para la ejecución de los diferentes *test*. Esta ejecución se llevará a cabo sobre la herramienta *QuestaSim*, concretamente en su versión *10.4b*.

En primer lugar, se debe tener acceso local a los archivos de la biblioteca *GRLIB IP*, el cual puede obtenerse desde la página web asociada al enlace www.gaisler.com. En esta página web se encuentra la versión más reciente de la biblioteca, que en el momento de elaboración de este TFM es la *2017.3-b4208*, si bien es cierto que en este caso se ha hecho uso de la versión *1.5.0-b4164*, pues es la que se utiliza en la implementación del *test bench* original. Tras incluir la biblioteca, se inicia la herramienta *QuestaSim* y se carga el proyecto *shyloc121.mpf*, que se encuentra en la carpeta *modelsim* y esta, a su vez, en el directorio principal.

Con el proyecto ya abierto en la herramienta *QuestaSim*, resulta necesario asegurarse de que las diferentes bibliotecas de *GRLIB IP* están correctamente enlazadas a la ruta local en la que se encuentran. Para comprobarlo, se accede a la ventana *Library* en la herramienta *QuestaSim* y se observa el valor de la variable *Path* para cada una de ellas. Si este valor no coincide con el directorio local de cada biblioteca, ha de modificarse manualmente. Un ejemplo de cómo debería ser el

aspecto de la ventana *Library*, una vez establecidos los valores correctos en las variables *Path* para algunas bibliotecas, es el que se muestra en la Figura 3.4.



Name	Type	Path
altera (empty)	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/altera
altera_mf (empty)	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/altera_mf
atc18	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/atc18
cycloneiii (empty)	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/cycloneiii
cypress	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/cypress
ec	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/ec
eclipsee	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/eclipsee
esa	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/esa
eth	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/eth
fmf	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/fmf
gaisler	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/gaisler
glibc	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/glibc
gsi	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/gsi
micron	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/micron
opencores	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/opencores
saed32 (empty)	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/saed32
secureip (empty)	Library	/home/users/master/divdsi/srodriguez/Compresion_IUMA/glibc-gpl-1.5.0-b4164/modelsim/secureip

Figura 3.4. Bibliotecas de **GRLIB IP** correctamente enlazadas a **QuestaSim**

Finalmente, para ejecutar el, o los *test* que se desee, se hace uso de la ventana de comandos de la herramienta *QuestaSim*. De este modo, se utilizará el comando `do` sobre alguno de los *scripts* de ejecución de *test*. Estos *scripts* se encuentran en la ruta `/modelsim/tb_scripts` desde el directorio raíz del proyecto.

Este proyecto está preparado para ejecutar la simulación de todos los *test* a partir del archivo `all_tests.do`, desde el que se irá invocando la ejecución secuencial de cada uno de los *test*. Como al abrir el proyecto `shyloc121.mpf` la consola se sitúa directamente en el directorio `modelsim`, habrá que ejecutar en la consola de la herramienta *QuestaSim* el comando que se muestra en el Código 3.13. En el caso de que se desee simular solamente un *test* en particular y, por ejemplo, visualizar sus formas de onda, se ejecutaría el *script* tipo `do` correspondiente, si bien es cierto que previamente habría que modificarlo pues, como ya se ha comentado, el entorno está diseñado para la ejecución secuencial de todos los *test*.

```
QuestaSim> do tb_scripts/all_tests.do
```

Código 3.13. Comando de ejecución de los *test* en **QuestaSim**

Capítulo 4. DESARROLLO DEL ENTORNO DE VERIFICACIÓN BASADO EN UVM

En este capítulo se presenta el entorno UVM desarrollado para la verificación funcional del IP121 descrito en el Capítulo 3. Así, se mostrará la estructura final del mismo y se detallará la implementación de cada uno de los componentes que lo integran. El desarrollo de este entorno UVM se basará, fundamentalmente, en la implementación del *test bench* original descrito en lenguaje VHDL, con el fin de mantener la compatibilidad de los aspectos considerados en la verificación funcional del DUV.

Sin embargo, en una primera aproximación, se pretende crear un entorno UVM funcional evitando la complejidad asociada al IP121 que actuará como DUV en el entorno de verificación final. Con ello, se comprobará la correcta asimilación e implementación de los diversos conceptos relacionados con el estándar UVM como, por ejemplo, el mecanismo de configuración a través de la base de datos, el uso adecuado de la *Factory* y de las fases de UVM, el mecanismo de mensajes, etc.

Por lo tanto, se requiere disponer de un IP más simple, para lo cual se utiliza la herramienta *HDL Coder*, de *MATLAB*, que permite la generación de código HDL (bien *Verilog*, o bien VHDL) a partir de funciones o modelos descritos en *MATLAB* [20]. Es por ello que, previamente al desarrollo del entorno de verificación basado en UVM para el IP121, se explicará el proceso seguido para la generación de un IP básico a partir de esta herramienta.

4.1 GENERACIÓN DE UN IP USANDO *HDL CODER*

En esta primera aproximación al desarrollo de un entorno de verificación UVM funcional, se ha decidido trabajar sobre una función ya descrita en *MATLAB*, que implementa un filtro de detección de bordes de tipo Sobel. En primer lugar, se accede a la ruta `/R2015b/toolbox/hdlcoder/hdlcoderdemos/matlabhdlcoderdemos` desde el directorio de instalación de *MATLAB*, que se encuentra instalado en su versión *R2015b*. Desde este directorio, se copiarán los archivos `mlhdlc_sobelfilter.m`, `mlhdlc_sobelfilter_tb.m` y `mlhdlc_img_yuv.tif` en un nuevo directorio de trabajo creado para la generación del IP. Estos ficheros se corresponden con la función *MATLAB* del filtro Sobel (cuya implementación se recoge

en el Código 4.1), su *test bench* (que se muestra en el Código 4.2), y la imagen sobre la que se ejecuta este *test bench*, respectivamente.

```

function [x_out, y_out, data_out] = ...
    mlhdlc_sobelfilter(x_in, y_in, data_in)

% Copyright 2011-2015 The MathWorks, Inc.

persistent lineBuffer1 lineBuffer2 k

WIDTH = 752;
HEIGHT = 480;
CMAX = 2000;
Kx = [ 1  0 -1; ...
      2  0 -2; ...
      1  0 -1];
Ky = [ 1  2  1; ...
      0  0  0; ...
      -1 -2 -1];

if isempty(k)
    k = zeros(3);
end
if isempty(lineBuffer1)
    lineBuffer1 = zeros(1,WIDTH);
    lineBuffer2 = zeros(1,WIDTH);
end

xTemp = x_in-1;
yTemp = y_in-1;
if xTemp < 0
    xOutTemp = CMAX;
else
    xOutTemp = xTemp;
end
if yTemp < 0
    yOutTemp = CMAX;
else
    yOutTemp = yTemp;
end
if x_in >= 0 && x_in < WIDTH
    dataValid = 1;
else
    dataValid = 0;
end
if dataValid == 1
    lbIndex = x_in+1;
else
    lbIndex = 1;
end
l1 = lineBuffer1(lbIndex);
l2 = lineBuffer2(lbIndex);
if dataValid == 1
    lb1WriteValue = l2;
    lb2WriteValue = data_in;
    l = [l1 l2 data_in]';

```

```

else
    lb1WriteValue = l1;
    lb2WriteValue = l2;
    l = zeros(3,1);
end
lineBuffer1(lbIndex) = lb1WriteValue;
lineBuffer2(lbIndex) = lb2WriteValue;
k = [k(:,2:3) l];

if yOutTemp == 0
    k(1,:) = k(2,:);
elseif yOutTemp == HEIGHT-1
    k(3,:) = k(2,:);
end
if xOutTemp == 0
    k(:,1) = k(:,2);
elseif xOutTemp == WIDTH-1
    k(:,3) = k(:,2);
end
%Gx = conv2(k,Kx,'valid');
%Gy = conv2(k,Ky,'valid');
Gx = 0;
Gy = 0;
for yi = 1:3,
    for xi = 1:3,
        Gx = Gx+k(yi,xi)*Kx(yi,xi);
        Gy = Gy+k(yi,xi)*Ky(yi,xi);
    end
end
G = abs(Gx) + abs(Gy);
Gd = floor(G/4);
Gdm = min(Gd,255);

x_out = xOutTemp;
y_out = yOutTemp;
if yOutTemp < HEIGHT && xOutTemp < WIDTH
    data_out = Gdm;
else
    data_out = 0;
end
end
    
```

 Código 4.1. Función *MATLAB* `mlhdlc_sobelfilter`

```

% Copyright 2011-2015 The MathWorks, Inc.
FRAMES = 1;
WIDTH = 752;
HEIGHT = 480;
HBLANK = 10;%748;
VBLANK = 10;%120;

vidData = double(imread('mlhdlc_img_yuv.tif'));

for f = 1:FRAMES
    xOut = 1;
    yOut = 1;
    vidOut = zeros(HEIGHT, WIDTH, 3);
    
```

```

for y = 0:HEIGHT+VBLANK-1
  for x = 0:WIDTH+HBLANK-1
    if y >= 0 && y < HEIGHT && x >= 0 && x < WIDTH
      pixData = vidData(y+1,x+1,1);
    else
      pixData = 0;
    end
    [xOut, yOut, dataOut] = ...
      mlhdlc_sobelfilter(x, y, pixData);
    if yOut >= 0 && yOut < HEIGHT && xOut >= 0 && xOut < WIDTH
      vidOut(yOut+1,xOut+1,:) = dataOut;
    end
  end
end

figure(1);
subplot(1,2,1);
imshow(uint8(vidData(:,:,1)));
subplot(1,2,2);
imshow(uint8(vidOut));
drawnow;

end

```

Código 4.2. Archivo `mlhdlc_sobelfilter_tb.m`

Una vez importados estos archivos al directorio específico, es posible iniciar el software *MATLAB*, acceder a este nuevo directorio, y ejecutar el *test bench*, con el fin de comprobar su correcto funcionamiento. El resultado de esta ejecución será el que se muestra en la Figura 4.1, con la imagen de entrada en escala de grises a la izquierda, y la imagen resultante de la aplicación del filtro Sobel a la derecha.

Figura 4.1. Resultado de la ejecución del *test bench* `mlhdlc_sobelfilter_tb.m`

Para poder trabajar con la herramienta *HDL Coder* y generar el código HDL de la función citada anteriormente, así como de su *test bench*, resulta necesario eliminar las funciones `imread` e `imshow` del *test bench*, relativas a la lectura y a la muestra en pantalla de un archivo de imagen en *MATLAB*. Esto se debe a que dichas funciones no están soportadas de forma nativa en código HDL.

Teniendo en cuenta el problema asociado a la lectura de la imagen (acción que resulta imprescindible), se almacena el contenido de la misma como una matriz de datos en un archivo tipo `dat`, que se denominará `imagen.dat`. De esta forma, es posible realizar la lectura de este archivo sin hacer uso de la función `imread`. Para efectuar este almacenamiento, se accede al directorio de trabajo en el software *MATLAB* y se ejecutan las sentencias que se muestran en el Código 4.3 desde la ventana de comandos de *MATLAB*.

```
>> vidData = double(imread('mlhdlc_img_yuv.tif'));
>> save imagen.dat vidData -tabs;
```

Código 4.3. Almacenamiento de la imagen `mlhdlc_img_yuv.tif` como matriz de datos

Seguidamente, se modifica la lectura de la imagen para hacerlo desde el nuevo archivo creado. Para ello, en el código del *test bench* se efectúa la sustitución de la línea de código que se muestra en el Código 4.4. Llegados a este punto, es posible verificar el correcto funcionamiento de la nueva implementación, para lo cual se volvería a ejecutar el *test bench* en *MATLAB*, comprobando que se obtienen los mismos resultados.

```
vidData = double(imread('mlhdlc_img_yuv.tif'));
↓
load -mat imagen.dat;
```

Código 4.4. Modificación del modo de lectura de la imagen en el *test bench*

Por otra parte, para el caso de la función `imshow`, el procedimiento es más sencillo, pues únicamente se debe eliminar el fragmento de código que se muestra en el Código 4.5, ya que la acción de mostrar por pantalla la imagen resultante no es de interés, ni tampoco realizable en un entorno hardware. Este código se encuentra situado al final del archivo `mlhdlc_sobelfilter_tb.m`.

```

figure(1);
subplot(1,2,1);
imshow(uint8(vidData(:,:,1)));
subplot(1,2,2);
imshow(uint8(vidOut));
drawnow;

```

Código 4.5. Código a eliminar del test bench

Habiendo llevado a cabo estos pasos, ya es posible comenzar la generación de la implementación HDL del IP. Para ello, se abre la aplicación *HDL Coder* desde el entorno *MATLAB* (pestaña *APPS*) y se crea un nuevo proyecto, que inicialmente estará vacío. El siguiente paso consiste en incluir, mediante la opción *Add files*, la función *MATLAB* desde la que se partirá para generar el IP descrito en HDL, así como su *test bench* (que ha sido modificado previamente).

Este proceso se muestra en la Figura 4.2, donde estos archivos ya han sido añadidos. Además, se puede observar cómo el software reconoce los parámetros de entrada de la función *MATLAB*, que serán establecidos como entradas del nuevo IP descrito en HDL. Llegados a este punto, se inicia la ejecución de *Workflow Advisor* mediante el botón habilitado para ello, que se abrirá en una nueva ventana, desde la cual se realizará el resto del procedimiento de generación del código HDL.

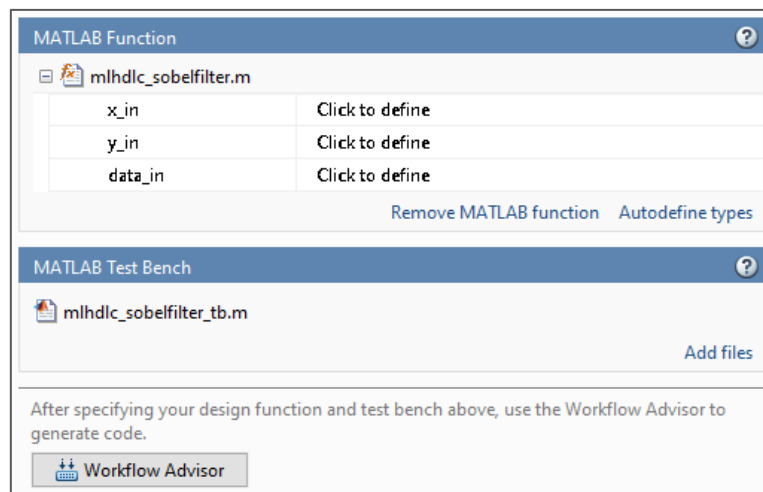


Figura 4.2. Ventana de un proyecto en HDL Coder

El procedimiento en *Workflow Advisor* consiste, en primer lugar, en trasladar el diseño *MATLAB* en punto flotante a un diseño en punto fijo, desde el cual es posible generar el código HDL. En la Figura 4.3 se representa un ejemplo de la ventana de *Workflow Advisor*, en la que se puede ver que ya se han efectuado correctamente algunos de los pasos indicados.

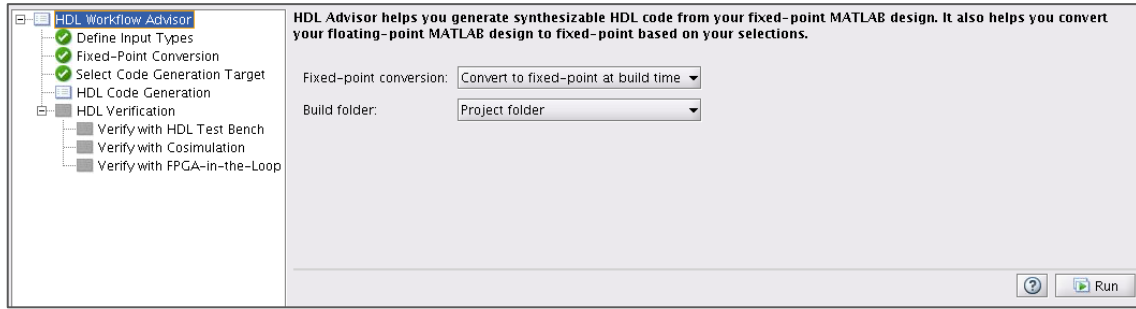


Figura 4.3. Ventana del *Workflow Advisor* de *HDL Coder*

De entre las diferentes tareas mostradas, las más importantes para el caso de una simulación usando el *test bench* son las siguientes:

- **Define Input Types.** En este paso se asigna un tipo a cada una de las entradas que tendrá el nuevo IP. Esto puede hacerse manualmente, o bien lanzar el software para que lo haga de forma automática.
- **Fixed-Point Conversion.** Ejecutando esta tarea, se efectúa la conversión de punto flotante a punto fijo del archivo *MATLAB* relativo a la función y se evalúa su validez mediante la ejecución del *test bench* sobre la misma. Tras ello, se permiten acciones como mostrar medidas de cobertura en el código de la función, o comparar los resultados obtenidos entre las implementaciones en punto flotante y en punto fijo.
- **HDL Code Generation.** Este paso es crítico y realiza la generación del código HDL, tanto del nuevo IP, como de su *test bench*, por lo que puede durar hasta varias horas, dependiendo de la complejidad de los mismos. Ofrece una amplia posibilidad de configuración, como puede ser el tipo de *reset* (síncrono o asíncrono, activo a nivel alto o a nivel bajo), el flanco de activación de reloj (subida o bajada), el nombre de los archivos a generar, las opciones de compilación y de simulación, etc. Sin embargo, el único aspecto importante en este momento es la selección del lenguaje HDL a utilizar, entre VHDL y *Verilog*, tal y como puede verse en la Figura 4.4. En este caso, para actuar de acuerdo al procedimiento que se seguirá con el IP multi-interfaz para la compresión de imágenes, se ha escogido VHDL.

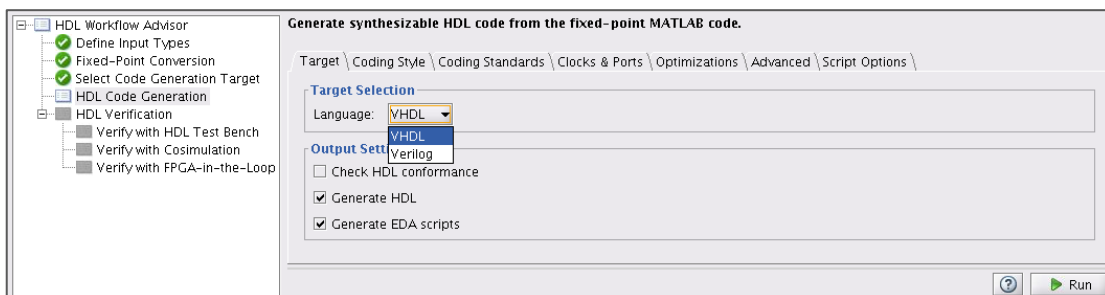


Figura 4.4. Selección del lenguaje para la generación del código HDL

- **Verify with HDL Test Bench.** En esta opción se ofrece la posibilidad de ejecución en *MATLAB* del *test bench* HDL creado, con el fin de comprobar su correcto funcionamiento. Sin embargo, resulta más fiable realizar una verificación mediante una simulación utilizando la herramienta *QuestaSim* y la visualización de las formas de onda. Esto último es sencillo, debido a que *HDL Coder* también genera un archivo tipo *do* ejecutable en *QuestaSim* con los comandos necesarios para ello.

Una vez finalizado el procedimiento de generación del código VHDL, todos los archivos relativos al mismo y a los ejecutables tipo *do* para su compilación y simulación, se encuentran en la ruta */codegen/mlhdlc_sobelfilter/hdlsrc* desde el directorio en el que se ha trabajado con *HDL Coder*. Los archivos más relevantes de entre todos los generados son: el IP VHDL (denominado *mlhdlc_sobelfilter_fixpt.vhd*), el *test bench* VHDL (que tiene por nombre *mlhdlc_sobelfilter_fixpt_tb.vhd*), y el ejecutable tipo *do* para la simulación en *QuestaSim* (*mlhdlc_sobelfilter_fixpt_tb_sim.do*).

Para poder detener la simulación y visualizar las formas de onda, es necesario eliminar dos líneas de código del ejecutable tipo *do* para la simulación, las cuales se muestran en el Código 4.6. En este código también se especifica la sentencia de ejecución del *test bench* desde la ventana de comandos de *QuestaSim*, la cual debe invocarse tras situar la consola en el directorio */codegen/mlhdlc_sobelfilter/hdlsrc*.

Eliminar del comienzo del ejecutable tipo <i>do</i> :	<code>onbreak {quit -f}</code>
Eliminar del final del ejecutable tipo <i>do</i> :	<code>quit -f</code>
Ejecutar el <i>test bench</i> :	<code>do mlhdlc_sobelfilter_fixpt_tb_sim.do</code>

Código 4.6. Simulación en QuestaSim del código VHDL generado en HDL Coder

Finalmente, es posible detener y reanudar la simulación en tiempo de ejecución, o bien esperar a que finalice. En cualquiera de los dos casos se puede comprobar en cada momento, y mediante la visualización de las formas de onda, el estado de las señales de entrada/salida del IP. De este modo, se verifica que las señales de salida obtenidas se corresponden con sus valores esperados, tal y como se muestra en la Figura 4.5. Si la simulación finaliza según lo esperado, también se mostrará un mensaje por consola indicando que el *test* ha sido exitoso.

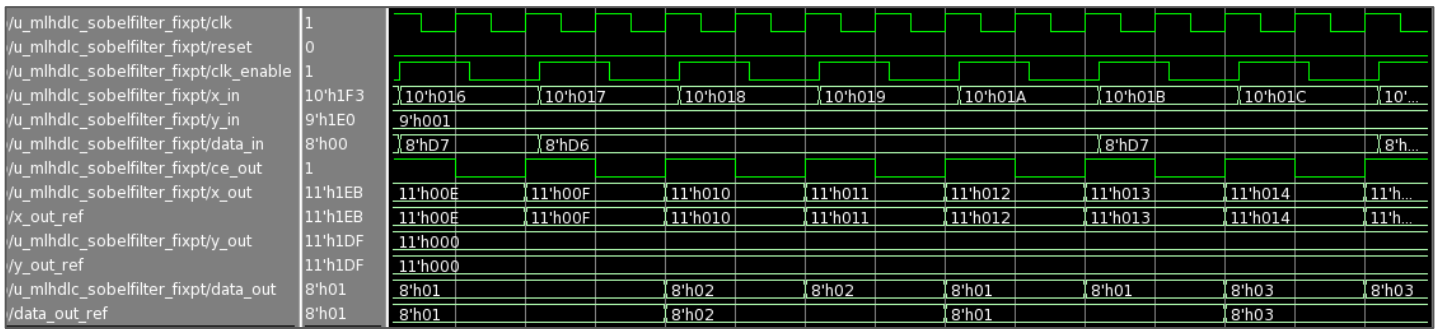


Figura 4.5. Formas de onda de la simulación en *QuestaSim* del IP generado en *HDL Coder*

4.2 INTEGRACIÓN DE UN DUV DESCRITO EN VHDL

Tanto el IP generado en *HDL Coder*, como el IP multi-interfaz para la compresión de imágenes cuya verificación funcional a partir de un entorno UVM es el objetivo de este TFM, están descritos en lenguaje VHDL. Sin embargo, como ya se comentó en el Capítulo 2, UVM usa el lenguaje hardware *SystemVerilog*. Por tanto, al contrario que para un DUV definido en *Verilog*, cuya interfaz con *SystemVerilog* es inmediata debido a que un lenguaje surge a partir del otro, en el caso de VHDL se requiere de una adaptación multi-lenguaje entre el DUV (VHDL) y el entorno de verificación UVM (*SystemVerilog*).

Para realizar dicha adaptación, se debe incluir la opción `-mixedsvvh` en el comando de compilación de la entidad VHDL que actuará como DUV. Este comando de compilación, en el caso de archivos VHDL, es `vcom`, mientras que para ficheros *Verilog* o *SystemVerilog* es `vlog` [21]. De este modo, el contenido de los archivos VHDL compilados usando la opción `-mixedsvvh` será directamente visible en los archivos *Verilog* o *SystemVerilog* que se compilen posteriormente, por lo que será posible la instanciación de la entidad del DUV VHDL en el módulo *Top*.

Finalmente, se deben eliminar ciertos *warnings* relativos a señales que toman valores de alta impedancia en la simulación. Esto se debe a que las variables `std_logic` de VHDL solamente pueden soportar los estados 0 y 1, mientras las variables `logic` de *SystemVerilog* incluyen, además, los estados X (desconocido) y Z (alta impedancia). Para solventarlos, se hace uso de la sentencia que se muestra en el Código 4.7 como primera orden tras lanzar la simulación (comando `vsim`) [21].

```
set NumericStdNoWarnings 1;
```

Código 4.7. Comando para la supresión de *warnings* en la simulación con un DUV VHDL

4.3 CREACIÓN DE UN WRAPPER COMO NUEVO DUV

Volviendo al IP multi-interfaz orientado a la compresión de imágenes, como se ha detallado en el apartado 3.2.3 Referencia de los componentes del *test bench*, su verificación funcional requiere utilizar dos componentes adicionales para la comunicación AHB con el IP121. Estos componentes se obtienen de la biblioteca *GRLIB IP* y se encuentran explícitamente definidos en VHDL. Por lo tanto, se decidió crear una nueva entidad VHDL, a modo de *wrapper*, entre el IP121 y estos componentes, la cual actuará como DUV del *test bench* UVM.

Para el desarrollo de este *wrapper* se reutiliza el código ya mostrado en el Código 3.6, el cual se agrupa en una nueva entidad, cuyo nombre es `ccsds121_duv`. Por tanto, el nuevo DUV a integrar en el entorno de verificación UVM estará definido en el archivo `ccsds121_duv.vhc`, cuyo aspecto es similar al ya visto en la Figura 3.2, pero incluyendo todos los componentes en una nueva entidad VHDL, tal y como se muestra en la Figura 4.6. Por otro lado, en el Código 4.8 se muestra el código correspondiente a la especificación de los puertos de esta nueva entidad VHDL.

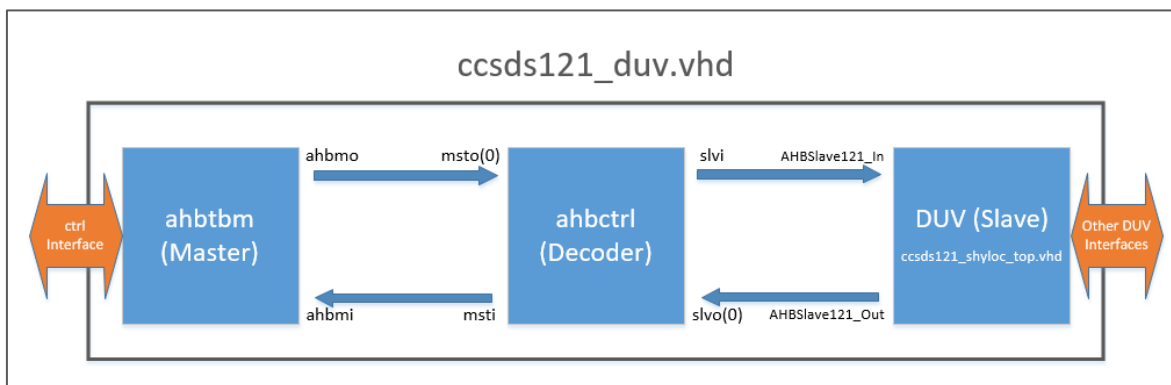


Figura 4.6. Diagrama de bloques de la entidad VHDL `ccsds121_duv`

```
entity ccsds121_shyloc_top is
  port (
    -- System Interface
    Clk_S: in std_logic;
    Rst_N: in std_logic;

    -- AMBA Interface
    Clk_AHB      : in  std_logic;
    Reset_AHB    : in  std_logic;
    ctrli        : in  ahbtbm_ctrl_in_type;
    ctrl_o       : out ahbtbm_ctrl_out_type;

    -- Data Input Interface
    DataIn       : in  std_logic_vector (D_GEN-1 downto 0);
    DataIn_NewValid: in  std_logic;
    IsHeaderIn   : in  std_logic;
    NbitsIn      : in  Std_Logic_Vector (5 downto 0);
```

```

-- Data Output Interface
DataOut          : out std_logic_vector (W_BUFFER_GEN-1 downto 0);
DataOut_NewValid: out std_logic;

-- Control Interface
ForceStop        : in  std_logic;
Ready_Ext        : in  std_logic;
AwaitingConfig   : out std_logic;
Ready            : out std_logic;
FIFO_Full        : out std_logic;
EOP              : out std_logic;
Finished         : out std_logic;
Error            : out std_logic;
);
end ccsds121_shyloc_top;

```

Código 4.8. Entidad VHDL del nuevo DUV `ccsds121_duv`

Como puede observarse en este código, la única diferencia con respecto a la entidad presentada en el Código 3.1 consiste en el reemplazamiento de los puertos `AHBSlave121_In` y `AHBSlave121_Out` por los puertos `ctrli` y `ctrl0`, respectivamente. Ello se debe a que los dos primeros puertos se utilizan en la comunicación AHB interna del *wrapper* entre el IP121, que actúa como dispositivo *Slave*, con el decodificador y el dispositivo *Master* de dicha comunicación. Al igual que se indicó en el apartado 3.2.4 Proceso de configuración del IP, el envío de los parámetros de configuración de las compresiones se efectuará mediante una comunicación directa con el módulo *Master* AHB, para lo cual se utilizan funciones de *GRLIB IP* y los puertos `ctrli` y `ctrl0`, cuyo tipo también está definido en *GRLIB IP*. El propósito de cada uno de estos puertos se detalla a continuación:

- `ctrli`. Conjunto de señales de entrada al dispositivo *Master* AHB para el envío de los parámetros de configuración al IP (tipo `ahbtbm_ctrl_in_type` definido en *GRLIB IP*).
- `ctrl0`. Conjunto de señales de salida del dispositivo *Master* AHB que integran respuestas a los datos enviados por el puerto `ctrli`, así como información del estado del dispositivo *Master* (tipo `ahbtbm_ctrl_out_type` definido en *GRLIB IP*).

El resto de puertos del *wrapper*, por su parte, se conectan internamente a sus puertos homólogos en el IP121. De este modo, en la práctica sería como interactuar directamente con estos últimos.

4.4 PAQUETES CON DEFINICIÓN DE CONSTANTES

En la implementación del *test bench* original se disponía de una serie de constantes para la configuración del IP121 durante su compilación, así como para la configuración del *test bench* y, en

caso de encontrarse habilitada, del IP121 en tiempo de ejecución. En este caso, el DUV sigue estando descrito en VHDL, pero para la implementación del entorno de verificación basado en UVM se hará uso del lenguaje *SystemVerilog*. Debido a este hecho, a la hora de efectuar cualquier parametrización en el *test bench*, se requiere utilizar un tipo especial de constante de este lenguaje, denominado `parameter`.

Por tanto, el paquete con la definición de constantes para la compilación del DUV quedará intacto, pero se modifica el *script Python* de modo que se genere, para cada *test*, el paquete con la definición de constantes `ccsds121_tb_parameters` descrito en *SystemVerilog*. En el nuevo *script*, que tiene como nombre `gen_config_tests_121.py`, también se ha omitido todo lo relativo a las posibilidades de síntesis que existían en el *script* original, pues en este caso no es de interés, además de la generación de los archivos ejecutables `do` para la compilación y la simulación, pues todo ello se efectuará ahora utilizando un *Makefile*. Finalmente, se elimina el argumento relativo al directorio en el que se almacenaban los datos de salida del IP para una comparación posterior, pues se ha decidido que resulta más dinámico y eficiente efectuar esta comparación *on the fly*.

Con todas estas modificaciones, el Código 4.9 hace referencia a la nueva ayuda de ejecución del *script*, donde el fichero CSV con los valores que tomarán los diferentes parámetros de configuración sigue siendo el mismo que en la implementación del *test bench* original.

```
# Some command line examples:
# gen_config_tests_121.py testcases_121.csv ../images/raw
# ../images/reference ../

#How to run this script
# gen_config_tests_121.py
#(0) *.csv file containing the desired parameters
#(1) path to the folder containing the raw images that will be
#    compressed during the simulation
#(2) path to the folder containing the reference compressed images
#(3) test environment database folder
```

Código 4.9. Ayuda de ejecución del *script* `gen_config_tests_121.py`

Tras la ejecución de este nuevo *script*, el paquete `ccsds121_tb_parameters`, que contiene los parámetros para la configuración del *test bench* y los estímulos para la ejecución de un *test*, es ahora un paquete definido en *SystemVerilog*. Un ejemplo de este nuevo paquete (concretamente el relativo al *Test 25*) se muestra en el Código 4.10.

```

// Include shyloc_functions file. Package with generic functions used
for the configuration parameters.
`include "shyloc_functions.sv"

// Import shyloc_functions package.
import shyloc_functions::*;

// ccstds121_tb_parameters package. Package with configuration values
used by the test.
package ccstds121_tb_parameters;

// TEST: 25_Test

parameter integer EN_RUNCFG_G = 1; // (0) Disables runtime
configuration; (1) Enables runtime configuration.
parameter integer RESET_TYPE = 1; // (0) Asynchronous reset;
(1) Synchronous reset.

parameter integer HSINDEX_121 = 3; // AHB slave index.
parameter integer HSCONFIGADDR_121 = 32'h0100; // ADDR field of the
AHB Slave.
parameter integer HSADDRMASK_121 = 32'h0FFF; // MASK field of the
AHB slave.

parameter integer Nx_tb = 256; // Number of columns.
parameter integer Ny_tb = 1; // Number of lines.
parameter integer Nz_tb = 1; // Number of bands.
parameter integer D_tb = 7; // Dynamic range of the input samples.
parameter integer ENDIANESS_tb = 0; // (0) Little-Endian;
(1) Big-Endian.
parameter integer J_tb = 64; // Block Size.
parameter integer REF_SAMPLE_tb = 4096; // Reference Sample
Interval (Determine how often to insert references not coded).
parameter integer CODESET_tb = 0; // Code Option (If specified and
the dynamic range D is <= 4, the restricted mode will be used).
parameter integer W_BUFFER_tb = 32; // Output word size.
parameter integer BYPASS_tb = 0; // (0) Compression;
(1) Bypass Compression.
parameter integer PREPROCESSOR_tb = 0; // (0) Preprocessor is not
present; (1) CCSDS123 preprocessor is present; (2) Any-other
preprocessor is present.
parameter integer DISABLE_HEADER_tb = 1; // Selects whether to
disable (1) or not (0) the header generation.
// Stimulis and reference files.

parameter string stim_file =
"/home/users/master/divdsi/srodriguez/Compresion_UVM/images/raw/test_p
256n07.dat";
parameter string ref_file =
"/home/users/master/divdsi/srodriguez/Compresion_UVM/images/reference/
comp_25.dat";

// Some other necessary parameters.
parameter integer D_G_tb = 16;
parameter integer W_BUFFER_G_tb = 32;
parameter integer N_SAMPLES_G_tb = (Nx_tb*Ny_tb*Nz_tb);
parameter integer W_N_SAMPLES_G_tb =
shyloc_functions::log2(N_SAMPLES_G_tb);
parameter integer CODESET_G_tb = 0;

```

```

parameter integer N_K_G_tb =
  shyloc_functions::get_n_k_options(D_G_tb, CODESET_G_tb);
parameter integer W_K_G_tb =
  shyloc_functions::maximum(3, shyloc_functions::log2(N_K_G_tb)+1);
parameter integer W_NBITS_K_G_tb =
  shyloc_functions::get_k_bits_option(W_BUFFER_G_tb, W_K_G_tb);

endpackage: ccsds121_tb_parameters

```

Código 4.10. Paquete *SystemVerilog* `ccsds121_tb_parameters` para el *Test 25*

En este archivo, se elimina la referencia al fichero en el que se almacenaría el flujo de datos de salida del IP121. Además, en las funciones utilizadas para generar los últimos parámetros se hace uso del prefijo `shyloc_functions::`. Esto se debe a que para invocar una función externa en *SystemVerilog* es necesario hacer referencia al paquete al que pertenece, además de importarlo en el comienzo del archivo. Estas funciones particulares, las cuales ya se encontraban implementadas en VHDL en el *test bench* original, han sido redefinidas en un paquete *SystemVerilog*, manteniendo su funcionalidad.

4.5 ESTRUCTURA DEL ENTORNO DE VERIFICACIÓN DEL IP121 BASADO EN UVM

Una vez generados los parámetros necesarios para la correcta ejecución de los *test* sobre el entorno de verificación a desarrollar, en la Figura 4.7 se presenta la estructura del entorno UVM planteado para la verificación del IP121 orientado a la compresión de imágenes.

Este entorno sigue, en esencia, la estructura ya mostrada en la Figura 2.11, con la salvedad de que en este caso se incluyen dos componentes *UVM Agent* activos. El primero de ellos se centrará en las interfaces de configuración y de control, mientras que el segundo se hará cargo de las interfaces de entrada/salida de datos. Por lo tanto, también se tienen dos tipos de interfaces y de secuencias diferenciadas, un tipo para cada componente *Agent*. Finalmente, el componente *Scoreboard* únicamente recibirá transacciones desde el componente *Monitor* de la interfaz de datos, con la información necesaria de salida del IP121 para efectuar, si es posible, las comparaciones pertinentes. El componente *Monitor* de la interfaz de configuración y control, por su parte, se encargará de comprobar que se cumple el patrón correcto de ejecución en las señales de control, notificando su estado mediante el mecanismo de mensajes.

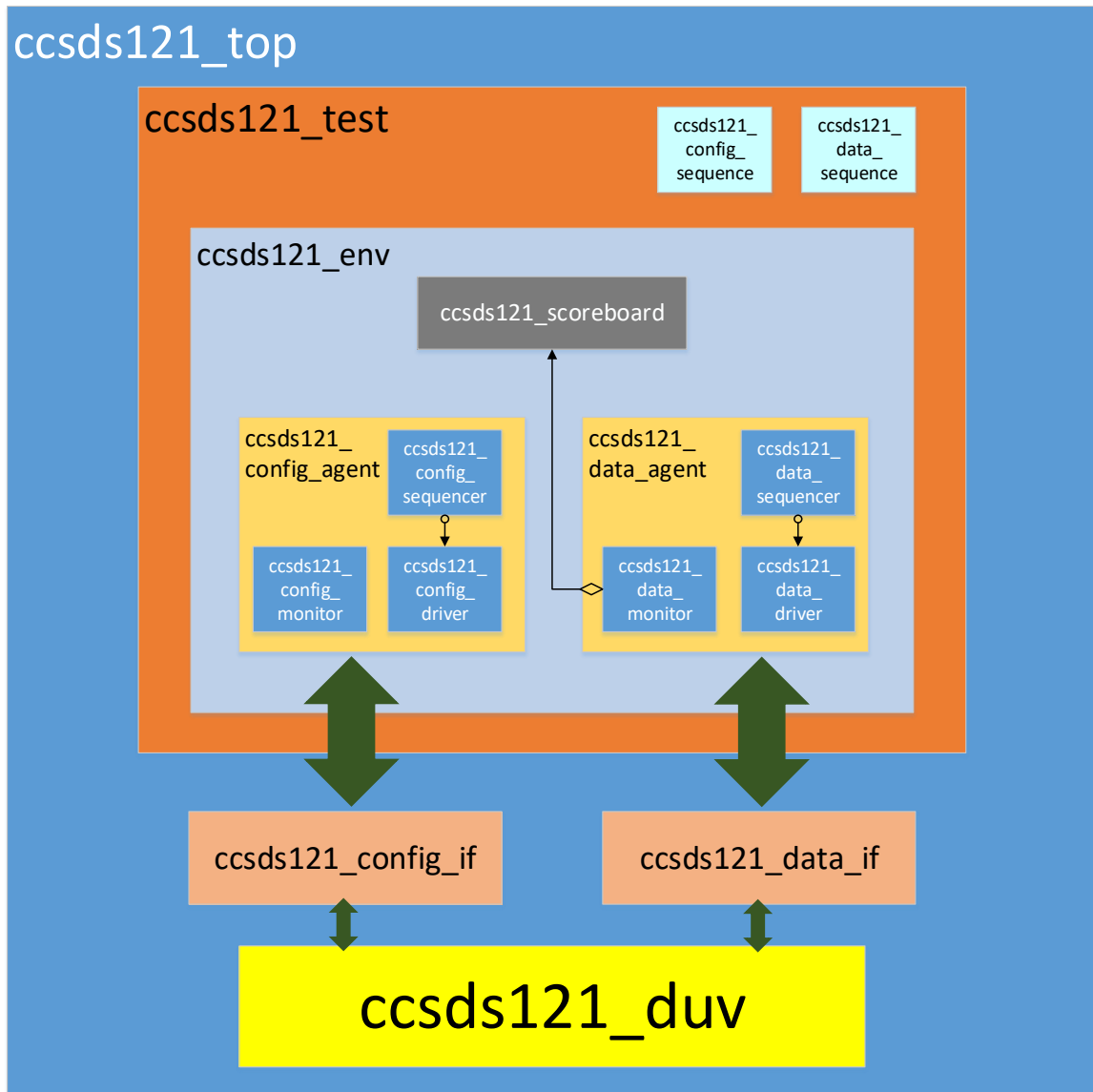


Figura 4.7. Estructura del entorno UVM específico creado

Habiendo visualizado el diagrama de bloques general del entorno UVM propuesto, se procede a detallar, en los siguientes puntos, la funcionalidad específica de cada uno de sus elementos y componentes. Para ello se seguirá, en principio, una metodología *top-down*, partiendo desde los componentes de mayor nivel de abstracción en la jerarquía del entorno de verificación UVM, si bien es cierto que, en algunos casos, será necesario alterar dicho orden por cuestiones de interdependencias entre componentes.

En este caso, cabe destacar que las funcionalidades desarrolladas en el *test bench* basado en UVM para la configuración del IP121 se corresponden con las del *test0*, el *test2*, el *test4* y el *test7* del *test bench* original, cuya definición se abordó en el apartado 3.2.3 Referencia de los componentes del

test bench. Como podrá comprobarse, se ha dejado de lado la funcionalidad del *test5*, ya que no se ha considerado relevante, pues no resulta significativo sobre los aspectos a verificar en el IP121.

4.5.1 INTERFACES VIRTUALES

En el entorno de verificación basado en UVM que se propone existen dos interfaces virtuales, las cuales se encuentran contenidas en el archivo `ccsds121_if.sv`. En estas interfaces se especifican las señales correspondientes a los puertos del DUV de interés para cada uno de los dos componentes *UVM Agent* existentes. De este modo, ambos componentes *UVM Agent* podrán acceder a dichos puertos del DUV a través de las interfaces virtuales, que actúan como nexo.

El Código 4.11 muestra la implementación de la interfaz para los puertos de configuración del DUV, denominada `ccsds121_config_if`. En este caso, la señal de control `ctrl` que se tenía en la implementación original para la interacción con el módulo *Master* de la comunicación AHB interna en el DUV, se ha dividido en dos señales diferenciadas, pues los puertos conjuntos de entrada/salida son más complejos de gestionar en *SystemVerilog*. Estas señales son `ctrl_i` y `ctrl_o`, de tipos `ahbtbm_ctrl_in_type` y `ahbtbm_ctrl_out_type`, respectivamente, tipos que se encuentran definidos en el paquete VHDL `ahbtbp`, por lo que deberá compilarse haciendo uso de la opción `-mixedsvvh`.

```
interface ccsds121_config_if (input logic Clk_S);

    // AMBA Signals
    ahbtbp::ahbtbm_ctrl_in_type  ctrl_i;
    ahbtbp::ahbtbm_ctrl_out_type ctrl_o;

    // Control Signals
    logic ForceStop;
    logic AwaitingConfig;
    logic Ready;
    logic Finished;
    logic Error;

endinterface: ccsds121_config_if
```

Código 4.11. Interfaz `ccsds121_config_if`

El Código 4.12, por su parte, representa la implementación de la interfaz para los puertos de datos del DUV, cuyo nombre es `ccsds121_data_if`. El aspecto más destacado de esta interfaz es que es parametrizable pues, como ya se ha comentado, el tamaño de los puertos de entrada y de salida de muestras de datos es variable y se establece en tiempo de compilación a través de determinados

parámetros específicos. Por lo demás, incluye una réplica exacta de las señales existentes en los puertos de entrada/salida de datos en el DUV.

```

interface ccstds121_data_if #(parameter integer D_GEN = 16,
                               parameter integer W_BUFFER_GEN = 32)
    (input logic Clk_S,
     input logic Rst_N);

    // Data Input Signals
    logic DataIn_NewValid;
    logic [D_GEN-1:0] DataIn;
    logic IsHeaderIn;
    logic [5:0] NbitsIn;

    // Data Output Signals
    logic DataOut_NewValid;
    logic [W_BUFFER_GEN-1:0] DataOut;

    // Control Signals
    logic ForceStop;
    logic AwaitingConfig;
    logic Ready;
    logic Finished;
    logic Error;

endinterface: ccstds121_data_if
    
```

Código 4.12. Interfaz `ccstds121_data_if`

Cabe destacar que en ambas interfaces se incluyen las señales de control `ForceStop`, `AwaitingConfig`, `Ready`, `Finished` y `Error`, además de la señal de reloj como entrada. Esto se debe a que son señales necesarias, tanto en el proceso de configuración del IP121, como en los procesos de transferencia del flujo de datos de entrada y de recepción del flujo de datos de salida. Además, la interfaz de configuración `ccstds121_config_if` no recibe como entrada la señal de *reset*, pues el comportamiento necesario en las señales de control de salida para poder efectuar el proceso de configuración, ya depende del *reset* inicial.

4.5.2 MÓDULO *TOP*

El módulo *Top*, implementado en el archivo `ccstds121_top.sv`, se encarga básicamente de las siguientes tareas:

- Generar las señales principales que gobernarán el DUV y el entorno de verificación. Estas señales coincidirán con las existentes en el *test bench* original.
- Referenciar e interconectar el DUV y las interfaces virtuales.
- Registrar las interfaces virtuales en la base de datos de UVM.

- Lanzar la ejecución del test.

El contenido de este archivo y, por consiguiente, la implementación de las tareas que han sido descritas anteriormente, se muestra en el Código 4.13.

```

`include "ccsds121_tb_parameters.sv"
`include "ccsds121_pkg.sv"
`include "ccsds121_if.sv"

module ccsds121_top;

    import uvm_pkg::*;
    import ccsds121_tb_parameters::*;
    import ccsds121_pkg::*;

    bit Clk_S, Rst_N;
    bit Clk_AHB, Reset_AHB;
    bit Ready_Ext;

    ccsds121_config_if config_vif (.Clk_S(Clk_S));

    ccsds121_data_if #(ccsds121_tb_parameters::D_G_tb,
                     ccsds121_tb_parameters::W_BUFFER_G_tb)
        data_vif (.Clk_S(Clk_S),
                 .Rst_N(Rst_N));

    ccsds121_duv duv (.Clk_S(Clk_S),
                    .Rst_N(Rst_N),
                    .Clk_AHB(Clk_AHB),
                    .Reset_AHB(Reset_AHB),
                    .ctrli(config_vif.ctrli),
                    .ctrllo(config_vif.ctrllo),
                    .DataIn_NewValid(data_vif.DataIn_NewValid),
                    .DataIn(data_vif.DataIn),
                    .IsHeaderIn(data_vif.IsHeaderIn),
                    .NbitsIn(data_vif.NbitsIn),
                    .DataOut_NewValid(data_vif.DataOut_NewValid),
                    .DataOut(data_vif.DataOut),
                    .ForceStop(config_vif.ForceStop),
                    .Ready_Ext(Ready_Ext),
                    .AwaitingConfig(config_vif.AwaitingConfig),
                    .Ready(config_vif.Ready),
                    .FIFO_Full(),
                    .EOP(),
                    .Finished(config_vif.Finished),
                    .Error(config_vif.Error));

    // Init Clocks and External Ready
    initial begin
        Clk_S = 1'b1;
        Clk_AHB = 1'b1;
        Ready_Ext = 1'b1;
    end

    // System Clock
    always #100 Clk_S = ~Clk_S;

```

```

// AMBA Clock
always #70 Clk_AHB = ~Clk_AHB;

// System Reset
initial begin
    Rst_N = 1'b0;
    #400 Rst_N = 1'b1;
end

// AMBA Reset
assign Reset_AHB = Rst_N;

// Control Ports in both two virtual interfaces
assign data_vif.AwaitingConfig = config_vif.AwaitingConfig;
assign data_vif.Ready          = config_vif.Ready;
assign data_vif.Finished       = config_vif.Finished;
assign data_vif.Error          = config_vif.Error;
assign data_vif.ForceStop      = config_vif.ForceStop;

initial begin
    uvm_config_db#(virtual ccsds121_config_if)::set(
        uvm_root::get(), "*.config_agent.*", "config_vif",
        config_vif);
    uvm_config_db#(virtual ccsds121_data_if
        #(ccsds121_tb_parameters::D_G_tb,
        ccsds121_tb_parameters::W_BUFFER_G_tb))::set(uvm_root::get(),
        "*.data_agent.*", "data_vif", data_vif);

    run_test();
end

endmodule: ccsds121_top

```

Código 4.13. Archivo `ccsds121_top.sv`

Al inicio del archivo se incluye el fichero correspondiente a la implementación de las interfaces virtuales. Además, se importan el paquete básico de UVM (`uvm_pkg`), el paquete *SystemVerilog* de parámetros de configuración para el *test bench* (`ccsds121_tb_parameters`), y un paquete adicional (`ccsds121_pkg`). Este paquete adicional simplemente incluye todos y cada uno de los archivos relativos a los diferentes componentes y elementos del entorno de verificación basado en UVM, de modo que sean visibles. Su contenido se muestra en el Código 4.14.

```
package ccsds121_pkg;
  `include "uvm_macros.svh"
  `include "ccsds121_packet.sv"
  `include "ccsds121_sequencer.sv"
  `include "ccsds121_sequences.sv"
  `include "ccsds121_config_driver.sv"
  `include "ccsds121_config_monitor.sv"
  `include "ccsds121_data_driver.sv"
  `include "ccsds121_data_monitor.sv"
  `include "ccsds121_agent.sv"
  `include "ccsds121_scoreboard.sv"
  `include "ccsds121_env.sv"
  `include "ccsds121_test.sv"
endpackage: ccsds121_pkg
```

Código 4.14. Archivo `ccsds121_pkg.sv`

4.5.3 TRANSACCIONES

Como se comentó en la introducción a la metodología UVM, la funcionalidad de un entorno de verificación UVM se basa en la transferencia de transacciones entre los diferentes componentes que integran el entorno. En este *test bench* particular se tienen dos tipos de transacciones diferenciados, que se corresponden con la transacción de configuración y la transacción de datos. Ambas transacciones se recogen en el archivo `ccsds121_packet.sv`.

4.5.3.1 TRANSACCIÓN DE CONFIGURACIÓN

La clase creada para implementar la transacción de configuración se denomina `ccsds121_config_packet`. En este tipo de transacción se especifican campos correspondientes a los parámetros necesarios para configurar el IP121 en la siguiente compresión a realizar. Además, se registra el objeto y sus variables en la *Factory* de UVM, y se define el constructor de la clase, acción que se implementará en todos y cada uno de los objetos y componentes que se vayan a presentar en adelante. Todo ello se muestra en el Código 4.15.

```

class ccsds121_config_packet extends uvm_sequence_item;

    rand bit [15:0] Nx_tb;
    rand bit      CODESET_tb;
    rand bit      DISABLE_HEADER_tb;
    rand bit [6:0] J_tb;
    rand bit [6:0] W_BUFFER_tb;
    rand bit [15:0] Ny_tb;
    rand bit [12:0] REF_SAMPLE_tb;
    rand bit [15:0] Nz_tb;
    rand bit [5:0] D_tb;
    rand bit      ENDIANESS_tb;
    rand bit [1:0] PREPROCESSOR_tb;
    rand bit      BYPASS_tb;

    `uvm_object_utils_begin(ccsds121_config_packet)
        `uvm_field_int(Nx_tb, UVM_DEFAULT)
        `uvm_field_int(CODESET_tb, UVM_DEFAULT)
        `uvm_field_int(DISABLE_HEADER_tb, UVM_DEFAULT)
        `uvm_field_int(J_tb, UVM_DEFAULT)
        `uvm_field_int(W_BUFFER_tb, UVM_DEFAULT)
        `uvm_field_int(Ny_tb, UVM_DEFAULT)
        `uvm_field_int(REF_SAMPLE_tb, UVM_DEFAULT)
        `uvm_field_int(Nz_tb, UVM_DEFAULT)
        `uvm_field_int(D_tb, UVM_DEFAULT)
        `uvm_field_int(ENDIANESS_tb, UVM_DEFAULT)
        `uvm_field_int(PREPROCESSOR_tb, UVM_DEFAULT)
        `uvm_field_int(BYPASS_tb, UVM_DEFAULT)
    `uvm_object_utils_end

    function new (string name = "ccsds121_config_packet");
        super.new(name);
    endfunction: new

endclass: ccsds121_config_packet

```

Código 4.15. Transacción `ccsds121_config_packet`

4.5.3.2 TRANSACCIÓN DE DATOS

La transacción de datos, por su parte, se implementa en la clase `ccsds121_data_packet`, que se encuentra parametrizada en función del tamaño del *buffer* de los datos de salida del DUV. En este caso, se ha decidido hacer uso de una única transacción para los datos de entrada y de salida, debido a que la cantidad de variables involucradas en ambos procesos no es elevada.

En la transacción se especifica un campo de 32 bits para los datos de entrada, pues como se indicará posteriormente, siempre se leerán bloques de 32 bits desde el fichero de estímulos para las muestras de entrada, así como otro campo para los datos de salida, cuyo tamaño es parametrizable. Además, se tienen dos campos adicionales, `Finished` y `NotCompare`, utilizados en el proceso de recepción de los datos de salida para informar al componente *UVM Scoreboard* sobre cuándo ha

terminado la compresión, y cuándo no será válida la comparación, respectivamente. La implementación de esta transacción se recoge en el Código 4.16.

```

class ccsds121_data_packet #(parameter W_BUFFER_GEN=32) extends
  uvm_sequence_item;

  rand bit [31:0]      DataIn;
  rand bit            Finished;
  rand bit [W_BUFFER_GEN-1:0] DataOut;
  rand bit            NotCompare;

  `uvm_object_param_utils_begin(ccsds121_data_packet#(W_BUFFER_GEN))
    `uvm_field_int(DataIn, UVM_DEFAULT)
    `uvm_field_int(Finished, UVM_DEFAULT)
    `uvm_field_int(DataOut, UVM_DEFAULT)
    `uvm_field_int(NotCompare, UVM_DEFAULT)
  `uvm_object_utils_end

  function new (string name = "ccsds121_data_packet");
    super.new(name);
  endfunction: new

endclass: ccsds121_data_packet

```

Código 4.16. Transacción `ccsds121_data_packet`

4.5.4 SECUENCIAS

Las transacciones con los estímulos de entrada al DUV se generan en las Secuencias UVM (*UVM Sequence*), secuencias que, a su vez, son introducidas en los componentes *UVM Sequencer* desde el componente *UVM Test*. Al igual que sucede con las transacciones se tienen, en el archivo `ccsds121_sequences.sv`, dos secuencias de distinta naturaleza, correspondientes a una secuencia de configuración y una secuencia de datos.

4.5.4.1 SECUENCIA DE CONFIGURACIÓN

La secuencia de configuración, denominada `ccsds121_config_sequence`, genera una transacción de configuración en su tarea `body()`, y la envía haciendo uso de la dupla de funciones `start_item()` y `finish_item()`. A los campos de dicha transacción se les asigna, siguiendo el formato correcto, los valores con la configuración seleccionada para el IP121, que se encuentran en el paquete `ccsds121_tb_parameters`. El código relativo a esta secuencia se muestra en el Código 4.17.

```

class ccstds121_config_sequence extends uvm_sequence
#(ccstds121_config_packet);

    `uvm_object_utils(ccstds121_config_sequence)

function new(string name = "ccstds121_config_sequence");
    super.new(name);
endfunction: new

virtual task body();
    req = ccstds121_config_packet::type_id::create("req");
    start_item(req);
    assert(req.randomize() with {
        Nx_tb == ccstds121_tb_parameters::Nx_tb[15:0];
        CODESET_tb == ccstds121_tb_parameters::CODESET_tb[0];
        DISABLE_HEADER_tb ==
            ccstds121_tb_parameters::DISABLE_HEADER_tb[0];
        J_tb == ccstds121_tb_parameters::J_tb[6:0];
        W_BUFFER_tb == ccstds121_tb_parameters::W_BUFFER_tb[6:0];
        Ny_tb == ccstds121_tb_parameters::Ny_tb[15:0];
        REF_SAMPLE_tb ==
            ccstds121_tb_parameters::REF_SAMPLE_tb[12:0];
        Nz_tb == ccstds121_tb_parameters::Nz_tb[15:0];
        D_tb == ccstds121_tb_parameters::D_tb[5:0];
        ENDIANESS_tb == ccstds121_tb_parameters::ENDIANESS_tb[0];
        PREPROCESSOR_tb ==
            ccstds121_tb_parameters::PREPROCESSOR_tb[1:0];
        BYPASS_tb == ccstds121_tb_parameters::BYPASS_tb[0];});
    finish_item(req);
endtask: body

endclass: ccstds121_config_sequence

```

Código 4.17. Secuencia `ccstds121_config_sequence`

4.5.4.2 SECUENCIA DE DATOS

La secuencia de datos, por su parte, tiene como nombre `ccstds121_data_sequence` y es más compleja. En primer lugar, obtiene el nombre del fichero de datos con los estímulos de entrada desde la base de datos (en caso de no encontrarlo, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación), crea una transacción de datos y habilita las respuestas no bloqueantes por parte del componente *UVM Driver*. Tras ello, se envía una transacción *idle* y se abre el archivo de estímulos. Hasta que se llega al final de este archivo, se leen 32 bits del mismo en el formato correcto y se envían en la transacción creada mediante la dupla de funciones `start_item()` y `finish_item()`. Finalmente, se llega a un estado de espera indefinida.

Al habilitar las respuestas no bloqueantes desde el componente *UVM Driver*, se implementa la función `response_handler` para gestionarlas. En este caso, esta función únicamente se encarga

de cerrar el archivo de estímulos y asignar el nivel alto a la variable `finish_sequence`, inicialmente a nivel bajo. Con el uso de esta función, así como el envío de una secuencia *idle* antes de comenzar con la transmisión de datos (con lo que se permiten respuestas en cualquier momento, ya que no son posibles sin que el componente *UVM Driver*, previamente, haya recibido una transacción) y la espera indefinida, se consigue que la secuencia sea universal para todos los test a ejecutar, algo que se entenderá mejor tras la definición del componente *UVM Test*.

La implementación de la secuencia de datos se recoge en el Código 4.18.

```
class ccsds121_data_sequence extends uvm_sequence
#(ccsds121_data_packet#(ccsds121_tb_parameters::W_BUFFER_G_tb));

integer stimulus_file;
string stim_name;
bit [31:0] Data;
bit finish_sequence = 1'b0;

`uvm_object_utils(ccsds121_data_sequence)

function new(string name = "ccsds121_data_sequence");
super.new(name);
endfunction: new

virtual task body();
if(!uvm_config_db#(string)::get(null, this.get_full_name(),
"stim_name", stim_name))
`uvm_fatal("NO_STIM_NAME", "File name must be set for the
stimulus file.")

use_response_handler(1); // Enable response handler
req = ccsds121_data_packet
#(ccsds121_tb_parameters::W_BUFFER_G_tb)
::type_id::create("req");

// Idle transaction to enable the error response
start_item(req);
finish_item(req);

stimulus_file = $fopen(stim_name, "r");
void'($rewind(stimulus_file));

forever begin
Data[7:0] = $fgetc(stimulus_file);
Data[15:8] = $fgetc(stimulus_file);
Data[23:16] = $fgetc(stimulus_file);
Data[31:24] = $fgetc(stimulus_file);

if ($feof(stimulus_file) || (finish_sequence == 1'b1)) begin
break;
end

start_item(req);
assert(req.randomize() with {DataIn == Data;});
finish_item(req);
end
```



```

// Wait for the Finished signal
wait(0);
endtask: body

function void response_handler (uvm_sequence_item response);
    $fclose(stimulus_file);
    finish_sequence = 1'b1;
endfunction: response_handler

endclass: ccsds121_data_sequence

```

Código 4.18. Secuencia `ccsds121_data_sequence`

4.5.5 COMPONENTE *TEST*

El componente *Test* es el primero en ejecutarse en la simulación cuando se invoca el comando correspondiente desde el módulo *Top*. Para el entorno de verificación basado en UVM desarrollado, se ha creado un componente *UVM Test* base, a partir del cual se derivará el resto, incluyendo su funcionalidad específica dependiendo del tipo de *test* a simular (todos ellos se encuentran en el archivo `ccsds121_test.sv`). El componente *UVM Test* base, denominado `ccsds121_base_test`, realiza las siguientes funciones básicas:

- **build_phase.** Crea un componente de tipo *UVM Environment* y un componente *printer*. Este último se utilizará para presentar la topología del *test bench* en la consola. Además, introduce en la base de datos de UVM los nombres de los archivos con los datos correspondientes a los estímulos de entrada y los valores de referencia, para que sean utilizados por la secuencia de datos y el componente *UVM Scoreboard*, respectivamente.
- **end_of_elaboration_phase.** Muestra en la consola la topología del *test bench* creado en las fases `build_phase` y `connect_phase`.
- **run_phase.** Ejecuta, de forma paralela y haciendo uso del mecanismo de *objections*, las tareas `send_config()` y `send_data()`.
 - **send_config.** Tarea que crea una secuencia de configuración y la inicia sobre el componente *UVM Sequencer* incluido en el componente *UVM Agent* asociado a la configuración del DUV.
 - **send_data.** Tarea que crea una secuencia de datos y la inicia sobre el componente *UVM Sequencer* incluido en el componente *UVM Agent* asociado a la transferencia del flujo de datos de entrada al DUV. Una vez que el componente *UVM Driver* notifique a la secuencia de la finalización de la misma mediante la transferencia de una respuesta (variable `finish_sequence` a nivel alto), se descarta la secuencia

activa y se inicializa el componente *UVM Sequencer*. Este proceso se ejecuta dos veces. Esta notificación de finalización de la secuencia se implementa en el componente *UVM Driver*, y se produce ante la finalización de una compresión (señal `Finished`), por activación de la señal de `Error`, o por la ejecución de un `ForceStop`, de modo que la secuencia creada permite cubrir todos los casos.

La implementación de este componente *Test* base se muestra en el Código 4.19.

```
class ccstds121_base_test extends uvm_test;

    ccstds121_env env;
    uvm_table_printer printer;

    `uvm_component_utils(ccstds121_base_test)

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

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = ccstds121_env::type_id::create("env", this);
        printer = new();

        uvm_config_db#(string)::set(this, "*", "stim_name",
            ccstds121_tb_parameters::stim_file);
        uvm_config_db#(string)::set(this, ".*.scoreboard", "ref_name",
            ccstds121_tb_parameters::ref_file);
    endfunction: build_phase

    virtual function void end_of_elaboration_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $sformatf("Printing the test bench
            topology:\n%s", this.sprint(printer)), UVM_LOW);
    endfunction: end_of_elaboration_phase

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);
        phase.phase_done.set_drain_time(this, 1500);

        phase.raise_objection(this);
        fork
            send_config();
            send_data();
        join
        phase.drop_objection(this);
    endtask

    virtual task send_config();
        ccstds121_config_sequence config_seq;
        config_seq =
            ccstds121_config_sequence::type_id::create("config_seq");
        config_seq.start(env.config_agent.sequencer);
    endtask: send_config
```

```

virtual task send_data();
  repeat (2) begin
    ccsds121_data_sequence data_seq;
    data_seq =
      ccsds121_data_sequence::type_id::create("data_seq");
    fork
      wait(data_seq.finish_sequence == 1'b1);
      data_seq.start(env.data_agent.sequencer);
    join_any
    disable fork;
    env.data_agent.sequencer.stop_sequences();
  end
endtask: send_data

endclass: ccsds121_base_test

```

Código 4.19. Componente `ccsds121_base_test`

Como se ha comentado, a partir de este componente base se derivan otros componentes *UVM Test*, por lo que heredarán sus funcionalidades. Cada uno de estos componentes *UVM Test* define uno de los *test* que pueden ejecutarse en el *test bench* original:

- **test0**: Componente *UVM Test* `ccsds121_normal_test`.
- **test2**: Componente *UVM Test* `ccsds121_reconfig_test`.
- **test4**: Componente *UVM Test* `ccsds121_forcestop_test`.
- **test7**: Componente *UVM Test* `ccsds121_error_test`.

Para ello, estos componentes *UVM Test* se diferencian entre sí en la fase de construcción `build_phase`, introduciendo unos valores u otros sobre el parámetro `test_type` en la base de datos, acción que determinará el comportamiento de los componentes *UVM Driver* y *UVM Monitor* de configuración. También se incluyen ciertas restricciones, como la necesidad de que la configuración del IP121 en tiempo de ejecución se encuentre habilitada para poder ejecutar determinados *test* (si esto no se cumple, se ejecutará `ccsds121_normal_test` por defecto). El código asociado a estos componentes *UVM Test* se recoge en el Código 4.20.

```

class ccsds121_normal_test extends ccsds121_base_test;

  `uvm_component_utils(ccsds121_normal_test)

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

```

```
function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(string)::set(this, "*.config_agent.*",
        "test_type", "normal");
endfunction: build_phase

endclass: ccsds121_normal_test

class ccsds121_forcestop_test extends ccsds121_base_test;

    `uvm_component_utils(ccsds121_forcestop_test)

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

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(string)::set(this, "*.config_agent.*",
        "test_type", "force_stop");
endfunction: build_phase

endclass: ccsds121_forcestop_test

class ccsds121_reconfig_test extends ccsds121_base_test;

    `uvm_component_utils(ccsds121_reconfig_test)

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

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd0) begin
        `uvm_warning(get_type_name(), "Data set does not allow
            configuration when running test. Reconfigure test can't
            be run. Instead, normal two compressions test will be
            executed.")
        uvm_config_db#(string)::set(this, "*.config_agent.*",
            "test_type", "normal");
    end
    else
        uvm_config_db#(string)::set(this, "*.config_agent.*",
            "test_type", "reconfig");
    endfunction: build_phase

endclass: ccsds121_reconfig_test

class ccsds121_error_test extends ccsds121_base_test;

    `uvm_component_utils(ccsds121_error_test)

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

```

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd0) begin
        `uvm_warning(get_type_name(), "Data set does not allow
        configuration when running test. Error test can't be run.
        Instead, normal two compressions test will be executed.")
        uvm_config_db#(string)::set(this, "*.config_agent.*",
        "test_type", "normal");
    end
    else
        uvm_config_db#(string)::set(this, "*.config_agent.*",
        "test_type", "error");
    endfunction: build_phase

endclass: ccsds121_error_test
    
```

Código 4.20. Componentes UVM Test asociados a la funcionalidad de cada test

4.5.6 COMPONENTE ENVIRONMENT

La implementación del componente *UVM Environment* se recoge en el archivo `ccsds121_env.sv` y, según su funcionalidad, actúa como contenedor. Las principales funciones que ejecuta este componente son:

- **build_phase.** Crea tres componentes de los tipos *UVM Agent* de configuración, *UVM Agent* de datos y *UVM Scoreboard*.
- **connect_phase.** Suscribe el puerto *TLM analysis export* del componente *UVM Scoreboard*, al puerto *TLM analysis port* del componente *UVM Monitor* de datos, de modo que el componente *UVM Scoreboard* recibirá las transacciones enviadas por el componente *UVM Monitor*.

El código *SystemVerilog* relativo a la implementación del componente *UVM Environment*, denominado `ccsds121_env`, se muestra en el Código 4.21.

```

class ccsds121_env extends uvm_env;

    ccsds121_config_agent config_agent;
    ccsds121_data_agent data_agent;
    ccsds121_scoreboard scoreboard;

    `uvm_component_utils(ccsds121_env)

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

```

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

    config_agent =
        ccsds121_config_agent::type_id::create("config_agent", this);
    data_agent =
        ccsds121_data_agent::type_id::create("data_agent", this);
    scoreboard =
        ccsds121_scoreboard::type_id::create("scoreboard", this);

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

function void connect_phase(uvm_phase phase);
    data_agent.monitor.output_collected_port.connect(
        scoreboard.output_packets_collected.analysis_export);
    `uvm_info(get_full_name(), "Connect stage completed.",
        UVM_LOW)
endfunction: connect_phase

endclass: ccsds121_env

```

Código 4.21. Componente `ccsds121_env`

4.5.7 COMPONENTE AGENT DE CONFIGURACIÓN

El componente *UVM Agent* de configuración, que tiene por nombre `ccsds121_config_agent`, se encuentra descrito en el archivo `ccsds121_agent.sv`. Al igual que en el componente *UVM Environment*, el componente *UVM Agent* se encarga de agrupar otros UVC que, en este caso, interactúan con una interfaz virtual específica. Por tanto, las funciones principales realizadas por el componente *UVM Agent* de configuración son:

- **build_phase**. Crea tres componentes de los tipos *UVM Sequencer* de configuración, *UVM Driver* de configuración y *UVM Monitor* de configuración.
- **connect_phase**. Interconecta los componentes *UVM Sequencer* y *UVM Driver* a través de sus puertos *TLM export* y *TLM port*, respectivamente.

La implementación del componente *UVM Agent* de configuración se recoge en el Código 4.22.

```

class ccsds121_config_agent extends uvm_agent;

    ccsds121_config_sequencer sequencer;
    ccsds121_config_driver driver;
    ccsds121_config_monitor monitor;

    `uvm_component_utils(ccsds121_config_agent)

```

```

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

function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    sequencer = ccsds121_config_sequencer::
        type_id::create("config_sequencer", this);
    driver = ccsds121_config_driver::
        type_id::create("config_driver", this);
    monitor = ccsds121_config_monitor::
        type_id::create("config_monitor", this);
    `uvm_info(get_full_name(), "Build stage completed.", UVM_LOW)
endfunction: build_phase

function void connect_phase (uvm_phase phase);
    driver.seq_item_port.connect(sequencer.seq_item_export);
    `uvm_info(get_full_name(), "Connect stage completed.",
        UVM_LOW)
endfunction: connect_phase

endclass: ccsds121_config_agent

```

Código 4.22. Componente `ccsds121_config_agent`

4.5.7.1 COMPONENTE SEQUENCER DE CONFIGURACIÓN

El componente *UVM Sequencer* de configuración se denomina `ccsds121_config_sequencer`, y se describe en el fichero `ccsds121_sequencer.sv`. Su implementación se recoge en el Código 4.23, gestionando únicamente transacciones de configuración, así como su registro en la *Factory* de UVM y su constructor. El resto de funcionalidades las hereda directamente de la clase `uvm_sequencer`.

```

class ccsds121_config_sequencer extends uvm_sequencer
    #(ccsds121_config_packet);

    `uvm_sequencer_utils(ccsds121_config_sequencer)

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

endclass: ccsds121_config_sequencer

```

Código 4.23. Componente `ccsds121_config_sequencer`

4.5.7.2 COMPONENTE DRIVER DE CONFIGURACIÓN

El componente *UVM Driver* de configuración, denominado `ccsds121_config_driver`, se implementa en el archivo `ccsds121_config_driver.sv`, y se encarga de llevar a cabo el proceso de configuración del DUV. Por lo tanto, su principal funcionalidad es la siguiente:

- **build_phase.** Extracción de la interfaz virtual de configuración y del nombre del tipo de *test* a ejecutar desde la base de datos de UVM, asignándolos a variables locales. En caso de no encontrar alguno de estos parámetros, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación.
- **run_phase.** Ejecución del proceso de configuración del DUV. Básicamente, se sigue el mismo procedimiento que en la implementación del *test bench* original, con la salvedad de que se incluye la recepción de la transacción de configuración desde el componente *UVM Sequencer* de configuración antes de invocar la ejecución de la tarea correspondiente al *test* (función `get` en el puerto `seq_item_port`, un puerto *TLM export* heredado de la clase `uvm_driver`).

La implementación del componente *UVM Driver* de configuración se muestra en el Código 4.24.

```
class ccsds121_config_driver extends uvm_driver
#(ccsds121_config_packet);

    virtual ccsds121_config_if config_vif;
    logic [31:0] config_reg [3:0];
    logic [31:0] address;
    string test_type;

    `uvm_component_utils(ccsds121_config_driver)

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

    function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual ccsds121_config_if)::get(this, "",
            "config_vif", config_vif))
            `uvm_fatal("NO_CONFIG_VIF", {"Virtual interface must be
                set for: ", get_full_name(), ".config_vif"})

        if (!uvm_config_db#(string)::get(this, "", "test_type",
            test_type))
            `uvm_fatal("NO_TEST_TYPE", "The type of test to run is not
                in the database")

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



```

virtual task run_phase (uvm_phase phase);
    config_vif.ForceStop <= 1'b0;
    while (config_vif.AwaitingConfig != 1'b1) begin
        @(posedge config_vif.Clk_S);
    end
    config_reg[0] = 32'b0;

    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd1) begin
        `uvm_info(get_type_name(), "Executing ahbtbmininit...",
            UVM_HIGH)
        ahbtbmininit();
    end

    `uvm_info(get_type_name(), "Asking for a config_sequence...",
        UVM_HIGH)
    seq_item_port.get(req);

    test();

    @(posedge config_vif.Clk_S);
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd1) begin
        `uvm_info(get_type_name(), "Executing ahbtbmdone...",
            UVM_HIGH)
        ahbtbmdone();
    end
endtask: run_phase

```

Código 4.24. Funciones principales del componente `ccsds121_config_driver`

La principal diferencia radica en la implementación de la tarea correspondiente a la ejecución del `test`. Mientras que en la implementación del `test bench` original se tenía un procedimiento para cada uno de los posibles `test` a ejecutar, lo cual daba lugar a bastante código repetitivo, la implementación realizada en este caso optimiza el código para hacer uso de una única tarea `test`, la cual recoge la funcionalidad de los cuatro `test` que se ha decidido considerar. El funcionamiento de esta tarea para el `test` normal seguiría el siguiente patrón durante dos ciclos:

1. Señal `AwaitingConfig` a nivel alto. Se envía la configuración del proceso de compresión.
2. Señal `AwaitingConfig` a nivel bajo y señal `Ready` a nivel alto. Se espera mientras el componente `UVM Agent` de datos transmite el flujo de datos de entrada y recibe el flujo de datos de salida.
3. Señal `Finished` a nivel alto. Se espera a que el IP121 se encuentre listo para una nueva configuración.

Sin embargo, durante la primera compresión se incluyen determinadas condiciones sobre la variable del tipo de `test` a ejecutar para cubrir casos como, por ejemplo, los de `Error` (tras un error de configuración, la señal `AwaitingConfig` vuelve a situarse a nivel alto para recibir una nueva

configuración) o ForceStop (tras forzar la detención de la compresión, la señal Finished se activa a nivel alto). La implementación de esta tarea se recoge en el Código 4.25.

```
// Test development
virtual task test ();
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd1) begin
        config_reg[1][31:16] = req.Nx_tb;
        config_reg[1][15]    = req.CODESET_tb;
        config_reg[1][14]    = req.DISABLE_HEADER_tb;
        config_reg[1][13:7]  = req.J_tb;
        if (test_type == "error")
            config_reg[1][6:0] = (req.D_tb-2);
        else
            config_reg[1][6:0] = req.W_BUFFER_tb;

        config_reg[2][31:16] = req.Ny_tb;
        config_reg[2][15:3]  = req.REF_SAMPLE_tb;
        config_reg[2][2:0]   = 3'b0;

        config_reg[3][31:16] = req.Nz_tb;
        config_reg[3][15:10] = req.D_tb;
        config_reg[3][8]     = req.ENDIANESS_tb;
        config_reg[3][7:6]   = req.PREPROCESSOR_tb;
        config_reg[3][5]     = req.BYPASS_tb;
        config_reg[3][4:0]   = 5'b0;

        config_reg[0][0] = 1'b0;

        address = 32'h10000000;
        @(posedge config_vif.Clk_S);
        `uvm_info(get_type_name(), "Sending a new configuration...",
            UVM_LOW)
        ahbwrite_burst(address, config_reg, 2'b10, 32'd4, 32'd2);
        // Single write of valid
        config_reg[0][0] = 1'b1;
        ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
            1'b1);
    end

    while (config_vif.AwaitingConfig != 1'b0) @(posedge
config_vif.Clk_S);

    if (test_type == "reconfig") begin
        // Wait a while to reconfigure
        repeat(50) @(posedge config_vif.Clk_S);
        `uvm_info(get_type_name(), "Reconfiguring the IP...", UVM_LOW)
        config_reg[0][0] = 1'b0;
        ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
            1'b1);
        ahbwrite_burst(address, config_reg, 2'b10, 32'd4, 32'd2);
        config_reg[0][0] = 1'b1;
        ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
            1'b1);
    end

    if (test_type != "error") begin
        if (test_type == "force_stop") begin
            // Wait a while to force the stop
            repeat(100) @(posedge config_vif.Clk_S);
        end
    end
end
```

```

        `uvm_info(get_type_name(), "ForceStop assertion", UVM_LOW)
        config_vif.ForceStop <= 1'b1;
        @(posedge config_vif.Clk_S);
        config_vif.ForceStop <= 1'b0;
    end

    while (config_vif.Finished == 1'b0) @(posedge
        config_vif.Clk_S);
end

while (config_vif.AwaitingConfig != 1'b1) @(posedge
    config_vif.Clk_S);

@(posedge config_vif.Clk_S);

if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd1) begin
    `uvm_info(get_type_name(), "Sending a new configuration...",
        UVM_LOW)
    if (test_type == "error") config_reg[1][6:0] =
        req.W_BUFFER_tb;
    config_reg[0][0] = 1'b0;
    ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
        1'b1);
    ahbwrite_burst(address, config_reg, 2'b10, 32'd4, 32'd2);
    config_reg[0][0] = 1'b1;
    ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
        1'b1);
end

while (config_vif.AwaitingConfig == 1'b1) begin
    @(posedge config_vif.Clk_S);
end

if (test_type == "reconfig") begin
    // Wait a while to reconfigure
    repeat(50) @(posedge config_vif.Clk_S);
    `uvm_info(get_type_name(), "Reconfiguring the IP...", UVM_LOW)
    config_reg[0][0] = 1'b0;
    ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
        1'b1);
    ahbwrite_burst(address, config_reg, 2'b10, 32'd4, 32'd2);
    config_reg[0][0] = 1'b1;
    ahbwrite(address, config_reg[0], 2'b10, 2'b10, 1'b1, 32'd2,
        1'b1);
end

while (config_vif.Finished == 1'b0) @(posedge config_vif.Clk_S);

while (config_vif.AwaitingConfig != 1'b1) @(posedge
    config_vif.Clk_S);
endtask: test

```

Código 4.25. Tarea test

Como se ha podido observar en los códigos correspondientes a este componente, se hace uso de las tareas `ahbtbminit`, `ahbtbmdone` y `ahbwrite`, cuya funcionalidad coincide con la de su implementación en el *test bench* original. Estas tareas han sido redefinidas localmente y de forma

manual en este componente *UVM Driver* (lenguaje *SystemVerilog*), adaptándolas también para su comunicación con la interfaz virtual.

4.5.7.3 COMPONENTE MONITOR DE CONFIGURACIÓN

El componente *UVM Monitor* de configuración se implementa en la clase `ccsds121_config_monitor`, que se encuentra en el archivo `ccsds121_config_monitor.sv`. En la descripción de este componente, las principales tareas que realiza son las siguientes:

- **build_phase.** Al igual que en el componente *UVM Driver* de configuración, efectúa la extracción de la interfaz virtual de configuración y del nombre del tipo de *test* a ejecutar desde la base de datos de UVM, asignándolos a variables locales. En caso de no encontrar alguno de estos parámetros, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación.
- **run_phase.** Comprobación, en una tarea *test* (siguiendo la misma terminología que en el componente *UVM Driver* de configuración), de que las señales de control siguen el patrón correcto en función del tipo de *test* que se ejecuta. Este patrón fue presentado, desde un punto de vista general, en el apartado 4.5.7.2 Componente Driver de configuración. El cumplimiento, o no, de este patrón por parte de las señales de control, se notifica mediante el uso del mecanismo de mensajes de UVM.

La implementación del componente *UVM Monitor* de configuración se muestra en el Código 4.26.

```
class ccsds121_config_monitor extends uvm_monitor;

    virtual ccsds121_config_if config_vif;
    string test_type;

    `uvm_component_utils(ccsds121_config_monitor)

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

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

        if (!uvm_config_db#(virtual ccsds121_config_if)::get(this, "",
            "config_vif", config_vif))
            `uvm_fatal("NO_CONFIG_VIF", {"Virtual interface must be set
            for: ", get_full_name(), ".config_vif"})
    endfunction: build_phase
endclass: ccsds121_config_monitor
```

```

if (!uvm_config_db#(string)::get(this, "", "test_type",
test_type))
    `uvm_fatal("NO_TEST_TYPE", "The type of test to run is not in
the database")

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

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

virtual task test ();
if (config_vif.Finished == 1'b1)
    `uvm_warning(get_type_name(), "Finished started with a high
value")

if (config_vif.Ready == 1'b1)
    `uvm_warning(get_type_name(), "Ready started with a high
value")

if (config_vif.AwaitingConfig == 1'b0)
    `uvm_warning(get_type_name(), "AwaitingConfig started with a
low value")

while (config_vif.AwaitingConfig != 1'b0) @(posedge
config_vif.Clk_S);

`uvm_info(get_type_name(), "AwaitingConfig lowered correctly
when configuration was received", UVM_LOW)

if (test_type != "error") begin
if (config_vif.Ready != 1'b1)
    `uvm_error(get_type_name(), "Ready not correctly asserted
when IP core has been configured")
else
    `uvm_info(get_type_name(), "Ready correctly asserted when
IP core is ready to receive new samples", UVM_LOW)

while (config_vif.Finished == 1'b0) begin
if (config_vif.Error != 1'b0)
    `uvm_error(get_type_name(), "Unexpected IP core error
during compression")
    @(posedge config_vif.Clk_S);
end

    `uvm_info(get_type_name(), "Finished correctly activated when
compression finished or stopped", UVM_LOW)

if (config_vif.Error != 1'b0)
    `uvm_error(get_type_name(), "Unexpected IP core error
after compression")
end
else begin
if (config_vif.Error == 1'b1)
    `uvm_info(get_type_name(), "Error has been correctly
asserted", UVM_LOW)
else
    `uvm_error(get_type_name(), "Error has not been correctly
asserted when error")

```

```

    if (config_vif.Finished == 1'b1)
        `uvm_info(get_type_name(), "Finished has been correctly
            asserted after an error", UVM_LOW)
    else
        `uvm_error(get_type_name(), "Finished has not been
            correctly asserted after an error")
    end

    while (config_vif.AwaitingConfig != 1'b1) @(posedge
        config_vif.Clk_S);
    `uvm_info(get_type_name(), "AwaitingConfig correctly activated
        after Finished", UVM_LOW)
    @(posedge config_vif.Clk_S);

    while (config_vif.AwaitingConfig == 1'b1) begin
        if (config_vif.Finished != 1'b1)
            `uvm_error(get_type_name(), "Error between sequential
                compressions, value of Finished shall be kept high")
            @(posedge config_vif.Clk_S);
        end

        `uvm_info(get_type_name(), "AwaitingConfig correctly lowered
            when configuration was received", UVM_LOW)

        if (config_vif.Ready != 1'b1)
            `uvm_error(get_type_name(), "Ready not correctly asserted
                when IP core has been configured")
        else
            `uvm_info(get_type_name(), "Ready correctly asserted when IP
                core is ready to receive new samples", UVM_LOW)

        if (config_vif.Finished != 1'b0)
            `uvm_error(get_type_name(), "Error for sequential
                compressions, Finished shall be de-asserted with
                AwaitingConfig")

        while (config_vif.Finished == 1'b0) begin
            if (config_vif.Error != 1'b0)
                `uvm_error(get_type_name(), "Unexpected IP core error
                    during compression")
                @(posedge config_vif.Clk_S);
            end

            `uvm_info(get_type_name(), "Finished correctly activated when
                compression finished", UVM_LOW)

            if (config_vif.Error != 1'b0)
                `uvm_error(get_type_name(), "Unexpected IP core error after
                    compression")

            while (config_vif.AwaitingConfig != 1'b1) @(posedge
                config_vif.Clk_S);

            `uvm_info(get_type_name(), "AwaitingConfig correctly activated
                after compression finished", UVM_LOW)
            `uvm_info(get_type_name(), "Test performed", UVM_LOW)
        endtask: test
    endclass: ccsds121_config_monitor

```

Código 4.26. Componente ccsds121_config_monitor

4.5.8 COMPONENTE *AGENT* DE DATOS

El componente *UVM Agent* de datos se denomina `ccsds121_data_agent`, y se encuentra implementado en el mismo archivo que el componente *UVM Agent* de configuración, siguiendo el mismo principio. La única discrepancia consiste en que los componentes *UVM Sequencer*, *UVM Driver* y *UVM Monitor* que integra, son de datos y no de configuración. La implementación del componente *UVM Agent* de datos se muestra en el Código 4.27.

```
class ccsds121_data_agent extends uvm_agent;

    ccsds121_data_sequencer sequencer;
    ccsds121_data_driver driver;
    ccsds121_data_monitor monitor;

    `uvm_component_utils(ccsds121_data_agent)

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

function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    sequencer = ccsds121_data_sequencer::
        type_id::create("data_sequencer", this);
    driver = ccsds121_data_driver::
        type_id::create("data_driver", this);
    monitor = ccsds121_data_monitor::
        type_id::create("data_monitor", this);
    `uvm_info(get_full_name(), "Build stage completed.", UVM_LOW)
endfunction: build_phase

function void connect_phase (uvm_phase phase);
    driver.seq_item_port.connect(sequencer.seq_item_export);
    `uvm_info(get_full_name(), "Connect stage completed.",
        UVM_LOW)
endfunction: connect_phase

endclass: ccsds121_data_agent
```

Código 4.27. Componente `ccsds121_data_agent`

4.5.8.1 COMPONENTE *SEQUENCER* DE DATOS

El componente *UVM Sequencer* de datos se denomina `ccsds121_data_sequencer`, y se implementa en el mismo archivo que el componente *UVM Sequencer* de configuración, siguiendo el mismo principio. Indica que se encarga de gestionar transacciones de datos, realiza su registro en la *Factory* de UVM, implementa su constructor, y hereda las funcionalidades de la clase

uvm_sequencer. El código *SystemVerilog* correspondiente a la implementación de este componente se recoge en el Código 4.28.

```
class ccstds121_data_sequencer extends uvm_sequencer
#(ccstds121_data_packet#(ccstds121_tb_parameters::W_BUFFER_G_tb));

    `uvm_sequencer_utils(ccstds121_data_sequencer)

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

endclass: ccstds121_data_sequencer
```

Código 4.28. Componente `ccstds121_data_sequencer`

4.5.8.2 COMPONENTE DRIVER DE DATOS

El componente *UVM Driver* de datos, denominado `ccstds121_data_driver`, se encuentra implementado en el archivo `ccstds121_data_driver.sv`, y realiza el proceso de transferencia del flujo de datos de entrada al DUV. Las principales tareas que ejecuta son las siguientes:

- **build_phase.** Extracción de la interfaz virtual de datos desde la base de datos de UVM, asignándola a una variable local. En caso de no encontrarla, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación.
- **run_phase.** Ejecución, en paralelo, de las tareas `reset` y `drive_inputs`. La primera de ellas se encarga de establecer las entradas de datos a unos valores por defecto en caso de `reset`, mientras que la segunda lleva a cabo el envío del flujo de datos correspondiente a los estímulos de entrada al IP121. Para esto último, se diseña una máquina de estados a partir de la ya existente en el entorno de verificación original.

La implementación del componente *UVM Driver* de datos se muestra en el Código 4.29.

```
class ccstds121_data_driver extends uvm_driver
#(ccstds121_data_packet#(ccstds121_tb_parameters::W_BUFFER_G_tb));

    virtual ccstds121_data_if#(ccstds121_tb_parameters::D_G_tb,
        ccstds121_tb_parameters::W_BUFFER_G_tb) data_vif;
    typedef enum logic [1:0] {idle, s0, s1, s2} state_type;
    state_type state_reg;
    integer J_tb, counter, words_prep_header;

    `uvm_component_utils(ccstds121_data_driver)
```



```

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

function void build_phase (uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual ccsds121_data_if
        #(ccsds121_tb_parameters::D_G_tb,
        ccsds121_tb_parameters::W_BUFFER_G_tb))
        ::get(this, "", "data_vif", data_vif))
        `uvm_fatal("NO_DATA_VIF", {"Virtual interface must be set
            for: ", get_full_name(), ".data_vif"})

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

virtual task run_phase (uvm_phase phase);
    fork
        reset();
        drive_inputs();
    join
endtask: run_phase

```

Código 4.29. Funciones principales del componente `ccsds121_data_driver`

En el Código 4.30 se muestra el código *SystemVerilog* correspondiente a la tarea `reset`. Como ya se ha comentado, esta tarea asigna unos valores por defecto a los puertos de entrada de datos del DUV en caso de que la señal de `reset` se encuentre activa a nivel bajo. Además, establece un estado determinado en la máquina de estados que efectúa la transferencia del flujo de datos de entrada.

```

// Reset task
virtual task reset ();
    forever begin
        if (data_vif.Rst_N == 1'b0) begin
            `uvm_info(get_type_name(), "Resetting signals...", UVM_HIGH)
            state_reg <= idle;
            data_vif.DataIn <= {ccsds121_tb_parameters::D_G_tb{1'b0}};
            data_vif.DataIn NewValid <= 1'b0;
            data_vif.IsHeaderIn <= 1'b0;
            data_vif.NbitsIn <= 6'b0;
            @(posedge data_vif.Rst_N);
        end
        @(posedge data_vif.Clk_S);
    end
endtask: reset

```

Código 4.30. Tarea `reset`

En este sentido, la tarea `drive_inputs` implementa la citada máquina de estados. En este caso, se parte de la implementación original, efectuando diversas modificaciones con el objetivo de optimizarla. Algunas de estas modificaciones son: supresión de un estado innecesario que existía

originalmente, eliminación de código repetitivo, e inclusión de la recepción de transacciones desde la secuencia de datos, así como el envío de respuestas a la misma. La funcionalidad original, sin embargo, se mantiene intacta. Una representación gráfica de la nueva máquina de estados implementada se muestra en la Figura 4.8.

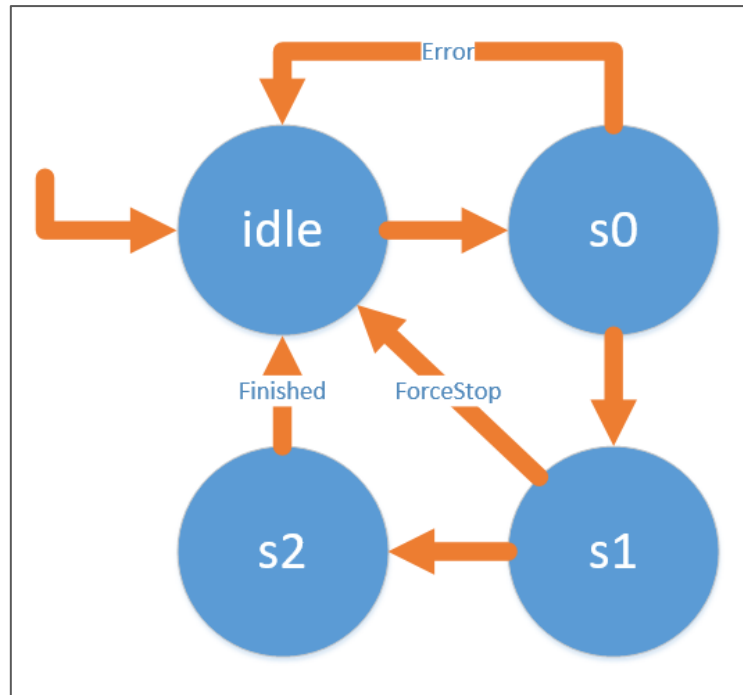


Figura 4.8. Diagrama de la nueva máquina de estados para el flujo de datos de entrada

Una vez presentada la máquina de estados implementada para transmitir el flujo de datos de entrada al DUV, se realizará una descripción detallada de cada uno de sus estados. Cabe destacar que previamente a la ejecución de la máquina de estados se crea una transacción de datos, que será utilizada para enviar respuestas a la secuencia de datos cuando sea necesario.

- **idle.** Estado en el que se recibe una transacción *idle* y se la asigna como identificador a la transacción ya creada para ser enviada como respuesta. Esto es necesario para poder enviar esta transacción a la secuencia de datos como respuesta. Seguidamente, se pasa al estado *s0*.
- **s0.** Estado en el que se espera a que el IP haya recibido la configuración correspondiente al proceso de compresión. En caso de que la configuración sea incorrecta (señal `Error` activa), se pasa al estado *idle* y se envía una respuesta a la secuencia de datos (función `put` sobre el puerto TLM), la cual conllevará que se finalice la secuencia. En el caso de una configuración correcta, se pasa al estado *s1*. Si el *test* a ejecutar introduce cabeceras de un

bloque preprocesador, estas se simulan y se envían al DUV antes de continuar al estado *s1*, mediante la tarea `get_prep_header_word` y un contador.

- **s1.** Estado en el que se reciben las transacciones de datos desde la secuencia de datos y se introducen en el DUV siguiendo el formato adecuado (según determinados parámetros como el tamaño del *buffer* de datos de entrada). En caso de que se fuerce una detención del proceso de compresión (señal `ForceStop` activa), se pasa al estado *idle* y se envía una respuesta a la secuencia de datos (función `put` sobre el puerto TLM), la cual provocará que se finalice la secuencia. En caso contrario, cuando se alcanza un determinado valor en un contador, se pasa al estado *s2*.
- **s2.** Estado en el que se reciben las transacciones de datos desde la secuencia de datos y se introducen en el DUV siguiendo el formato adecuado (según determinados parámetros como el tamaño del *buffer* de datos de entrada). Cuando la compresión finalice (señal `Finished` activa), se pasa al estado *idle* y se envía una respuesta a la secuencia de datos (función `put` sobre el puerto TLM), la cual conllevará que se finalice la secuencia.

Finalmente, la implementación *SystemVerilog* de la tarea `drive_inputs` se recoge en el Código 4.31.

```
// Task to drive the inputs and change the state in a synchronous way
virtual task drive_inputs ();
    integer word, prep_header;
    logic [31:0] varaux;
    bit new_read_req;

    J_tb = ccstds121_tb_parameters::J_tb - 32'd1; // Variable to analyze
        the counter state
    counter          = 32'd0;
    word             = 32'b0;
    words_prep_header = 32'd0;
    prep_header      = 32'b0;
    new_read_req     = 1'b1;

    if (((ccstds121_tb_parameters::D_tb <= 32'd8) &&
        (ccstds121_tb_parameters::EN_RUNCFG_G == 32'b1)) ||
        ((ccstds121_tb_parameters::D_G_tb <= 32'd8) &&
        (ccstds121_tb_parameters::EN_RUNCFG_G == 32'b0)))
        word = 32'd1;
    else
        word = 32'd2;

    if ((ccstds121_tb_parameters::DISABLE_HEADER_tb == 32'b0) &&
        (ccstds121_tb_parameters::PREPROCESSOR_tb != 32'b0))
        prep_header = 32'd17; // num_prep_header_words = 17

    rsp = ccstds121_data_packet#(ccstds121_tb_parameters::W_BUFFER_G_tb)
        ::type_id::create("rsp");
```

```

forever begin
  @(posedge data_vif.Rst_N);
  while (data_vif.Rst_N == 1'b1) begin
    `uvm_info(get_type_name(), "Driving inputs...", UVM_DEBUG)
    @(posedge data_vif.Clk_S);
    data_vif.DataIn_NewValid <= 1'b0;
    data_vif.IsHeaderIn      <= 1'b0;
    data_vif.NbitsIn        <= 6'b0;

    case (state_reg)
      idle: begin
        state_reg <= s0;
        seq_item_port.get(req); // Idle transaction
        rsp.set_id_info(req); // Enable response with the request ID
      end

      s0: begin
        if (data_vif.AwaitingConfig != 1'b1) begin
          if (data_vif.Ready == 1'b1) begin
            if (words_prep_header < prep_header)
              get_prep_header_word();
            else begin
              state_reg      <= s1;
              words_prep_header <= 32'b0;
            end
          end

          if (data_vif.Error == 1'b1) begin
            state_reg <= idle;
            counter  <= 32'b0;
            // Restart the stimuli file
            seq_item_port.put(rsp);
          end
        end
      end

      s1: begin
        if (data_vif.ForceStop == 1'b1) begin
          state_reg <= idle;
          counter  <= 32'b0;
          // Restart the stimuli file
          seq_item_port.put(rsp);
        end
        else begin
          // It always reads 32 bits, so we have to decide which
          // part to get every moment, and to consider the endianness
          if (data_vif.Ready == 1'b1) begin
            if (counter % (4/word) == 32'd1) begin
              if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1)
                begin
                  if (word == 32'd2)
                    data_vif.DataIn[15:0] <= {varaux[23:16],
                                                  varaux[31:24]};
                  else
                    data_vif.DataIn[7:0] <= varaux[15:8];
                end
            end
            else begin
              if ((ccsds121_tb_parameters::ENDIANESS_tb ==
                    32'b0) && (word == 32'd2))
                data_vif.DataIn[(ccsds121_tb_parameters::
                                  D_G_tb-1):0] <= {varaux[23:16], varaux
                                                       [24+: (ccsds121_tb_parameters::D_G_tb-8)]};
            end
          end
        end
      end
    end
  end

```

```

        else
            data_vif.DataIn[(ccsds121_tb_parameters::
                D_G_tb-1):0] <= varaux
                [(word*8)+:ccsds121_tb_parameters::D_G_tb];
        end
        data_vif.DataIn_NewValid <= 1'b1;
    end
else if (counter % (4/word) == 32'd2) begin
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1)
        data_vif.DataIn[7:0] <= varaux[23:16];
    else
        data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0]
            <= varaux[16+:ccsds121_tb_parameters::D_G_tb];
        data_vif.DataIn_NewValid <= 1'b1;
    end
end
else if (counter % (4/word) == 32'd3) begin
    if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1)
        data_vif.DataIn[7:0] <= varaux[31:24];
    else
        data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0]
            <= varaux[24+:ccsds121_tb_parameters::D_G_tb];
        data_vif.DataIn_NewValid <= 1'b1;
    end
end
else begin
    // if not endfile(stim)
    seq_item_port.try_next_item(req);
    if (req != null) begin
        varaux = req.DataIn;
        seq_item_port.item_done();
        `uvm_info(get_full_name(), $sformatf("Data read from
            the input file: %h - S1", varaux), UVM_FULL)

        if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1) begin
            if (word == 32'd2)
                data_vif.DataIn[15:0] <= {varaux[7:0],
                    varaux[15:8]};
            else
                data_vif.DataIn[7:0] <= varaux[7:0];
            end
        else begin
            if ((ccsds121_tb_parameters::ENDIANESS_tb == 32'b0)
                && (word == 32'd2))
                data_vif.DataIn[(ccsds121_tb_parameters::
                    D_G_tb-1):0] <= {varaux[7:0],
                    varaux[8+: (ccsds121_tb_parameters::D_G_tb-8)]};
            else
                data_vif.DataIn[(ccsds121_tb_parameters::
                    D_G_tb-1):0] <=
                    varaux[(ccsds121_tb_parameters::D_G_tb-1):0];
            end
        end
        data_vif.DataIn_NewValid <= 1'b1;
    end
end
end
change_counter();
end
end
end
end

s2: begin
    if (data_vif.Finished == 1'b0) begin
        if ((data_vif.Ready == 1'b1) && (data_vif.AwaitingConfig !=
            1'b1)) begin

```

```

if (counter % (4/word) == 32'd1) begin
  new_read_req = 1'b1;
  if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1) begin
    if (word == 32'd2)
      data_vif.DataIn[15:0] <= {varaux[23:16],
        varaux[31:24]};
    else
      data_vif.DataIn[7:0] <= varaux[15:8];
    end
  else begin
    if ((ccsds121_tb_parameters::ENDIANESS_tb == 32'b0) && (word ==
      32'd2))
      data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0] <=
        {varaux[23:16], varaux[24+: (ccsds121_tb_parameters::
          D_G_tb-8)]};
    else
      data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0] <=
        varaux[(word*8)+:ccsds121_tb_parameters::D_G_tb];
    end
    data_vif.DataIn_NewValid <= 1'b1;
  end
else if (counter % (4/word) == 32'd2) begin
  if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1)
    data_vif.DataIn[7:0] <= varaux[23:16];
  else
    data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0] <=
      varaux[16+:ccsds121_tb_parameters::D_G_tb];
    data_vif.DataIn_NewValid <= 1'b1;
  end
else if (counter % (4/word) == 32'd3) begin
  if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1)
    data_vif.DataIn[7:0] <= varaux[31:24];
  else
    data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0] <=
      varaux[24+:ccsds121_tb_parameters::D_G_tb];
    data_vif.DataIn_NewValid <= 1'b1;
  end
else begin
  // if (not endfile(stim) and new_read_req = '1')
  if (new_read_req == 1'b1) begin
    seq_item_port.try_next_item(req);
    if (req != null) begin
      varaux = req.DataIn;
      seq_item_port.item_done();
      new_read_req = 1'b0;
      `uvm_info(get_full_name(), $sformatf("Data read from the
        input file: %h - S2", varaux), UVM_FULL)
    end
  end
end

  if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'b1) begin
    if (word == 32'd2)
      data_vif.DataIn[15:0] <= {varaux[7:0],varaux[15:8]};
    else
      data_vif.DataIn[7:0] <= varaux[7:0];
    end
  else begin
    if ((ccsds121_tb_parameters::ENDIANESS_tb == 32'b0) &&(word ==
      32'd2))
      data_vif.DataIn[(ccsds121_tb_parameters::D_G_tb-1):0] <=
        {varaux[7:0], varaux[8+: (ccsds121_tb_parameters::
          D_G_tb-8)]};

```

```

        else
            data_vif.DataIn[(ccsds121_tb_parameters
                ::D_G_tb-1):0] <= varaux
                [(ccsds121_tb_parameters::D_G_tb-1):0];
        end
        data_vif.DataIn_NewValid <= 1'b1;
    end
    change_counter();
end
else begin
    state_reg          <= idle;
    counter            <= 32'd0;
    words_prep_header <= 32'd0;
    // Restart the stimuli file
    seq_item_port.put(rsp);
end
end
endcase
end
endtask: drive_inputs

// Task for simulating an input preprocessing header
virtual task get_prep_header_word ();
    data_vif.DataIn <=
        words_prep_header[ccsds121_tb_parameters::D_G_tb-1:0];
    data_vif.NbitsIn          <= 6'd8;
    words_prep_header        <= words_prep_header + 32'd1;
    data_vif.DataIn_NewValid <= 1'b1;
    data_vif.IsHeaderIn     <= 1'b1;
endtask: get_prep_header_word

// Task for changing the value of the "counter" variable and the state
if necessary
virtual task change_counter ();
    if (counter == J_tb) begin
        counter <= 32'd0;
        if (state_reg == s1) state_reg <= s2;
    end
    else
        counter <= counter + 32'd1;
endtask: change_counter

```

Código 4.31. Tarea drive_inputs

4.5.8.3 COMPONENTE MONITOR DE DATOS

El componente *UVM Monitor* de datos se denomina `ccsds121_data_monitor`, y se encuentra implementado en el archivo `ccsds121_data_monitor.sv`. Su principal funcionalidad es la de recibir el flujo de datos de salida del DUV tras finalizar el proceso de compresión y enviarlo al componente *UVM Scoreboard* para realizar las comprobaciones pertinentes. Las tareas que ejecuta para ello el componente *UVM Monitor* de datos, son las siguientes:

- **build_phase.** Extracción de la interfaz virtual de datos desde la base de datos de UVM, asignándola a una variable local. En caso de no encontrarla, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación. Creación del puerto *TLM analysis port* y de las transacciones de datos necesarias para enviar los datos de salida al componente *UVM Scoreboard*.
- **run_phase.** En este caso, también se parte de la implementación original del proceso de recepción del flujo de datos de salida. En cada ciclo de reloj se evalúa el estado de las señales de salida de datos y de control. En caso de existir un dato válido en la salida (señal `DataOut_NewValid` activa), se incluye el mismo en una transacción de datos y se envía al componente *UVM Scoreboard*. Si se activan las señales `ForceStop` o `Error`, se envía una transacción al componente *UVM Scoreboard* indicando que la comparación que se está realizando no es válida. Finalmente, si se activa la señal de finalización del proceso de compresión (`Finished`), también se informa al componente *UVM Scoreboard* mediante una transacción, para que finalice el proceso de comparación.

La implementación del componente *UVM Monitor* de datos se muestra en el Código 4.32.

```
class ccsds121_data_monitor extends uvm_monitor;

    virtual ccsds121_data_if#(ccsds121_tb_parameters::D_G_tb,
        ccsds121_tb_parameters::W_BUFFER_G_tb) data_vif;

    uvm_analysis_port#(ccsds121_data_packet
        #(ccsds121_tb_parameters::W_BUFFER_G_tb)) output_collected_port;
    ccsds121_data_packet#(ccsds121_tb_parameters::W_BUFFER_G_tb)
        output_data_collected, output_data_clone;

    `uvm_component_utils(ccsds121_data_monitor)

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

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

        if (!uvm_config_db#(virtual ccsds121_data_if
            #(ccsds121_tb_parameters::D_G_tb, ccsds121_tb_parameters::
                W_BUFFER_G_tb))::get(this, "", "data_vif", data_vif))
            `uvm_fatal("NO_DATA_VIF", {"Virtual interface must be set
                for: ", get_full_name(), ".data_vif"})

        output_collected_port = new("output_collected_port", this);
        output_data_collected = ccsds121_data_packet
            #(ccsds121_tb_parameters::W_BUFFER_G_tb)::
            type_id::create("output_data_collected");
        output_data_clone = ccsds121_data_packet
            #(ccsds121_tb_parameters::W_BUFFER_G_tb)::
            type_id::create("output data clone");
```



```

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

virtual task run_phase (uvm_phase phase);
    bit fin, error;
    fin    = 1'b0;
    error  = 1'b1;
    forever begin
        @(posedge data_vif.Clk_S);
        if (!data_vif.Rst_N) begin
            fin = 1'b0;
        end
        else if (data_vif.ForceStop) begin
            fin    = 1'b0;
            error  = 1'b0;
            output_data_collected.NotCompare = 1'b1;
            $cast(output_data_clone, output_data_collected.clone());
            output_collected_port.write(output_data_clone);
            #1000; // Wait for the IP to be steady
        end
        else if (data_vif.Error) begin
            if (error) begin
                fin    = 1'b0;
                error  = 1'b0;
                output_data_collected.NotCompare = 1'b1;
                $cast(output_data_clone,
                    output_data_collected.clone());
                output_collected_port.write(output_data_clone);
            end
        end
        else begin
            if ((data_vif.DataOut_NewValid) &&
                (!data_vif.AwaitingConfig)) begin
                fin = 1'b1;
                output_data_collected.DataOut    = data_vif.DataOut;
                output_data_collected.NotCompare = 1'b0;
                output_data_collected.Finished   = 1'b0;
                $cast(output_data_clone,
                    output_data_collected.clone());
                output_collected_port.write(output_data_clone);
            end

            if (data_vif.Finished) begin
                if (fin) begin
                    fin    = 1'b0;
                    error  = 1'b0;
                    output_data_collected.NotCompare = 1'b0;
                    output_data_collected.Finished   = 1'b1;
                    $cast(output_data_clone,
                        output_data_collected.clone());
                    output_collected_port.write(output_data_clone);
                end
            end
        end
    end
endtask: run_phase

endclass: ccstds121_data_monitor

```

Código 4.32. Componente ccstds121_data_monitor

4.5.9 COMPONENTE SCOREBOARD

El componente *UVM Scoreboard*, denominado `ccsds121_scoreboard`, se define en el archivo `ccsds121_scoreboard.sv`. Este componente efectuará las tareas de comparación del flujo de datos de salida del IP121 tras finalizar el proceso de compresión con los valores de referencia. Sus principales funciones son:

- **build_phase.** Creación del puerto *TLM analysis export* y de la transacción de datos necesaria para recibir los datos de salida desde el componente *UVM Monitor* de datos. Extracción del nombre del archivo que contiene los datos de salida de referencia desde la base de datos de UVM. En caso de no encontrarlo, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación.
- **run_phase.** En esta fase se ejecuta el procedimiento de comparación que, al contrario que en su implementación original, se efectuará *on the fly*, sin almacenar el flujo de datos de salida en un archivo externo. Como ya se ha comentado al describir el componente *UVM Test* del entorno de verificación basado en UVM desarrollado en este TFM, cada *test* ejecuta dos secuencias de datos, por lo que el proceso de comparación también se realizará dos veces. Este proceso, en primer lugar, abre el archivo de datos de referencia comprimidos. Tras ello, se evalúa si el *test* en cuestión incluye cabeceras de un bloque preprocesador, en cuyo caso se desecha la cantidad de transacciones de datos recibidas correspondientes a estas cabeceras. A continuación, se reciben transacciones de datos válidas y se efectúa la comparación, *byte a byte*, con los datos del fichero de referencia (si se trata de un *test* de valores residuales, se descartan algunos *bytes* y se comparan otros). En cualquier caso, si se recibe una transacción con los campos `Finished` (se han recibido todos los datos comprimidos) o `NotCompare` (la comparación no es válida) activos a nivel alto, se finaliza el proceso de comparación. Además de comprobar que los datos comparados coinciden, también se tiene en cuenta que la cantidad de datos recibidos, y de referencia, sea la misma.
- **report_phase.** Recuento del número de errores obtenidos durante el proceso de comparación. En función de este valor, se utilizará el mecanismo de mensajes de UVM para notificar el éxito, o no, de la ejecución del *test*.

La implementación *SystemVerilog* del componente *UVM Scoreboard* se recoge en el Código 4.33.

```

class ccsds121_scoreboard extends uvm_scoreboard;

    uvm_tlm_analysis_fifo#(ccsds121_data_packet#
        (ccsds121_tb_parameters::W_BUFFER_G_tb)) output_packets_collected;
    ccsds121_data_packet#(ccsds121_tb_parameters::W_BUFFER_G_tb)
        output_packet;

    string ref_name;
    integer reference_file;
    integer error, size, counter, bytes_discard_top;
    integer outputs, remainder; // To by-pass preprocessing headers
    byte reference_byte, output_byte;
    bit ini, fin, not_compare;

    logic [(ccsds121_tb_parameters::W_BUFFER_G_tb-1):0] data_received;

    `uvm_component_utils(ccsds121_scoreboard)

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

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

        output_packets_collected = new("output_packets_collected",
            this);
        output_packet = ccsds121_data_packet#(ccsds121_tb_parameters::
            W_BUFFER_G_tb)::type_id::create("output_packet");

        if(!uvm_config_db#(string)::get(this, "", "ref_name", ref_name))
            `uvm_fatal("NO_REF_NAME", "File name must be set for the
                reference file.")

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

    virtual task run_phase (uvm_phase phase);
        super.run_phase(phase);
        error = 0;
        ini = 1'b0;
        fin = 1'b0;
        not_compare = 1'b0;

        if (ccsds121_tb_parameters::EN_RUNCFG_G == 32'd1) begin
            size = ccsds121_tb_parameters::W_BUFFER_tb;
            bytes_discard_top = (size/8) -
                ceil(ccsds121_tb_parameters::D_G_tb,8);
        end
        else begin
            size = ccsds121_tb_parameters::W_BUFFER_G_tb;
            bytes_discard_top = (size/8) -
                ceil(ccsds121_tb_parameters::D_G_tb,8);
        end

        repeat(2) begin
            // Opening reference file
            reference_file = $fopen(ref_name, "r");
            void'($rewind(reference_file));
        end
    endtask: run_phase
endclass: ccsds121_scoreboard
    
```

```

// Checking if it is necessary to bypass some preprocessing header
if (ccsds121_tb_parameters::DISABLE_HEADER_tb == 32'd0) begin
  if(ccsds121_tb_parameters::PREPROCESSOR_tb != 32'd0) begin
    outputs = 17/(size/8);
    remainder = 17%(size/8);
    repeat (outputs) begin // num_prep_header_words = 17
      output_packets_collected.get(output_packet);
      if (output_packet.NotCompare) begin
        not_compare = 1'b1;
        continue;
      end
    end
    if (not_compare) continue;
    output_packets_collected.get(output_packet);
    if (output_packet.NotCompare) continue;
    data_received = output_packet.DataOut;
    data_received = data_received << remainder*8;
    repeat ((size/8)-remainder) check_byte();
  end
end

forever begin
  output_packets_collected.get(output_packet);
  if (output_packet.NotCompare) begin
    ini = 1'b0;
    fin = 1'b0;
    break;
  end
  else if (output_packet.Finished) begin
    if (fin) begin
      ini = 1'b0;
      break;
    end
  end
  else begin
    if (!ini) begin
      ini = 1'b1;
      fin = 1'b1;
    end

    // Read the output words and comparison part (byte per byte)
    data_received = output_packet.DataOut;
    `uvm_info(get_full_name(), $sformatf("Data received from the
      DUV: %h", data_received), UVM_FULL)

    if (ccsds121_tb_parameters::BYPASS_tb == 32'd0) begin
      repeat (size/8) check_byte();
    end
    else begin
      // Check the bypassed residuals
      data_received = data_received << (8*bytes_discard_top);
      repeat ((ccsds121_tb_parameters::W_BUFFER_G_tb/8) -
        bytes_discard_top) check_byte();
    end
  end
end

if (fin) begin
  fin = 1'b0;
  forever begin
    reference_byte = $fgetc(reference_file);
  end
end

```

```

        if ($feof(reference_file)) break;
        if (reference_byte != 8'b0) begin
            error = error + 1;
            `uvm_error(get_type_name(), "Reference file has more
            samples")
        end
    end
end

    $fclose(reference_file);
end
endtask: run_phase

virtual task check_byte ();
    output_byte = data_received[(size-1)-:8];
    reference_byte = $fgetc(reference_file);
    if ($feof(reference_file)) begin
        if (output_byte != 8'b0) begin
            error = error + 1;
            `uvm_error(get_type_name(), "Output file has more
            samples")
        end
    end
    else if (reference_byte != output_byte) begin
        error = error + 1;
        `uvm_error(get_type_name(), "Problems in final stream")
    end
    data_received = data_received << 8;
endtask: check_byte

function integer ceil (input integer A,
                       input integer B);

    integer q, r;
    q = A/B;
    r = A - q*B;
    if (r > 32'd0)
        q = q + 1;
    return q;
endfunction: ceil

function void report_phase (uvm_phase phase);
    if (error == 0)
        `uvm_info(get_type_name(), "Test was run successfully.",
        UVM_LOW)
    else if (error == 1)
        `uvm_error(get_type_name(), "The execution of the test was
        not successful: There was 1 error.")
    else
        `uvm_error(get_type_name(), $sformatf("The execution of the
        test was not successful: There were %d errors.", error))
endfunction: report_phase

endclass: ccsds121_scoreboard

```

Código 4.33. Componente ccsds121_scoreboard

4.5.10 MAKEFILE

Finalmente, la ejecución de los *test* sobre el entorno de verificación desarrollado se realizará mediante un *Makefile*. En este archivo, se detallan los comandos necesarios para la compilación de los diferentes ficheros involucrados en la simulación, así como para su ejecución. El principal contenido de este archivo, denominado *Makefile*, se muestra en el Código 4.34.

```

DATA = 25_Test
UVM_VERB = UVM_LOW
UVM_TEST = ccsds121_normal_test
SRC = /home/users/master/divdsi/srodriguez/Compresion_UVM/src
TEST_PARAM = /home/users/master/divdsi/srodriguez/Compresion_UVM/
test_stimuli/$(DATA)
VLOG_OPT = +incdir+/home/users/master/divdsi/srodriguez/
Compresion_UVM/ +incdir+/home/users/master/divdsi/srodriguez/
Compresion_UVM/test_stimuli/ +incdir+/home/users/master/divdsi/
srodriguez/Compresion_UVM/test_stimuli/$(DATA)/
VSIM_OPT = -voptargs=+acc
+uvm_set_action=*, ILLEGALNAME, UVM_WARNING, UVM_NO_ACTION
+uvm_set_action=*, uvm_link_transaction, UVM_ERROR, UVM_NO_ACTION

ifeq ($(UVM_TEST), ccsds121_forcestop_test)
    DATA = 04_Test
endif

lib:
    vlib work
    vlib shyloc_121
    vlib shyloc_utils

comp:
ifeq ($(UVM_TEST), ccsds121_forcestop_test)
    echo "Force Stop test only allows 04_Test data set"
endif
    make compile_duv
    vlog -incr $(VLOG_OPT) ccsds121_top.sv

run:
    vsim \
    +UVM_TESTNAME=$(UVM_TEST) +UVM_VERBOSITY=$(UVM_VERB) \
    $(VSIM_OPT) -classdebug -msgmode both -uvmcontrol=all \
    -gui ccsds121_top -do "set NumericStdNoWarnings 1; run 0; do
    wave.do; run -all"

runc:
    vsim \
    +UVM_TESTNAME=$(UVM_TEST) +UVM_VERBOSITY=$(UVM_VERB) \
    $(VSIM_OPT) \
    -c \
    -do "set NumericStdNoWarnings 1; run -all; quit -f" \
    ccsds121_top

all: clean lib comp run

allc: clean lib comp runc

```

```
clean:
  rm -rf *~ work shyloc_121 shyloc_utils vsim.wlf* \
  *.log questa.tops transcript *.vstf
```

Código 4.34. Contenido principal del archivo *Makefile*

Así, en primer lugar, se definen una serie de variables, entre las que destacan las siguientes:

- **DATA.** Nombre del *set* de estímulos de datos y parámetros de configuración.
- **UVM_VERB.** Nivel de *verbosity* a asignar a la variable global del entorno `UVM_VERBOSITY`.
- **UVM_TEST.** Nombre del *test* a ejecutar.
- **TEST_PARAM.** Ruta del directorio en el que se encuentran los archivos de parámetros.

Seguidamente, se detallan los diferentes grupos de acciones que pueden ejecutarse, siendo los más destacados los siguientes:

- **lib.** Creación de las diferentes bibliotecas lógicas locales en las que se compilarán los archivos fuente del proyecto.
- **comp.** En primer lugar, compila todos los archivos relativos al DUV, para lo que se reutiliza el código de compilación de la implementación original, añadiendo la compilación del DUV creado, e incluyendo la opción `-mixedsvvh` en los casos que sea necesario. Posteriormente, se compila módulo `Top ccsds121_top.sv`, a partir del cual también se compilarán el resto de componentes y objetos del entorno de verificación basado en UVM.
- **run.** Ejecución de la simulación en la herramienta *QuestaSim*, utilizando el componente *UVM Test* y el nivel de *verbosity* especificados. En este caso, se utiliza un archivo `wave.do` para visualizar, en dicha herramienta, las formas de onda de las principales señales involucradas en la simulación.
- **runc.** Similar al grupo de comandos `run`, pero de forma “silenciosa”, sin abrir la herramienta *QuestaSim* (opción `-c` para el comando `vsim`). De este modo, no se visualizan las formas de onda, y los diferentes pasos y resultados de la simulación se muestran en la consola.
- **clean.** Eliminación de las bibliotecas lógicas creadas por el grupo de comandos `lib`, el fichero `log`, y otros archivos residuales de la ejecución de una simulación previa.
- **all.** Ejecución, de forma secuencial, de los grupos de comandos `clean`, `lib`, `comp` y `run`.
- **allc.** Ejecución, de forma secuencial, de los grupos de comandos `clean`, `lib`, `comp` y `runc`.

Finalmente, cabe destacar que para el *test* `ccsds121_forcestop_test`, únicamente se permite hacer uso del *set* de estímulos `04_Test`. Esto se debe a que la implementación del *test bench* original así lo contemplaba.

4.6 ESTRUCTURA DE DIRECTORIOS

Para una mejor visión del entorno implementado con el fin de verificar la funcionalidad del IP121 mediante la metodología UVM, se detalla en este apartado la estructura de directorios asociada al proyecto desarrollado. La carpeta principal del proyecto es *Compresion_UVM*, y su contenido se describe a continuación:

- ***config_files_gen***. Directorio que contiene el *script Python* `gen_config_tests_121.py` y el fichero CSV `testcases_121.csv`. Estos archivos se utilizan para la generación de los archivos con los parámetros de configuración correspondientes a cada *test*.
- ***gplib-gpl-1.5.0-b4164***. Carpeta en la que se incluyen todos los archivos relativos a la biblioteca *GRLIB IP*. Esta carpeta ha sido copiada directamente del proyecto original.
- ***images***. Directorio con las carpetas *raw* y *reference*, en las cuales se encuentran los datos de las imágenes a utilizar como flujo de datos de entrada y como datos de referencia en los diferentes *test*, respectivamente.
- ***src***. Directorio con los archivos fuente del IP121. Este directorio ha sido copiado directamente del proyecto original. En la carpeta *tb* se ha eliminado el archivo VHDL correspondiente al *test bench* original y se ha añadido la nueva entidad VHDL creada para actuar como DUV en el *test bench* basado en UVM.
- ***test_stimuli***. Directorio en el que se almacenarán los archivos de parámetros de configuración para cada *test*, generados a partir del *script Python*. También se incluye en esta carpeta el fichero *SystemVerilog* con el paquete de funciones utilizado en los archivos de parámetros de configuración.
- En el directorio raíz del proyecto se encuentran todos los archivos descritos en el apartado 4.5 Estructura del entorno de verificación del IP121 basado en UVM. También se sitúan en este directorio el fichero `wave.do` para visualizar, en la herramienta *QuestaSim*, las formas de onda de las principales señales involucradas en la simulación, así como un archivo `modelsim.ini`, en el que se han mapeado las bibliotecas de *GRLIB IP* de forma local al entorno (este archivo se cargará en el inicio de cada simulación).

4.7 FLUJO DE EJECUCIÓN

El flujo de ejecución para conseguir que el entorno de verificación sea funcional y poder simular los diferentes *test* considerados, se resume en dos pasos básicos. El primero de ellos consiste en generar los archivos correspondientes a los parámetros de configuración del IP121 y del *test bench* (*ccsds121_parameters.vhd* y *ccsds121_tb_parameters.sv*, respectivamente) para los diferentes *sets* de estímulos de *test*. Para ello, se accede al directorio *Compresion_UVM/config_files_gen* desde la consola o *Terminal* y se ejecuta el *script Python*. El modo de realizarlo se muestra en el Código 4.35.

```
>> cd Compresion_UVM/config_files_gen
>> python gen_config_tests_121.py testcases_121.csv ../images/raw
    ../images/reference ../
```

Código 4.35. Generación de los archivos con los parámetros de configuración

A continuación, se sale de la carpeta *config_files_gen* hasta el directorio principal del proyecto y se hace uso del *Makefile* para ejecutar los *test* que se desee. Para ello, se asignan valores a las variables del *Makefile*, modificando su valor por defecto. Las variables de interés, en este caso, son *DATA*, *UVM_VERB* y *UVM_TEST* (definidas en el apartado 4.5.10 *Makefile*), si bien es cierto que el nivel de *verbosity* establecido por defecto es el adecuado y solamente debería modificarse para tareas de depuración. Por tanto, si no se asigna un valor a estas variables, se mantiene el valor por defecto definido en el *Makefile*.

El Código 4.36 muestra una secuencia de tres comandos en consola. Así, el primero de ellos sale de la carpeta *config_files_gen* hasta el directorio principal del proyecto. En el segundo comando, se ejecuta la simulación en “modo silencioso” del *test* por defecto *ccsds121_normal_test* con el *set* de estímulos *18_Test* (sin abrir la herramienta *QuestaSim*). Finalmente, el tercer comando simula, visualizando las formas de onda, el *set* de estímulos *05_Test* sobre el *test* *ccsds121_error_test*.

```
>> cd ..
>> make allc DATA=18_Test
>> make all DATA=05_Test UVM_TEST=ccsds121_error_test
```

Código 4.36. Ejecución de dos test sobre el entorno de verificación creado

Capítulo 5. RESULTADOS

En este capítulo se recogen los resultados que se han obtenido tras la ejecución de los cuatro *test* que se han contemplado para el entorno de verificación basado en UVM desarrollado. Los resultados pueden presentarse de dos formas:

- En modo texto por consola, en el que se podrán observar las salidas generadas por los diferentes componentes del entorno de verificación desarrollado haciendo uso del mecanismo de mensajes de UVM. El módulo IP121 y el resto de módulos que conforman la comunicación AHB interna, por su parte, también incluyen su propio sistema de mensajería por consola, de modo que se mostrarán ciertos mensajes relativos a, por ejemplo, la configuración recibida. Este modo de presentación de los resultados se utiliza siempre. En el caso de ejecutar la simulación “en modo silencioso”, estos mensajes se recibirán en el *Terminal* de ejecución, mientras que en la simulación mediante *QuestaSim*, se mostrarán en la ventana de comandos de dicha herramienta.
- En modo gráfico, en el que se visualizarán las formas de onda de las señales de entrada/salida del DUV en función del tiempo, pudiendo establecer una comparativa con los resultados de la simulación original. Esta visualización solamente es posible cuando se ejecuta la simulación mediante la herramienta *QuestaSim*.

Finalmente, este capítulo también incluirá un apartado dedicado al análisis de los tiempos de ejecución de los diferentes *test* obtenidos, pues se trata de un aspecto determinante a la hora de contemplar la aplicación real del entorno de verificación basado en UVM que ha sido implementado en este TFM.

5.1 SALIDA DE TEXTO POR CONSOLA

Por motivos de extensión, este apartado se centra en los mensajes recibidos por consola durante la simulación, omitiendo otros aspectos que no se consideran relevantes, como los mensajes de compilación, o la inclusión de las bibliotecas y los componentes necesarios previa a la simulación. Por otro lado, debe considerarse que los mensajes relativos al sistema de mensajería de UVM siguen la siguiente estructura:

1. Tipo de mensaje: UVM_INFO, UVM_WARNING, UVM_ERROR o UVM_FATAL.
2. Archivo y línea de código donde se encuentra la macro que envía el mensaje.

3. Tiempo de ejecución en el que se envía el mensaje.
4. Componente del entorno de verificación que envía el mensaje.
5. Contenido del mensaje.

5.1.1 TEST NORMAL

En este caso, el *test* utilizado sigue la configuración por defecto que se encuentra en el archivo *Makefile*, relativa a la ejecución de un *test* `ccsds121_normal_test` (dos compresiones consecutivas) utilizando el *set* de estímulos `25_Test`.

```
# UVM_INFO @ 0: reporter [RNTST] Running test ccsds121_normal_test...
# UVM_INFO ccsds121_env.sv(20) @ 0: uvm_test_top.env
[uvvm_test_top.env] Build stage completed.
# UVM_INFO ccsds121_agent.sv(18) @ 0: uvm_test_top.env.config_agent
[uvvm_test_top.env.config_agent] Build stage completed.
# UVM_INFO ccsds121_config_driver.sv(22) @ 0:
uvvm_test_top.env.config_agent.config_driver
[uvvm_test_top.env.config_agent.config_driver] Build stage completed.
# UVM_INFO ccsds121_config_monitor.sv(21) @ 0:
uvvm_test_top.env.config_agent.config_monitor
[uvvm_test_top.env.config_agent.config_monitor] Build stage
completed.
# UVM_INFO ccsds121_agent.sv(45) @ 0: uvm_test_top.env.data_agent
[uvvm_test_top.env.data_agent] Build stage completed.
# UVM_INFO ccsds121_data_driver.sv(19) @ 0:
uvvm_test_top.env.data_agent.data_driver
[uvvm_test_top.env.data_agent.data_driver] Build stage completed.
# UVM_INFO ccsds121_data_monitor.sv(24) @ 0:
uvvm_test_top.env.data_agent.data_monitor
[uvvm_test_top.env.data_agent.data_monitor] Build stage completed.
# UVM_INFO ccsds121_scoreboard.sv(30) @ 0: uvm_test_top.env.scoreboard
[uvvm_test_top.env.scoreboard] Build stage completed.
# UVM_INFO ccsds121_agent.sv(23) @ 0: uvm_test_top.env.config_agent
[uvvm_test_top.env.config_agent] Connect stage completed.
# UVM_INFO ccsds121_agent.sv(50) @ 0: uvm_test_top.env.data_agent
[uvvm_test_top.env.data_agent] Connect stage completed.
# UVM_INFO ccsds121_env.sv(25) @ 0: uvm_test_top.env
[uvvm_test_top.env] Connect stage completed.
# UVM_INFO ccsds121_test.sv(22) @ 0: uvm_test_top
[ccsds121_normal_test] Printing the test bench topology:
# -----
# Name                               Type                               Size  Value
# -----
# uvm_test_top                       ccsds121_normal_test              -     @467
#   env                               ccsds121_env                      -     @475
#     config_agent                   ccsds121_config_agent             -     @491
#       config_driver                 ccsds121_config_driver            -     @639
#         rsp_port                     uvm_analysis_port                 -     @656
#           seq_item_port              uvm_seq_item_pull_port            -     @647
#             config_monitor           ccsds121_config_monitor           -     @665
#               config_sequencer       ccsds121_config_sequencer         -     @516
#                 rsp_export           uvm_analysis_export               -     @524
#                   seq_item_export    uvm_seq_item_pull_imp              -     @630
```

```

# arbitration_queue array 0 -
# lock_queue array 0 -
# num_last_reqs integral 32 'd1
# num_last_rsps integral 32 'd1
# data_agent ccsds121_data_agent - @499
# data_driver ccsds121_data_driver - @806
# rsp_port uvm_analysis_port - @823
# seq_item_port uvm_seq_item_pull_port - @814
# data_monitor ccsds121_data_monitor - @832
# output_collected_port uvm_analysis_port - @842
# data_sequencer ccsds121_data_sequencer - @683
# rsp_export uvm_analysis_export - @691
# seq_item_export uvm_seq_item_pull_imp - @797
# arbitration_queue array 0 -
# lock_queue array 0 -
# num_last_reqs integral 32 'd1
# num_last_rsps integral 32 'd1
# scoreboard ccsds121_scoreboard - @507
# output_packets_collected uvm_tlm_analysis_fifo #(T) - @864
# analysis_export uvm_analysis_imp - @908
# get_ap uvm_analysis_port - @899
# get_peek_export uvm_get_peek_imp - @881
# put_ap uvm_analysis_port - @890
# put_export uvm_put_imp - @872
# -----
#
# ahbctrl: AHB arbiter/multiplexer rev 1
# ahbctrl: Common I/O area at 0xffff0000, 1 Mbyte
# ahbctrl: AHB masters: 1, AHB slaves: 1
# ahbctrl: Configuration area at 0xfffff000, 4 kbyte
# ** Warning: AHB slave 0 appears to be disabled, but the slave config
# record is not driven to zero (check vendor ID or drive unused bus
# index with appropriate values).
# Time: 2 ns Iteration: 0 Instance: /ccsds121_top/duv/ahbtbctrl
# UVM_INFO ccsds121_config_driver.sv(154) @ 1200:
# uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
# Sending a new configuration...
# Time: 2100ns Write[0x10000000]: 0x00000000
# Time: 2240ns Write[0x10000004]: 0x01006020
# Time: 2380ns Write[0x10000008]: 0x00018000
# Time: 2520ns Write[0x1000000c]: 0x00011X00
# Time: 2800ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(40) @ 4200:
# uvm_test_top.env.config_agent.config_monitor
# [ccsds121_config_monitor] AwaitingConfig lowered correctly when
# configuration was received
# UVM_INFO ccsds121_config_monitor.sv(46) @ 4200:
# uvm_test_top.env.config_agent.config_monitor
# [ccsds121_config_monitor] Ready correctly asserted when IP core is
# ready to receive new samples
# UVM_INFO ccsds121_config_monitor.sv(54) @ 73000:
# uvm_test_top.env.config_agent.config_monitor
# [ccsds121_config_monitor] Finished correctly activated when
# compression finished or stopped
# UVM_INFO ccsds121_config_monitor.sv(72) @ 73200:
# uvm_test_top.env.config_agent.config_monitor
# [ccsds121_config_monitor] AwaitingConfig correctly activated after
# Finished
# UVM_INFO ccsds121_config_driver.sv(190) @ 73400:
# uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
# Sending a new configuration...

```

```

# Time: 74060ns Write[0x10000000]: 0x00000000
# Time: 74340ns Write[0x10000000]: 0x00000000
# Time: 74480ns Write[0x10000004]: 0x01006020
# Time: 74620ns Write[0x10000008]: 0x00018000
# Time: 74760ns Write[0x1000000c]: 0x00011X00
# Time: 75040ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(81) @ 76600:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly lowered when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(86) @ 76600:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_monitor.sv(97) @ 145200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(104) @ 145400:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(105) @ 145400:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Test performed
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
  146700: reporter [TEST_DONE] 'run' phase is ready to proceed to the
  'extract' phase
# UVM_INFO ccsds121_scoreboard.sv(154) @ 146700:
  uvm_test_top.env.scoreboard [ccsds121_scoreboard] Test was run
  successfully.
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    26
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [RNTST]      1
# [TEST_DONE]  1
# [ccsds121_config_driver]      2
# [ccsds121_config_monitor]     9
# [ccsds121_normal_test]       1
# [ccsds121_scoreboard]        1
# [uvm_test_top.env]           2
# [uvm_test_top.env.config_agent]      2
# [uvm_test_top.env.config_agent.config_driver]      1
# [uvm_test_top.env.config_agent.config_monitor]     1
# [uvm_test_top.env.data_agent]      2
# [uvm_test_top.env.data_agent.data_driver]      1
# [uvm_test_top.env.data_agent.data_monitor]      1
# [uvm_test_top.env.scoreboard]      1
# ** Note: $finish :
# /home/soft/eucad/mentor/2015/questa_sv_af_10.4b_Linux/questasim/
# linux_x86_64/./verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
# Time: 146700 ns Iteration: 53 Instance: /ccsds121_top

```

Código 5.1. Mensajes durante la ejecución de la simulación del test ccsds121_normal_test

En primer lugar, tras la ejecución de la fase `build_phase`, todos los componentes del entorno de verificación emiten un mensaje informando de su finalización. Los componentes *UVM Sequencer* no realizan esta acción, debido a que no han redefinido la implementación de esta fase, sino que ejecutan el comportamiento de la misma en la clase `uvm_sequencer`. Seguidamente, al finalizar la fase `connect_phase`, los componentes *UVM Agent* indican que han interconectado sus componentes *UVM Sequencer* y *UVM Driver*, mientras el componente *UVM Environment* efectúa la misma acción sobre los componentes *UVM Monitor* de datos y *UVM Scoreboard*. La siguiente que se ejecuta es `end_of_elaboration_phase`, en la que el objeto *printer* generado en el componente *UVM Test* muestra la estructura del entorno de verificación que se ha creado. Todas las acciones indicadas se ejecutan en tiempo 0 ns.

A continuación comienza la fase `run_phase` en la que, en su inicio, el módulo *Decoder* de la comunicación AHB interna del DUV (`ahbctrl`) presenta su configuración. Tras ello, se muestran mensajes en diferentes instantes de tiempo, procedentes desde los componentes *UVM Driver* y *UVM Monitor* de configuración, y en los que se observa la correcta ejecución del proceso de configuración del IP121 (si no fuese así, se recibirían mensajes de error). También se deduce que las comprobaciones que se están realizando en el componente *UVM Scoreboard* son satisfactorias pues, en caso de existir errores, dicho componente emitiría mensajes alertando de la situación. En caso de incrementar el nivel de *verbosity* para, por ejemplo, llevar a cabo labores de depuración del código desarrollado, se mostraría un mayor número de mensajes provenientes de los diferentes componentes.

En la fase `report_phase`, el componente *UVM Scoreboard* realiza un recuento del número de errores resultantes de la comparación entre el flujo de datos de salida del DUV tras finalizar el proceso de compresión, y los valores de referencia. Si la comparación no ha sido exitosa, envía un mensaje con la cantidad de errores detectados, y en caso contrario, informa de que el *test* se ha pasado correctamente. Finalmente, se muestra un resumen con la cantidad de mensajes recibidos, clasificados según su tipo (macro que lo emite) y el componente que lo invoca.

5.1.2 TEST DE ERROR

Para esta simulación, se ejecuta el *test* `ccsds121_error_test` (envío de una configuración inválida y, tras la indicación de error por parte del IP121, envío de una configuración válida) utilizando el *set* de estímulos `05_Test`. El texto recibido por la consola para este caso se recoge en el Código 5.2. Tanto para este *test*, como para los restantes, únicamente se mostrará la salida de

texto en la consola desde el mecanismo de mensajes de UVM y para la fase `run_phase`, ya que los mensajes recibidos en el resto de fases son idénticos, independientemente del `test` ejecutado.

```
# UVM_INFO ccsds121_config_driver.sv(154) @ 1000:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 1960ns Write[0x10000000]: 0x00000000
# Time: 2100ns Write[0x10000004]: 0x0100080e
# Time: 2240ns Write[0x10000008]: 0x00010800
# Time: 2380ns Write[0x1000000c]: 0x00014X00
# Time: 2660ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(40) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig lowered correctly when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(61) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Error has been correctly asserted
# UVM_INFO ccsds121_config_monitor.sv(66) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished has been correctly asserted after
  an error
# UVM_INFO ccsds121_config_monitor.sv(72) @ 4400:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  Finished
# UVM_INFO ccsds121_config_driver.sv(190) @ 4600:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 5180ns Write[0x10000000]: 0x00000000
# Time: 5460ns Write[0x10000000]: 0x00000000
# Time: 5600ns Write[0x10000004]: 0x01000820
# Time: 5740ns Write[0x10000008]: 0x00010800
# Time: 5880ns Write[0x1000000c]: 0x00014X00
# Time: 6160ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(81) @ 7800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly lowered when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(86) @ 7800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_monitor.sv(97) @ 66800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(104) @ 67000:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(105) @ 67000:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Test performed
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
  68300: reporter [TEST_DONE] 'run' phase is ready to proceed to the
  'extract' phase
```



```
# UVM_INFO ccsds121_scoreboard.sv(154) @ 68300:
  uvm_test_top.env.scoreboard [ccsds121_scoreboard] Test was run
  successfully.
```

Código 5.2. Mensajes recibidos en la fase run_phase del test ccsds121_error_test

En esta simulación, es posible comprobar a partir de los mensajes mostrados que, tras la primera configuración errónea, se informa que la señal `Error` se activa correctamente. Tras ello, directamente se solicita una nueva configuración mediante la señal `AwaitingConfig`. Finalmente, al recibir la segunda configuración, que es válida, el proceso de compresión se ejecuta correctamente.

5.1.3 TEST DE FORCESTOP

En este caso, se hace uso del `test ccsds121_forcestop_test` (detención de la ejecución de una compresión mediante la señal `ForceStop`) utilizando el `set` de estímulos `04_Test`. El texto recibido por la consola durante la fase `run_phase` de esta simulación se muestra en el Código 5.3.

```
# UVM_INFO ccsds121_config_driver.sv(154) @ 1000:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 1960ns Write[0x10000000]: 0x00000000
# Time: 2100ns Write[0x10000004]: 0x01009018
# Time: 2240ns Write[0x10000008]: 0x00010400
# Time: 2380ns Write[0x1000000c]: 0x00014X80
# Time: 2660ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(40) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig lowered correctly when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(46) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_driver.sv(176) @ 24200:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  ForceStop assertion
# UVM_INFO ccsds121_config_monitor.sv(54) @ 24600:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished or stopped
# UVM_INFO ccsds121_config_monitor.sv(72) @ 24800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  Finished
# UVM_INFO ccsds121_config_driver.sv(190) @ 25000:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 25620ns Write[0x10000000]: 0x00000000
```

```

# Time: 25900ns Write[0x10000000]: 0x00000000
# Time: 26040ns Write[0x10000004]: 0x01009018
# Time: 26180ns Write[0x10000008]: 0x00010400
# Time: 26320ns Write[0x1000000c]: 0x00014X80
# Time: 26600ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(81) @ 28200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly lowered when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(86) @ 28200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_monitor.sv(97) @ 94800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(104) @ 95000:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(105) @ 95000:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Test performed
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
  96300: reporter [TEST_DONE] 'run' phase is ready to proceed to the
  'extract' phase
# UVM_INFO ccsds121_scoreboard.sv(154) @ 96300:
  uvm_test_top.env.scoreboard [ccsds121_scoreboard] Test was run
  successfully.

```

Código 5.3. Mensajes recibidos en la fase `run_phase` del test `ccsds121_forcestop_test`

En los mensajes emitidos por los componentes *UVM Driver* y *UVM Monitor* de configuración se observa que, durante el primer proceso de compresión se fuerza su detención mediante la señal `ForceStop`, tras lo cual la señal `Finished` se activa correctamente. Finalmente, el segundo proceso de compresión se ejecuta de forma satisfactoria.

5.1.4 TEST DE RECONFIGURACIÓN

En este caso, se ejecuta el *test* `ccsds121_reconfig_test` (intento de reconfigurar el IP121 mientras está realizando una compresión) utilizando el *set* de estímulos `20_Test`. El texto recibido por la consola durante la fase `run_phase` de esta simulación se muestra en el Código 5.4.

```

# UVM_INFO ccsds121_config_driver.sv(154) @ 1200:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 2100ns Write[0x10000000]: 0x00000000
# Time: 2240ns Write[0x10000004]: 0x01006020
# Time: 2380ns Write[0x10000008]: 0x00018000

```

```

# Time: 2520ns Write[0x1000000c]: 0x00013X00
# Time: 2800ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(40) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig lowered correctly when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(46) @ 4200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_driver.sv(165) @ 14200:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Reconfiguring the IP...
# ** Note: Attempt to send new configuration during compression
#   Time: 14840 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 14840 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 14980ns Write[0x10000000]: 0x00000000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 15260 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 15260 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 15400ns Write[0x10000000]: 0x00000000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 15540 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 15540 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 15680ns Write[0x10000004]: 0x01006020 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 15820 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 15820 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 15960ns Write[0x10000008]: 0x00018000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 16100 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 16100 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 16240ns Write[0x1000000c]: 0x00013X00 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 16520 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 16520 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 16660ns Write[0x10000000]: 0x00000001 [ERROR]
# UVM_INFO ccsds121_config_monitor.sv(54) @ 74200:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished or stopped
# UVM_INFO ccsds121_config_monitor.sv(72) @ 74400:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  Finished

```

```
# UVM_INFO ccsds121_config_driver.sv(190) @ 74600:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Sending a new configuration...
# Time: 75180ns Write[0x10000000]: 0x00000000
# Time: 75460ns Write[0x10000000]: 0x00000000
# Time: 75600ns Write[0x10000004]: 0x01006020
# Time: 75740ns Write[0x10000008]: 0x00018000
# Time: 75880ns Write[0x1000000c]: 0x00013X00
# Time: 76160ns Write[0x10000000]: 0x00000001
# UVM_INFO ccsds121_config_monitor.sv(81) @ 77800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly lowered when
  configuration was received
# UVM_INFO ccsds121_config_monitor.sv(86) @ 77800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Ready correctly asserted when IP core is
  ready to receive new samples
# UVM_INFO ccsds121_config_driver.sv(205) @ 87800:
  uvm_test_top.env.config_agent.config_driver [ccsds121_config_driver]
  Reconfiguring the IP...
# ** Note: Attempt to send new configuration during compression
#   Time: 88480 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 88480 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 88620ns Write[0x10000000]: 0x00000000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 88900 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 88900 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 89040ns Write[0x10000000]: 0x00000000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 89180 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 89180 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 89320ns Write[0x10000004]: 0x01006020 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 89460 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 89460 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 89600ns Write[0x10000008]: 0x00018000 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 89740 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 89740 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# Time: 89880ns Write[0x1000000c]: 0x00013X00 [ERROR]
# ** Note: Attempt to send new configuration during compression
#   Time: 90160 ns Iteration: 2 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
# ** Note: Attempt to send new configuration during compression
#   Time: 90160 ns Iteration: 6 Instance:
#     /ccsds121_top/duv/uut_shyloc/ahbslv
```

```

# Time: 90300ns Write[0x10000000]: 0x00000001 [ERROR]
# UVM_INFO ccsds121_config_monitor.sv(97) @ 147600:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Finished correctly activated when
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(104) @ 147800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] AwaitingConfig correctly activated after
  compression finished
# UVM_INFO ccsds121_config_monitor.sv(105) @ 147800:
  uvm_test_top.env.config_agent.config_monitor
  [ccsds121_config_monitor] Test performed
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @
  149100: reporter [TEST_DONE] 'run' phase is ready to proceed to the
  'extract' phase
# UVM_INFO ccsds121_scoreboard.sv(154) @ 149100:
  uvm_test_top.env.scoreboard [ccsds121_scoreboard] Test was run
  successfully.

```

Código 5.4. Mensajes recibidos en la fase run_phase del test ccsds121_reconfig_test

En esta simulación, los dos procesos de compresión ejecutados se llevan a cabo de forma satisfactoria. Sin embargo, es posible observar cómo se intenta enviar una configuración al IP121 mientras está realizando una compresión. Ante esta situación, el IP121 responde correctamente, denegando la configuración.

5.2 FORMAS DE ONDA

En la Figura 5.1 se muestran las formas de onda de las principales señales que conforman los puertos de entrada/salida del DUV. En este caso particular, se muestra el instante de la simulación en el que comienza la recepción del flujo de datos comprimidos de salida para la simulación descrita en el apartado 5.1.1 Test normal.

Observando las formas de onda a lo largo del tiempo de simulación para el *test bench* original y para la implementación basada en UVM, es posible comprobar que las señales de control, de configuración y de datos de entrada/salida coinciden. Además, esta técnica permite detectar algunos fallos de forma sencilla cuando se obtienen errores en la salida por consola, mediante la visualización de anomalías en las señales, sin necesidad de incluir demasiadas sentencias de depuración en el código.

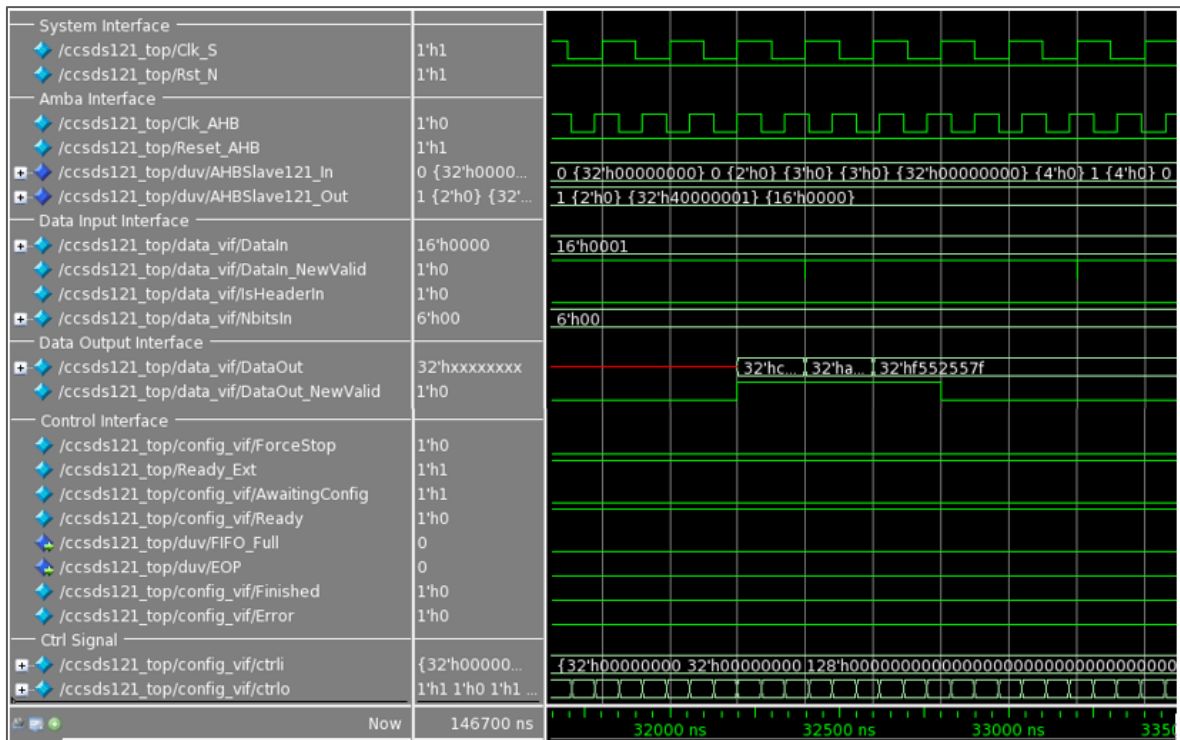


Figura 5.1. Formas de onda en el comienzo de la recepción del flujo de datos de salida

5.3 TIEMPO DE EJECUCIÓN

Atendiendo al tiempo de simulación, los *test* contemplados siguen un patrón temporal muy similar. En este caso, se tiene un tiempo de simulación promedio de aproximadamente 12 segundos. Por otra parte, el tiempo medio de simulación, haciendo uso del *test bench* original, es de 14-15 segundos. Por tanto, se obtiene una mejora relativa en el tiempo de simulación que, aunque para un único *test* no resulte significativa, sí que lo será en el caso de ejecutarse múltiples *test* de forma secuencial.

Sin embargo, el tiempo de ejecución total sigue estando determinado por el tiempo de compilación del IP121 para cada *test*, que se sitúa en torno a 18-20 segundos, y es independiente del entorno de verificación utilizado. Este proceso de compilación es estrictamente necesario debido a que determina la configuración fundamental del IP121, definiendo parámetros indispensables como, por ejemplo, el tamaño de los *buffers* de entrada y salida de datos.

Por lo tanto, además de haber comprobado la funcionalidad del entorno de verificación basado en UVM objetivo de este TFM, se obtiene también un mejor rendimiento asociado al tiempo global de ejecución de un *test*, en comparación con el *test bench* original.

Capítulo 6. CONCLUSIONES

En el procedimiento llevado a cabo en este TFM se han contextualizado y detallado los diferentes pasos llevados a cabo durante su realización. Llegado este punto, se expondrán las conclusiones generales a las que se llega a raíz de las tareas completadas a lo largo del procedimiento descrito, así como la propuesta de futuras ampliaciones a realizar sobre el TFM desarrollado.

6.1 CONCLUSIONES GENERALES

Una vez obtenidos los resultados de los diferentes *test* considerados para el entorno UVM desarrollado para la verificación funcional del IP121, resulta posible confirmar la validez de su implementación. Por tanto, se puede considerar que el objetivo principal del TFM ha sido satisfecho, logrando integrar y verificar este IP haciendo uso del entorno UVM desarrollado y de la adaptación multilenguaje implementada entre ambos.

Así, en primer lugar se ha llevado a cabo un estudio exhaustivo de la metodología UVM y del lenguaje *SystemVerilog*, a partir del cual se han adquirido los conceptos necesarios para poder crear y aplicar un entorno de verificación funcional basado en UVM. Asociado al estudio de la metodología, se realizó un proceso de familiarización con el uso de *Makefile*, así como con las herramientas *QuestaSim* y *MATLAB*, si bien es cierto ambas habían sido utilizadas en algunas ocasiones previas.

Además, tras un análisis en profundidad, se consiguió asimilar la descripción detallada del funcionamiento del IP121 y su *test bench* original, ambos carentes de documentación asociada. Se trata de un IP complejo y altamente configurable, por lo que la existencia de documentación explicativa tiene un valor fundamental a la hora de desarrollar ampliaciones futuras sobre la línea de trabajo iniciada, o bien para hacer uso del IP121 en nuevos proyectos.

Por último, se ha desarrollado el entorno de verificación basado en UVM en el que se fundamenta este TFM, tomando como referencia la implementación del *test bench* original. En este caso, se ha mantenido su funcionalidad y se han optimizado algunos de sus procesos a partir de las deficiencias identificadas a raíz de su análisis. Con ello, se obtiene un nuevo entorno de verificación más sencillo de interpretar y mejor estructurado, además de reutilizable, características inexistentes en el *test bench* original.

Finalizado el procedimiento llevado a cabo durante el desarrollo del presente TFM, y tras la experiencia adquirida con ello, se pueden establecer las siguientes conclusiones:

- El entorno de verificación desarrollado es estructurado, y con una jerarquía que separa la gestión de los protocolos de comunicación (asociada a los componentes *UVM Driver*, *UVM Monitor*, *UVM Agent*, etc.) de la declaración de la intencionalidad de cada *test* (recogida en los objetos *UVM Sequence* y el componente *UVM Test*).
- El entorno de verificación basado en UVM es totalmente reutilizable para la verificación de la funcionalidad de otros IP/DUV que dispongan de la misma interfaz de entrada/salida, siendo necesario modificar únicamente el propósito de cada *test*, es decir, las secuencias definidas, y utilizando los mismos UVC. En caso de que estas interfaces se modifiquen en número, bastaría con reutilizar los componentes definidos.
- El uso de un módulo *wrapper* hace que el entorno de verificación desarrollado pueda ser utilizado sobre cualquier otro IP/DUV, independientemente del lenguaje HDL utilizado para su codificación.
- El uso de la metodología UVM facilita significativamente la implementación de un Plan de Verificación, en el que se especifican todos los *test* a ejecutar para verificar la funcionalidad de un IP/DUV, y que representa en la actualidad un mecanismo obligado en las principales empresas de verificación. Cada *test* contemplado en este Plan de Verificación constituye una secuencia independiente. Además, y para aumentar su reusabilidad, UVM contempla la agrupación de estas secuencias en una biblioteca de secuencias, lo que facilita enormemente su uso.
- El uso de la metodología UVM simplifica la incorporación de métricas de cobertura orientadas a *Functional Verification* mediante la definición de *covergroups/coverpoints* en el entorno de verificación desarrollado.
- La documentación específica ofrecida por *Accellera Systems Initiative* y *Mentor Graphics Corporation* acerca de la metodología UVM, sus características, sus particularidades y el modo de utilizarla, se encuentra bastante detallada y resulta de elevada utilidad.

- La metodología UVM facilita la inclusión de nuevas funcionalidades en el entorno de verificación. Mientras para la definición de un nuevo *test* en el *test bench* original sería necesario implementar un nuevo procedimiento de configuración, incurriendo en una cantidad considerable de líneas de código, la inclusión de un nuevo *test* en el entorno basado en UVM es más sencilla, con leves modificaciones en el componente *UVM Driver* de configuración y, si fuese necesario, la creación de una nueva secuencia.

6.2 LÍNEAS FUTURAS

A continuación, se destacan las líneas futuras de trabajo más importantes que se proponen tras la realización del presente TFM:

- Inclusión de la capa RAL (*Register Abstraction Layer*) de UVM en el entorno de verificación desarrollado. Con este mecanismo es posible generar, de forma sencilla, transacciones de lectura y escritura sobre registros del DUV mediante el uso de un modelo externo en el que se mapean las posiciones de memoria útiles del DUV.
- Aplicación de métricas de cobertura que permitan valorar el grado de verificación alcanzado sobre el conjunto de propiedades y funcionalidades del DUV, con lo cual será posible determinar en cada momento la compleción, o no, de un Plan de Verificación.
- Extender el uso de la metodología UVM sobre el codificador descrito en VHDL que, en principio, se utiliza como módulo preprocesador para proveer de datos de entrada al IP121. De este modo, se podría llevar a cabo el proceso de verificación funcional mediante UVM para ambos, tanto de manera conjunta, como independientemente.

REFERENCIAS

- [1] J. O. Grady, *System Verification: Proving the Design Solution Satisfies the Requirements*. Elsevier Science, 2016.
- [2] A. Dasso and A. Funes, *Verification, Validation and Testing in Software Engineering*. Idea Group Pub., 2007.
- [3] H. Height, *A Practical Guide to Adopting the Universal Verification Methodology (UVM) Second Edition*. Lulu.com, 2013.
- [4] V. R. Cooper, *Getting Started with Uvm: A Beginner's Guide*. Verilab Publishing, 2013.
- [5] J. R. Montesdeoca Mateo, V. de Armas Sosa, and F. B. Tobajas Guerrero, "Creación de un entorno de verificación basado en UVM para sistemas digitales descritos en SystemC." 2016.
- [6] V. de Armas Sosa, F. B. Tobajas Guerrero, and Z. Perdomo Pérez, "Aplicación de UVM en el desarrollo de un entorno de verificación sobre plataforma Cadence," ULPGC, 2013.
- [7] S. Vasudevan, *Effective Functional Verification: Principles and Processes*. Springer US, 2006.
- [8] B. Bailey, *The Functional Verification of Electronic Systems: An Overview from Various Points of View*. International Engineering Consortium, 2005.
- [9] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*. Springer US, 2012.
- [10] Wilson Research Group, "Wilson Research Group ASIC/IC and FPGA Functional Verification Study," 2016. [Online]. Available: <https://www.mentor.com/products/fv/multimedia/the-2016-wilson-research-group-asic-ic-and-fpga-functional-verification-study>. [Accessed: 27-Dec-2017].
- [11] P. Aggarwal, "Introduction To System Verilog," 2010. [Online]. Available: <http://www.asicguru.com/system-verilog/tutorial/introduction/1/>. [Accessed: 28-Dec-2017].
- [12] G. Allan, M. Baird, R. Edelman, A. Erickson, M. Horn, M. Peryer, A. Rose, and K. Schwartz, "UVM/OVM Cookbook." 2012.

- [13] www.verificationguide.com, “UVM Tutorial.” [Online]. Available: <http://www.verificationguide.com/p/uvm-introduction.html>. [Accessed: 28-Dec-2017].
- [14] Accellera Systems Initiative Inc., “Universal Verification Methodology (UVM) 1.2 Class Reference.” 2014.
- [15] Accellera Systems Initiative Inc., “Universal Verification Methodology (UVM) 1.2 User’s Guide.” 2015.
- [16] European Space Agency, “ESA HDL IP Cores Portfolio Overview,” *ESA IP Cores*, 2016. [Online]. Available: http://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/ESA_HDL_IP_Cores_Portfolio_Overview. [Accessed: 05-Jan-2017].
- [17] The Consultative Committee for Space Data Systems, “Recommended Standard CCSDS 121.0-B-2.” 2012.
- [18] L. Santos, A. Gómez, R. Sarmiento, L. Fossati, and D. Merodio, “Architectural design and implementation of CCSDS-123 and CCSDS-121 IP cores for lossless satellite data compression,” in *On-Board Payload Data Compression Workshop*, 2016.
- [19] Cobham Gaisler AB, “GRLIB IP Library User’s Manual.” .
- [20] The MathWorks Inc., “Generate VHDL and Verilog code for FPGA and ASIC designs.” [Online]. Available: <https://www.mathworks.com/products/hdl-coder.html>. [Accessed: 15-Jan-2018].
- [21] Mentor Graphics Corporation, “Questa SIM User’s Manual.” 2011.

PLIEGO DE CONDICIONES

PLIEGO DE CONDICIONES

El procedimiento desarrollado, así como los resultados obtenidos a lo largo del presente TFM, son válidos para los modelos del hardware y las versiones del software que se indican en la Tabla PL.1 y en la Tabla PL.2, respectivamente.

RECURSOS HARDWARE

Tabla PL.1. Condiciones hardware

Equipo	Modelo
Estación de trabajo	Servidor Sun Fire X2200 M2

RECURSOS SOFTWARE

Tabla PL.2. Condiciones software

Aplicación	Versión
Sistema Operativo	Red Hat Enterprise Linux Server
Python	Python v2.7
Entorno de simulación	QuestaSim 10.4b
Biblioteca GRLIB IP	GRLIB IP 1.5.0-b4164
UVM	UVM 1.1d

PRESUPUESTO

PRESUPUESTO

Durante la realización del presente TFM ha sido necesario utilizar diferentes recursos. En este apartado se recogen todos los costes asociados a los recursos materiales y humanos implicados en el proyecto.

La cuantía del trabajo elaborado se ha fijado según las indicaciones de contrataciones de la Universidad de Las Palmas de Gran Canaria. De este modo, el total del presupuesto se ha desglosado en las siguientes secciones, en las que se representan los distintos costes según su naturaleza.

- Coste de Recursos Humanos
- Coste de Recursos Hardware
- Coste de Recursos Software
- Coste de Material Fungible
- Coste Total

RECURSOS HUMANOS

Para determinar el coste asociado a los recursos humanos (RRHH), se hace uso de las últimas tablas de honorarios publicadas el 4 de noviembre de 2010 por la Universidad de Las Palmas de Gran Canaria. Según estas tablas, el coste total mensual de un Licenciado, Arquitecto o Ingeniero, con una dedicación de 20 horas semanales, asciende a 1.032,63€. Por lo tanto, se tiene que el coste aproximado asociado a una hora de trabajo será de 12,91€.

Para la realización del presente TFM, el número de horas invertidas se corresponde con el tiempo asociado a 12 créditos ECTS, es decir, 300 horas. Por lo tanto, el coste de recursos humanos será:

$$\text{Honorarios (coste RRHH)} = 12,91\text{€/h} \cdot 300\text{h} = \mathbf{3.873,00\text{€}}$$

RECURSOS HARDWARE

La Tabla P.1 especifica los recursos hardware amortizables que han sido utilizados a lo largo del proyecto y sus costes asociados en función del número de usuarios, y de su tiempo de uso con respecto al periodo de amortización, que se ha establecido en 3 años (36 meses).

Tabla P.1. Coste de recursos hardware

Recurso	Coste	Número de usuarios	Tiempo de uso	Amortización
Ordenador portátil Medion Akoya P66 MD99173	770,00€	1	5 meses	106,94€
Servidor Sun Fire X2200 M2	23.075,00€	150	5 meses	21,37€
TOTAL				128,31€

RECURSOS SOFTWARE

De igual modo, los recursos software que han sido necesarios durante el desarrollo del proyecto y sus costes derivados se reflejan en la Tabla P.2, donde la herramienta *QuestaSim* tiene una vida útil de 5 años (60 meses) y un tiempo de uso de 5 meses.

Tabla P.2. Coste de recursos software

Recurso	Tipo de licencia	Valor de adquisición	Coste
Python v2.7	Pública	-	-
QuestaSim 10.4b	Pago	20.000,00€	1.666,67€
Gedit v3.22.0	Pública	-	-
Notepad++ v7.5.4	Pública	-	-
Mendeley Desktop	Pública	-	-
Microsoft Office 2016	Universitaria	79,00€	79,00€
TOTAL			1745,67€

MATERIAL FUNGIBLE

Los costes relacionados con el material fungible utilizado en la realización de este TFM se detallan en la Tabla P.3.

Tabla P.3. Coste de material fungible

Recurso	Coste
Impresión y encuadernación	20,00€
3 x CD-ROM	2,10€
TOTAL	22,10€

COSTE TOTAL DEL PROYECTO

Así, en la Tabla P.4 se refleja la suma de los costes originados de los distintos recursos utilizados, además de la aplicación del Impuesto General Indirecto Canario (IGIC), que es del 7%, dando lugar al coste total del proyecto.

Tabla P.4. Coste total

Recursos	Coste
Recursos humanos	3.873,00€
Recursos hardware	128,31€
Recursos software	1.745,67€
Material fungible	22,10€
SUBTOTAL	5.769,08€
IGIC (7%)	403,84€
TOTAL	6.172,92€

El presupuesto total del Trabajo Fin de Máster *“Entorno UVM para la verificación funcional de un IP multi-interfaz orientado a la compresión de imágenes”* asciende a la cantidad de *seis mil ciento setenta y dos euros con noventa y dos céntimos* (6.172,92 euros).

Las Palmas de Gran Canaria, a 2 de febrero de 2018

Fdo.: D. Samuel Rodríguez Rodríguez

ANEXOS

ANEXO – CONTENIDO DEL CD-ROM

En este Anexo se presenta el contenido del CD-ROM adjunto a este documento, que se corresponde con la siguiente estructura:

- Memoria del TFM “*Entorno UVM para la verificación funcional de un IP multi-interfaz orientado a la compresión de imágenes*” en formato PDF.
- Directorio “*COMPRESION_UVM*” en el que se encuentra el entorno de verificación basado en UVM desarrollado. Este directorio sigue la estructura detallada en el apartado 4.6 Estructura de directorios del documento correspondiente a la Memoria.