ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO DE FIN DE MÁSTER

Diseño de un *testbench* UVM para la verificación de un procesador RISC-V

Titulación: Máster Universitario en Ingeniería de Telecomunicación

Autor: **D. Francisco Duque El Ayachi**

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

Fecha: Julio 2024

AGRADECIMIENTOS

Quisiera expresar mi más sincero agradecimiento a todas las personas y entidades que han hecho posible la realización de este Trabajo de Fin de Máster.

En primer lugar, agradezco a mis tutores, por su constante apoyo, orientación y valiosos consejos a lo largo de mi carrera como estudiante. Un agradecimiento especial a mis tutores de TFM, Valentín y Félix. Por su dedicación, paciencia y por supuesto su afán de hacer llegar sus conocimientos a mí de la mejor manera, no sólo con el fin de realizar el presente TFM sino para formarme y ser capaz de afrontar con mayor confianza mi futuro como profesional.

A mis compañeros y amigos, quienes han brindado su apoyo moral y han compartido este viaje conmigo, muchas gracias por su camaradería y estímulo. A mi familia, por su amor incondicional y comprensión durante este periodo de intenso trabajo, siendo el pilar fundamental de mi vida, a quien incluyo como familia a mi pareja, que me ha brindado las fuerzas y motivación para lograr mis objetivos, gracias de corazón.

Este trabajo no habría sido posible sin el apoyo y la colaboración de todas las personas mencionadas.

ABSTRACT

Verification of integrated circuits is a crucial process in hardware design, as it ensures that systems function correctly before fabrication. In this context, the Universal Verification Methodology (UVM) has become an essential tool. UVM is an extension of SystemVerilog that provides a structured framework for developing reusable and scalable verification environments. The importance of UVM lies in its ability to reduce development time and improve design quality by automating common tasks and promoting good verification practices.

The need for UVM arises from the increasing complexity of modern digital systems. As designs become more sophisticated, manual verification becomes impractical. UVM addresses this challenge by offering a set of standard libraries and components that facilitate the creation of robust and efficient verification environments. Additionally, UVM enables the integration of advanced verification techniques, such as random test generation and coverage-driven verification, significantly improving error detection.

The choice of the RISC-V architecture for this work is due to several reasons. RISC-V is an open and free instruction set architecture (ISA) that has gained popularity for its flexibility and extensibility. Unlike proprietary architectures, RISC-V allows designers to modify and extend the ISA according to their specific needs, making it ideal for research and development in both academic and professional settings. Furthermore, the growing adoption of RISC-V in the industry highlights the relevance of having solid verification environments for processors based on this architecture.

The work conducted focuses on the implementation of a UVM-based verification environment for a processor with a RISC-V architecture. Initially, an exhaustive study of SystemVerilog and UVM was carried out to understand the theoretical and practical foundations of verification. During this phase, a priority queue module (PIFO) was used as a case study to develop basic verification environments and familiarize with the tools and methodologies.

Subsequently, a deep analysis of the RISC-V architecture and the specific processor to be verified was performed. This analysis included the review of the RISC-V specification, the identification of the main functional blocks of the processor, and the definition of the verification objectives. With this information, the design and development of the UVM verification environment were undertaken.

RESUMEN

La verificación de circuitos integrados es un proceso crucial en el diseño de hardware, ya que garantiza que los sistemas funcionen correctamente antes de su fabricación. En este contexto, la Metodología de Verificación Universal (UVM) se ha convertido en una herramienta esencial. UVM es una extensión de *SystemVerilog* que proporciona un marco estructurado para desarrollar entornos de verificación reusables y escalables. La importancia de UVM radica en su capacidad para reducir el tiempo de desarrollo y mejorar la calidad del diseño mediante la automatización de tareas comunes y la promoción de buenas prácticas en verificación.

La necesidad de UVM se deriva de la complejidad creciente de los sistemas digitales modernos. A medida que los diseños se vuelven más sofisticados, la verificación manual se vuelve impracticable. UVM aborda este desafío al ofrecer un conjunto de bibliotecas y componentes estándar que facilitan la creación de entornos de verificación robustos y eficientes. Además, UVM permite la integración de técnicas de verificación avanzadas, como la generación de tests aleatorios y la verificación dirigida por cobertura, lo que mejora significativamente la detección de errores.

La elección de la arquitectura RISC-V para este trabajo responde a varias razones. RISC-V es una arquitectura de conjunto de instrucciones (ISA) abierta y libre, que ha ganado popularidad por su flexibilidad y extensibilidad. A diferencia de las arquitecturas propietarias, RISC-V permite a los diseñadores modificar y extender la ISA según sus necesidades específicas, lo que la hace ideal para la investigación y el desarrollo en el ámbito académico y profesional. Además, la creciente adopción de RISC-V en la industria subraya la relevancia de contar con entornos de verificación sólidos para procesadores basados en esta arquitectura.

El trabajo realizado se centra en la implementación de un entorno de verificación basado en UVM para un procesador con arquitectura RISC-V. Inicialmente, se llevó a cabo un estudio exhaustivo de *SystemVerilog* y UVM para comprender las bases teóricas y prácticas de la verificación. Durante esta fase, se utilizó un módulo de cola prioritaria (PIFO) como caso de estudio para desarrollar entornos de verificación básicos y familiarizarse con las herramientas y metodologías. Posteriormente, se realizó un análisis profundo de la arquitectura RISC-V y del procesador específico a verificar. Este análisis incluyó la revisión de la especificación de RISC-V, la identificación de los principales bloques funcionales del procesador y la definición de los objetivos de verificación. Con esta información, se procedió a diseñar y desarrollar el entorno de verificación UVM.

Tabla de contenido

CAPÍTULO 1. INTRODUCCIÓN Y ANTECEDENTES 19

1.1. ANTECEDENTES 19

1.1.1. Universal Verification Methodology (UVM) 20

1.1.2. Reduced Instruction Set Computer V (RISC-V) 22

1.2. OBJETIVOS 25

1.3. PETICIONARIO 25

CAPÍTULO 2. UNIVERSAL VERIFICATION METHODOLOGY 27

2.1. VERIFICACIÓN DE SISTEMAS ELECTRÓNICOS DIGITALES 27

2.2. DESCRIPCIÓN Y VERIFICACIÓN HARDWARE *SYSTEMVERILOG* 29

2.3. EJEMPLO DE ENTORNO DE VERIFICACIÓN EN *SYSTEMVERILOG* 33

2.3.1. Objeto Transaction 35

2.3.2. Componente Generator 37

2.3.3. Interfaces en SystemVerilog 39

2.3.4. Componente Driver 40

2.3.5. Componente Monitor 43

2.3.6. Componente Scoreboard 44

2.3.7. Componente Agent 50

2.3.8. Componente Environment 52

2.3.9. Programa Test y módulo TestBench Top 53

2.4. CONCEPTOS GENERALES DE LA METODOLOGÍA UVM 55

2.4.1. Biblioteca de clases UVM 55

2.4.2. Mecanismo de fases en UVM 57

2.4.3. Mecanismo de Factory 59

2.4.4. Mecanismo de mensajes 60

2.4.5. Modelado y comunicación TLM 60

2.5. ENTORNO DE VERIFICACIÓN UVM. 62

2.5.1. Transacciones 63

2.5.2. Interfaz Virtual 64

2.5.3. Componente UVM Sequencer 65

2.5.4. Componente UVM Driver 66

2.5.5. Componente UVM Monitor 68

2.5.6. UVM Scoreboard 71

2.5.7. Componente UVM Agent 75

2.5.8. UVM Environment 77

2.5.9. UVM Test 79

2.5.10. UVM Testbench *83*

2.5.11. Secuencias UVM *85*

2.5.12. Ejecución del testbench UVM sobre el módulo PIFO *88*

CAPÍTULO 3. ARQUITECTURA RISC-V 93

3.1. VARIANTES DE RISC-V 94

3.2. CARACTERÍSTICAS TÉCNICAS DE RISC-V 95

3.3. FORMATO DE INSTRUCCIONES RISC-V 96

3.3.1. Computación Entera *97*

3.3.2. Load y Store *97*

3.3.3. Saltos Condicionales *97*

3.3.4. Saltos Incondicionales *99*

3.3.5. Otras instrucciones *99*

3.4. EJECUCIÓN DE INSTRUCCIONES 99

3.5. REGISTROS 106

3.6. MODOS DE DIRECCIONAMIENTO 107

3.7. MODOS PRIVILEGIADOS DEL PROCESADOR 107

3.7.1. Modo Máquina *108*

3.7.2. Modo usuario *111*

3.7.3. Modo supervisor *112*

3.8. MEMORIA VIRTUAL 113

3.9. COMPARACIÓN DE RISC-V CON OTRAS ARQUITECTURAS 115

3.10. CONCLUSIONES 116

CAPÍTULO 4. PROCESADOR RISC-V A VERIFICAR 117

4.1. DESCRIPCIÓN DEL PROCESADOR 117

4.2. *TESTBENCH* ORIGINAL DEL PROCESADOR RISC-V 120

4.3. EJEMPLO DE TEST DEL PROCESADOR RISC-V 122

4.4. SIMULACIÓN DEL TEST EN QUESTASIM 126

4.4.1. Ejecución de un test completo en el procesador RISC-V *130*

CAPÍTULO 5. CREACIÓN DEL ENTORNO VERIFICACIÓN UVM PARA RISC-V 135

5.1. ADAPTACIÓN PREVIA DEL DISEÑO 135

5.2. ESTRUCTURA DEL ENTORNO DE VERIFICACIÓN DEL RISC-V EN UVM 137

5.2.1. Transacción de instrucción y Transacción de datos *138*

5.2.2. Interfaces Virtuales *140*

5.2.3. Componente Monitor *141*

5.2.4. Componente Agent *143*

5.2.5. Componente Subscriber *144*

5.2.6. Componente Environment *152*
5.2.7. Módulo TOP *153*
5.2.8. Componente Test *156*
5.3. RESULTADOS DEL TEST COBERTURA FIBONACCI 158
5.4. RESULTADOS TEST DE COBERTURA SIMPLE 162

CAPÍTULO 6. CONCLUSIONES Y TRABAJOS FUTUROS 165

6.1. CONCLUSIONES 165
6.2. LÍNEAS FUTURAS 167

BIBLIOGRAFÍA 169

PLIEGO DE CONDICIONES 175

PRESUPUESTO 179

6.3. RECURSOS HUMANOS 179
6.4. RECURSOS MATERIALES 180
6.5. RECURSOS *HARDWARE* 181
6.6. RECURSOS *SOFTWARE* 181
6.7. MATERIAL FUNGIBLE 182
6.8. REDACCIÓN DE LA MEMORIA 182
6.9. DERECHOS DE VISADO DEL COIT 183
6.10. GASTOS DE TRAMITACIÓN Y ENVÍO 183
6.11. COSTE TOTAL DEL PROYECTO 183

Índice de figuras

Figura 1 Porcentaje del tiempo del proyecto ASIC dedicado a la verificación.................... 20
Figura 2 Metodología ASIC y tendencia de adopción de bibliotecas de clases.................. 21
Figura 3 Tendencias de adopción de los lenguajes de verificación y descripción Hardware 22
Figura 4. Publicaciones Anuales relacionadas con RISC-V en Google Academic y IEE Xplore 24
Figura 5 Modelos de verificación 28
Figura 6 Funcionalidades de *SystemVerilog* [13]... 29
Figura 7 Capas *Signal* y *Command*... 31
Figura 8 *Testbench* con capa funcional añadida.. 31
Figura 9 *Testbench* con capa *scenario* añadida 32
Figura 10 Entorno de test basado en componentes SystemVerilog [14]........................... 33
Figura 11 Interfaz de E/S del módulo PIFO... 33
Figura 12 Entorno Verificación *SytemVerilog* PIFO 35
Figura 13 Conexión de dos componentes mediante `mailbox`.. 38
Figura 14 Mecanismo de inserción mediante método `push_front`.............................. 45
Figura 15 Mecanismo de inserción mediante método `Insert`....................................... 47
Figura 16 Mecanismo de inserción mediante método `push_back`................................ 47
Figura 17 Mecanismo de inserción natural de un dato.. 48
Figura 18 Jerarquía de la biblioteca de clases UVM... 57
Figura 19 Fases UVM y orden ejecución 58
Figura 20 TLM modo *Push* 61
Figura 21 TLM modo *Pull*... 61
Figura 22 TLM modo FIFO... 61
Figura 23 TLM modo *Broadcast*.. 62
Figura 24 Modelo básico de un entorno de verificación UVM.. 62
Figura 25 In-order scoreboard UVM. 71
Figura 26 Out-of-order scoreboard UVM 72
Figura 27 Diferencia entre UVM Agent activo y pasivo... 75
Figura 28 Fase de ejecución del test `t_r00`.. 89
Figura 29 Fase de ejecución del test `t_i04`.. 91
Figura 30 Fase de ejecución del test `t_r02`.. 92
Figura 31 Resultados ejecución test t_r02 mecanismo mensajes UVM 92
Figura 32 Formato de instrucciones [21]... 96
Figura 33. Mapa de opcodes de RV32I [21] 98
Figura 34. Etapa de Fetch del caso práctico addi x1, x0, 32.. 101
Figura 35. Etapa de DECODE 102
Figura 36. Etapa de ALU (Execute) 103
Figura 37. Etapa de escritura en memoria (Mem/Reg) y PC... 104
Figura 38. Fetch de la siguiente instrucción 105
Figura 39. Causas de excepciones e interrupciones en RISC-V [21]................................ 108
Figura 41. Estructura del registro mstatus del nivel de máquina 109
Figura 42. Estructura registros mie y mip... 110
Figura 43. Estructura del registro mcause.. 110
Figura 44. Estructura del registro mtvec 110

Figura 44. Estructura de configuraciones de PMP en los CSRs pmpcfg. 111
Figura 45. Estructura del registro sstatus del nivel de supervisor 113
Figura 47. Una entrada de la tabla de páginas (PTE) de RV32 Sv32 113
Figura 47. CSR satp 114
Figura 48 Diagrama del proceso de traducción de direcciones Sv32 115
Figura 49 Diagrama bloques del diseño RISC-V a verificar 118
Figura 50 Interfaz de E/S del procesador RISC-V original 119
Figura 51 Codificación de la instrucción `add x4, x5, x6` 126
Figura 52 Resultados de simulación para la etapa *Fetch* 127
Figura 53 Resultados de simulación para la etapa *Execute* 128
Figura 54 Resultados de simulación de la etapa escritura de resultados 129
Figura 55 Esquema de funcionamiento de la unidad *forwarding* RISC-V a verificar 129
Figura 56 Simulación de la ejecución unidad forwarding 130
Figura 57 Resultado de la simulación del test serie Fibonacci - Inicialización de los registros 131
Figura 58 Resultado de la simulación del test serie Fibonacci - Actualización de los registros 132
Figura 59 Resultado de la simulación test serie Fibonacci - Cálculo del décimo número 133
Figura 60 Resultado de la simulación del test serie Fibonacci - Almacenamiento en memoria de datos 133
Figura 61 Adaptación de la interfaz E/S del módulo CPU 137
Figura 62 Estructura del entorno UVM creado para la verificación del procesador RISC-V 138
Figura 63 Porcentaje de cobertura de cada *covergroup* del Test Fibonacci 158
Figura 64 Resultados de cobertura para instrucciones Tipo-R - test Fibonacci 159
Figura 65 Resultados de cobertura para instrucciones Tipo-I - test Fibonacci 160
Figura 66 Resultados de cobertura para instrucciones Tipo-S - test Fibonacci 161
Figura 67 Resultados de cobertura para instrucciones Tipo-B - test Fibonacci 161
Figura 68 Resultados de cobertura para el test de cobertura simple............................. 162

Índice de tablas

Tabla 1 Descripción señales E/S PIFO 34
Tabla 2 Batería tests para verificación módulo PIFO 87
Tabla 3 Secuencias para verificación módulo PIFO 88
Tabla 4 Características técnicas RISC-V [20] 95
Tabla 5. Convención de registros en RISC-V [23] 106
Tabla 6 Niveles de privilegio de RISC-V 108
Tabla 7 Descripción señales E/S RISC-V original 120
Tabla 8 Condiciones Hardware 175
Tabla 9 Condiciones Software 175
Tabla 10 Factor de corrección según horas trabajadas 180
Tabla 11 Coste total de los recursos hardware 181
Tabla 12 Amortización de los recursos Software 181
Tabla 13 Coste total del Trabajo Fin de Máster 184

Índice de Códigos

Código 1 Transaction_in del módulo PIFO en *SystemVerilog* 36
Código 2 Transaction_out del módulo PIFO en *SystemVerilog* 37
Código 3 Componente Generator_in del módulo PIFO en *SystemVerilog* 37
Código 4 Interfaz Interface_in del módulo PIFO en *SystemVerilog* 39
Código 5 Interfaz Interface_out del módulo PIFO en *SystemVerilog* 39
Código 6 Componente Driver_in del módulo PIFO en *SystemVerilog* 40
Código 7 Driver_out del módulo PIFO en *SystemVerilog* 42
Código 8 Componente Monitor_in del módulo PIFO en *SystemVerilog* 43
Código 9 Scoreboard-Instancia PIFO *SystemVerilog* 44
Código 10 Componente *Scoreboard*: Tarea insert() – 1er caso 45
Código 11 Componente *Scoreboard*: Tarea insert() – 2do caso 46
Código 12 Componente *Scoreboard*: Tarea insert() – 3er caso 48
Código 13 Componente *Scoreboard*: Tarea insert() – 4o caso 48
Código 14 Componente *Scoreboard*: Tarea remove() 49
Código 15 Componente *Scoreboard*: Tarea main() 50
Código 16 Componente Agent_in del módulo PIFO en *SystemVerilog* 51
Código 17 Agent_in - Tareas PIFO *SystemVerilog*. 51
Código 18 Componente *environment* del módulo PIFO en *SystemVerilog* 52
Código 19 Program Test del módulo PIFO en *SystemVerilog* 53
Código 20 Componente testbench del módulo PIFO en *SystemVerilog*. 54
Código 21 Testbench UVM PIFO: transacción interfaz *insert* 64
Código 22 Testbench UVM PIFO: transacción interfaz *remove* 64
Código 23 Testbench UVM PIFO: Interfaz *insert* 65
Código 24 Testbench UVM PIFO: Componente UVM Sequencer interfaz *insert* 65
Código 25 Testbench UVM PIFO: Componente UVM Sequencer interfaz remove 66
Código 26 Testbench UVM PIFO: Componente UVM Driver - Parte I 66
Código 27 Testbench UVM PIFO: Componente UVM Driver - Parte II 68
Código 28 Testbench UVM PIFO: Componente UVM Monitor interfaz *insert* - Parte I 69
Código 29 Testbench UVM PIFO: Componente UVM Monitor interfaz *insert* - Parte II 70
Código 30 Testbench UVM PIFO: Componente UVM Scoreboard parte I 73
Código 31 Testbench UVM PIFO: Componente UVM Scoreboard parte II 74
Código 32 Testbench UVM PIFO: Componente UVM Scoreboard parte III 74
Código 33 Testbench UVM PIFO: Componente UVM Agent interfaz insert 76
Código 34 Testbench UVM PIFO: Componente UVM Environment interfaz insert 78
Código 35 Testbench UVM PIFO: Componente UVM Environment 79
Código 36 Testbench UVM PIFO: Componente UVM Test – base_test 81
Código 37 Testbench UVM PIFO: Componente UVM Test – t_r01 82
Código 38 Testbench UVM PIFO: Testbench Top parte I 83
Código 39 Testbench UVM PIFO: Testbench Top parte II 84
Código 40 Testbench UVM PIFO: Secuencia insert_data_min_prio 86
Código 41 Testbench UVM PIFO: Secuencia insert_into_full 86
Código 42 Testbench UVM PIFO: Secuencia remove_into_empty 87
Código 43 Fase de ejecución del test t_r00 89
Código 44 Fase de ejecución del test t_i04 90

Código 45 Fase de ejecución del test t_r02 91
Código 46 Módulo top del *testbench* original 121
Código 47 Inicialización y carga de instrucciones del *testbench* original 122
Código 48 Código ensamblador para el cálculo de un determinado número de la serie Fibonacci 124
Código 49 Código hexadecimal correspondiente al programa Fibonacci descrito en ensamblador 125
Código 50 Test básico con dependencia de datos 126
Código 51 Extracto módulo ALU.v 128
Código 52 Adaptación de la Interfaz E/S módulo CPU 136
Código 53 Transacción inst_packet 139
Código 54 Transacción data_packet 140
Código 55 Interfaz de instrucciones inst_if 141
Código 56 Interfaz de datos data_if 141
Código 57 Componente *Monitor* - inst_monitor 142
Código 58 Componente *Monitor* - data_monitor 143
Código 59 Componente Agent - inst_agent 144
Código 60 Componente UVM Subscriber 145
Código 61 Componente *Subscriber* - covergroup para instrucciones Tipo-R 147
Código 62 Componente *Subscriber* - covergroup para instrucciones Tipo-I 149
Código 63 Componente *Subscriber* - covergroup para instrucciones Tipo-S 150
Código 64 Componente Subscriber - covergroup instrucciones tipo-B 151
Código 65 Componente Subscriber - Función write() 152
Código 66 Componente Environment 153
Código 67 Primera parte del módulo riscv_top 154
Código 68 Segunda parte del módulo riscv_top 155
Código 69 Tercera parte del módulo riscv_top 156
Código 70 Componente Test 157

Acrónimos

ABI	*Application Binary Interface*
ABV	*Assertion-Based Verification*
ALU	*Arithmetic Logic Unit*
ARM	*Advanced RISC Machine*
ASIC	*Application-Specific Integrated Circuit*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DPI	*Direct Programming Interface*
DUV	*Design Under Verification*
EITE	Escuela de Ingeniería de Telecomunicación y Electrónica
EPI	*European Processor Initiative*
FIFO	*First in, first out*
FPGA	*Field Programable Gate Array*
HDL	*Hardware Description Language*
HVL	*Hardware Verification Language*
IP	*Intelectual Property*
ISA	*Instruction Set Architecture*
MUIT	Máster Universitario en Ingeniería de Telecomunicación
OOP	*Object Oriented Programming*
PC	*Program Counter*
PIFO	*Push In First Out*
RISC-V	*Reduced Instruction Set Computer*
RTL	*Register Transfer Level*
TFM	Trabajo Fin de Máster
TLM	*Transaction Level Modeling*
ULPGC	Universidad Las Palmas de Gran Canarias
UVM	*Universal Verification Methodology*
VIP	*Verification IP*

Capítulo 1. Introducción y antecedentes

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

1.1. Antecedentes

En este apartado se exponen distintos aspectos que presentan el problema y definen el contexto del desarrollo del Trabajo Fin de Máster.

En el diseño de sistemas digitales, la verificación es una etapa crucial para asegurar que el hardware funcione correctamente y cumpla con las especificaciones deseadas. Con la creciente complejidad de los diseños modernos, se han desarrollado diversas metodologías y estándares para facilitar y sistematizar este proceso. Uno de los estándares más adoptados en la industria es el Universal Verification Methodology (UVM), especialmente útil para la verificación de diseños complejos como es el procesador RISC-V con el que se plantea trabajar en este TFM.

RISC-V es una arquitectura de conjunto de instrucciones (ISA) abierta y libre que ha ganado popularidad debido a su flexibilidad, simplicidad y extensibilidad. A diferencia de otras arquitecturas propietarias como x86 de Intel o ARM, RISC-V permite a los diseñadores personalizar y optimizar sus procesadores para aplicaciones específicas sin necesidad de licencias costosas. Esta libertad para modificar y extender la ISA es una de las principales ventajas de RISC-V frente a sus

competidores. El código abierto de RISC-V es proporcionado principalmente por la Fundación RISC-V (RISC-V International), una organización sin fines de lucro que administra la arquitectura RISC-V y promueve su adopción. La fundación proporciona especificaciones y documentación detallada sobre la arquitectura RISC-V previamente verificada, y facilita la colaboración entre sus miembros, que incluyen empresas, universidades e instituciones de investigación.

De la misma manera que ser una ISA modular presenta una gran ventaja, esto también supone un reto a la hora de verificar los procesadores RISC-V, ya que los cores previamente verificados son de aquellos con el juego de instrucciones por defecto que proporciona RISC-V, sin embargo, en caso de querer añadir nuevas instrucciones implica nuevos casos de verificación. Aquí es donde entra en juego una metodología robusta de verificación como UVM.

1.1.1. Universal Verification Methodology (UVM)

En la actualidad, el procedimiento de verificación desempeña un papel crucial en el proceso de desarrollo de productos, especialmente en el diseño de sistemas hardware digitales. De acuerdo con estudios realizados, la etapa de verificación puede llegar a consumir más del 60% del tiempo total de los proyectos, como se ilustra en la Figura 1. Es importante señalar que en los proyectos donde se destina menos tiempo a la verificación, es debido a la presencia de una amplia cantidad de módulos IP diseñados y verificados previamente dentro del propio proyecto. Estos proyectos tienden a incluir cada vez más ingenieros de verificación en comparación con los ingenieros de diseño, e incluso los ingenieros de diseño asignan una parte de su tiempo a tareas de verificación [1].

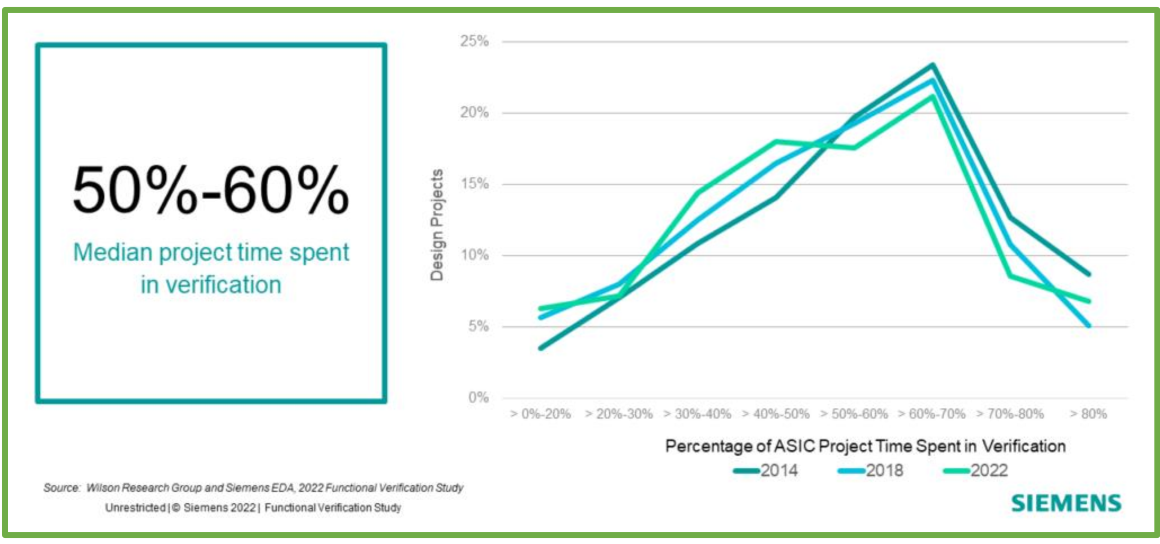


Figura 1 Porcentaje del tiempo del proyecto ASIC dedicado a la verificación

Los avances tecnológicos continuos y la creciente demanda del mercado en el sector han propiciado que los dispositivos y sistemas desarrollados integren cada vez más componentes, los cuales desempeñan un número creciente de funciones, volviéndolos, en consecuencia, más complejos. Esta complejidad conlleva una necesidad de verificación más rigurosa y sofisticada. Como respuesta a esta demanda, ha surgido una metodología de verificación universal conocida como UVM (*Universal Verification Methodology*), la cual se ha establecido como un estándar de facto en la industria. Esta metodología, desarrollada como el conjunto de varias tecnologías de verificación utilizadas previamente por importantes empresas del sector, ha demostrado su eficacia desde su concepción hasta la fecha actual. Como se puede apreciar en la Figura 2, la metodología UVM se ha mantenido consistentemente como la metodología más ampliamente empleada para la creación de bancos de pruebas, tanto en diseños ASIC como en diseños basados en FPGA [2].

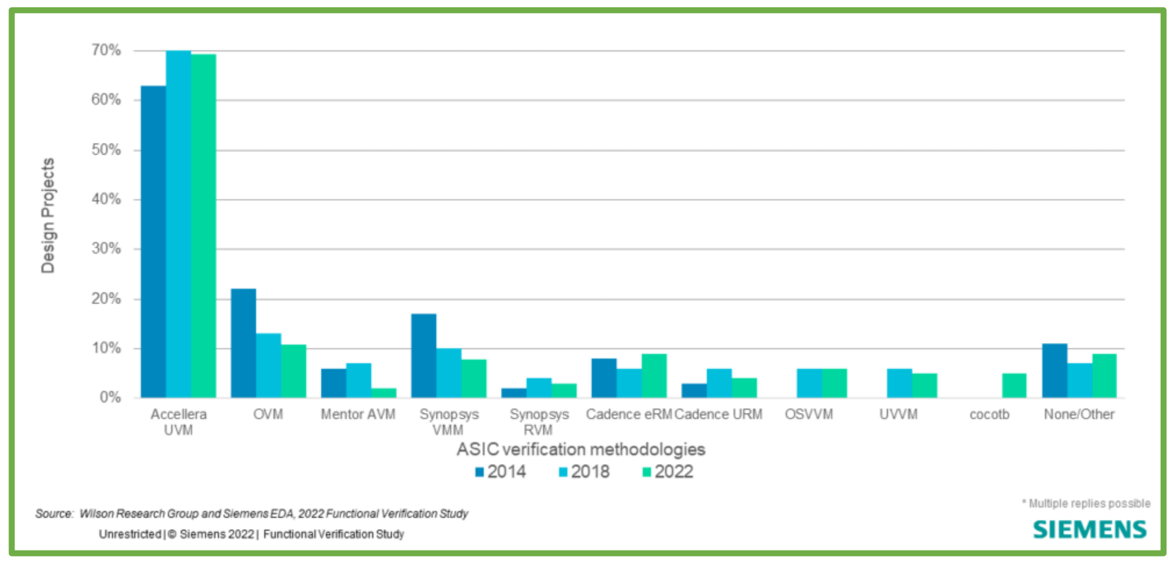


Figura 2 Metodología ASIC y tendencia de adopción de bibliotecas de clases

Esta metodología está basada en el lenguaje de descripción y verificación Hardware HDL (*Hardware Description Language*) SystemVerilog. En la Figura 3 se muestra la adopción de los diferentes lenguajes de verificación Hardware en los últimos años [2].

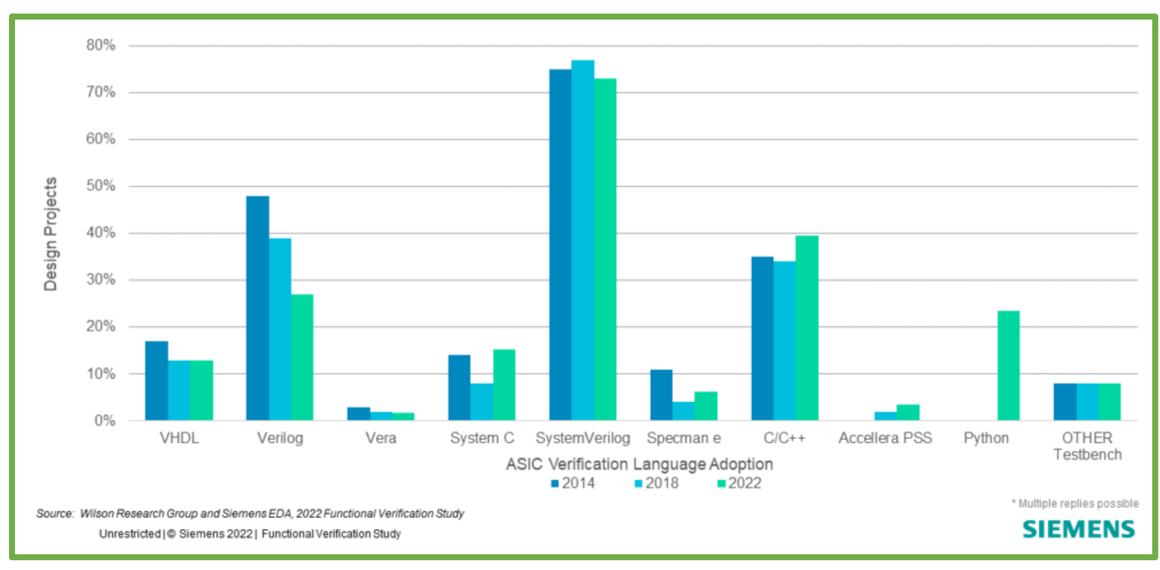


Figura 3 Tendencias de adopción de los lenguajes de verificación y descripción Hardware

Es fundamental destacar la importancia de adquirir habilidades en verificación funcional utilizando la metodología UVM, dada la alta demanda de profesionales en este ámbito. Esta demanda se refleja claramente en numerosas ofertas de empleo disponibles en reconocidos portales especializados en el diseño electrónico, como Chipcript Jobs [3]. En dicho portal, prácticamente todas las ofertas de trabajo con títulos como "*Verification Engineer*" o similares, hacen hincapié en la necesidad de poseer conocimientos sobre UVM como requisito mínimo. En algunos casos, estas ofertas incluso exigen habilidades sólidas en verificación basadas en SystemVerilog/UVM y en la metodología de verificación funcional. Esta tendencia subraya una vez más la importancia actual del dominio de esta tecnología y el manejo de las herramientas asociadas a ella.

1.1.2. Reduced Instruction Set Computer V (RISC-V)

RISC-V es una arquitectura de juegos de instrucciones (*Instruction Set Architecture* - ISA) libre y abierta, creada mediante una colaboración abierta entre la comunidad científica para permitir el desarrollo de procesadores en distintos dominios de aplicación, permitiendo el desarrollo estratégico de la industria de semiconductores. A diferencia de otras ISA, RISC-V ofrece un conjunto muy reducido de instrucciones básicas (47) y la posibilidad de añadir distintos niveles de complejidad en función del dominio de aplicación (ISA modular). Este tipo de diseño sigue la filosofía de desarrollo de tipo RISC (*Reduced Instruction Set Computer*), el cual comenzó a desarrollarse en la Universidad de California en Berkeley y que ha tomado especial interés en la comunidad científica desde 2010, teniendo en la actualidad numerosos colaboradores y

desarrolladores externos a la institución. Dicha colaboración se organiza a través de la fundación RISC-V International, con sede en Suiza [4].

Alguna de las principales razones que justifican la creación de una nueva ISA se pueden resumir en los dos puntos siguientes [5].

1. Todas las ISAs comerciales populares son propietarias. Sus proveedores obtienen beneficios de las distintas implementaciones que se realizan de su ISA, ya sea en forma de núcleos IP o silicio, o bien mediante el pago de royalties.

2. En RISC-V, cuando se habla de "un juego de instrucciones reducido" no hace referencia a que la arquitectura RISC soporte menos tipos de instrucciones [6]. Realmente hace referencia a que las instrucciones en RISC son más simples. Para los procesadores RISC, una instrucción de carga de datos en memoria no hace más operaciones. El procesador espera una nueva instrucción que le diga lo que tiene que hacer con esos datos. Implementar esas instrucciones en el procesador es mucho más sencillo. Adicionalmente ocupa menos espacio en los bloques lógicos, pudiéndolos hacer mucho más pequeños. Esto, a su vez, permite obtener mayores velocidades de procesamiento [7]. Por tanto, el objetivo es acelerar al máximo las operaciones más comunes, para lo cual se requiere una cooperación muy estrecha con el compilador.

El interés de RISC-V hace que en la actualidad sean más de 80 grandes compañías tecnológicas las que dan soporte a RISC-V, entre las que se encuentran Google, Qualcomm, Nvidia, Intel, Synopsis, IBM o Huawei. En esta misma dirección, Red Hat (IBM) se ha unido a la fundación de RISC-V como miembro oficial para ayudar a impulsar este proyecto con sus conocimientos del sector open source.

RISC-V es la arquitectura de referencia dentro de la EPI (*European Processor Initiative*), cuyo objetivo es la creación de un microprocesador propio para la Unión Europea, limitando la dependencia del exterior [8].

El Barcelona *Supercomputing Center* – Centro Nacional de Supercomputación (BSC - CNS) y la empresa estadounidense Intel [9] han anunciado en el ISC de Hamburgo (Alemania) sus planes de crear conjuntamente un laboratorio pionero para desarrollar una nueva generación de supercomputadores que rompa la barrera de la zettascale [10]. Para ello, el laboratorio conjunto diseñará microprocesadores o chips con tecnología basada en el hardware de código abierto RISC-V. Este laboratorio conjunto ayudará a Europa a ser autónoma en este tipo de chips, que podrán utilizarse en todo el mundo en terrenos como el diseño de coches autónomos o de dispositivos para aplicaciones de inteligencia artificial. Este laboratorio conjunto recibirá hasta 400 millones de euros

de inversión en 10 años. Estos fondos provendrán de Intel y del Gobierno español a través del PERTE Chip como parte del Plan de Recuperación, Transformación y Resiliencia, aprobado por el Consejo de ministros recientemente.

Es tal el crecimiento de RISC-V que cada vez se utiliza más en diferentes ámbitos, cubriendo prácticamente todo el panorama de las telecomunicaciones donde se necesite cualquier tipo de procesador. Y todo indica a que su importancia y visibilidad sea cada vez mayor como índica la Figura 4, donde se observa como cada año el número de publicaciones acerca de RISC-V va en aumento siendo cada vez más rápido.

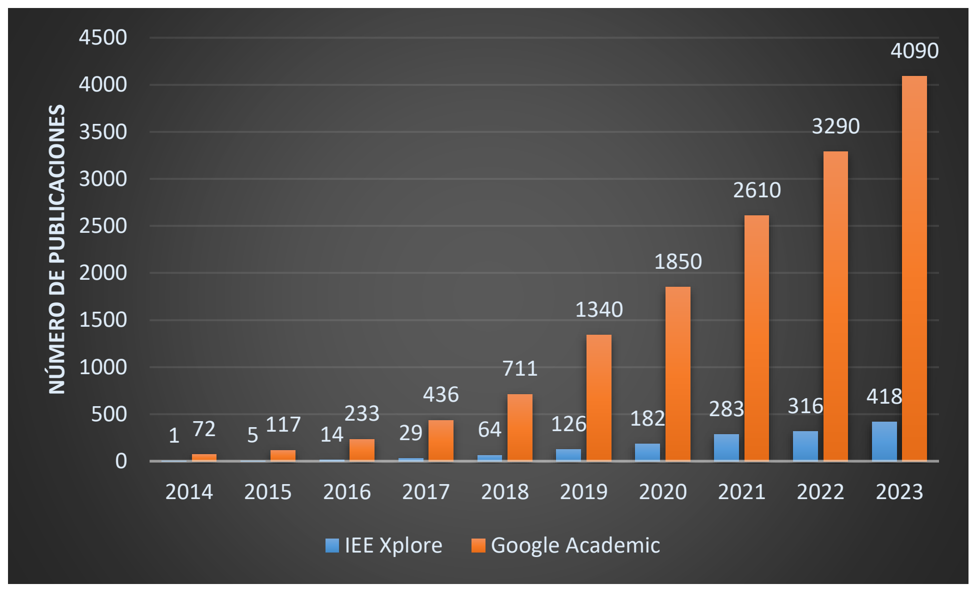


Figura 4. Publicaciones Anuales relacionadas con RISC-V en Google Academic y IEE Xplore

1.2. Objetivos

El objetivo de este proyecto es el estudio de la metodología UVM para la implementación para verificar un IP de procesamiento con la arquitectura RISC-V. En este caso, se hará uso de uno de los procesadores que proporciona lowRISC [11]. Debido a la complejidad de diseñar un *testbench* completo y con funciones de autoverificación basada en modelos de referencia, para un procesador de estas características, el objetivo principal es estudiar el funcionamiento del entorno UVM desarrollado por lowRISC, así como realizar su ejecución con el fin de analizar los resultados y, finalmente, modificar este entorno para desarrollar test propios.

Para alcanzar este objetivo, se propone concretar el trabajo en los siguientes objetivos:

- O1. Estudiar en detalle la arquitectura del módulo RISC-V elegido.
- O2. Estudiar en profundidad los conceptos necesarios de la metodología UVM, principalmente aquellos más relacionados con el IP a verificar.
- O3. Comprender el entorno de verificación UVM implementado por el grupo lowRISC, ejecutar el *testbench* y analizar los resultados obtenidos. Implementar el módulo RISC-V al flujo de diseño ASIC para su futura fabricación.
- O4. Añadir al *testbench* UVM determinados test para la verificación propia.
- O5. Documentar el trabajo realizado

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).

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 dará una introducción al proceso de verificación y su importancia en el flujo de diseño de sistemas hardware, centrándose particularmente en la verificación funcional. Para poder entender la arquitectura basada en componentes que utiliza UVM es necesario en primer lugar introducir la verificación que ofrece *SystemVerilog*, que ya introduce gran parte de la idea que propone UVM, además de ser el lenguaje de verificación hardware utilizado por esta metodología, para ello, se creará un entorno completo de verificación basado en *SystemVerilog* para un módulo IP independiente. En este caso, se ha elegido una PIFO [12] ya que su diseñador incluye, además de su descripción, su pequeño *testbench*. Posteriormente, el entorno desarrollado se pasará a UVM mostrando las ventajas que esta metodología ofrece.

2.1. Verificación de sistemas electrónicos digitales

La verificación, es esencial en el proceso de desarrollo de un diseño, garantiza que este cumpla con las especificaciones establecidas y se satisfagan las necesidades previstas. La creciente complejidad y diversidad de funciones en los diseños electrónicos contemporáneos aumentan el desafío de la verificación, especialmente con la presión por acelerar el tiempo de llegada al mercado. Por ello, la verificación ha ganado importancia y ha impulsado avances rápidos en herramientas especializadas.

Contrario al diseño, la verificación evalúa una implementación para confirmar su conformidad con las especificaciones. Por consiguiente, cada fase del diseño de un sistema digital requiere verificaciones adecuadas para asegurar la integridad del diseño en distintos niveles de abstracción.

Existen varias formas de llevar a cabo la verificación, que pueden variar según el objetivo, la estrategia de seguimiento o el método de ejecución. Entre estos métodos se encuentran la verificación estática (formal) y la verificación dinámica (funcional). Esta última ha ganado relevancia, siendo el método utilizado en este Trabajo Fin de Máster utilizando simulación o emulación. La Figura 5 ilustra las tres filosofías de implementación de la verificación funcional en relación con la información del DUV: *Black-Box*, *White-Box* y *Grey-Box*.

Figura 5 Modelos de verificación

Las principales características de estos tres métodos son:

- ***Black-Box***: En este enfoque, el sistema se evalúa sin conocimiento detallado de su diseño interno o cómo interactúan sus componentes. Se enfoca en observar qué hace el dispositivo, sin preocuparse por cómo lo hace. Se analizan las entradas y salidas del sistema, estimulando las entradas para observar el comportamiento de las salidas. Se considera una verificación exitosa si las salidas coinciden con los valores esperados para las entradas dadas, como lo haría un usuario final.
- ***White-Box***: En este modelo, el ingeniero de verificación tiene un conocimiento completo del diseño bajo prueba (DUV). A diferencia del enfoque *Black-Box*, se accede a la estructura interna del diseño, lo que facilita la depuración, por ejemplo, mediante el uso de propiedades. Se pueden evaluar y controlar todas las señales internas del dispositivo durante la ejecución del *testbench*.
- ***Grey-Box***: En este enfoque, el ingeniero de verificación tiene un conocimiento parcial de la estructura interna del dispositivo. Combina aspectos del enfoque *Black-Box* y *White-Box*, lo que permite un equilibrio entre la abstracción del *Black-Box* y la visibilidad interna del *White-Box*. Este enfoque permite controlar el diseño a nivel de sus interfaces, mientras se

gestionan las señales internas durante la ejecución. Este enfoque está particularmente pensado para diseños con módulos IP interconectados con interfaces estándar.

2.2. Lenguaje de Descripción y verificación hardware *SystemVerilog*

SystemVerilog es un estándar universal para la descripción y verificación de hardware, estandarizado como IEEE 1800. Se utiliza en el diseño y verificación de sistemas electrónicos y es una extensión de Verilog-2001, que incorpora conceptos de otros lenguajes como VHDL, C y C++. En la Figura 6 se ilustran las diferentes funcionalidades asociadas a este lenguaje.

La metodología UVM se desarrolla junto con el lenguaje *SystemVerilog*, que abarca tanto la descripción del hardware (*Hardware Description Language* - HDL) como la verificación del hardware (*Hardware Verification Language* - HVL).

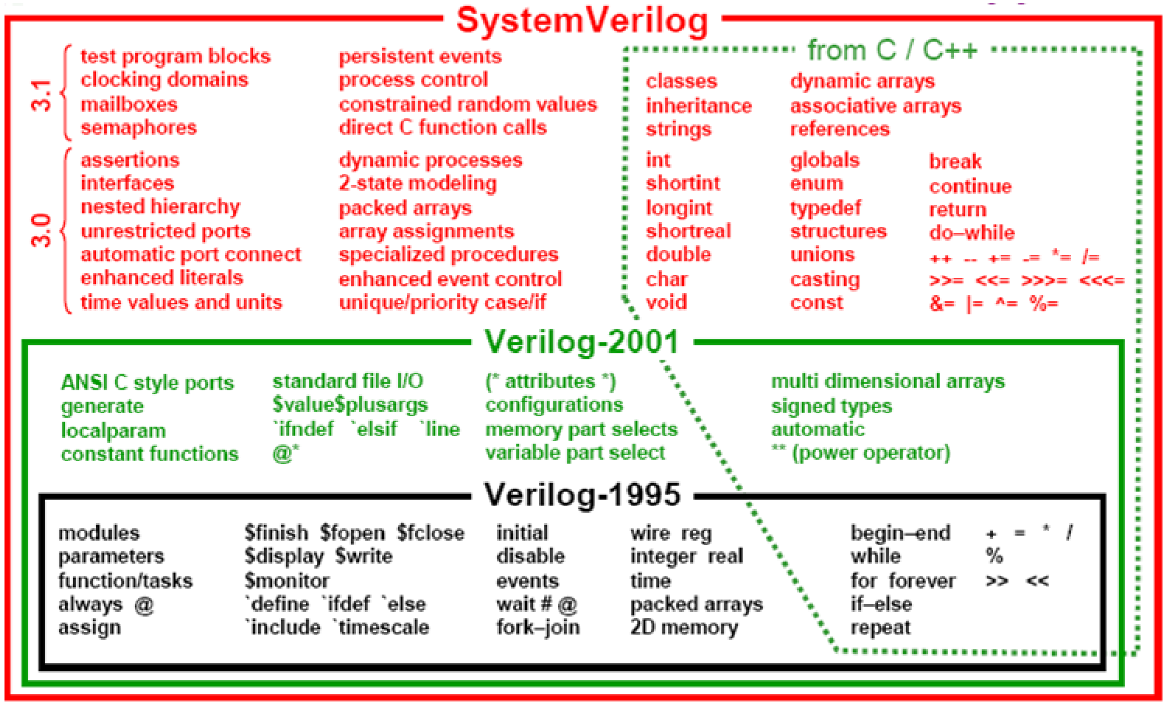


Figura 6 Funcionalidades de *SystemVerilog* [13]

SystemVerilog ha evolucionado desde el lenguaje de descripción hardware *Verilog* y ha integrado funciones de otros lenguajes. Su principal objetivo está en la verificación del correcto funcionamiento de los sistemas electrónicos. Algunas razones para su importancia son:

- Utiliza técnicas de programación orientada a objetos (OOP), lo que facilita el desarrollo de entornos de verificación más complejos mediante el uso de clases y herencia.
- Emplea paquetes para compartir código entre módulos, incluyendo declaraciones y definiciones de constantes, tipos, funciones y clases.

- Permite el uso de propiedades *assertions*, así como la realización automática de medidas de cobertura, con soporte para realizar una verificación tipo ABV (*Assertion-Based Verification*) y adquisición de datos flexible.
- Ofrece una amplia variedad de tipos de datos, tanto dinámicos como estáticos.
- Permite la aleatorización de datos, importante para la metodología UVM al estimular el DUV, con la capacidad de aplicar restricciones.
- Facilita la comunicación entre elementos mediante el uso de interfaces, con soporte para comunicaciones TLM, lo que permite la reutilización de entornos de verificación en diferentes niveles de abstracción.
- La interfaz DPI (*Direct Programming Interface*) permite referenciar directamente funciones de C en el código *SystemVerilog*.

La descripción de un hardware consiste principalmente en varios archivos *Verilog* (o *SystemVerilog*) con un módulo principal, en el cual se referencian todos los submódulos restantes para lograr el comportamiento y la funcionalidad deseados. Hoy en día, para la verificación en *SystemVerilog* se requiere un entorno llamado *testbench*. La idea es ejecutar, haciendo uso de este *testbench*, diferentes test para observar sus salidas y compararlas con los valores esperados y poder verificar si el diseño hace el comportamiento esperado.

Para hacer esto, se instancia el módulo de diseño de nivel superior con el entorno *testbench* y los puertos se conectan con las señales apropiadas del componente *testbench*. Se aplican ciertos valores a las entradas del diseño, para los cuales conocemos cómo debería operar el diseño. Se analizan las salidas y se comparan con los valores esperados para verificar si el comportamiento del diseño es correcto.

Para verificar el comportamiento funcional del diseño, se escribe un *testbench*. El proceso de verificación permite a los ingenieros de verificación encontrar errores y verificar la corrección de la descripción RTL basada en la especificación del diseño. El primer paso en el proceso de verificación es elaborar un plan de verificación que esté estrechamente vinculado con la especificación del diseño e involucre qué características deben ser probadas y qué técnicas se deben utilizar para verificar el diseño bajo prueba (DUV). El *testbench* es responsable de:

- Generar los estímulos de entrada.
- Conducir un estímulo de entrada al DUV.
- Monitorizar la actividad del diseño a nivel de salida y entrada.
- Comprobar la corrección de la transacción de salida basada en el estímulo de entrada.
- La convergencia de la función de muestreo y la corrección de las afirmaciones.

Un concepto clave para cualquier metodología de verificación moderna es el banco de pruebas basado en capas. Aunque se puede pensar que este proceso hace el banco de pruebas más complejo, en realidad se facilita la tarea al dividir el código en piezas más pequeñas que pueden ser desarrolladas por separado. No debe intentarse escribir una única rutina que pueda generar aleatoriamente todos los tipos de estímulos, tanto legales como ilegales, además de inyectar errores con un protocolo multinivel. De hacerlo, esta rutina se vuelve rápidamente compleja e inmantenible. Además, se permite la reutilización y encapsulación de la Verificación IP (VIP) con un enfoque en capas, que son conceptos de la programación orientada a objetos (OOP).

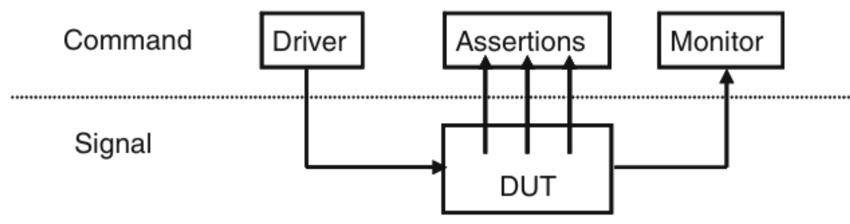


Figura 7 Capas *Signal* y *Command*

La Figura 7 muestra las capas bajas de un *testbench.* En la parte inferior se encuentra la capa de señales (*Signals*) que contiene el diseño bajo prueba y las señales con las que se conecta al banco de pruebas. El siguiente nivel superior es representado por la capa de comandos (*Commands*). En este nivel aparecen dos componentes muy importantes. Las entradas del DUV son accionadas por el componente *driver*, el cual ejecuta comandos individuales, como leer o escribir en el bus. Las salidas del DUV son monitorizadas por el componente *monitor*, el cual analiza las transiciones de las señales del interfaz y las agrupa en comandos. Las propiedades o aserciones (*assertions)* son mecanismos que también cruza las capas de comandos/señales. Estos observan señales individuales dentro del DUV, así como cambios a lo largo de un comando completo, con la finalidad de verificar el correcto funcionamiento de la propiedad definida.

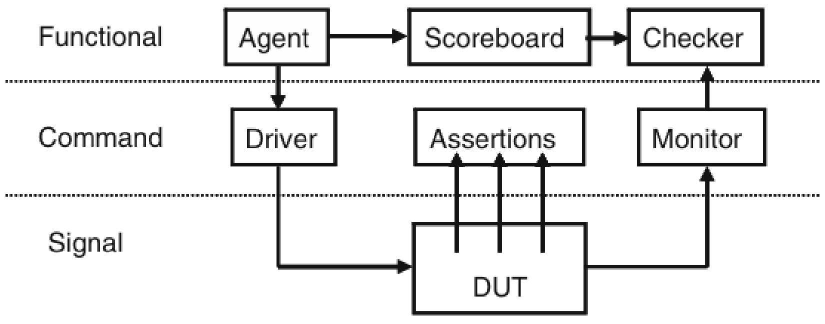


Figura 8 *Testbench* con capa funcional añadida

En la Figura 8 se muestra el *testbench* con la capa funcional añadida, que es alimentada hacia la capa de comandos. En esta capa las transacciones de alto nivel, como puede ser lecturas o escrituras de acceso directo a memoria (DMA), son recibidas por el bloque agente y son divididas en comandos o transacciones individuales más sencillos. Estos comandos son también enviados a un nuevo componente, denominado *scoreboard*, que predice los resultados de la transacción. Los comandos obtenidos por el componente monitor son enviados al componente *checker,* que los comparará con los comandos obtenidos del *scoreboard*.

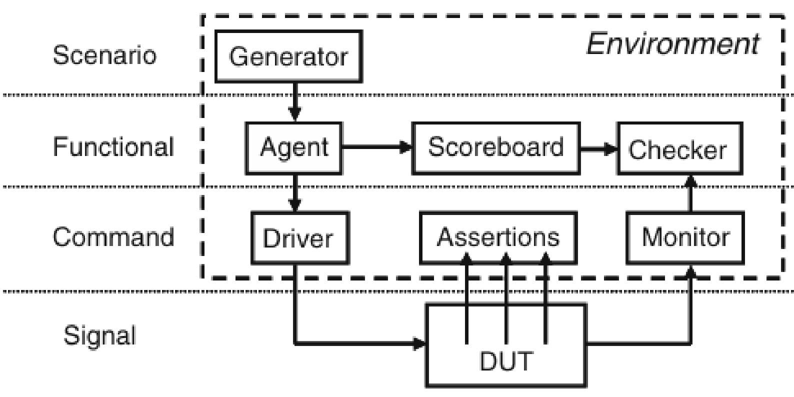


Figura 9 *Testbench* con capa *scenario* añadida

La capa funcional es activada por el componente generador ubicado en la capa de escenario, como se muestra en la Figura 9. El ingeniero de verificación debe asegurarse de que el dispositivo cumpla con su tarea prevista en los diferentes escenarios. Un ejemplo de dispositivo es un reproductor de MP3 que puede reproducir música de forma concurrente desde su almacenamiento, descargar nueva música de un host y responder a la entrada del usuario, como ajustar el volumen y los controles de la pista. Cada una de estas operaciones es un escenario. La descarga de un archivo de música implica varios pasos, como lecturas y escrituras de registros de control para configurar la operación, múltiples escrituras DMA para transferir la canción, seguido de otro grupo de lecturas y escrituras. La capa de escenario del banco de pruebas orquesta todos estos pasos con valores aleatorios, o no restringidos por parámetros, como el tamaño de la pista y la ubicación de la memoria.

SystemVerilog define un contenedor, denominado *Environment,* que agrupa todas las capas definidas en la Figura 9. Todos los bloques definidos en este contenedor se escriben al comienzo del desarrollo del proyecto de verificación. Durante el proyecto estos componentes pueden evolucionar y se puede nueva añadir funcionalidad, sin embargo, estos bloques no deberían cambiar para la ejecución de pruebas individuales. Esto se hace posible con el uso de dejando

"*hooks*" en el código lo que permite que una prueba pueda cambiar el comportamiento de estos bloques sin tener que reescribir el código del mismo.

SystemVerilog propone materializar la estructura en capas de la Figura 9 en una arquitectura basada en componentes y enlazada con el DUV mediante uno o varios interfaces de entrada/salida, tal y como se muestra en la Figura 10. Esta organización es la base sobre la que se levanta la arquitectura de un *testbench* UVM, tal y como se verá en la última parte de este capítulo.

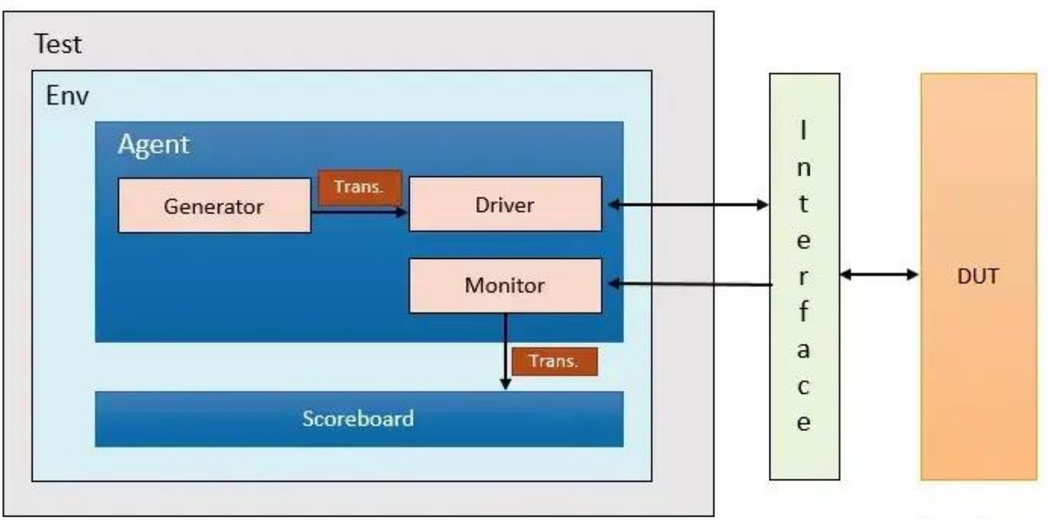


Figura 10 Entorno de test basado en componentes SystemVerilog [14]

2.3. Ejemplo de Entorno de Verificación en *SystemVerilog*

Con el fin de cumplir con los objetivos de conocimientos sobre UVM, en el proceso de estudio se ha elaborado un entorno completo basado en *SystemVerilog* para verificar un módulo IP [15] que implementa un módulo PIFO, para aplicar los conocimientos obtenidos de manera que se llegue al objetivo final con una base más consolidada.

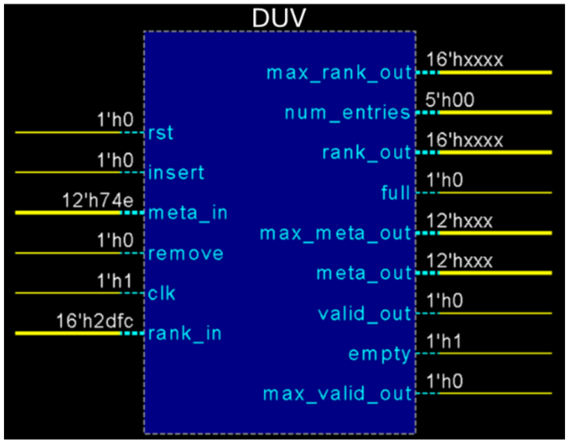


Figura 11 Interfaz de E/S del módulo PIFO

La Tabla 1 muestra las señales de la interfaz de E/S de este módulo (Figura 11).

Tabla 1 Descripción señales E/S PIFO

Señal	I/O	Descripción
rst	Input	Señal de reinicio, a nivel alto reinicia la PIFO, limpiando todas las entradas y señales internas.
clk	Input	Señal de reloj, sincroniza las operaciones del módulo.
insert	Input	Señal de inserción, a nivel alto, activa la inserción de una nueva entrada en la PIFO.
remove	Input	Señal de eliminación, a nivel alto, activa la eliminación de la entrada con la mayor prioridad (menor rango) de la PIFO.
rank_in	Input	Señal de 16 bits que indica la prioridad de la nueva entrada a insertar.
meta_in	Input	Señal de 12 bits, es el propio dato asociado a la prioridad indicada en la señal anterior.
num_entries	Output	Señal que indica el número actual de entradas almacenadas en la PIFO.
rank_out	Output	Señal que indica el rango de la entrada con la mayor prioridad (menor rango) de la PIFO.
meta_out	Output	Señal que indica el dato asociado a aquel con la prioridad **rank_out**.
max_rank_out	Output	Señal que indica la prioridad de la entrada con la menor prioridad (mayor rango).
max_meta_out	Output	Señal que indica el dato asociado a aquel con la prioridad **max_rank_out**.
valid_out	Output	Señal que indica si las señales **rank_out** y **meta_out** son válidas.
max_valid_out	Output	Señal que indica si las señales **max_rank_out** y **max_meta_out** son válidas.
full	Output	Señal que indica si la PIFO está llena o no.
empty	Output	Señal que indica si la PIFO está vacía o no.

Este módulo consiste en una *Push In First Out* (PIFO) de tamaño configurable con el parámetro `REG_WIDTH`. La PIFO señaliza la validez (`VALID_OUT`) de los elementos en el registro y mantiene contadores para el número de entradas, verificando en todo momento si la PIFO está llena o vacía. A diferencia de una FIFO, la PIFO, al insertar un elemento, comprueba el rango (clase) del mismo, insertándolo en el lugar correspondiente. El módulo elegido codifica la clase a través del puerto `RANK_IN` de la interfaz de entrada/salida.

De tal manera que, cuando se inserta un elemento (`INSERT`), el módulo comprueba si el registro está lleno. Si no está lleno, el nuevo elemento se inserta. Si está lleno, compara el rango (`RANK_IN`) del nuevo elemento con el rango máximo almacenado en el registro. Si el rango del

nuevo elemento es más pequeño (mayor prioridad), este se inserta en el lugar correspondiente. Al mismo tiempo, se elimina el elemento menos prioritario.

La extracción de un elemento se lleva a cabo cuando se activa la señal `REMOVE`. El módulo verifica si hay elementos presentes en el registro (`NUM_ENTRIES > 0`) y luego procede a validar el elemento de prioridad más alta.

El primer paso para la creación del entorno de verificación es la definición de su estructura. En este caso, la estructura es ligeramente diferente a la estructura propuesta en la Figura 10, debido a que una PIFO cuenta con 2 interfaces, una primera para realizar las operaciones `INSERT` y una segunda para las operaciones `REMOVE`. Con esta consideración, la estructura queda como se muestra en la Figura 12.

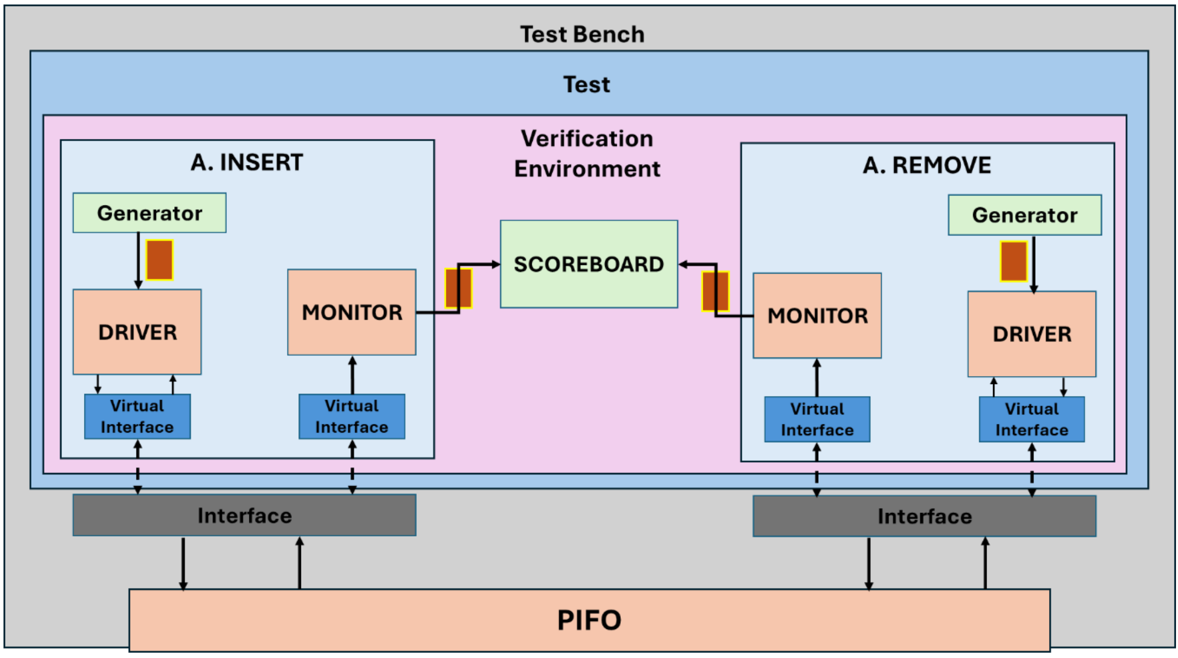


Figura 12 Entorno Verificación *SytemVerilog* PIFO

2.3.1. Objeto *Transaction*

El componente *transaction* define un objeto que contiene toda la información necesaria para poder realizar una operación en un interfaz. Para definir el objeto, se crea una representación abstracta de un conjunto de datos que se intercambian entre diferentes componentes del entorno. Estos contienen los datos específicos que se van a transmitir, en este caso al tratarse de una PIFO se envía los datos correspondientes a los datos del mismo. En este ejemplo, al usar dos interfaces, se han creado dos transacciones, una para la interfaz de `INSERT` (`transaction_in`) y otra para la interfaz `REMOVE` (`transaction_out`).

Para crear una transacción, *sytemverilog* hace uso de las clases que son la clave del paradigma de programación orientada a objetos (OOP) que el lenguaje soporta. Las clases permiten definir estructuras de datos complejas y comportamientos asociados, facilitando la creación de entornos de verificación reutilizables y escalables.

```
1  class transaction_in;
2    bit insert;
3    rand bit [15:0] rank_in;
4    rand bit [11:0] meta_in;
5    bit full;
6    bit [15:0] max_rank_out;
7    bit [11:0] max_meta_out;
8    bit [4:0] num_entries;
9    bit [1:0] cnt;
10   function void display();
11   $display("----[TRANS_IN] post_randomize---");
12   $display("- insert = %0d",insert);
13   $display("- rank_in = %0d",rank_in);
14   $display("- meta_in = %0d",meta_in);
15   $display("-------------------------");
16 endclass
```

Código 1 `Transaction_in` del módulo PIFO en *SystemVerilog*

En la transacción de entrada (Código 1) se definen los ítems que corresponden con los datos del DUV a tener en cuenta cuando se produzca una operación de `INSERT`. Estos datos vienen a ser las propiedades que encapsula la clase. Dentro de una clase se puede hacer uso de funciones como `$display()` (línea 10 del Código 1). Esta función no consume tiempo de simulación y permite realizar una sencilla depuración mostrando el valor de los datos que forman la transacción.

Otra herramienta importante que soporta *SystemVerilog* es la aleatorización de variables. Esto permite generar estímulos variados y realistas para el DUV. En el caso de la PIFO es muy útil para generar datos aleatorios con prioridades aleatorias. Para ello, si en la definición de un campo, como ocurre en las líneas 3 y 4, se usa la macro `rand`, esto permite generar valores aleatorios para dichos campos en cada llamada a `randomize()`. Existen otras posibilidades como `randc` (*cyclic random*), que genera valores aleatorios cíclicos asegurando que todos los valores posibles se generen antes de que se repita cualquier valor. Sin embargo, para el entorno de verificación de la PFO no interesa que se genere un valor aleatorio de forma cíclica cada vez que el *generator* crea una nueva transacción llamando al método `randomize()`.

A diferencia de la transacción de entrada, la transacción de salida debe contener los ítems `REMOVE`, `RANK_OUT` y `META_OUT`. Además, a la hora de eliminar un dato es necesario contemplar si la PIFO está vacía. Por lo tanto, se ha añadido el ítem `EMPTY`. Por último, el protocolo de la PIFO es que no se puede eliminar ningún dato si este no está validado, así que se añade la señal `VALID_OUT` tal y como muestra el Código 2.

```
1  class transaction_out;
2    bit remove;
3    bit [15:0] rank_out;
4    bit [11:0] meta_out;
5    bit empty;
6    bit valid_out;
7    bit max_valid_out;
8    bit [4:0] num_entries;
9    bit [1:0] cnt;
10 endclass
```

Código 2 `Transaction_out` del módulo PIFO en *SystemVerilog*

2.3.2. Componente *Generator*

El componente *generator* es un componente del *testbench* encargado de crear las transacciones previamente comentadas y enviarlas al DUV como estímulos a través del componente *driver*. Para ello, se crea como una clase al igual que las transacciones para hacer uso de las funcionalidades que ofrece.

Al igual que ocurría con las transacciones, hay que definir un componente *generator* por cada una de las interfaces del *testbench.* El Código 3 muestra la descripción en *SystemVerilog* del componente *generator* para el interfaz `INSERT`.

```
1  class generator_in;
2    rand transaction_in trans, tr; //declarar tr class
3    mailbox gen2driv_in;  //declarar buzón
4    event ended;  //declarar evento fin de tr
5    int repeat_count;  // contador num items a generar
6    //constructor
7    function new(mailbox gen2driv_in, event ended);
8      this.gen2driv_in = gen2driv_in;
9      this.ended = ended;
10     trans = new();;
11   endfunction
12   // crea, randomiza y envia tr al driver
13   task main();
14     repeat(repeat_count) begin
15     if(!trans.randomize())$fatal("Gen:: trans randomization failed");
16     gen2driv_in.put(trans);
17     end
18     -> ended;
19   endtask
20 endclass
```

Código 3 Componente `Generator_in` del módulo PIFO en *SystemVerilog*

En la línea 2 del Código 3 se declara el manejador de la clase *transaction* definida para su interfaz haciendo uso de la macro `rand` para más adelante crear transacciones, aleatorizar los datos y enviarla al componente *driver*. Para poder enviar una transacción entre dos procesos concurrentes de manera sincronizada y manteniendo el código reutilizable, es decir sin necesidad de conocer el nivel jerárquico del entorno, *SystemVerilog* implementa el mecanismo de `mailbox`. En este caso, y para poder implementar la comunicación entre los componentes *generator* y *driver,*

se añade un `mailbox`, tal y como se muestra en la línea 3. En este caso, se declara un **mailbox** denominado `gen2driv_in`. Al no especificar tamaño este es ilimitado, aunque se puede especificar un tamaño límite. Un `mailbox` admite los siguientes métodos:

- **put():** Este método se utiliza para enviar datos al `mailbox`. Es bloqueante, lo que significa que si el `mailbox` está lleno (en el caso de un `mailbox` de tamaño limitado), el proceso esperará hasta que haya espacio disponible.
- **get():** Este método se utiliza para recibir datos del `mailbox`. También es bloqueante, por lo que, si el `mailbox` está vacío, el proceso esperará hasta que haya datos disponibles.
- **try_put()** y **try_get():** Estos métodos son versiones no bloqueantes de los métodos `put()` y `get()`, respectivamente. Devuelven un valor de éxito/fallo sin llegar a bloquear la ejecución del código que sigue a las llamadas.
- **num():** Este método devuelve el número de elementos actualmente en el `mailbox`.

Por lo tanto, como se muestra en la Figura 13, ambos procesos, tanto *generator* como *driver* deben compartir el mismo `mailbox` para una comunicación exitosa.

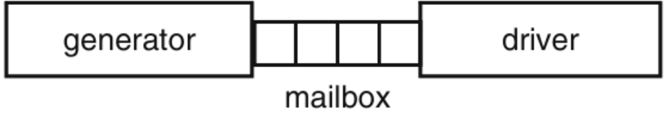


Figura 13 Conexión de dos componentes mediante `mailbox`

En *SystemVerilog,* a diferencia de C/C++, es posible manejar la construcción de objetos de manera sencilla y automática, evitando los problemas comunes como fugas de memoria presentes en lenguajes de programación. La línea 7 del Código 3 define la función `new()`, que es el constructor de la clase. Tal y como refleja la definición de su constructor, para crear el componente *generator* es necesario pasarle por parámetros el **mailbox** y, así como un evento que utilizará para declarar el fin de la transacción y que se comenta más adelante. En la línea 10 se crea la transacción a utilizar por este componente con la misma función `new()`. Sin embargo, tanto el **mailbox** como el evento `ended` (líneas 8 y 9) deben crearse haciendo uso de la macro previa `this` propia de la programación orientada a objetos y que indica que se va a crear un objeto de la clase “padre” correspondiente a cada uno.

Finalmente, en la línea 13 se declara la tarea `main()`. Esta tarea, contiene su bucle que se repite tantas veces como indica la variable `repeat_count`. Esta variable se establece en la capa Test, la de mayor jerarquía asignando así el número de transacciones que se van a generar por parte

del componente *generator*. Dentro del bucle se utiliza la función `randomize()` para aleatorizar la transacción y se envía la transacción al componente *driver* (línea 16) a través del `mailbox` previamente comentado. Por último, se activa el evento `ended` con el activador de eventos '->', indicando que el generador ha terminado su tarea, gracias al uso de eventos se sincronizan los diferentes procesos sin necesidad de añadir más variables o señales.

2.3.3. Interfaces en *SystemVerilog*

Systemverilog ofrece directamente el constructor de una interfaz que proporciona un medio para agrupar y abstraer las señales de interconexión entre módulos. En el caso de la verificación y en el caso del ejemplo utilizado en este capítulo, es utilizado con el fin de conectar el *testbench* con el DUV facilitando la creación de un entorno modular y escalable.

En el Código 4 se declara la interfaz de `INSERT`. La definición de un interfaz utiliza las palabras claves *`interface`* y *`endinterface`*. Dentro de la interfaz se declaran las señales y buses que se utilizarán para la comunicación entre módulos. Estas señales y buses pueden ser de entrada o salida.

```
1  interface intf_in(input logic clk, reset);
2    logic insert;
3    logic [15:0] rank_in;
4    logic [11:0] meta_in;
5    logic full;
6    logic [15:0] max_rank_out;
7    logic [11:0] max_meta_out;
8    logic [4:0] num_entries;
9  endinterface
```

Código 4 Interfaz `Interface_in` del módulo PIFO en *SystemVerilog*

Las señales en *SystemVerilog* sobre todo aquellas definidas dentro de una interfaz, se declaran tipo `logic`. Este tipo representa una mejora sobre el tipo `reg` utilizado en *Verilog*. Esto es debido a que permite que la señal pueda tomar valores binarios, indefinidos o de alta impedancia. De esta manera, cualquier señal declarada como `logic` se puede usar tanto en bloques de procedimiento como en declaración de asignaciones.

```
1  interface intf_out(input logic clk, reset);
2    logic remove;
3    logic [15:0] rank_out;
4    logic [11:0] meta_out;
5    logic empty;
6    logic [15:0] valid_out;
7    logic [11:0] max_valid_out;
8    logic [4:0] num_entries;
9  endinterface
```

Código 5 Interfaz `Interface_out` del módulo PIFO en *SystemVerilog*

2.3.4. Componente *Driver*

En el Código 6 se implementa el componente *driver* del interfaz INSERT, este componente es responsable de recibir los estímulos generados por el componente generador y enviarlos hacia el DUV asignando los valores de los ítems de cada transacción a las señales de la interfaz correspondiente. El *driver* es el componente de verificación que realiza la manipulación de señal a nivel de pines (*Pin-Wiggling*) del DUV.

```
1  class driver_in;
2    int num_trans;
3    virtual intf_in vif_in;
4    mailbox gen2driv_in;
5    //constructor
6    function new(virtual intf_in vif_in, mailbox gen2driv_in);
7      this.vif_in=vif_in;  //obtener la interfaz
8      this.gen2driv_in=gen2driv_in; //obtener el buzón
9    endfunction
10   task reset; //resetea las señales de la interfaz al valor por defect
11     wait(vif_in.reset);
12     $display("[ DRIVER_IN ] --- Reset Started ---" );
13         vif_in.insert <= 0;
14     wait(!vif_in.reset);
15     $display("[ DRIVER_IN ] --- Reset Ended ---" );
16   endtask
17   task drive; //transacción del item hacia la señal de la interfaz
18     transaction_in trans;
19     gen2driv_in.get(trans);
20     $display("--------- [DRIVER_IN-TRANSFER: %0d] ------",num_trans);
21     @(posedge vif_in.clk);
22     vif_in.insert <= 1;
23     vif_in.meta_in <= trans.meta_in;
24     vif_in.rank_in <= trans.rank_in;
25     @(posedge vif_in.clk);
26     vif_in.insert <= 0;
27     @(posedge vif_in.clk);
28     num_trans++;
29   endtask
30   task main;
31     forever begin
32       fork
33         //Thread-1: Waiting for reset
34         begin
35           wait(vif_in.reset);
36         end
37         //Thread-2: Calling drive task
38         begin
39           forever
40             drive();
41         end
42       join_any
43       disable fork;
44     end
45   endtask
46 endclass
```

Código 6 Componente Driver_in del módulo PIFO en *SystemVerilog*

En primer lugar, se crea el *driver* como una clase y se declara la variable `num_trans`, utilizada para contar el número de transacciones que se han enviado al DUV. De esta manera en el *Environment* se podrá comparar el número de transacciones enviada por el *driver* con el número de transacciones recibidas por el componente *scoreboard,* tal y como se explicará más adelante.

En la línea 3 del Código 6 se define un apuntador a la interfaz, este interfaz debe ser de tipo virtual. *SystemVerilog* cuenta con las interfaces virtuales, muy útiles en los entornos de verificación basado en componentes, debido a que son esencialmente un puntero a la interfaz física concreta utilizada para el conexionado con el DUV. Este puntero permite a las tareas y funciones del *driver* acceder a la interfaz sin necesidad de conocer su implementación física exacta. La asociación entre la interfaz virtual y la interfaz física se realiza durante la configuración del banco de pruebas en el *testbench.* El procedimiento consiste en conectar la interfaz virtual a la instancia real de la interfaz que conecta el componente *Environment* con el DUV. En la línea 4 se declara el `mailbox` de donde recibirá las transacciones puestas por el componente *generator*. De la línea 6 a 9 se define el constructor del componente *driver,* pasando por parámetros la interfaz virtual y el `mailbox` y haciendo uso del operador `this` para indicar en todo momento que hace referencia a la clase padre de cada ítem.

En la línea 11 del Código 6 se crea la tarea `reset`, esta tarea espera a que la señal de *reset* esté activa (línea 12), tras detectar el *reset*, pone las señales de la interfaz en sus valores por defecto (líneas 14 a la 16). En este caso, se establece el valor de la señal *insert* en la interfaz virtual. La asignación de valores a señales hace uso del operador '<='. En *SystemVerilog* este operador se utiliza para realizar asignaciones no bloqueantes. Estas asignaciones evalúan el lado derecho de la asignación inmediatamente, pero la actualización de la variable a la izquierda se programa para que ocurra al final del bloque de tiempo simulado actual. Esto asegura que todas las asignaciones no bloqueantes en un bloque `always` o `initial` se actualicen simultáneamente, una vez que todas las evaluaciones se hayan completado.

En la línea 17 comienza la tarea `drive`, aquí es donde realmente se comete el objetivo del *driver* que es activar el protocolo de comunicación con el DUV. En las líneas 18 y 19 se obtiene la transacción del *generator* a partir del `mailbox` `gen2drive_in`. En las líneas 22 a 24 se activa la señal de *insert* y se copia los datos de la transacción `meta_in` y `rank_in` en sincronismo con la señal de reloj. Además, en la línea 27, tras un ciclo de reloj se desactiva la señal de *insert*. Por último, se incrementa el contador de transacciones.

Para finalizar, en la línea 30 comienza la tarea `main`, donde se ejecuta un bucle infinito. Dentro de este bucle se pretende ejecutar dos códigos en paralelo, de forma concurrente. Para ello,

se hace uso de la estructura `fork join_any` de *SystemVerilog*. En esta estructura se crea un hilo por cada código a ejecutar en paralelo. En este componente se han creado dos hilos:

- **Thread-1:** Espera a que la señar de *reset* esté activa.
- **Thread-2:** Ejecuta la tarea `drive` en un bucle infinito.

Cuando cualquiera de los hilos termina, el bloque `fork` se deshabilita con la línea 43. Esto permite que la tarea `drive` se ejecute continuamente al implementar un bucle infinito, pero se detendrá y reiniciará si se detecta un *reset*. Esto permite describir el comportamiento real del hardware asociado.

Los cambios más importantes del componente *driver* de salida (Código 7) con respecto al de entrada son las tareas de `reset()` y `drive()`. En este caso y al tratarse de la extracción de un elemento de la PIFO, sólo se envía al módulo PIFO la señal activa de *remove* durante un ciclo de reloj. Esta señal provoca la extracción del dato presente en el interfaz de salida. Además, este *driver* contiene una lógica añadida, debido a que la PIFO no admite operaciones de `REMOVE` mientras no se haya validado el último dato. Este hecho es necesario tenerlo en cuenta a la hora de activar la señal *remove*. En la línea 16 se observa cómo mientras la señal de `VALID_OUT` no esté activa, se sigue esperando otro ciclo de reloj antes de activar el proceso de extracción.

```
1  //Reset task, resetea las señales de la interfaz al valor por defecto
2    task reset;
3      wait(vif_out.reset);
4      $display("[ DRIVER_OUT ] --- Reset Started ---" );
5      vif_out.remove <= 0;
6      vif_out.empty <= 1;
7      vif_out.max_valid_out <= 0;
8      vif_out.valid_out <= 0;
9      wait(!vif_out.reset);
10     $display("[ DRIVER_OUT ] --- Reset Ended ---" );
11   endtask
12   //drive la transacción del item hacia la señal de la interfaz
13   task drive;
14     transaction_out trans;
15     gen2driv_out.get(trans);
16       while (vif_out.valid_out == 0) begin
17       @(posedge vif_out.clk);
18     end
19     $display("--------- [DRIVER_OUT-TRANSFER: %0d] -----",num_trans);
20     @(posedge vif_out.clk);
21     vif_out.remove <= 1;
22     @(posedge vif_out.clk);
23     vif_out.remove <= 0;
24     @(posedge vif_out.clk);
25     num_trans++;
26   endtask
```

Código 7 `Driver_out` del módulo PIFO en *SystemVerilog*

2.3.5. Componente *Monitor*

El componente monitor (Código 8) se encarga de "muestrear" las señales de la interfaz correspondiente y convertir la actividad a nivel de señal a un nivel de transacciones y las envía al componente *scoreboard* para su verificación. El monitor se declara como una clase de *SystemVerilog*. En las líneas 4 y 6 se declara la interfaz virtual y el `mailbox` `mon2scb_in` que se utilizará para comunicar las transacciones capturadas desde el monitor hacia el *scoreboard*. En la línea 8 se define el constructor de la clase `monitor_in`, donde se inicializa las variables del objeto creado.

En la línea 14 se declara la tarea `main`. Esta tarea, para emular el comportamiento hardware, ejecuta un bucle infinito para que el monitor permanezca capturando transacciones continuamente mientras dure la simulación. En las líneas 16 y 17 se declara y crea un nuevo objeto de `transacción_in`, que se utilizará para almacenar los valores de las señales capturadas en la PIFO en cada ciclo de reloj.

```
1   class monitor_in;
2
3     //interfaz virtual
4     virtual intf_in vif_in;
5     //manejador mailbox
6     mailbox mon2scb_in;
7     //constructor
8     function new(virtual intf_in vif_in, mailbox mon2scb_in);
9       this.vif_in = vif_in;  //getting the interface
10      this.mon2scb_in = mon2scb_in; //getting the mailbox handles from
11  environment
12    endfunction
13    //captura señales y las envia por paquete al scoreboard
14    task main;
15      forever begin
16        transaction_in trans_in;
17        trans_in = new();
18        @(posedge vif_in.clk);
19        wait(vif_in.insert);
20         trans_in.insert = vif_in.insert;
21         trans_in.meta_in = vif_in.meta_in;
22         trans_in.rank_in = vif_in.rank_in;
23         trans_in.max_rank_out <= vif_in.max_rank_out;
24        @(posedge vif_in.clk);
25           trans_in.max_meta_out <= vif_in.max_meta_out;
26           trans_in.num_entries <= vif_in.num_entries;
27           trans_in.full <= vif_in.full;
28        mon2scb_in.put(trans_in);
29        $display("--------- [MONITOR_IN] Detectado --------- ");
30      end
31    endtask
32
33  endclass
```

Código 8 Componente `Monitor_in` del módulo PIFO en *SystemVerilog*

Para mantener la sincronización con el reloj, en la línea 18, se espera el flanco de subida del reloj haciendo uso de la interfaz virtual. En la línea 19 se espera hasta que la señal de *insert* se active, debido a que el monitor sólo avisará de que se ha producido una operación de INSERT al *scoreboard* cuando efectivamente se detecte un INSERT.

Desde la línea 20 hasta la línea 27 se realiza la captura de señales, asignando los valores de las señales de la interfaz a los campos correspondientes en la transacción. En la línea 28 finalmente se envía la transacción capturada al mailbox que comunica con el *scoreboard.* De esta manera, el *scoreboard* puede procesar los valores de las señales de la PIFO y compararlos con los resultados esperados para verificar el correcto funcionamiento del DUV.

2.3.6. Componente *Scoreboard*

El componente *scoreboard* tiene como objetivo recibir los datos proporcionados por el *monitor* y compararlos con los valores esperados. En este caso, al tratarse de una PIFO el *scoreboard* se ha diseñado como una cola de *SystemVerilog (queues)*, de tal manera que irá simulando el comportamiento de la PIFO con el fin de comparar esta cola con los datos obtenidos por ambos monitores y comprobar que los datos coinciden.

En primer lugar (Código 9), se referencia los mailbox de entrada y salida, se declara la cola con el tamaño igual al tamaño propio del diseño y se obtienen los manejadores de ambos mailbox mediante el constructor. Además, se declaran las variables de apoyo necesarias como son los contadores de transacciones con el fin de establecer el número de paquetes recibidos y compararlo con el número de transacciones enviadas por el *driver*. Por último, se necesita una variable para ir almacenando cada transacción que se extrae de la PIFO y poder compararla con la que se elimina en la cola del *scoreboard.*

```
1  class scoreboard;
2    //manejador mailbox y transacciones
3    mailbox mon2scb_in;
4    mailbox mon2scb_out;
5    transaction_in trans_in;
6    transaction_out trans_out;
7    //contador de numero de transacciones
8    int num_trans_in;
9    int num_trans_out;
10   //variables
11   bit [15:0] DatoEsperado;
12   localparam QUEUE_WIDTH = 16;
13   bit [15:0] queue_rank [$:QUEUE_WIDTH-1];
14   //constructor
15   function new(mailbox mon2scb_in, mailbox mon2scb_out);
16     this.mon2scb_in = mon2scb_in;
17     this.mon2scb_out = mon2scb_out;
18   endfunction
```

Código 9 Scoreboard-Instancia PIFO *SystemVerilog*

Tal y como se comentó anteriormente para el diseño del *scoreboard* se han implementado dos tareas, `insert()` y `remove()`, las cuales se ejecutarán cuando el *scoreboard* reciba alguna transacción desde el monitor correspondiente, emulando así el comportamiento de un módulo PIFO.

La tarea `insert()` contempla 4 escenarios implementados por las estructuras condicionales `if`. El primer escenario es que se produzca un *insert* mientras la PIFO se encuentra vacía y se trate del primer elemento insertado. En este caso, se inserta al principio de la cola el `rank_in` de la transacción tal y como se muestra en el Código 10 y se muestra por pantalla la cola completa. En la línea 23 se comprueba que la cola se encuentra vacía o que el valor de prioridad del dato a insertar sea menor que el primer elemento en la cola. El acceso al valor de la prioridad del dato de entrada, se realiza mediante el campo `rank_in` de la transacción, esto es `trans_in.rank_in`.

```
21  task insert();
22      //si hay espacio libre se inserta al comienzo
23      if((queue_rank.size()==0)||((queue_rank[0]>trans_in.rank_in&&
24  primera_in == 0)))begin
25          Maximo = trans_in.max_rank_out;
26          primera_in=1;
27          queue_rank.push_front(trans_in.rank_in);
28           $display("\t [SCB] Push_front queue_rank = %p", queue_rank);
29      end
```

Código 10 Componente *Scoreboard*: Tarea `insert()` – 1er caso

En la línea 25 se guarda el registro del dato con mayor prioridad para compararlo más adelante cuando sea necesario debido a que este valor puede resultar modificado. En la línea 27 se inserta el dato en la cola creada dentro del *scoreboard*. Al usar el método `push_front`, el dato se inserta en la parte inicial de la cola como muestra la Figura 14.

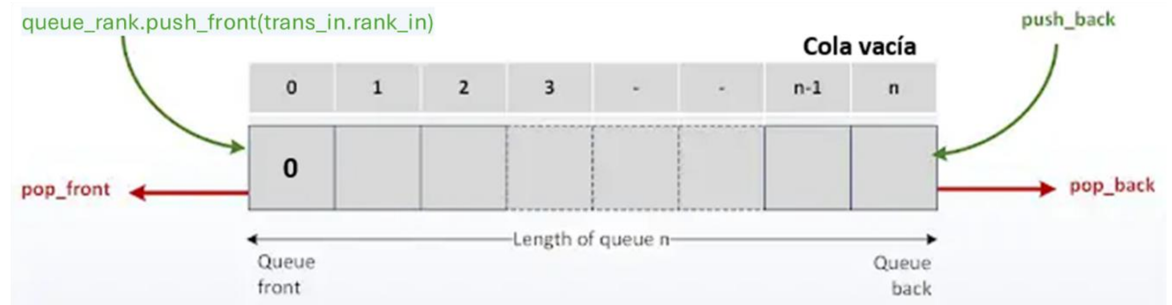


Figura 14 Mecanismo de inserción mediante método `push_front`

El segundo escenario sería encontrar la PIFO llena, pero con una prioridad del nuevo elemento superior a la prioridad mínima almacenada en la PIFO. En caso de que la prioridad de entrada sea inferior a la mínima, el comportamiento de la PIFO es ignorar la operación y descartar el nuevo dato. Si se cumple la condición de prioridad, en primer lugar, se debe eliminar el elemento

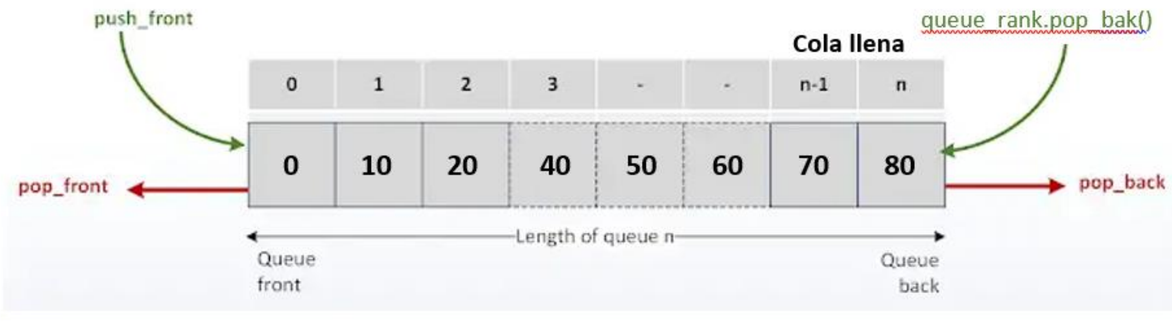
de menor prioridad e insertar el nuevo dato en su posición correspondiente siguiendo el orden de prioridades (Código 11).

```
29 //si está lleno y no es máxima prio, primero se elimina el de mayor
30 prio y se inserta ordenado
31     else if(queue_rank.size()==QUEUE_WIDTH &&
32 (trans_in.rank_in<queue_rank[queue_rank.size()-1]))begin
33         Maximo = trans_in.max_rank_out;
34         DatoEsperado = queue_rank.pop_back();
35         foreach(queue_rank[i])
36           if(trans_in.rank_in<queue_rank[i])begin
37             queue_rank.insert(i,trans_in.rank_in);
38             break;
39           end
40        $display("\t [SCB] Insert-Full queue_rank = %p", queue_rank);
41         if(DatoEsperado == Maximo)begin
42        $display("\t [SCB-PASS!] DatoEsperado = %0d, DatoEliminado =
43 %0d", DatoEsperado, Maximo);
44         end
45         else begin
46         $error("\t [SCB-FAIL!] DatoEsperado = %0d, DatoEliminado =
47 %0d", DatoEsperado, Maximo);
48         end
49     end
```

Código 11 Componente *Scoreboard*: Tarea `insert()` – 2do caso

En la línea 31 se valora la condición del segundo escenario, donde se comprueba: que cuando se inserte un nuevo dato, la cola se encuentra llena comparando el tamaño de la cola `queue_rank.size()` con el parámetro `QUEUE_WIDTH` que indica el tamaño de la PIFO en el diseño; y se valora que también se cumpla que la prioridad del nuevo dato sea mayor que la prioridad del último dato en la cola. Si se cumplen estas condiciones se vuelve a guardar el registro correspondiente al dato con menor prioridad de la PIFO (línea 33) y en la línea 34 se elimina el último dato de la cola guardándolo en la variable `DatoEsperado`. Las líneas 35 hasta la 39 describe el proceso para recorrer la cola completa comparando el valor de las prioridades entre el nuevo dato y la cola para insertar el nuevo dato en su posición correspondiente simulando así el correcto funcionamiento de la PIFO.

Finalmente, para verificar en este caso el correcto funcionamiento, además de imprimir por pantalla el contenido de la cola en todo momento se comparan las variables `DatoEsperado` y `Maximo`. De esta manera, si ambos son iguales quiere decir que el elemento eliminado ha sido el correcto. Un ejemplo de este caso se muestra en la Figura 15.



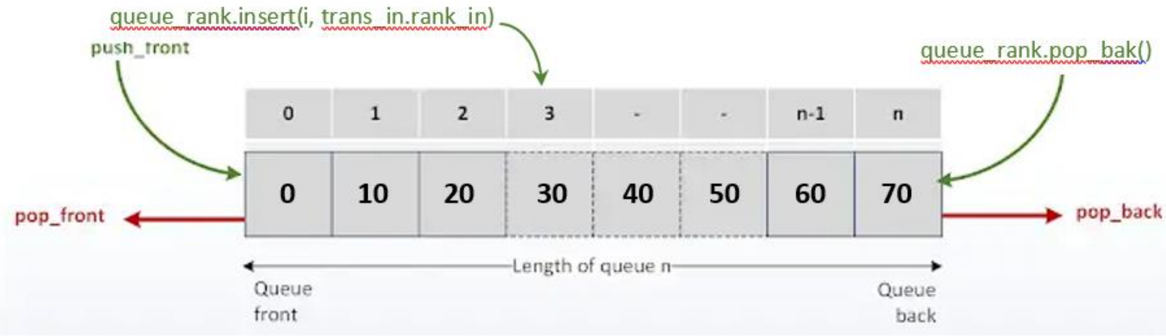


Figura 15 Mecanismo de inserción mediante método `Insert`

El tercer escenario es que el nuevo dato tenga la mínima prioridad, pero la PIFO sigue sin estar llena. En este caso, el nuevo elemento se inserta al final de la cola del *scoreboard*, ya que, el *scoreboard* debe comportarse según el diseño de la PIFO. De esta manera, conociendo que el nuevo dato es el de menor prioridad, la inserción se hace directamente al final de la cola gracias a las funciones que proporciona *SystemVerilog* para el manejo del tipo de datos de colas; tal y como se muestra en la Figura 16.

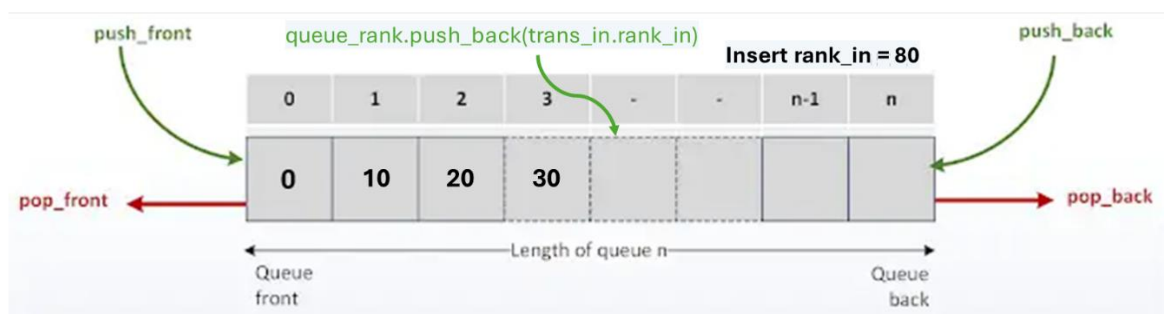


Figura 16 Mecanismo de inserción mediante método `push_back`

En el Código 12 se muestra la programación de este escenario. En la línea 52 se valora la condición de que la prioridad del nuevo dato sea menor que la del último elemento de la cola y que, además, la cola no esté llena. En la línea 54, como en los demás casos, se mantiene el registro del valor de la mínima prioridad y se inserta al final de la cola. Como en todos los casos anteriores también se imprime por pantalla el contenido de la cola para facilitar la depuración del código.

```
50 //si el nuevo rank es el de mayor prio y no está lleno se inserta al
51 final
52 else if((trans_in.rank_in>queue_rank[queue_rank.size()-1]) &&
53 queue_rank.size()!=QUEUE_WIDTH)begin
54     Maximo = trans_in.max_rank_out;
55     queue_rank.push_back(trans_in.rank_in);
56     $display("\t [SCB] Push_back queue_rank = %p", queue_rank);
57 end
```

Código 12 Componente *Scoreboard*: Tarea `insert()` – 3er caso

Por último, en cualquier otro escenario, el nuevo dato se inserta en su orden correspondiente, tal y como muestra la Figura 17, donde una nueva instrucción de *insert* se realiza cuando la cola no se encuentra llena y la prioridad del dato no es la máxima.

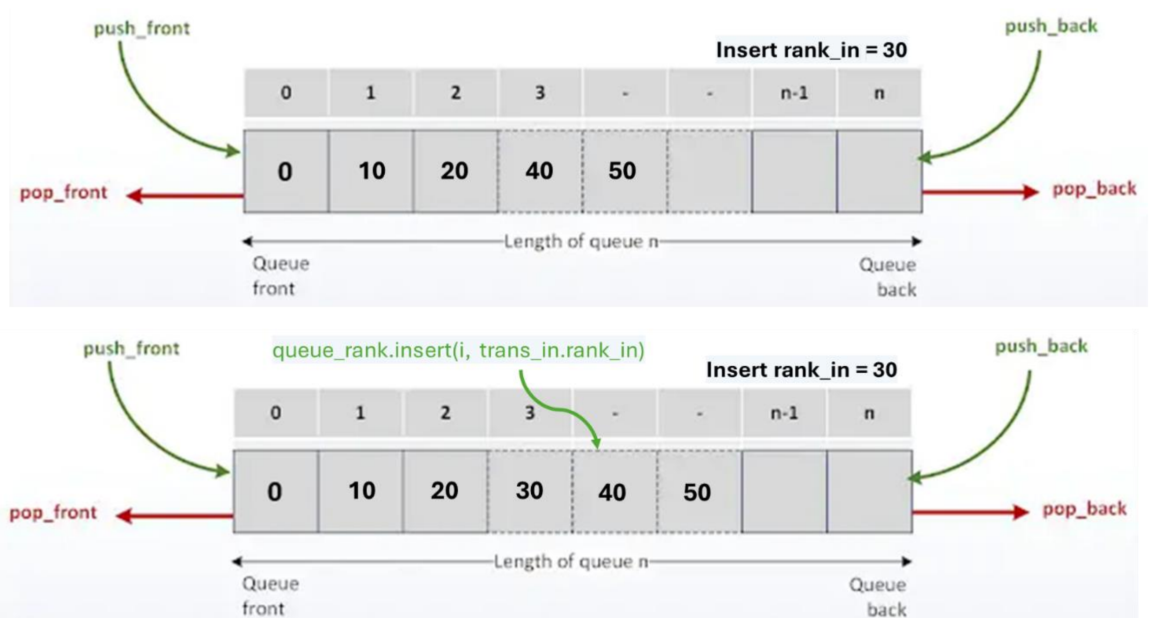


Figura 17 Mecanismo de inserción natural de un dato

El Código 13 muestra cómo se recorre la cola comparando las prioridades de los datos en la cola con el nuevo dato a insertar. En la línea 63, cuando se detecta que la prioridad del nuevo dato sea menor que la del siguiente en la cola, el nuevo dato es insertado en dicha posición.

```
58 //sino, se inserta ordenado
59     else begin
60       Maximo = trans_in.max_rank_out;
61       foreach(queue_rank[i])
62         if(trans_in.rank_in<queue_rank[i])begin
63           queue_rank.insert(i,trans_in.rank_in);
64           break;
65         end
66       $display("\t [SCB] Insert queue_rank = %p", queue_rank);
67     end
68 endtask
```

Código 13 Componente *Scoreboard*: Tarea `insert()` – 4o caso

Por otro lado, la tarea `remove()` se ejecuta cuando el *scoreboard* reciba una transacción a través del `mailbox` del monitor encargado de la interfaz `REMOVE`. Una vez se detecta una

operación de remove(), el componente *scoreboard* ejecuta la siguiente tarea descrita en el Código 14:

```
70  task remove();
71      //se elimina el rank de menor prioridad
72      if(queue_rank.size()>0)begin
73        DatoEsperado = queue_rank.pop_front();
74        if(DatoEsperado == trans_out.rank_out)begin
75          $display("\t [SCB-PASS!] DatoEsperado = %0d, DatoEliminado =
76  %0d", DatoEsperado, trans_out.rank_out);
77          $display("\t [SCB] Remove queue_rank = %p", queue_rank);
78        end
79        else begin
80          $error("\t [SCB-FAIL!] DatoEsperado = %0d, DatoEliminado =
81  %0d", DatoEsperado, trans_out.rank_out);
82          $display("\t [SCB] Remove queue_rank = %p", queue_rank);
83        end
84      end
85      else begin
86        $display("\t [SCB-PASS!] queue_rank.size() = %0d y
87  trans_out.empty = %0d", queue_rank.size(), trans_out.empty);
88        $display("\t [SCB] Remove queue_rank = %p", queue_rank);
89      end
90   endtask
```

Código 14 Componente *Scoreboard*: Tarea remove()

El comportamiento de la PIFO cuando se ejecuta una operación de REMOVE es eliminar el dato de mayor prioridad de la PIFO, que justamente es el dato situado en el *front* de la cola. Gracias a utilizar la cola en el *scoreboard*, en la tarea remove tan solo hay que eliminar el último dato insertado en la cola. En la línea 72, en primer lugar, se comprueba que la cola no esté vacía, de tal manera que si se produce una instrucción de REMOVE mientras la cola está vacía la respuesta del *scoreboard* es mostrar por pantalla el tamaño de la cola y mostrar el valor de la señal EMPTY con el fin de verificar que se ha producido una operación de REMOVE y que tanto la cola como la PIFO se encuentran vacías.

En caso de no estar vacía, el comportamiento se basa en comparar el valor de la variable DatoEsperado, donde se ha ido guardando el dato eliminado de la cola (línea 73) y compararlo con el de mayor prioridad de la PIFO haciendo uso de la señal RANK_OUT (línea 74). En caso de ser iguales, entonces se muestra por pantalla que la eliminación ha sido correcta y se muestra el contenido de la cola.

Por último, en el Código 15, se muestra la tarea main, la cual es responsable de coordinar las operaciones de inserción y extracción de transacciones en la cola de prioridades. Estas operaciones se ejecutan en paralelo mediante el uso del mecanismo fork-join_any, lo que permite procesar simultáneamente las transacciones de entrada y salida. En la línea 95 se bloquea la ejecución de las operaciones de inserción hasta que se reciba una transacción de entrada. Por su

lado, en la línea 101 se bloquea las operaciones de extracción hasta que no se reciba una transacción de extracción enviada por los generadores correspondientes. Una vez recibida las solicitudes, se ejecuta las tareas `insert()` o `remove()` y se incrementa el valor de la variable `num_trans_in`, para las operaciones de inserción, y/o `num_trans_out`, para las operaciones de extracción.

```
 91  task main;
 92    fork
 93      // Transacciones de INSERT
 94      forever begin
 95        mon2scb_in.get(trans_in);
 96        insert();
 97        num_trans_in++;
 98      end
 99      // Transacciones de REMOVE
100      forever begin
101        mon2scb_out.get(trans_out);
102        remove();
103        num_trans_out++;
104      end
105    join_any
106  endtask
107
```

Código 15 Componente *Scoreboard*: Tarea `main()`

2.3.7. Componente *Agent*

El siguiente componente es el agente, quien se encarga de agrupar en un solo componente los subcomponentes *generator*, *driver* y *monitor*. Además, gestiona la sincronización y comunicación entre ellos. En el Código 16 se declara la clase agente y las referencias de las demás clases, `generator_in`, `driver_in` y `monitor_in` (líneas 4-6). En las líneas 8 y 9 se declaran los `mailbox` para la comunicación entre generador y el driver `gen2driv_in` para la interfaz de inserción, y entre el monitor y el *scoreboard* `mon2scb_in` para la interfaz de extracción. En la línea 11 se declara el evento para la sincronización entre el generador y el test que se pasará por parámetro a la hora de crear el generador. La línea 13 contiene la declaración de la interfaz virtual para conectar el agente con la PIFO. Y desde la línea 15 hasta la 26 se define el constructor de la clase `agent_in`. Dentro del constructor se inicializan las variables de la interfaz, se crea los objetos correspondientes a los `mailbox` así como los componentes `generator`, `driver` y `monitor`.

```
1  class agent_in;
2
3    //generator and driver instance
4    generator_in gen_in;
5    driver_in driv_in;
6    monitor_in mon_in;
7    //mailbox handle's
8    mailbox gen2driv_in;
9    mailbox mon2scb_in;
10   //event for synchronization between generator and test
11   event gen_ended;
12   //virtual interface
13   virtual intf_in vif_in;
14   //constructor
15   function new(virtual intf_in vif_in);
16     //get the interface from test
17     this.vif_in = vif_in;
18     //creating the mailbox (Same handle will be shared across
19 generator and driver)
20     gen2driv_in = new();
21     mon2scb_in  = new();
22     //creating generator and driver
23     gen_in   = new(gen2driv_in,gen_ended);
24     driv_in = new(vif_in,gen2driv_in);
25     mon_in   = new(vif_in,mon2scb_in);
26   endfunction
```

Código 16 Componente `Agent_in` del módulo PIFO en *SystemVerilog*

En el Código 17 se describe las tareas de preconfiguración, ejecución y el post procesamiento. Además, se describe la tarea que invoca las tres anteriores. En la línea 27 se describe la tarea `pre_test()`, que ejecuta la tarea `reset` ya explicada anteriormente en el componente *driver* para establecer las señales de la interfaz a su valor por defecto.

```
27 task pre_test();
28     driv_in.reset();
29   endtask
30
31   task test();
32     fork
33     gen_in.main();
34     driv_in.main();
35     mon_in.main();
36     join_any
37   endtask
38   task post_test();
39     wait(gen_ended.triggered);
40     wait(gen_in.repeat_count == driv_in.num_trans);
41   endtask
42
43   task run;
44     pre_test();
45     test();
46     post_test();
47   endtask
48 endclass
```

Código 17 Agent_in - Tareas PIFO *SystemVerilog*.

En la línea 31, la tarea test() hace uso del mecanismo fork para, de manera concurrente, ejecutar las tareas principales del generator, driver y monitor, simulando así el comportamiento realista y simultáneo del sistema. En la línea 38, la tarea post_test() comienza esperando a que se active el evento gen_ended. Una vez activado, verifica que el número de transacciones generadas sea igual al número de transacciones conducidas. Por último, la tarea run que ejecuta las demás tareas y que se utilizará para inicializar el agente.

2.3.8. Componente *Environment*

El componente *environment,* al tratarse de un nivel superior en la jerarquía de componentes, agrupa el entorno completo, el agente de entrada, el agente de salida y el componente *scoreboard*, haciendo uso de las interfaces virtuales correspondientes. De la línea 3 a la línea 8 del Código 18 se declaran los apuntadores de las clases agent_in, agent_out y scoreboard. De esta manera se puede decir que el componente *environment* aglutina la generación de estímulos, el envío de las transacciones, la monitorización de las señales y la verificación de los resultados.

```
1  class environment;
2    //instancia generator y driver
3    agent_in age_in;
4    agent_out age_out;
5    scoreboard scb;
6    //interfaz
7    virtual intf_in vif_in;
8    virtual intf_out vif_out;
9    //Constructor
10   function new(virtual intf_in vif_in, virtual intf_out vif_out);
11     this.vif_in=vif_in;
12     this.vif_out=vif_out;
13     age_in = new(vif_in);
14     age_out = new(vif_out);
15     scb=new(age_in.mon2scb_in, age_out.mon2scb_out);
16   endfunction
17   //run task
18   task run;
19     fork
20       age_in.run();
21       age_out.pre_test();
22       scb.main();
23     join_any
24     wait(age_in.gen_in.repeat_count == scb.num_trans_in);
25     fork
26       age_out.run();
27       scb.main();
28     join_any
29     wait(age_out.gen_out.repeat_count == scb.num_trans_out);
30     $finish;
31   endtask
32
33 endclass
```

Código 18 Componente *environment* del módulo PIFO en *SystemVerilog*

De la línea 11 a 17 se define el constructor de la clase, donde se inicializan las interfaces virtuales, se crea los componentes de los agentes y el *scoreboard*. Además de crear el *scoreboard*, en la línea 16, se establecen las conexiones necesarias pasando por parámetros los apuntadores de los `mailbox` correspondientes.

En la línea 19 se define la tarea `run`, que coordina la ejecución del entorno de verificación. Esta tarea hace uso del mecanismo `fork` y `join_any` de manera que se crean múltiples hilos de ejecución en paralelo. En la línea 21 se ejecuta la tarea `run` del agente de entrada y en la línea 23 se ejecuta la tarea `main` del *scoreboard*. En la línea 25 espera a que el número de repeticiones generadas por el generador del agente de entrada sea igual al número de transacciones de entrada procesadas por el *scoreboard*, asegurando que todas las transacciones generadas por el agente de entrada en este caso hayan sido procesadas antes de continuar. El resto del código es exactamente lo mismo, pero para el agente de salida.

2.3.9. Programa *Test* y módulo *TestBench Top*

El Test (Código 19), a diferencia del resto de componentes, se trata de un objeto program de SystemVerilog y no una clase. El objetivo es definir el entorno de prueba en *SystemVerilog* para verificar la PIFO usando los agentes y el *scoreboard*, especifica el punto de entrada para la simulación y proporciona el contexto para la prueba, incluyendo las interfaces virtuales.

```
1  program test(intf_in intf_in, intf_out intf_out);
2
3      //declaración environment
4      environment env;
5      initial begin
6      //creacion de environment
7      env=new(intf_in, intf_out);
8      //configuracion del numero de transacciones a ser generadas
9      env.age_in.gen_in.repeat_count = 20;
10     env.age_out.gen_out.repeat_count = 20;
11     //inicialización de los estimulos
12     env.run();
13   end
14
15 endprogram
```

Código 19 Program Test del módulo PIFO en *SystemVerilog*.

En la línea 1 se define el programa `test` en *SystemVerilog.* Se trata de un objeto estático especial diseñado para contener el código de prueba que interactúa con el DUV. Permite separar claramente el código de prueba o *test* y el de diseño, lo cual es importante para evitar que el código de verificación que se está desarrollando interfiera con la funcionalidad de la PIFO debido, por ejemplo, a problemas de sincronización. El programa, a diferencia de una tarea en el interior de una clase, se ejecuta en una región de simulación diferente llamada la *program region*, que se encuentra después de la evaluación de la lógica combinacional y antes de que los valores se

propaguen a las señales en la siguiente región de simulación. Esto asegura que el código definido en el programa test se ejecute en una fase específica del ciclo de simulación, lo que ayuda a evitar problemas de *race conditions* [16].

En la línea 4 se declara el apuntador del componente *environment*. A partir de la línea 5 se define el bloque `inicial` de *SystemVerilog*. En la línea 7 se crea el entorno pasando como parámetros las interfaces virtuales y, posteriormente se configura el número de transacciones. En este caso particular cada uno de los generadores generarán 20 transacciones dentro de cada agente. Y, por último, en la línea 12 se llama a la tarea `run` del entorno para iniciar el test.

Por último, hay que definir el módulo *testbench* Top, Código 20. El módulo `tbench_top` es el archivo con mayor nivel en la jerarquía de componentes. En este nivel se conecta el DUV con el *testbench*, por lo que es necesario referenciar el DUV a verificar. Además, hay que referenciar el `Test` previamente mencionado, así como las interfaces. Además de referenciar, estas últimas conectan a nivel de señal el DUV con el entorno de verificación.

```
1  module tbench_top;
2    parameter RST_WIDTH = 20;
3    bit clk; //declaración de señales reset y clk
4    bit reset;
5    always #5 clk=~clk;    //generacion clk
6    initial        //generacion reset
7           begin
8             reset = 0;
9           #RST_WIDTH
10            reset = 1;
11          #RST_WIDTH reset = 0;
12          end
13   intf_in intf_in(clk, reset);      //creacion instancia interfaz
14   intf_out intf_out(clk, reset);
15   test t1(intf_in, intf_out);      //instancia testcase
16   pifo_reg DUT (         //creacion DUT y conectar señales interfaz
17      .rst              (reset),
18     .clk               (clk),
19     .full              (intf_in.full),
20     .insert            (intf_in.insert),
21     .rank_in           (intf_in.rank_in),
22     .meta_in           (intf_in.meta_in),
23     .valid_out         (intf_out.valid_out),
24     .remove            (intf_out.remove),
25     .rank_out          (intf_out.rank_out),
26     .meta_out          (intf_out.meta_out),
27     .max_valid_out     (intf_out.max_valid_out),
28     .max_rank_out      (intf_in.max_rank_out),
29     .max_meta_out      (intf_in.max_meta_out),
30     .num_entries       (intf_in.num_entries),
31     .empty             (intf_out.empty)
32   );
33   initial begin           //generacion del dump
34   $dumpfile("dump.vcd"); $dumpvars;
35   end
36 endmodule
```

Código 20 Componente `testbench` del módulo PIFO en *SystemVerilog*.

En las líneas 2, 3 y 4 se declaran las señales de *reset* y reloj además de definir la duración del pulso de *reset* a partir del parámetro `RST_WIDTH`. En la línea 5 se genera la señal de reloj con un periodo de 10 unidades de tiempo. Entre las líneas 6 – 12 se genera la señal de *reset.*

Las líneas 13 y 14 crean las interfaces tanto de entrada como salida y en la línea 16 se referencia la PIFO. En esta referenciación, se conecta sus puertos a las señales de las interfaces correspondientes. Por último, entre las líneas 33 – 35 se genera un archivo de volcado de simulación que permite la visualización de las señales en formas de onda para analizar el comportamiento de la PIFO.

2.4. Conceptos generales de la metodología UVM

UVM es una metodología de verificación que se basa en *SystemVerilog*. Este hecho explica que herede de este lenguaje algunas características básicas, lo que justifica el haber estudiado precisamente este lenguaje y la arquitectura típica de un *testbench* en *SystemVerilog*. Es precisamente este punto, la arquitectura de un *testbench*, uno de los principales puntos que aporta *SystemVerilog* a la metodología UVM.

La metodología UVM es hoy en día la metodología de verificación utilizada como referencia para la verificación de sistemas electrónicos. UVM es una biblioteca de clases que facilita la creación de componentes y objetos modulares, parametrizables y reutilizables para la creación de entornos de verificación. Basada en *SystemVerilog*, utiliza los principios de la programación orientada a objetos, como es el patrón de *Factory*; utiliza una comunicación basada en transacciones (TLM) entre componentes; define un potente y parametrizable sistema de mensajes para facilitar la depuración; y añade un mecanismo de fases para controlar todo los pasos involucrados en el proceso de verificación, desde la creación del *testbench* hasta la comprobación de los resultados, pasando por las etapas de conexionado y ejecución entre otros.

En este punto se presenta, muy brevemente, cada una de estas aportaciones.

2.4.1. Biblioteca de clases UVM

Para construir el entorno de verificación de manera estructurada y que permita ser una metodología estandarizada, son necesarios una serie de elementos activos, como los componentes (*drivers, sequencers, monitors,* etc.) y otros elementos pasivos, denominados objetos, como las transacciones. UVM facilita esta tarea proporcionando una librería de clases. En la Figura 18 se puede ver cómo se estructura la biblioteca de clases en UVM siguiendo un orden jerárquico donde

unas clases van derivando de otras. Estos componentes y objetos son controlados a través de un conjunto de fases (*phases*) para inicializar, ejecutar y completar cada uno de los test.

En la Figura 18 se representa el árbol de herencia de las principales clases base de UVM, aunque existen muchas otras clases. El árbol de clases se estructura en grupos, según su finalidad.

- Manejo de estímulos. Las clases más importantes en este campo son `uvm_sequence`, `uvm_sequence_library`, `uvm_sequence_library`, `uvm_sequence_item`, ambos heredan de la clase base `uvm_object`. Están asociados a los estímulos utilizados por el *testbench* para enviar y/o recoger del DUV.
- Componentes de configuración. Las clases más importantes en este grupo son aquellos asociados a los componentes ya definidos en `uvm_driver`, `uvm_sequencer`, `uvm_monitor`, `uvm_agent`, `uvm_scoreboard`, `uvm_env`, `uvm_test`. Estas clases están definidas con funciones que implementan el comportamiento básico del componente que modelan. Todos ellos heredan de `uvm_component`.
- Configuración. Las clases más importantes en este grupo son `uvm_config_db` y `uvm_resource`. Todos ellos heredan de `uvm_object`. Estas clases permiten el acceso a la base de datos de configuración para, de forma dinámica, realizar cambios en la configuración de un *testbench* y, sobre todo, realizar el paso de objetos entre los diferentes componentes de un *testbench.*
- Capa de registros. Las clases más importantes en este grupo son `uvm_reg`, `uvm_reg_block`, `uvm_reg_file`, `uvm_reg_field` y `uvm_mem`. El uso principal de estas clases es el de crear dentro del *testbench* una réplica del banco de registros del sistema a verificar. Este grupo no será tratado en el desarrollo de este TFM.

Además de estas clases, se definen otras más especializadas para, por ejemplo, el manejo de puertos relacionados con la comunicación, clases de informes para gestionar las tareas de *debug*, clases de fábrica para manejar la configuración del *testbench*, entre otros. Algunas de estas clases se explicarán en detalle en los próximos capítulos.

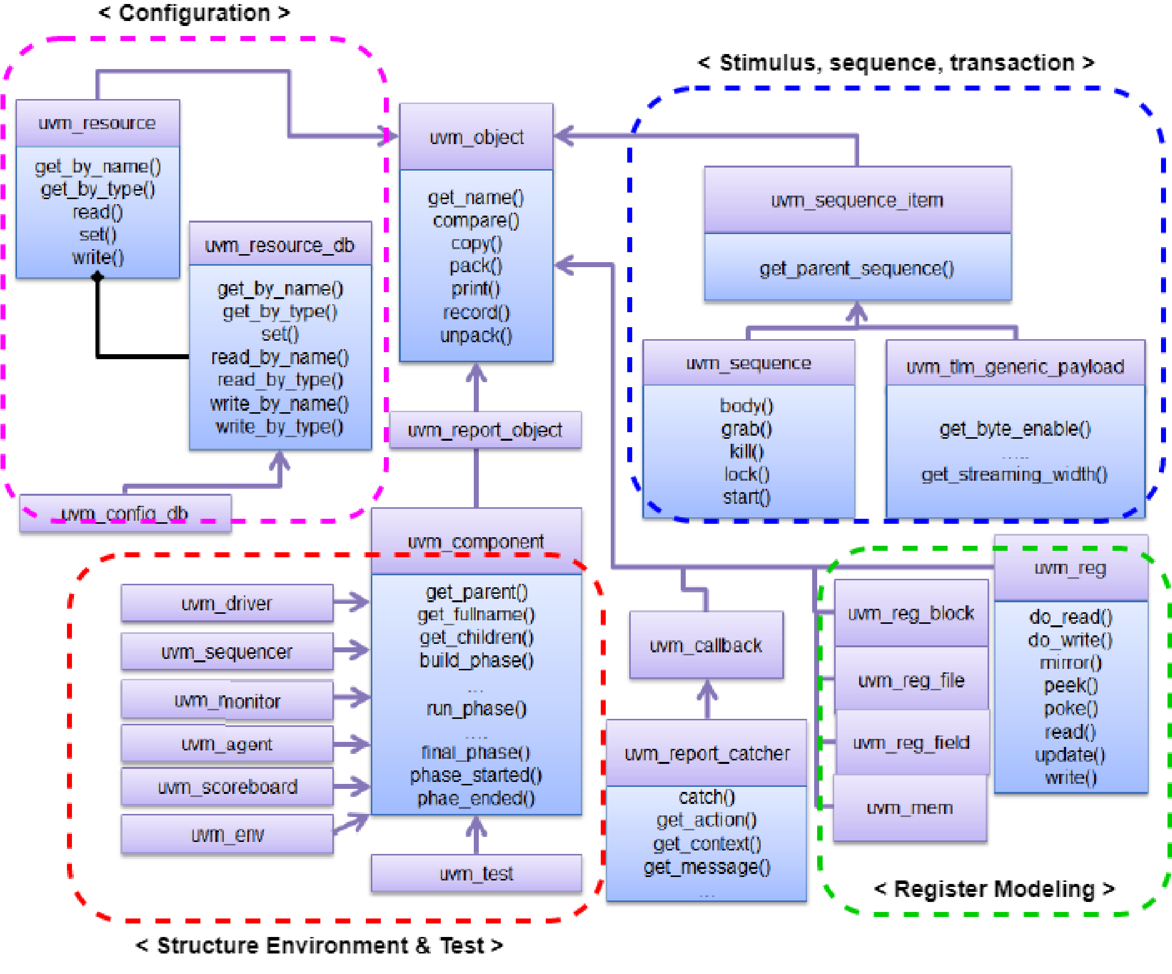


Figura 18 Jerarquía de la biblioteca de clases UVM

2.4.2. Mecanismo de fases en UVM

A diferencia de *Verilog*, donde todos los componentes del banco de pruebas se describen en un contenedor estático llamado módulo, UVM se basa en *SystemVerilog* y por tanto utiliza contenedores dinámicos, características de programación orientada a objetos, por lo que su construcción y procesamiento varían desde el inicio de la simulación. Para garantizar la reutilización y escalabilidad del entorno de verificación en UVM, la sincronización general entre componentes dinámicos es un requisito importante. En la metodología UVM esta sincronización se consigue a través del mecanismo de fases. Las fases y los dominios se heredan de la clase base `uvm_component` y se utilizan para llamar a funciones y tareas en un orden predefinido y, según este orden, el usuario realiza la operación correspondiente para cada fase en su componente, que habrá sido creado a partir de su clase base. En otras palabras, las clases que se extienden desde `uvm_component` ejecutan métodos correspondientes a una serie de fases en forma de devoluciones de llamada. En este momento, las tareas que no requieren consumo de tiempo de simulación se implementan como `function()`, mientras que aquellas que consumen tiempo de

simulación se implementan como `task()`. Las fases se ejecutan en el orden que muestra la Figura 19.

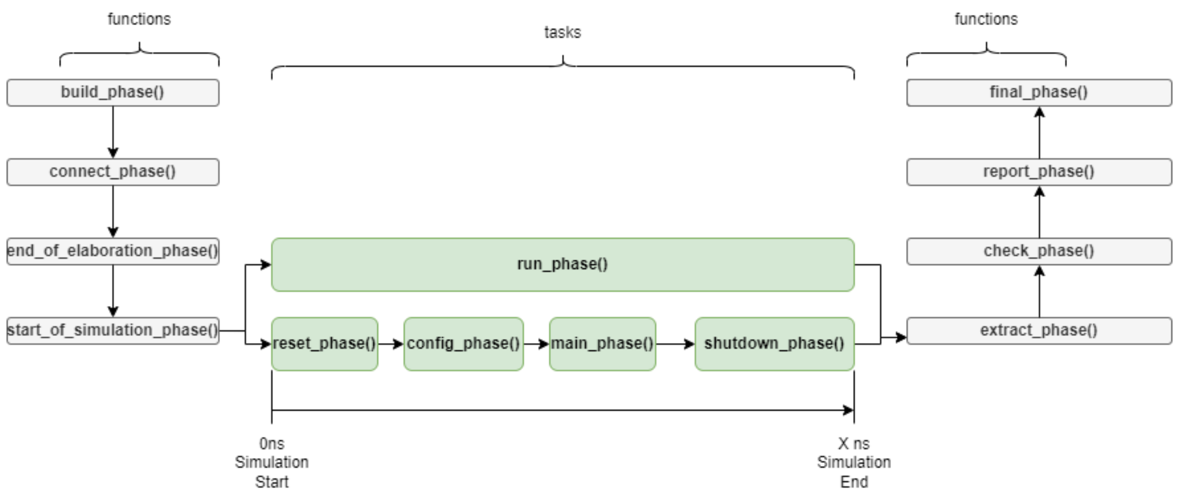


Figura 19 Fases UVM y orden ejecución

A continuación, se detallan brevemente casa una de las principales fases que proporciona UVM para la realización del test de un sistema.

- **`build_phase()`.** Esta es la primera fase del flujo de ejecución. Consiste en la creación de los componentes UVM y su jerarquía siguiendo una filosofía *top-down*, es decir, la jerarquía se construirá desde el nivel más alto al más bajo.
- **`connect_phase()`.** La función de esta fase es realizar el conexionado de los componentes creados en la `build_phase()`. En este caso, se sigue una filosofía *bottom up*, donde se conectarán primero los componentes de más bajo nivel hasta llegar al nivel más alto de la jerarquía.
- **`end_of_elaboration_phase()/start_of_simulation_phase()`.** Estas fases permiten mostrar en pantalla información de relevancia tras la creación y conexionado de los componentes, como puede ser la topología final de la jerarquía del entorno o información de su configuración.
- **`run_phase()`.** Es la única fase que es implementada como una tarea (*task*) y consume tiempo, a diferencia de las fases anteriores, que se ejecutan en tiempo cero. Los eventos o hilos que se necesita que ocurran durante la ejecución de la simulación deberían ir todos ellos en la fase `run_phase`. En esta fase se describe la implementación de los componentes que conforman el entorno de verificación. Como se observa en la Figura 19, la fase `run_phase` engloba otras sub-fases. Sin embargo, para un test sin mucha complejidad, estas no resultan de utilidad. Todo el código de esta fase está bajo la única denominación de `run_phase`. Debido a que esta fase se trata de una tarea que se

ejecuta durante el tiempo que dure la simulación, resulta conveniente poder controlar o determinar cuándo finaliza la ejecución. Para ello se utiliza el mecanismo de *objections*.

La clase `uvm_objection` proporciona contadores para conocer cuántos componentes y secuencias siguen activos durante la fase `run_phase`. Cada uno de estos componentes puede activar (*raise*) o eliminar (*drop*) *objections* de forma asíncrona, lo que aumenta o disminuye el valor del contador. Una vez el contador de *objections* llega a cero, se produce una condición denominada "*all dropped*" y se da por finalizada la simulación. Para controlar los *objections* se usan tres métodos: `raise_objection()`, que señaliza cuando un componente inicia la fase de `run_phase`; `drop_objection()`, que señaliza cuando un componente finaliza la fase de `run_phase`, y el método `set_drain_time()`, que permite, una vez alcanzado la condición "*all dropped*", esperar un tiempo adicional con la intención de sincronizar los componentes y dar un tiempo extra por si algún recurso o dato no haya terminado de ejecutarse.

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

2.4.3. Mecanismo de *Factory*

El objetivo principal de este mecanismo, a veces conocido como patrón *Factory,* es permitir la creación dinámica de objetos durante la simulación sin depender de tipos específicos en el código fuente. El mecanismo *Factory* permite crear objetos en tiempo de simulación basándose en nombres de tipos, lo que proporciona una gran flexibilidad al entorno de verificación. Esto es particularmente útil para la sustitución y extensión de componentes sin cambiar el código existente, lo que también evita la necesidad de compilar de nuevo todo el sistema.

En este mecanismo antes de crear un tipo, este debe estar registrado en el *Factory*. Los tipos de componentes y objetos de transacción se registran típicamente en el constructor estático (*static function*) de la clase usando métodos como `uvm_component_utils` y `uvm_object_utils`. Además, el uso del mecanismo *Factory* permite la sustitución de un tipo registrado por otro. Esto es útil para reemplazar componentes por otros especializados o para introducir *mocks* o *stubs* sin modificar el código original. Estas macros de remplazo de componentes implementan, primero, la función `get_type_name()`, que devolverá el objeto o componente como una variable de tipo *string*. Tras esto, ejecutan la función `create()` como constructor y

después registran el objeto o componente en la *Factory* utilizando la variable tipo *string* generada al comienzo de la macro.

2.4.4. Mecanismo de mensajes

El mecanismo de mensajes en UVM es una herramienta fundamental para la comunicación, el registro y la depuración dentro de un entorno de verificación. Este sistema facilita la generación, el control y la gestión de mensajes de diferentes tipos (informativos, de advertencia, de error, etc.), permitiendo a los ingenieros obtener información detallada sobre el comportamiento del sistema DUV y el entorno de verificación.

UVM proporciona una serie de macros que simplifican la generación de mensajes que pueden ser utilizados con diferentes niveles de prioridad. Independientemente de la macro a utilizar, todas comienzan por `uvm_report_` y, seguidamente, el tipo de mensaje que puede ser `_info`, `_error`, `_warning`, `_fatal`. Como se comentó, UVM tiene hasta 6 niveles para cada mensaje indicando el nivel de importancia. El formato de estas macros es el que se muestra:

- `uvm_report_*(“TAG”,$sformatf(“[Message]”), VERBOSITY_LEVEL);`

El nivel de prioridad se indica a través del parámetro `VERBOSITY_LEVEL`. Este nivel puede tomar los siguientes valores en orden creciente: `UVM_NONE`, `UVM_LOW`, `UVM_MEDIUM`, `UVM_HIGH`, `UVM_FULL` y `UVM_DEBUG`.

2.4.5. Modelado y comunicación TLM

Una de las características más importantes de UVM es la abstracción en la comunicación entre componentes, sin preocuparse por detalles de bajo nivel como señales y sincronización. Sin contar la interfaz con el DUV, la comunicación entre los componentes de verificación se realiza a nivel de transacciones. Una transacción en UVM es una clase que contiene toda la información para describir la actividad de bajo nivel del protocolo utilizado para comunicarse con el DUV. Para ello, UVM adopta y utiliza el estándar TLM (*Transaction Level Modeling*)[17]. En este estándar existen diferentes tipos de conexiones para las diferentes comunicaciones entre componentes. Dependiendo, además, de la complejidad del entorno, en determinadas arquitecturas hará falta utilizar conexiones más elaboradas.

`Connect()`: UVM proporciona una capa de interfaz para el manejo de la comunicación TLM. Esta interfaz contiene diferentes tipos de puertos, cuyo uso depende del tipo de componente y del hilo de uso que se le vaya a dar. Además, todos estos puertos proporcionan diferentes métodos que posibilitan la comunicación entre componentes.

La comunicación TLM implica la existencia de dos componentes, un productor y uno o varios componentes consumidores. Atendiendo a cómo se establece la comunicación.

A continuación, se muestra los modos comúnmente utilizados:

- **Modo push:** En este método, una transacción se pasa llamando al método `put()` del puerto definido en el componente productor. La implementación del método `put()` se realiza en el componente consumidor. Este método se ejecuta gracias a la conexión realizada entre productor y consumidor para recibir y procesar la transacción pasada. Este método se utiliza para la conexión entre los componentes `sequencer` y `driver`.

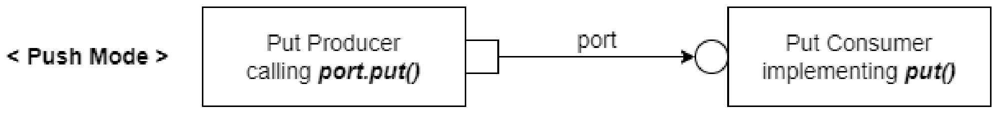


Figura 20 TLM modo *Push*

- **Modo pull:** En este método, es el componente consumidor quien inicia la transferencia llamando al método `get()` del puerto. Es el componente productor quien implementa en este caso el método `get()` de acuerdo con la conexión realizada y envía la transacción. Este método se utiliza para la conexión entre los componentes `sequencer` y `driver`.

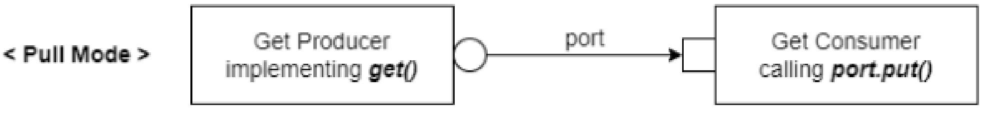


Figura 21 TLM modo *Pull*

- **Modo FIFO:** Se utiliza para intermediar entre el productor y el consumidor, donde el productor genera transacciones y las envía a la FIFO utilizando el método `put()`. Cada llamada al método `put()` agrega una nueva transacción al final de la FIFO. El consumidor recibe las transacciones desde el FIFO utilizando el método `get()`. Cada llamada al método `get()` extrae la transacción que está en la parte frontal del FIFO, manteniendo el orden en que las transacciones fueron añadidas. El canal FIFO (`uvm_tlm_fifo`) actúa como un buffer que almacena las transacciones temporalmente. Este canal asegura que las transacciones se entreguen en el orden en que fueron recibidas.

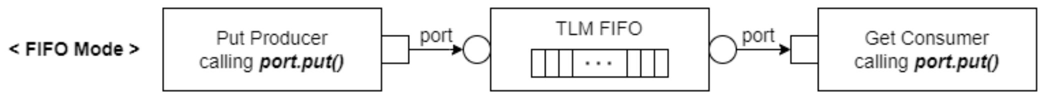


Figura 22 TLM modo FIFO

- **Modo Broadcast:** Este método, también conocido como modo de transmisión (comunicación de análisis) se produce cuando el productor llama al método `write()` y comparte una transacción. Los consumidores conectados a él reciben y procesan la transacción compartida implementando el método `write()`. En este caso, a diferencia de los métodos `put()` y `get()`, en lo que la comunicación era uno a uno, en este caso los componentes se denominan Productor y Suscriptor y el único método que interviene es el método `write()`. Se utiliza para la conexión `monitor` y `scoreboard`.

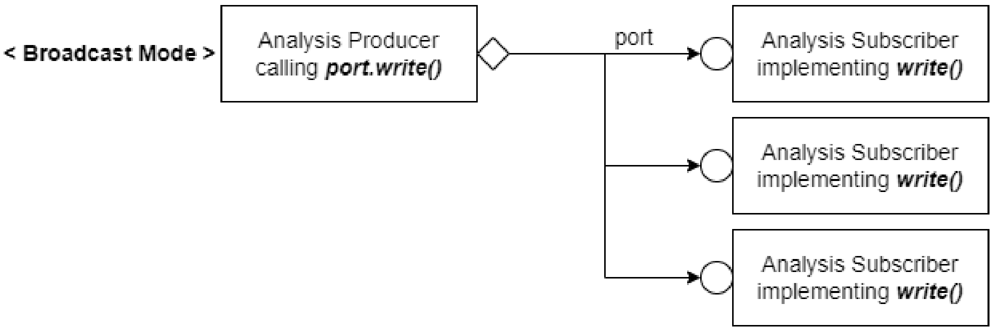


Figura 23 TLM modo *Broadcast*

2.5. Entorno de verificación UVM.

Como se explicó en el apartado 2.3, UVM está basado, en la filosofía de verificación propuesta por *SystemVerilog*. En este sentido, un entorno de verificación UVM es muy similar al descrito en la Figura 10, pero añadiendo todas las características propias de la metodología UVM. En este caso, el entorno UVM se crea a partir de la referencia de los componentes de todas las clases que heredan de `uvm_object`. Sigue siendo un entorno jerárquico y la relación entre los componentes define la posición del componente en la jerarquía del *testbench*, desde el nivel más alto hasta el más bajo.

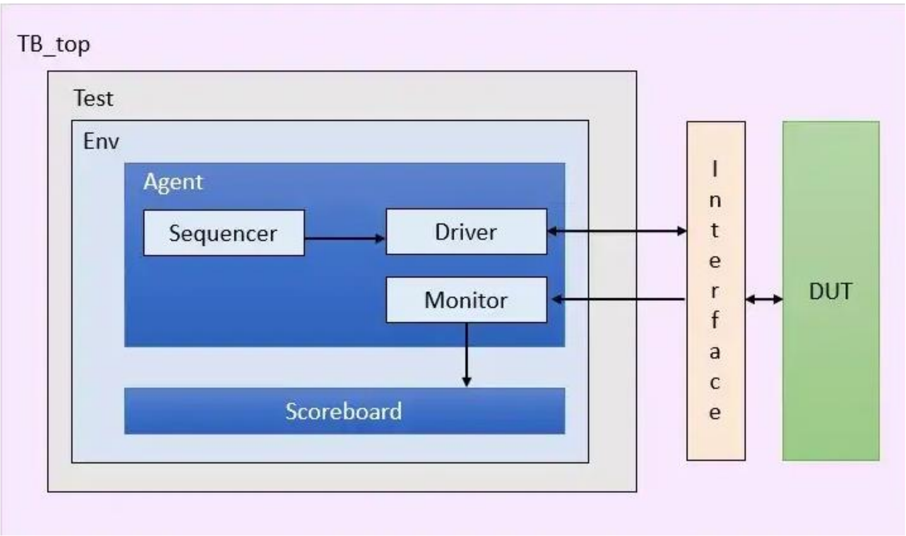


Figura 24 Modelo básico de un entorno de verificación UVM

La Figura 24 muestra un modelo de entorno UVM básico que sirve de referencia jerárquica. Se parte de un módulo principal, denominado `top`, donde se referencia el sistema que se quiere verificar y el entorno de verificación. Para que ambos se comuniquen y se pueda gestionar los estímulos y salidas del DUV, en este módulo también se referencia la interfaz de comunicación y se conecta el DUV a dicha interfaz. Dentro del entorno UVM, el nivel más alto es el componente `Test`, con la finalidad de habilitar el control de esta interfaz por parte del *testbench*, en este módulo, se introduce en la base de datos de UVM una referencia a la misma que contiene los componentes UVM `Environment` necesarios. Dentro del componente UVM `Environment` se encuentran, entre otros, los componentes de verificación UVM `Agent`. Estos últimos componentes representan la unidad básica, a nivel de reusabilidad, de cualquier entorno de verificación. Por su parte, un componente UVM `Agent` está, a su vez, compuesto por un componente UVM `Sequencer`, un componente UVM *Driver* y un componente UVM `Monitor`. Todos ellos integrados en el componente UVM `Agent`.

Para la explicación de los componentes UVM se ha desarrollado un *testbench* UVM completo para la verificación del módulo PIFO. El detalle de cada uno de los componentes UVM hará uso del código desarrollado para el citado *testbench*.

2.5.1. Transacciones

La transacción `data_packet_i` es la utilizada por los componentes encargados de la gestión de las operaciones de *insert*, el Código 21 muestra la implementación de esta transacción, que deriva del objeto `uvm_sequence_item` de UVM. Su función es actuar como contenedor de variables que se utilizarán para estimular y registrar la señales del DUV. De la línea 2 a 8 se definen las señales que forman esta transacción y en la línea 10 un nuevo ítem adicional de tipo *random* con la finalidad de establecer un *delay* entre transacciones, aunque como se verá más adelante, se puede especificar un valor determinado haciendo uso de macros aún haya sido definido como *random*. En la línea 11, además, se define la restricción de que los valores aleatorios de *delay* deben estar entre 1 y 5.

Por otro lado, de la línea 12 a 21 se registra el objeto y las variables en la *Factory* de UVM con las etiquetas en modo `DEFAULT`, permitiendo que los campos sean automáticamente manipulados por las utilidades UVM. Entre las líneas 22-24 se define el constructor `new`. Este constructor invoca a la función `new` de su clase padre, `uvm_sequence_item`, mediante el *token* "super". Y por último, entre las líneas 25-30, se define la tarea `displayAll()`. Esta tarea hace uso de la macro `uvm_info` para mostrar los datos que contiene el paquete.

```
1  class data_packet_i extends uvm_sequence_item;
2      bit insert;
3      rand bit [15:0] rank_in;
4      rand bit [11:0] meta_in;
5      bit full;
6      bit [15:0] max_rank_out;
7      bit [11:0] max_meta_out;
8      bit [4:0] num_entries;
9      //delay utilizado entre transacciones
10     rand int delay;
11     constraint timing {delay inside {[1:5]};}
12    `uvm_object_utils_begin(data_packet_i)
13        `uvm_field_int(insert, UVM_DEFAULT)
14        `uvm_field_int(rank_in, UVM_DEFAULT)
15        `uvm_field_int(meta_in, UVM_DEFAULT)
16        `uvm_field_int(full, UVM_DEFAULT)
17        `uvm_field_int(max_rank_out, UVM_DEFAULT)
18        `uvm_field_int(max_meta_out, UVM_DEFAULT)
19        `uvm_field_int(num_entries, UVM_DEFAULT)
20        `uvm_field_int(delay, UVM_DEFAULT)
21     `uvm_object_utils_end
22    function new(string name = "data_packet_i");
23        super.new(name);
24     endfunction: new
25     virtual task displayAll( );
26        `uvm_info("DP", $sformatf("insert = %0h rank_in = %0h meta_in =
27 %0h full = %0d max_rank_out = %0d max_meta_out = %0d num_entries =
28 %0d delay = %0d", insert, rank_in, meta_in, full, max_rank_out,
29 max_meta_out, num_entries, delay), UVM_LOW)
30     endtask: displayAll
31 endclass: data_packet_i
```
Código 21 Testbench UVM PIFO: transacción interfaz *insert*

Para la interfaz REMOVE se implementó la transacción data_packet_r de la misma manera que para la interfaz INSERT, pero asignando los ítems deseados para dicha interfaz como son: remove, rank_out, meta_out, empty, valid_out, max_valid_out, num_entries y delay, que se muestra en el Código 22.

```
1  class data_packet_r extends uvm_sequence_item;
2      bit remove;
3      bit [15:0] rank_out;
4      bit [11:0] meta_out;
5      bit empty;
6      bit valid_out;
7      bit max_valid_out;
8      bit [4:0] num_entries;
9      rand int delay;
```
Código 22 Testbench UVM PIFO: transacción interfaz *remove*

2.5.2. Interfaz Virtual

La interfaz permite estimular y recibir respuestas del DUV, conectando los componentes *driver* para el estímulo del DUV y el componente *monitor* para recibir las respuestas del DUV. En UVM se implementa las interfaces directamente al igual que en la verificación *SystemVerilog*, esto

es mediante interfaces virtuales debido a las ventajas ya mencionadas en capítulos anteriores por lo que en este caso no se hace uso de nuevas clases para la definición de una interfaz.

La implementación se muestra en el Código 23 y como se puede ver, es igual que en el caso del desarrollado en *Systemverilog*.

```
1 interface pifo_if_i(input logic clk, reset);
2   logic insert;
3   logic [15:0] rank_in;
4   logic [11:0] meta_in;
5   logic full;
6   logic [15:0] max_rank_out;
7   logic [11:0] max_meta_out;
8   logic [4:0] num_entries;
9 endinterface: pifo_if_i
```

Código 23 Testbench UVM PIFO: Interfaz *insert*

2.5.3. Componente UVM Sequencer

El componente *sequencer* es un mediador que establece una conexión entre la secuencia y el *driver*. En última instancia, pasa transacciones o elementos de secuencia al *driver* para que puedan ser conducidos al DUV.

Se recomienda que un *sequencer* definido por el usuario se extienda desde la clase base `uvm_sequencer` línea 1 del Código 24 que está parametrizada por tipos de elementos de solicitud (REQ) y respuesta (RSP). El uso de elementos de respuesta es opcional. Por lo tanto, la mayoría de las clases de secuenciador se extienden desde una clase base que sólo tiene un elemento REQ.

```
1 class pifo_sequencer_i extends uvm_sequencer #(data_packet_i);
2   `uvm_component_utils(pifo_sequencer_i)
3    function new(string name, uvm_component parent);
4       super.new(name, parent);
5    endfunction: new
6 endclass: pifo_sequencer_i
```

Código 24 Testbench UVM PIFO: Componente UVM Sequencer interfaz *insert*

En la línea 3, se hace uso de la macro `uvm_component_utils` para registrar el componente en la *Factory*. Esto permitirá su creación usando el método `create` de la misma, en vez de tener que invocar el constructor.

Los métodos `seq_item_export` y `seq_item_port` TLM *connect* están definidos en las clases `uvm_sequencer` y `uvm_driver`, siguiendo un protocolo de comunicación *handshake* a través de los puertos TLM que se heredan de sus respectivas clases padre, este protocolo se explicará en detalle en un apartado en específico por su relevancia importancia.

El código previamente comentado corresponde a la interfaz *insert*, por lo tanto, existe otro componente *sequencer* dedicado a la interfaz de *remove* implementado en el Código 25, completamente idéntico pero que la diferencia entre ambos reside en las conexiones que se realizará más adelante con el resto de los componentes.

```
1 class pifo_sequencer_r extends uvm_sequencer #(data_packet_r);
2
3   `uvm_component_utils(pifo_sequencer_r)
4
5    function new(string name, uvm_component parent);
6       super.new(name, parent);
7    endfunction: new
8 endclass: pifo_sequencer_r
```

Código 25 Testbench UVM PIFO: Componente UVM Sequencer interfaz remove

2.5.4. Componente UVM Driver

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

Para la realización del `testbench` del módulo PIFO, es necesario implementar dos componentes driver, uno por cada interfaz; interfaz de entrada, asociada a las operaciones INSERT, e interfaz de salida, asociada a las operaciones REMOVE. El Código 26 muestra la definición del componente *Driver* desarrollado para la verificación del IP PIFO correspondiente a la interfaz de entrada. Tras la declaración de la clase, heredada de la clase base `uvm_driver`, el componente se registra en la Factory, línea 3 del Código 26 y se define su constructor, líneas 5 a 7.

```
1  class pifo_driver_i extends uvm_driver #(data_packet_i);
2    virtual pifo_if_i vif;
3   `uvm_component_utils(pifo_driver_i)//registrar driver en la factory
4    //Constructor UVM
5    function new(string name, uvm_component parent);
6     super.new(name, parent);
7     endfunction: new
8     //build_phase, se usa el uvm_config_db para obtener la int virtual
9    function void build_phase(uvm_phase phase);
10    super.build_phase(phase);
11    if(!uvm_config_db#(virtual pifo_if_i)::get(this, "", "in_intf",
12 vif))
13     `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
14 get_full_name( ), ".vif"})
15     `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
16   endfunction
```

Código 26 Testbench UVM PIFO: Componente UVM Driver - Parte I

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

El acceso a la interfaz virtual se realiza durante la fase `build`, mediante el método `get` de la clase `uvm_config_db`, tal y como se muestra en la línea 11 del Código 26. En este proceso se asigna la interfaz virtual al manejador `vif` para, posteriormente, poder activar las señales de interfaz que conectan con el DUV.

Para finalizar la descripción del componente, únicamente hace falta implementar la fase `run` o de ejecución, líneas 22 y 27. Esta fase, definida como tarea, tal y como se comentó anteriormente, ejecuta dos hilos en paralelo, gestión de reset y manejo de las operaciones de inserción.

La tarea `reset`, definida entre las líneas 29 a 34, está continuamente esperando la activación de la señal de `reset` del sistema para, una vez detectada y en sincronismo con la señal de reloj, inicializar las señales de control del interfaz de entrada.

La tarea `get_and_drive` se encarga de gestionar las transacciones recibidas desde el componente `sequencer` y enviarlas al DUV según el protocolo establecido. En la línea 45 se solicita una transacción al componente `sequencer` a través de la llamada al método `get_next_item` implementado en el puerto TLM `seq_item_port`. Una vez recibida la transacción, esta se envía al DUV haciendo uso de la tarea `drive_packet`.

La tarea `drive_packet` se define entre las líneas 50 y 58. Su función es la de ejecutar un ciclo de INSERT. Para la ejecución activa, entre las líneas 53 y 57, las señales y los buses de la interfaz de entrada del DUV, en este caso particular, la señal `insert` y los buses `rank_in` y `meta_in`.

Una vez enviada la transacción, este componente debe avisar al componente `sequencer` que ha terminado de enviarla. Este proceso se realiza en la línea 45 y es sumamente importante ya que, de no hacerlo, no se completaría el protocolo de `handshake` entre `driver` y `sequencer`. Esta operación se realizar invocando el método `item_done` implementado en el puerto TLM `seq_item_port`.

```
21 //run_phase
22  virtual task run_phase(uvm_phase phase);
23     fork
24        reset( );
25        get_and_drive( );
26     join
27  endtask: run_phase
28  //en el siguiente ciclo se resetean las señales propias al protocolo
29  virtual task reset( );
30    `uvm_info(get_type_name( ), "Resetting signals ... ", UVM_HIGH)
31     forever begin
32       @(posedge vif.reset);
33       vif.insert <= 0;      end
34  endtask: reset
35  virtual task get_and_drive( );
36     forever begin
37       @(negedge vif.reset);
38       while(vif.reset != 1'b1) begin
39          //coge el item del sequencer (bloqueante)
40          seq_item_port.get_next_item(req);
41          //envia a la interfaz el contenido de la transaccion
42          drive_packet(req);
43         `uvm_info(get_type_name,"just drove a packet", UVM_HIGH);
44          //se finaliza
45          seq_item_port.item_done( );
46         `uvm_info(get_type_name,"just returned item_doen", UVM_HIGH);
47        end
48     end
49  endtask: get_and_drive
50  virtual task drive_packet(data_packet_i pkt);
51    repeat(pkt.delay) @(posedge vif.clk);
52    `uvm_info(get_type_name,"in the drive_packet_i", UVM_HIGH);
53     vif.insert <= 1;
54     vif.meta_in <= pkt.meta_in;
55     vif.rank_in <= pkt.rank_in;
56     @(posedge vif.clk);
57     vif.insert <= 1'b0;
58  endtask
59 endclass:pifo_driver_i
```

Código 27 Testbench UVM PIFO: Componente UVM Driver - Parte II

2.5.5. Componente UVM Monitor

Un componente monitor UVM es un componente pasivo utilizado para capturar señales DUV utilizando una interfaz virtual y traducirlas a un formato de elemento de secuencia. Estos elementos de secuencia o transacciones se transmiten a otros componentes como el *scoreboard* UVM, el colector de cobertura, etc. Utiliza un puerto de análisis TLM para transmitir transacciones.

Para la realización del `testbench` del módulo PIFO, es necesario implementar dos componentes monitores, uno por cada interfaz por el mismo motivo que ocurre con el componente *driver*. El Código 28 muestra la definición del componente monitor desarrollado para la verificación del IP PIFO correspondiente a la interfaz de entrada. Tras la declaración de la clase, heredada de la clase base `uvm_monitor`, el componente se registra en la *Factory*, línea 11 del Código 28 y se define su constructor, líneas 13 a 15.

```
1  class pifo_monitor_i extends uvm_monitor;
2   virtual pifo_if_i vif;
3   string monitor_intf_i;
4   int num_pkts;
5   uvm_analysis_port #(data_packet_i) item_collected_port;
6   //data_colleted utilizado para depositar los datos de la int
7   data_packet_i data_collected;
8   //data_clone clon data_collected para enviar al analisis port
9   data_packet_i data_clone;
10
11 `uvm_component_utils(pifo_monitor_i)
12
13  function new(string name, uvm_component parent);
14     super.new(name, parent);
15  endfunction: new
16
17  function void build_phase(uvm_phase phase);
18     super.build_phase(phase);
19    if(!uvm_config_db#(string)::get(this, "", "monitor_intf_i",
20 monitor_intf_i))
21       `uvm_fatal("NOSTRING", {"Need interface name for: ",
22 get_full_name( ), ".monitor_intf_i"})
23
24       `uvm_info(get_type_name( ), $sformatf("INTERFACE USED = %0s",
25 monitor_intf_i), UVM_HIGH)
26       if(!uvm_config_db#(virtual pifo_if_i)::get(this, "",
27 monitor_intf_i, vif))
28         `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
29 get_full_name( ), ".vif"})
30
31    item_collected_port = new("item_collected_port", this);
32    data_collected = data_packet_i::type_id::create("data_collected");
33    data_clone = data_packet_i::type_id::create("data_clone");
34     `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
35  endfunction: build_phase
```

Código 28 Testbench UVM PIFO: Componente UVM Monitor interfaz *insert* - Parte I

Entre las líneas 17 a 36 se define la fase `build_phase`, donde se obtiene el nombre de la interfaz a partir de la base de datos de configuración UVM (línea 19) mediante el método `get` de la clase `uvm_config_db` y, por otro lado, se configura con el mismo mecanismo la interfaz virtual `vif` (línea 26) usando el nombre de la interfaz obtenida en la línea 19.

En la línea 31 se crea una instancia del puerto de análisis `item_collected_port` para la comunicación TLM con el componente *scoreboard*, previamente se declara el puerto mencionado en la línea 5 donde se pasa por parámetro la transacción correspondiente a la interfaz *insert* para almacenar los resultados obtenidos del DUV a dicha transacción y enviarlo a través del puerto.

Para evitar problemas de concurrencia e integridad de datos entre los componentes *monitor* y *drivers* que acceden al mismo recurso, se crea dos transacciones, la primera llamada `data_collected`, creada en la línea 32 mediante el método `create` y accediendo a la *Factory*, el objetivo de esta transacción es almacenar los datos recolectados a través de la interfaz provenientes del DUV, la segunda transacción creada de la misma manera en la línea 33 será una

copia de `data_collected` y será la enviada por el puerto hacía el componente *scoreboard*, de manera que la integridad de los datos independientemente de que ocurran cambios que peligren los datos, `data_clone` asegura que los datos originales no se vean afectados.

```
37 virtual task run_phase(uvm_phase phase);
38       collect_data( );
39    endtask: run_phase
40
41    virtual task collect_data( );
42       forever begin
43         @(posedge vif.clk);
44         wait(vif.insert)
45          data_collected.insert <= vif.insert;
46          data_collected.meta_in <= vif.meta_in;
47          data_collected.rank_in <= vif.rank_in;
48          data_collected.max_meta_out <= vif.max_meta_out;
49          data_collected.max_rank_out <= vif.max_rank_out;
50          @(posedge vif.clk);
51          data_collected.max_meta_out <= vif.max_meta_out;
52          data_collected.max_rank_out <= vif.max_rank_out;
53          data_collected.num_entries <= vif.num_entries;
54          data_collected.full <= vif.full;
55          $cast(data_clone, data_collected.clone( ));
56          item_collected_port.write(data_clone);
57          num_pkts++;
58       end
59    endtask: collect_data
60
61    virtual function void report_phase(uvm_phase phase);
62       `uvm_info(get_type_name( ), $sformatf("REPORT: COLLECTED
63 PACKETS = %0d", num_pkts), UVM_HIGH)
64    endfunction: report_phase
65 endclass: pifo_monitor_i
```

Código 29 Testbench UVM PIFO: Componente UVM Monitor interfaz *insert* - Parte II

Entre las líneas 37 a 39 se define la fase de ejecución (`run_phase`), donde el monitor llama a la tarea `collect_data` para empezar a recoger los datos del DUV a través de la interfaz, la tarea `collect_data` se define en la línea 41 y consiste en un bucle infinito en el que se espera por cada flanco de reloj para evaluar que la señal *insert* esté activa. Si el *monitor* detecta una operación de *insert* entre las líneas 45 a 49 almacena en la transacción `data_collected` el valor de cada ítem para la interfaz de *insert* y un ciclo de reloj después (línea 50) se almacena el resto de ítems siguiendo así la funcionalidad del módulo PIFO.

En la línea 55 se realiza la clonación de transacciones mediante el método `data_collected.clone()` y seguidamente en la línea 56 se escribe `data_clone` en el puerto de análisis `item_collected_port`, por último en este bucle se aumenta la variable `num_pkts` para controlar el número de paquetes tratados por el componente *monitor*, y que se imprimirá usando el mecanismo de mensajes de UVM en la fase `report_phase` implementado en las líneas 61 a 64.

2.5.6. UVM Scoreboard

El *scoreboard* UVM es un componente que verifica la funcionalidad del DUV. Recibe las transacciones del monitor utilizando los puertos de análisis con fines de verificación. El *scoreboard* extiende de `uvm_scoreboard`, que a su vez deriva de `uvm_component`.

El *scoreboard* tiene un modelo de referencia para comparar con el comportamiento de diseño. El modelo de referencia también se conoce como un predictor que implementa el comportamiento de diseño para que el marcador pueda comparar el resultado del DUV con el resultado del modelo de referencia para el mismo estímulo impulsado. Dependiendo de la funcionalidad del diseño, el *scoreboard* puede implementarse de dos maneras.

- In-order scoreboard
- Out-of-order scoreboard

El **In-order scoreboard** es útil para el diseño cuyo orden de salida es el mismo que el de los estímulos impulsados. El *scoreboard* comparará las secuencias de salida esperadas y reales en el mismo orden. Llegarán de manera independiente. Por lo tanto, la evaluación debe bloquearse hasta que estén presentes tanto las transacciones esperadas como las reales (Figura 25).

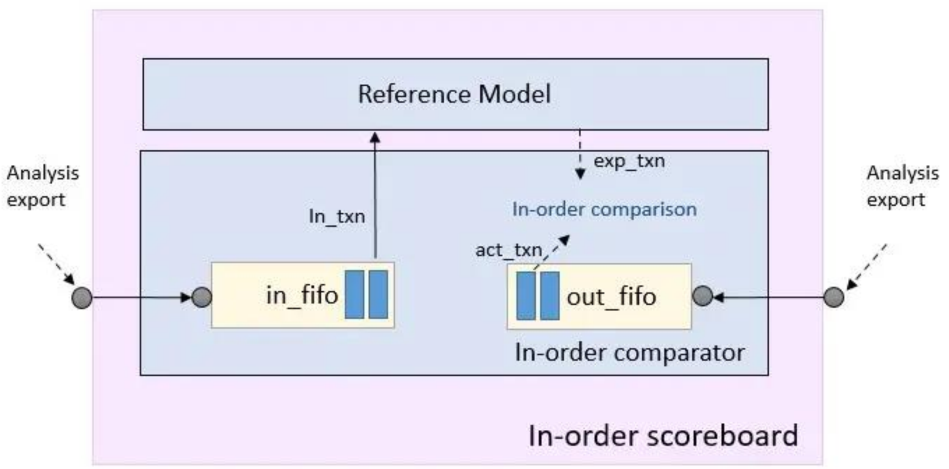


Figura 25 In-order scoreboard UVM.

Para implementar tales *scoreboards*, una forma más sencilla sería implementar FIFOs de análisis TLM. Para obtener más detalles, ver la sección Modelado y comunicación TLM.

El **Out-of-order scoreboard** es útil para el diseño cuyo orden de salida es diferente del de los estímulos de entrada impulsados. Basándose en los estímulos de entrada de referencia, el modelo de referencia generará el resultado esperado del DUV y se espera que la salida real llegue en cualquier orden. Por lo tanto, es necesario almacenar tales transacciones no coincidentes generadas a partir del estímulo de entrada hasta que se reciba la salida correspondiente del DUV

para ser comparada. Para almacenar tales transacciones, se utiliza ampliamente una matriz asociativa. Basándose en el valor del índice, las transacciones se almacenan en las matrices asociativas esperadas y reales. Las entradas de las matrices asociativas se eliminan cuando ocurre la comparación para el índice de matriz coincidente (Figura 26).

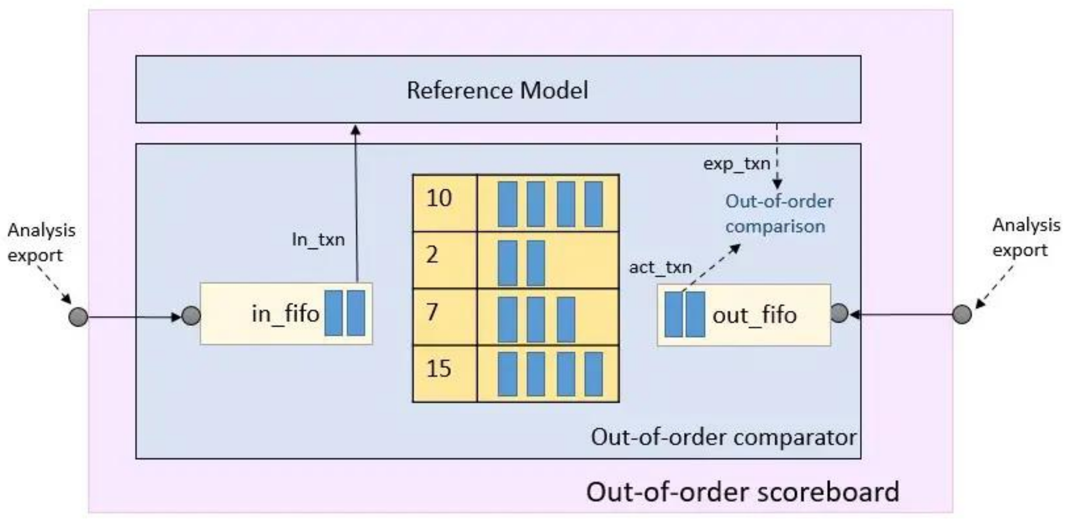


Figura 26 Out-of-order scoreboard UVM

En el Código 30 se muestra la implementación del componente *scoreboard* para la verificación del módulo PIFO, para esto caso y, como ya se explicó para el entorno de verificación en *SystemVerilog*, el modelo de referencia se realiza mediante el tipo de dato `queue` para emular el comportamiento de la PIFO. De esta manera, sabiendo el comportamiento del DUT estamos ante un *scoreboard out-of-order* ya que éste procesa las transacciones según una métrica, en este caso la prioridad del dato independientemente del orden en que fueron recibidas.

El aspecto más importante y diferenciador con respecto al *scoreboard* desarrollado en *SystemVerilog* es la manera de comunicarse con el resto de los componentes. En la línea 2 y 3 se declara dos puertos de análisis con la macro `uvm_analysis_imp_decl`, estos puertos permiten la recepción de datos que no requieren una confirmación de recepción por parte del consumidor (*scoreboard*). Es decir, el emisor envía datos al puerto y no espera una respuesta directa del consumidor de que los datos fueron recibidos o procesados correctamente. Además, se añade un identificador para más adelante como se explicará poder usarse para crear instancias concretas de estos puertos en los diferentes interfaces, estos identificadores son `_PORT_INSERT` y `_PORT_REMOVE`. En la línea 4 se crea el componente *scoreboard* heredada de la clase `uvm_scoreboard` y de la línea 6 a 11 se define las variables y parámetros necesarios para implementar la lógica del componente. En las líneas 13 y 14 se instancia los paquetes que almacenarán las transacciones de datos para inserción y extracción respectivamente.

En las líneas 16 a 19 se definen los puertos de análisis del componente *scoreboard*, `uvm_analysis_imp_PORT_INSERT` es el nombre del puerto de análisis con el sufijo adicional personalizado que se explicó anteriormente y, `#(data_packet_i, pifo_scoreboard)` es la lista de parámetros donde se indica el tipo de dato que el puerto de análisis recibirá, en este caso el paquete correspondiente a la interfaz *insert* y el componente que va a implementar el método `write` para manejar los datos recibidos, en este caso el *scoreboard*. Por último se especifica la instancia del puerto de análisis con estos parámetros es `analysis_imp_I` para el caso de la interfaz *insert*.

En la línea 21 se registra el componente *scoreboard* en la base de datos mediante el mecanismo *Factory* y en las líneas 23 a 25 el constructor de la clase. En las líneas 27 a 35 se ejecuta la fase `build_phase`, donde se crea las instancias de los puertos de análisis y los paquetes de *insert* y *remove* usando el mecanismo de *Factory*.

```
1  //definir analysis_imp ports
2  `uvm_analysis_imp_decl(_PORT_INSERT)
3  `uvm_analysis_imp_decl(_PORT_REMOVE)
4  class pifo_scoreboard extends uvm_scoreboard;
5    //Parámetros
6    localparam QUEUE_WIDTH = 8;
7    bit [15:0] queue_rank [$:QUEUE_WIDTH-1];
8    //variables
9    bit [15:0] DatoEsperado;
10   bit primera_in = 0;
11   bit [15:0] Maximo;
12   //Transacciones
13   data_packet_i insert_packet;
14   data_packet_r remove_packet;
15   // declarar analysis_imp ports
16   uvm_analysis_imp_PORT_INSERT #(data_packet_i, pifo_scoreboard)
17 analysis_imp_I;
18   uvm_analysis_imp_PORT_REMOVE #(data_packet_r, pifo_scoreboard)
19 analysis_imp_R;
20
21    `uvm_component_utils(pifo_scoreboard)
22
23    function new(string name, uvm_component parent);
24       super.new(name, parent);
25    endfunction: new
26
27    function void build_phase(uvm_phase phase);
28       super.build_phase(phase);
29       //crear analysis_imp ports
30       analysis_imp_I = new("analysis_imp_I", this);
31       analysis_imp_R = new("analysis_imp_R", this);
32       insert_packet = data_packet_i::type_id::create("insert_packet");
33       remove_packet = data_packet_r::type_id::create("remove_packet");
34       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
35    endfunction: build_phase
```
Código 30 Testbench UVM PIFO: Componente UVM Scoreboard parte I

En el componente *scoreboard,* cada puerto de análisis en UVM debe implementar un método `write` que especifica cómo manejar los datos recibidos, en el Código 31 se implementa el método `write` responsable de las operaciones de *insert* como funciones de *SystemVerilog*. En este punto, el código es el mismo que el utilizado para el *scoreboard* en *SystemVerilog*, donde se contempla los distintos escenarios a la hora de recibir una operación de *insert*, por lo que se muestra únicamente la estructura debido a la cantidad de líneas de código.

```
36 function void write_PORT_INSERT(data_packet_i insert_packet);
37  //si hay espacio libre se inserta al comienzo
38  if((queue_rank.size()==0) || ((queue_rank[0]>insert_packet.rank_in &&
39 primera_in == 0)))begin
40 
41  end
42  //si está lleno y no es máxima prio, primero se elimina el de mayor
43 prio y se inserta ordenado
44  else if(queue_rank.size()==QUEUE_WIDTH &&
45 (insert_packet.rank_in<queue_rank[queue_rank.size()-1]))begin
46 
47  end
48  //si el nuevo rank es el de mayor prio y no está lleno se inserta al
49 final
50  else if((insert_packet.rank_in>queue_rank[queue_rank.size()-1]) &&
51 queue_rank.size()!=QUEUE_WIDTH)begin
52 
53  end
54  //sino, se inserta ordenado
55  else begin
56 
58  end
59 endfunction
```

Código 31 Testbench UVM PIFO: Componente UVM Scoreboard parte II

Por último, en el Código 32 se define el método `write` para manejar las operaciones recibidas de la interfaz *remove*, siguiendo el protocolo del módulo PIFO y como también ya se explicó en el *scoreboard* en *SystemVerilog*, además en la línea 71 se llama a `run_phase` de la clase base `uvm_scoreboard` para ejecutar dicha fase.

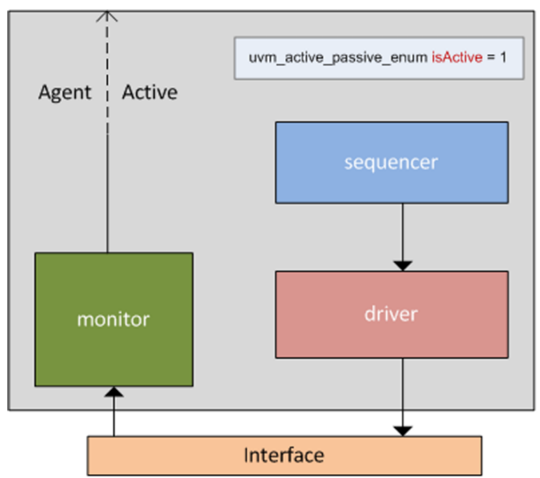
```
60 function void write_PORT_REMOVE(data_packet_r remove_packet);
61  //Primero ver que no está vacío, se elimina el rank de menor prio
62  if(queue_rank.size()>0)begin
63 
64  end
65  else begin
66 
67  end
68 endfunction
69 
70 virtual task run_phase(uvm_phase phase);
71    super.run_phase(phase);
72 endtask: run_phase
73 
74 endclass: pifo_scoreboard
```

Código 32 Testbench UVM PIFO: Componente UVM Scoreboard parte III

2.5.7. Componente UVM Agent

Un componente UVM *Agent* es un contenedor que contiene y conecta los componentes *driver, monitor*, y *sequencer* ya explicados. El agente desarrolla una jerarquía estructurada basada en el protocolo o el requisito de la interfaz.

El componente *Agent* se puede configurar como un elemento activo o pasivo. En el caso de estar configurado como activo, el UVM *Agent* tiene la función de conducir los estímulos generados en el test hacia el DUV. Por el contrario, un componente UVM *Agent* pasivo solamente monitoriza las señales de la interfaz del dispositivo. Por esta razón, un componente UVM *Agent* pasivo no necesita ni un componente UVM *Sequencer* ni un componente UVM *Driver*. Es capaz de cumplir su función únicamente con un UVM Monitor. La Figura 27 muestra la diferencia entre la configuración activa y pasiva de este componente.



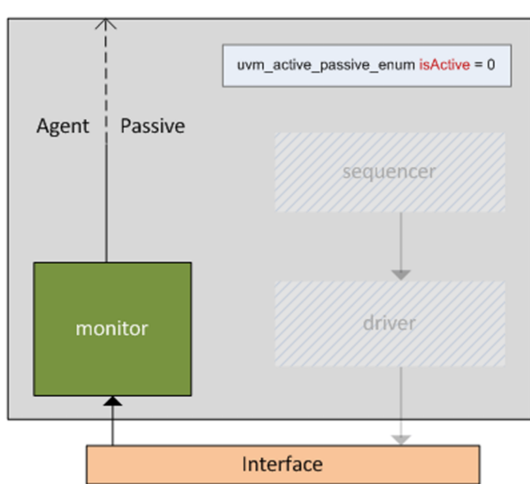


Figura 27 Diferencia entre UVM Agent activo y pasivo.

Un agente es usualmente instanciado en la clase *environment* UVM. Por lo tanto, se puede configurar en el entorno o en cualquier otra clase de componente donde se instancie un agente utilizando el parámetro de configuración `int is_active` como se muestra a continuación:

```
set_config_int("<path_to_agent>", "is_active", UVM_ACTIVE);
set_config_int("<path_to_agent>", "is_active", UVM_PASSIVE);
```

El Código 33 muestra la implementación en UVM del componente *Agent* para la verificación del módulo PIFO, después de crear el componente heredado de la clase `uvm_agent`, en la línea 2 se define el comportamiento del agente, como se explicó, al establecer el valor `UVM_ACTIVE` indica que tendrá comportamiento activo, incluyendo los tres componentes mencionados. Para el caso de la PIFO, ambos agentes tanto para la interfaz de *insert*, como la interfaz de *remove* se definen activo ya que es necesario interactuar con el DUV incluyendo los componentes *driver* y *sequencer*. Gracias a esta parametrización, en las líneas 16 a 23 cuando se produce la fase

`build_phase` se puede acceder al valor de `is_active` y crear el *Agent* según su condición, en este caso al ser activo se crea los componentes *sequencer* y *driver* mediante el método `create` de la *Factory* y en la línea 23 independientemente del valor de `is_active` siempre se crea el componente *monitor*. En las líneas 4-6 se instancia los tres componentes que conforman un agente y se registra en la *Factory* (líneas 8-10).

En las líneas 12-14 se define el constructor de la clase, en este caso del componente *Agent* con la función `new()` propia de *SystemVerilog* con el fin de heredar las variables y métodos.

Por último, durante la fase `connect_phase`, en la línea 30 se realiza la conexión entre componente *driver* y *sequencer* a través de los mecanismos TLM ya explicados en capítulos anteriores.

```
1   class pifo_agent_i extends uvm_agent;
2    protected uvm_active_passive_enum is_active = UVM_ACTIVE;
3
4    pifo_sequencer_i sequencer;
5    pifo_driver_i    driver;
6    pifo_monitor_i   monitor;
7
8   `uvm_component_utils_begin(pifo_agent_i)
9      `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
10   `uvm_component_utils_end
11
12  function new(string name, uvm_component parent);
13     super.new(name, parent);
14  endfunction
15
16  function void build_phase(uvm_phase phase);
17     super.build_phase(phase);
18     if(is_active == UVM_ACTIVE) begin
19       sequencer = pifo_sequencer_i::type_id::create("sequencer_i", this);
20       driver = pifo_driver_i::type_id::create("driver_i", this);
21     end
22
23    monitor = pifo_monitor_i::type_id::create("monitor_i", this);
24
25     `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
26  endfunction: build_phase
27
28  function void connect_phase(uvm_phase phase);
29     if(is_active == UVM_ACTIVE)
30        driver.seq_item_port.connect(sequencer.seq_item_export);
31        `uvm_info(get_full_name( ), "Connect stage complete.", UVM_HIGH)
32     endfunction: connect_phase
33  endclass: pifo_agent_i
```
Código 33 Testbench UVM PIFO: Componente UVM Agent interfaz insert

2.5.8. UVM Environment

El componente *Environment* contiene múltiples componentes de verificación reutilizables y define su configuración predeterminada según lo requerido por la aplicación. Por ejemplo, en el entorno UVM para el módulo PIFO, contiene múltiples agentes para diferentes interfaces, un *scoreboard* común y podría contener también un recolector de cobertura funcional y verificadores adicionales. También puede contener otros entornos más pequeños que han sido verificados a nivel de bloque y ahora se integran en un subsistema. Esto permite que ciertos componentes y secuencias utilizados en la verificación a nivel de bloque se reutilicen en el plan de verificación a nivel de sistema.

Técnicamente, es posible inicializar los agentes y *scoreboard* directamente en la clase `uvm_test`, sin embargo, no es recomendable realizarlo de esta manera por las siguientes razones:

- El *testbench* dejaría de ser reutilizable porque se basan en una estructura de entorno específica para cada caso.
- Los cambios en la topología del sistema obligarían a la actualización de varios módulos del *testbench*.
- El ingeniero de verificación necesitaría saber cómo configurar el entorno.

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

Para la verificación del módulo PIFO se implementó un primer componente *environment* que contenga la fase `build_phase` para la creación del componente *Agent*, uno para cada interfaz y de esta manera crear un componente *environment* final que contenga estos subcomponentes además del componente *scoreboard.*

En el Código 34 se define el componente *environment* para la interfaz *insert*, tras crear el componente en la línea 1 heredada de la clase `uvm_env`, en la línea 3 se declara el componente *Agent*. En la línea 5 se registra el componente en la *Factory*. Entre las líneas 7 a 9 se declara el constructor new y entre las líneas 11 a 16 se ejecuta la fase `build_phase`, donde se crea el componente agente mediante el método `create` de la *Factory.* De manera idéntica, se crea el para la interfaz *remove* creando en este caso el agente responsable de la interfaz *remove*.

```
1  class pifo_env_i extends uvm_env;
2
3     pifo_agent_i agent;
4
5    `uvm_component_utils(pifo_env_i)
6
7     function new(string name, uvm_component parent);
8        super.new(name, parent);
9     endfunction
10
11    function void build_phase(uvm_phase phase);
12       super.build_phase(phase);
13
14      agent = pifo_agent_i::type_id::create("agent_i", this);
15       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
16    endfunction: build_phase
17 endclass: pifo_env_i
```

Código 34 Testbench UVM PIFO: Componente UVM Environment interfaz insert

En el Código 35 se muestra la implementación del componente *environment* global para la verificación del módulo PIFO, después de crear el componente heredado de la clase `uvm_env`, se declaran los componentes que conforman el componente *environment* (líneas 3-5), estos componentes son los subcomponentes *environments* que crea los componentes *Agent* para ambas interfaces y el componente *scoreboard*. En la línea 7 se registra en la base de datos con el mecanismo *Factory* y entre las líneas 9 a 11 su correspondiente constructor.

A partir de la línea 13, se ejecuta la fase `build_phase`, en primer lugar se configura los agentes de inserción `pifo_insert.agent_i` y extracción `pifo_remove.agent_r` como componentes agentes activos (líneas 16 a 19) y, por otro lado, entre las líneas 20 a 23 se configura las interfaces virtuales para ambos componentes *monitors* responsables de las interfaces *insert* y *remove*, esto es posible gracias al método `set` de la clase `uvm_config_db`.

En las líneas 25 y 26 se crea los subcomponentes *environments* de inserción y extracción que viene a ser los componentes agentes respectivamente utilizando el mecanismo *Factory* y de la misma manera en la línea 27 se crea el componente *scoreboard*.

El componente *environment* se encarga de la conexión de los componentes en la fase `connect_phase`, donde en la línea 33 se establece la conexión entre el *monitor* de inserción y el *scoreboard* y en la línea 34 la conexión entre *monitor* de extracción y el *scoreboard* mediante el mecanismo de comunicación UVM basado en TLM explicado en capítulos anteriores y a través de los puertos de análisis creados y explicados en el componente *scoreboard*.

```
1  class dut_env extends uvm_env;
2
3     pifo_env_i       pifo_insert;
4     pifo_env_r       pifo_remove;
5     pifo_scoreboard sb;
6
7     `uvm_component_utils(dut_env)
8
9     function new(string name, uvm_component parent);
10       super.new(name, parent);
11    endfunction
12
13    function void build_phase(uvm_phase phase);
14       super.build_phase(phase);
15
16       uvm_config_db#(int)::set(this, "pifo_insert.agent_i", "is_active",
17 UVM_ACTIVE);
18       uvm_config_db#(int)::set(this, "pifo_remove.agent_r", "is_active",
19 UVM_ACTIVE);
20       uvm_config_db#(string)::set(this, "pifo_insert.agent_i.monitor_i",
21 "monitor_intf_i", "in_intf");
22       uvm_config_db#(string)::set(this, "pifo_remove.agent_r.monitor_r",
23 "monitor_intf_r", "re_intf");
24
25       pifo_insert = pifo_env_i::type_id::create("pifo_insert", this);
26       pifo_remove = pifo_env_r::type_id::create("pifo_remove", this);
27       sb = pifo_scoreboard::type_id::create("sb", this);
28
29       `uvm_info(get_full_name( ), "Build stage complete.", UVM_HIGH)
30    endfunction: build_phase
31
32    function void connect_phase(uvm_phase phase);
33 pifo_insert.agent.monitor.item_collected_port.connect(sb.analysis_imp_I);
34 pifo_remove.agent.monitor.item_collected_port.connect(sb.analysis_imp_R);
35       `uvm_info(get_full_name( ), "Connect phase complete.", UVM_HIGH)
36    endfunction: connect_phase
37 endclass: dut_env
```

Código 35 Testbench UVM PIFO: Componente UVM Environment

2.5.9. UVM Test

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

Se recomienda tener un test base que incluya todas las configuraciones del *testbench*, así como las instancias de la clase *environment*, crear configuraciones, etc., y una variedad de test según el plan de verificación que se puedan extender desde el test base para evitar repetir el mismo código en cada test. Los tests derivados también pueden establecer configuraciones basadas en los requisitos del test.

La ejecución del test es una actividad que consume tiempo, ya que ejecuta una secuencia o secuencias para la funcionalidad del DUV. Al ejecutar el test, se construye una estructura completa del *testbench* UVM en la fase `build_phase` y las actividades que consumen tiempo se realizan en la `run_phase` con la ayuda de secuencias. Es obligatorio registrar los test*s* en la fábrica de UVM; de lo contrario, la simulación terminará con `UVM_FATAL` (test no encontrado).

La tarea `run_test` es una tarea global declarada en la clase `uvm_root` y es responsable de ejecutar un caso de test.

- La tarea `run_test` inicia el mecanismo de fases que ejecuta las fases en un orden predefinido.
- Se llama en el bloque inicial del banco de pruebas principal y acepta `test_name` como una cadena de texto.

Pasar un argumento a la tarea `run_test` provoca la recompilación al ejecutar diferentes test*s*. Para evitarlo, UVM proporciona una forma alternativa utilizando el argumento de línea de comandos `+UVM_TESTNAME`. En este caso, la tarea `run_test` no requiere pasar ningún argumento en el bloque inicial del `testbench` principal. Después de la ejecución de todas las fases, `run_test` finalmente llama a la tarea `$finish` para la salida del simulador.

A continuación, se muestra en el Código 36, la implementación del Test base para la verificación del módulo PIFO, del cual se origina el resto de tests que forman el plan de verificación, con el fin de no extender demasiado la memoria, se explicará el desarrollo del Test base y uno de los test específicos, en este caso el test responsable de insertar datos en la PIFO hasta llenarla y seguidamente eliminar todos los datos hasta vaciar la PIFO desarrollado en el Código 37.

Una vez creado el componente Test que hereda de la clase `uvm_test` (línea 1), se registra el componente Test con el mecanismo de *Factory*, permitiendo la creación dinámica de instancias de este componente. En la línea 4 se instancia el entorno de verificación y en la línea 5 se instancia un objeto `uvm_table_printer`, este objeto se utiliza para imprimir la topología del entorno de verificación completo de manera organizada y legible en formato tabular.

En las líneas 7 a 9 se construye el componente `base_test` llamando al constructor de la clase `uvm_test`, a partir de la línea 11 comienza la fase `build_phase`, en este nivel de la jerarquía, en la línea 13 se crea el componente *environment* mediante el método créate de *Factory* y en las líneas 14 se crea el objeto `printer`, para definir la profundidad a nivel de jerarquía que se quiere mostrar con este mecanismo se utiliza el método `printer.knobs.depth`, señalando en este caso, hasta 5 niveles de profundidad. Para mostrar el contenido de `printer`, en la línea

18 se ejecuta la fase `end_of_elaboration_phase`, donde el método `sprint` en la línea 20 genera una cadena de texto recorriendo la jerarquía de componentes organizando la información en un formato legible.

Por último, en la tarea `run_phase` (líneas 23 a 25) se ejecuta la fase de ejecución, configurando un tiempo de drenaje de 500 unidades de tiempo en la línea 24 con el uso del método `phase.phase_done.set_drain_time` para asegurar la finalización ordenada de los procesos activos.

```
1  class base_test extends uvm_test;
2     `uvm_component_utils(base_test)
3  
4     dut_env env;
5     uvm_table_printer printer;
6  
7     function new(string name, uvm_component parent);
8        super.new(name, parent);
9     endfunction: new
10 
11    function void build_phase(uvm_phase phase);
12       super.build_phase(phase);
13       env = dut_env::type_id::create("env", this);
14       printer = new( );
15       printer.knobs.depth = 5;
16    endfunction:build_phase
17 
18    virtual function void end_of_elaboration_phase(uvm_phase phase);
19       `uvm_info(get_type_name( ), $sformatf("Printing the test topology
20 :\n%s", this.sprint(printer)), UVM_LOW)
21    endfunction: end_of_elaboration_phase
22 
23    virtual task run_phase(uvm_phase phase);
24      phase.phase_done.set_drain_time(this, 500);
25    endtask: run_phase
26 endclass: base_test
```

Código 36 Testbench UVM PIFO: Componente UVM Test – `base_test`

Como se comentó, en este punto se implementa los diferentes test para la verificación del DUV, para el caso del módulo PIFO se definieron hasta 12 test que entre todos verifican los distintos escenarios donde la PIFO puede verse sometido. En el Código 37 se muestra el caso particular de verificar la PIFO insertando paquetes hasta que la PIFO esté llena y luego eliminar paquetes hasta que esté vacío.

Para ello, en la línea 2 se crea el componente `t_r01` que hereda de la clase `base_test` previamente explicado. Como siempre, en la línea 3 se registra con el *Factory*, en las líneas 5 a 7 se llama al constructor de la clase base y se implementa la fase de construcción `build_phase` entre las líneas 9 y 11 sin realizar ninguna acción adicional, con la intención de crear los componentes definidos en `base_test` y `dut_env`.

```
 1  //SE INSERTA HASTA FULL ALEATORIO Y SE ELIMINA HASTA EMPTY
 2  class t_r01 extends base_test;
 3    `uvm_component_utils(t_r01)
 4
 5    function new(string name, uvm_component parent);
 6        super.new(name, parent);
 7     endfunction: new
 8
 9    function void build_phase(uvm_phase phase);
10        super.build_phase(phase);
11     endfunction: build_phase
12
13    virtual task run_phase(uvm_phase phase);
14      insert_into_full seq_i;
15      remove_into_empty seq_r;
16      super.run_phase(phase);
17      phase.raise_objection(this);
18      seq_i = insert_into_full::type_id::create("seq_i");
19      seq_r = remove_into_empty::type_id::create("seq_r");
20      seq_i.start(env.pifo_insert.agent.sequencer);
21      seq_r.start(env.pifo_remove.agent.sequencer);
22      phase.drop_objection(this);
23    endtask: run_phase
24  endclass: t_r01
```

Código 37 Testbench UVM PIFO: Componente UVM Test – `t_r01`

En la tarea que define la fase `run_phase` es donde se define la lógica del propio test, en la línea 14 se declara la secuencia `insert_into_full`, su función es enviar transacciones al DUV, en este caso hasta llenar la PIFO, más adelante en el apartado dedicado a secuencias se explicará más en detalle cada una de ellas. Por otro lado, en la línea 15 se declara la secuencia `remove_into_empty` cuya función es enviar transacciones al DUV hasta vaciar la PIFO. En la línea 16 se asegura que la clase `base_test` también se ejecute.

Para la gestión de objeciones en UVM se utiliza en la línea 17 y línea 22 un mecanismo que permite indicar al resto de componentes que aún está ocupado y que la simulación no debe avanzar a la siguiente fase. Esta técnica es especialmente útil en la fase `run_phase` donde múltiples tareas y secuencias pueden estar ejecutándose simultáneamente. Para ello, `phase.raise_objection(this)` indica que este componente está ocupado y que la simulación no debe finalizar la fase actual en ningún momento hasta que esta objeción sea retirada, para terminar la objeción en la línea 22 se retira mediante `phase.drop_objection(this)`.

En medio de la gestión de objeciones se produce la creación (líneas 18 y 19) y ejecución (líneas 20 y 21) de las secuencias. Para ello, se utiliza el método `start` que inicia la ejecución de la secuencia en el componente *sequencer* asociado con los agentes correspondientes a cada interfaz y que coordina la generación y entrega de las transacciones al componente *driver* que interactúa directamente con el DUV.

2.5.10. UVM *Testbench*

Todos los componentes de verificación, interfaces y DUV son referenciados en el módulo Top (también conocido como UVM *Testbench*), que es del tipo module en *SystemVerilog*. Por lo tanto, este componente es un contenedor estático que contiene todos los componentes que conforman el entorno de verificación UVM (incluido el componente UVM Test) y el DUV. Este componente juega un papel importante en la metodología UVM y, a continuación, en el Código 38 se implementa el módulo *Testbench* para la verificación del IP PIFO.

La principal función de este componente es referenciar y configurar la conexión entre el entorno de verificación UVM y el DUV. Para ello, en primer lugar, se crea y referencia las interfaces virtuales para las operaciones de *insert* y *remove* respectivamente que permitirán la comunicación entre el DUV y el entorno UVM definidas en las líneas 8 y 9. Estas interfaces definirán todas las señales del DUV a las que se quiera tener acceso durante la verificación del mismo. Destacar, que la interfaz responsable de las operaciones de *remove* no solo se conecta a las señales de reloj y *reset*, también se conecta directamente a la señal de *insert*. Esto es debido a que el monitor encargado de detectar las operaciones de *remove* debe conocer también el estado de la señal *insert*.

En segundo lugar, se conectarán físicamente las señales de los puertos del DUV con sus homólogos situados en las interfaces virtuales anteriormente creadas (líneas 10 a 26).

```
1  module top;
2     import uvm_pkg::*;
3     import pifo_pkg::*;
4     localparam RST_WIDTH = 20;
5     bit clk;
6     bit reset;
7     pifo_if_i ivif(.clk(clk), .reset(reset));
8     pifo_if_r rvif(.clk(clk), .reset(reset), .insert(ivif.insert));
9  
10    pifo_reg DUV (
11      .rst              (reset),
12      .clk              (clk),
13      .full             (ivif.full),
14      .insert           (ivif.insert),
15      .rank_in          (ivif.rank_in),
16      .meta_in          (ivif.meta_in),
17      .valid_out        (rvif.valid_out),
18      .remove           (rvif.remove),
19      .rank_out         (rvif.rank_out),
20      .meta_out         (rvif.meta_out),
21      .max_valid_out    (rvif.max_valid_out),
22      .max_rank_out     (ivif.max_rank_out),
23      .max_meta_out     (ivif.max_meta_out),
24      .num_entries      (ivif.num_entries),
25      .empty            (rvif.empty)
26   );
```

Código 38 Testbench UVM PIFO: Testbench Top parte I

Otra función de este componente es la generación de las señales de reloj y *reset*. En un diseño cuyos bloques digitales operan en múltiples frecuencias de reloj, el componente UVM *Testbench* debe generar múltiples relojes. Por lo tanto, la generación de la señal de reloj no siempre será tan sencilla de implementar. Además, para comprobar diferentes funcionalidades del dispositivo se pueden modificar determinados parámetros como la frecuencia, el ciclo de trabajo y la fase de manera dinámica, por lo que el componente UVM *Testbench* necesitaría cierta infraestructura para tolerar estas operaciones dinámicas. En la línea 27 del Código 39 se genera la señal de reloj mediante un proceso `always` con un periodo de 10 unidades de tiempo. Entre las líneas 29 a 35 se define la secuencia de `reset`, primero desactivado, luego activado durante `RTS_WIDTH` unidades de tiempo y finalmente se desactiva nuevamente.

Este componente debe incluir las interfaces virtuales creadas en la base de datos, de esta manera, los diferentes componentes que forman parte del entorno, como el UVM *Driver* y el UVM *Monitor*, podrán acceder a las interfaces para interactuar con el DUV, con el fin de estimular y monitorizar dicho dispositivo. Para ello, se configura con el método `set` de la clase `uvm_config_db` la interfaz virtual para que esté disponible en los componentes UVM necesarios.

Por último, realiza la llamada del método `run_test()`, el cual inicia la ejecución de las distintas fases que se explicaron en apartados anteriores, lo que provoca el comienzo de la simulación UVM y por parámetro se selecciona el test, en este caso `t_r01` explicado anteriormente.

```
27    always #5 clk = ~clk;
28
29    initial
30          begin
31            reset = 0;
32          #RST_WIDTH
33            reset = 1;
34          #RST_WIDTH reset = 0;
35          end
36
37     initial begin
38       uvm_config_db#(virtual pifo_if_i)::set(uvm_root::get( ) ,
39  "*.agent_i.*" , "in_intf", ivif);
40       uvm_config_db#(virtual pifo_if_r)::set(uvm_root::get( ) ,
41  "*.agent_r.*" , "re_intf", rvif);
42       uvm_config_db#(virtual pifo_if_i)::set(uvm_root::get( ) ,
43  "*.monitor_i" , "in_intf", ivif);
44       uvm_config_db#(virtual pifo_if_r)::set(uvm_root::get( ) ,
45  "*.monitor_r" , "re_intf", rvif);
46       //hay que espicificar el nombre del test
47       run_test("t_r01");
48     end
49  endmodule
```

Código 39 Testbench UVM PIFO: Testbench Top parte II

2.5.11. Secuencias UVM

Una de las mayores aportaciones de la metodología UVM, en comparación con la estructura de un *testbench* en *SystemVerilog* es el uso de las secuencias para enviar las transacciones al DUV. Las secuencias son una clase de la biblioteca de clases de UVM que se ejecutan sobre el componente `sequencer`. A diferencia del resto de componentes, las secuencias son objetos dinámicos, derivados de la clase base `uvm_object`, que aprovechan el protocolo de *handshake* que se establece entre los componentes `sequencer` y `driver`. Esto permite adaptar el comportamiento de los estímulos de entrada en función del estado del interfaz del DUV en cada momento.

Al igual que ocurre con la definición de los test, el manual de buenas prácticas de UVM aconseja la definición de una librería de secuencias básicas a partir de las cuales, y de ser necesario, se puedan generar secuencias más complejas que ayuden a la verificación del sistema, tanto en modo aleatorio como en modo dirigido.

El Código 40 muestra la descripción de la secuencia base `insert_data_min_prio`. Esta secuencia se deriva de la clase base `uvm_sequence`. Las clases asociadas a una secuencia deben estar parametrizadas con el tipo de transacción que vayan a manejar, en este caso `data_packet_i`. Este tipo de transacción debe ser la misma que declaran los componentes `sequencer` y `driver`, sobre los que se va a enviar dicha transacción. En la línea 2 se registra la secuencia en la *Factory*, haciendo uso de la macro `uvm_objet_utils`. Las líneas 4 a 7 definen el constructor de la secuencia.

Por último, la parte más importante de una secuencia corresponde a su tarea `body`, definida entre las líneas 9 y 11. Esta tarea contiene las operaciones de creación, aleatorización y envío de las transacciones que componen la secuencia al componente `sequencer`. Estas acciones pueden realizarse de manera individual o bien se puede hacer uso de las macros que UVM ofrece para tal efecto. En este caso son dos las macros que UVM ofrece para este propósito, `uvm_do` y `uvm_do_with`. Ambas macros reciben, como parámetro, la transacción a enviar al componente `driver` a través del `sequencer`. En este caso, la transacción es `req` que, si bien no se declara en la secuencia, se declara en la clase base `uvm_sequence`.

En la línea 10 se llama a la macro `uvm_do_with`, que permite, además de especificar la transacción a enviar, fijar determinadas restricciones a la hora de aleatorizar la transacción. En este caso en particular, se ha establecido que el rango de la transacción sea `16'h0000` y que el retardo de envío de la misma sea igual a `2`.

```
1  class insert_data_min_prio extends uvm_sequence #(data_packet_i);
2    `uvm_object_utils(insert_data_min_prio)
3
4    function new(string name = "insert_data_min_prio");
5      super.new(name);
6      req = new("data_packet_i");
7     endfunction:new
8
9    virtual task body();
10     `uvm_do_with(req, {req.rank_in == 16'h0000; req.delay == 2;});
11   endtask: body
12 endclass: insert_data_min_prio
```

Código 40 Testbench UVM PIFO: Secuencia `insert_data_min_prio`

En el componente Test, se comentó cómo usando las secuencias `insert_into_full` y `remove_into_empty` se implementa el test `t_r01`, que verifica el comportamiento de la PIFO insertando datos hasta llenar la PIFO y seguidamente eliminar hasta vaciar. A continuación, se muestra el desarrollo de ambas secuencias, comentando los aspectos más relevantes.

De la misma manera que la anterior, la secuencia `insert_into_full` hereda de `uvm_sequence` y está parametrizada con el tipo `data_packet_i` (línea 1 del Código 41). En la línea 4 se registra con el sistema de *Factory* y se construye la clase creando también un nuevo paquete llamado `req`. En la tarea principal `body()` se ejecuta un bucle que se repetirá tantas veces como el tamaño de la PIFO y, mediante la macro `uvm_do_with` se configura los campos de `req` estableciendo el valor de `rank_in` al valor `order` y `delay` en 2. De manera que, después de enviar cada transacción, el valor de `order` se incrementa en 10 haciendo que se envíen transacciones con orden de prioridad creciente en valores de 10 hasta llenar la PIFO.

```
1  class insert_into_full extends uvm_sequence #(data_packet_i);
2    int width = 8;
3    int order = 10;
4    `uvm_object_utils(insert_into_full)
5
6    function new(string name = "insert_into_full");
7      super.new(name);
8      req = new("data_packet_i");
9     endfunction:new
10
11   virtual task body();
12     for(int i = 0; i < width; i++) begin
13       `uvm_do_with(req, {req.rank_in == order; req.delay == 2;});
14       order = order + 10;
15     end
16   endtask: body
17 endclass: insert_into_full
```

Código 41 Testbench UVM PIFO: Secuencia `insert_into_full`

En el Código 42, se implementa la secuencia remove_into_empty, muy similar a la anterior con la diferencia de; asignar por parámetro el tipo data_packet_r encargada de las operaciones de *remove* y, en la tarea principal body(), se realiza nuevamente un bucle con la finalidad de enviar transacciones pero esta vez haciendo uso de la macro uvm_do debido a que en este caso, el objetivo es eliminar los datos de la PIFO.

```
1  class remove_into_empty extends uvm_sequence #(data_packet_r);
2    int width = 8;
3    `uvm_object_utils(remove_into_empty)
4    function new(string name = "remove_into_empty");
5      super.new(name);
6      req = new("data_packet_i");
7     endfunction:new
8
9    virtual task body();
10     for(int i = 0; i < width; i++) begin
11       `uvm_do(req);
12     end
13   endtask: body
14 endclass: remove_into_empty
```

Código 42 Testbench UVM PIFO: Secuencia remove_into_empty

A modo de resumen, se lista los diferentes test utilizados para cubrir el plan de verificación con los diferentes escenarios que aseguren la funcionalidad y condiciones de la PIFO se comportan correctamente, así como las secuencias necesarias para desarrollar dichos test.

Tabla 2 Batería tests para verificación módulo PIFO

Test	Descripción
t_i00	Inserta elemento con prioridad máximo y, seguido un nuevo dato de prioridad mínima.
t_i01	Inserta elemento con prioridad mínima y, seguido un nuevo dato de prioridad máxima.
t_i02	Inserta elementos hasta llenar la PIFO en orden creciente de prioridades.
t_i03	Inserta elementos hasta llenar la PIFO con prioridades aleatorias.
t_i04	Inserta elemento de prioridad mínima mientras la PIFO está llena.
t_i05	Inserta elemento de prioridad máxima mientras la PIFO está llena.
t_r00	Inserta dos elementos aleatorios y se eliminan.
t_r01	Inserta elementos en orden creciente de prioridades hasta llenar la PIFO y se eliminan hasta vaciar.
t_r02	Inserta elementos con prioridades aleatorias hasta llenar la PIFO y se eliminan hasta vaciar.
t_r03	Elimina elemento mientras la PIFO está vacía.
t_ir00	Inserta y elimina elemento al mismo instante.
t_ir01	Inserta y elimina completamente aleatorio.

Tabla 3 Secuencias para verificación módulo PIFO

Secuencias	Interfaz	Descripción
insert_data_min_prio	insert	Envía una transacción con prioridad rank_in 16'h0000.
insert_data_max_prio	insert	Envía una transacción con prioridad rank_in 16'hFFFF.
insert_into_full	insert	Envía número de transacciones igual al tamaño de la PIFO en orden creciente de prioridades.
random_sequence_i	insert	Envía una transacción con prioridad aleatoria.
many_random_sequence_i	insert	Envía número de transacciones igual al tamaño de la PIFO con prioridades aleatorias.
remove_data	remove	Envía una transacción.
remove_into_empty	remove	Envía un número de transacciones igual al tamaño de la PIFO.

2.5.12. Ejecución del *testbench* UVM sobre el módulo PIFO

Si bien en este TFM se han desarrollado una gran batería de test para la verificación, en este apartado se presentan únicamente los resultados de ejecución de tres de los test realizados. En este caso se trata de los test t_r00, t_i04 y t_r02.

- **Test t_r00.** El Código 43 muestra la fase de ejecución del test t_r00. Antes de su ejecución, se definen dos secuencias, una secuencia aleatoria (línea 2) y una secuencia de extracción de datos, línea 3. El comienzo de la tarea se señaliza activando el mecanismo de objections, línea 5, lo que se consigue con la llamada al método raise_objections de la clase phase.

  En las líneas 6 y 7 se crean las secuencias mediante el mecanismo de *Factory*, las cuales se denominan seq_1, para atacar la interfaz de entrada, y seq_2, para la interfaz de salida. Finalmente, se envían dichas secuencias al componente *driver* a través del componente *sequencer* correspondiente. Esto se muestra en las líneas 8 y 9. Destacar que el mecanismo de comunicación *driver – sequencer* se activa mediante la llamada al método start de cada una de las secuencias.

```
1  virtual task run_phase(uvm_phase phase);
2      random_sequence_i seq_1;
3      remove_data seq_2;
4      super.run_phase(phase);
5      phase.raise_objection(this);
6      seq_1 = random_sequence_i::type_id::create("seq_1");
7      seq_2 = remove_data::type_id::create("seq_2");
8      seq_1.start(env.pifo_insert.agent.sequencer);
9      seq_2.start(env.pifo_remove.agent.sequencer);
10     phase.drop_objection(this);
11   endtask: run_phase
```

Código 43 Fase de ejecución del test `t_r00`

La Figura 28 muestran las formas de ondas tras la ejecución del test descrito. Tras la fase de `reset`, en el instante t_1, se produce una operación se inserción activando la señal `insert`, con rango (`rank_in`) `0x0a` y dato (`meta_in`) `0xdcf`. Dos ciclos más tarde, instante t_2, el DUV valida el dato a la salida activando la señal `valid_out`, con el rango y el dato presente en los buses `rank_out` y `meta_out`, respectivamente.

El segundo ciclo de inserción se produce en el instante t_3, esta vez con un par de rango inferior, `rank_in` de `0x14`. Tal y como indican las especificaciones funcionales del módulo, tras un ciclo de inserción, la señal `valid_out` se desactiva ya que el módulo está comprobando la prioridad del dato de entrada, tal y como se puede observar en el instante t_4.

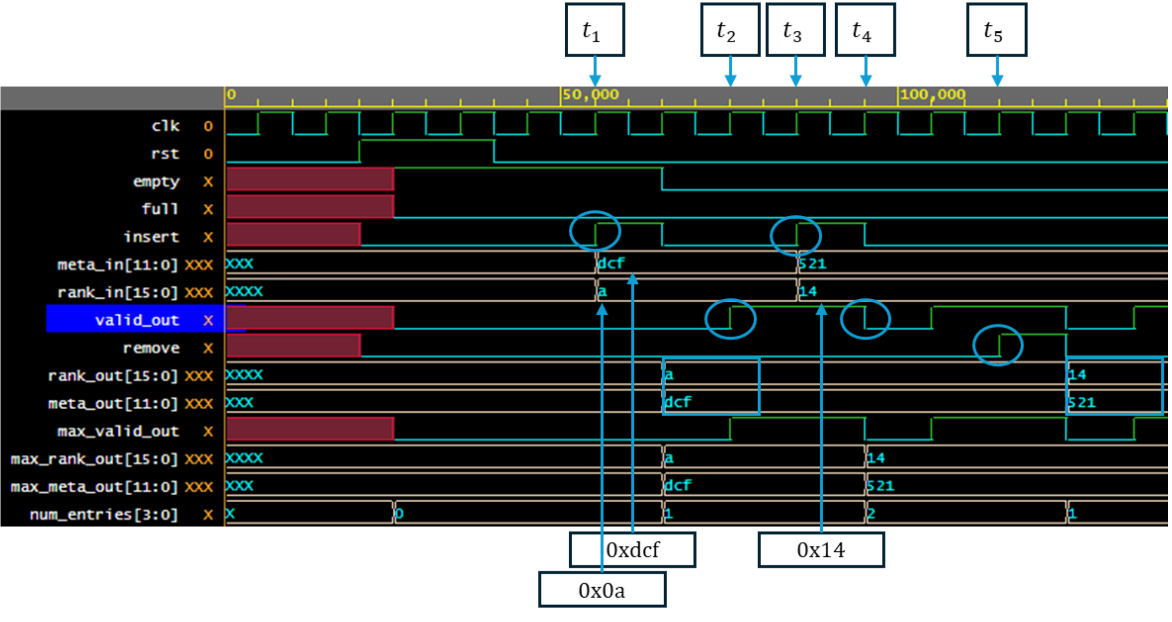


Figura 28 Fase de ejecución del test `t_r00`

En el instante t_5 se produce una transacción de extracción, señalizada al activarse la señal remove. El interfaz de salida está validando el dato de mayor prioridad, con el par 0x0a, 0xdcf. Tras el ciclo de extracción, el DUV valida el siguiente dato almacenado, en este caso el par 0x14, 0x521, tal y como se esperaba.

- **Test t_i04:** El Código 44 muestra la fase de ejecución del test t_i04. Antes de su ejecución, se definen dos secuencias, una secuencia many_random_sequence_i (línea 2) que llena la PIFO con datos aleatorios y una secuencia de inserción de datos many_random_sequence_i, línea 3 que inserta un dato con la prioridad 16'h0000. El comienzo de la tarea se señaliza activando el mecanismo de objections, línea 5, lo que se consigue con la llamada al método raise_objections de la clase phase.
  En las líneas 6 y 7 se crean las secuencias mediante el mecanismo de *Factory*, las cuales se denominan seq_1, para atacar la interfaz de entrada, y seq_2, para la interfaz de salida. Finalmente, se envían dichas secuencias al componente *driver* a través del componente *sequencer* correspondiente. Esto se muestra en las líneas 8 y 9.

```
1  virtual task run_phase(uvm_phase phase);
2     many_random_sequence_i seq_1;
3     insert_data_min_prio seq_2;
4     super.run_phase(phase);
5     phase.raise_objection(this);
6     seq_1 = many_random_sequence_i l::type_id::create("seq_1");
7     seq_2 = insert_data_min_prio::type_id::create("seq_2");
8     seq_1.start(env.pifo_insert.agent.sequencer);
9     seq_2.start(env.pifo_insert.agent.sequencer);
10    phase.drop_objection(this);
11  endtask: run_phase
```

Código 44 Fase de ejecución del test t_i04

La Figura 29 muestran las formas de ondas tras la ejecución del test descrito. Tras la fase de reset, en el instante t_1, se produce la primera operación se inserción activando la señal insert.

En el instante t_2, se produce la segunda operación de inserción, con rango menor, esto es mayor prioridad reflejándose así en las señales rank_out y meta_out, respectivamente tras la validación del dato en ese mismo instante. En el instante t_3, ocurre exactamente lo mismo y a partir de aquí se produce operaciones de inserción hasta el instante t_4, donde, tras validarse el dato número 8 como muestra la señal num_entries, se activa la señal full indicando que la PIFO se encuentra llena, en el instante t_5, se comprueba el escenario de insertar un dato con el mínimo rango mientras está llena la PIFO, y se puede ver como efectivamente en el siguiente ciclo la señal rank_out aparece este último dato insertado tras ser validado, eliminándose así el dato con mayor rango indicado por la señal max_rank_out.

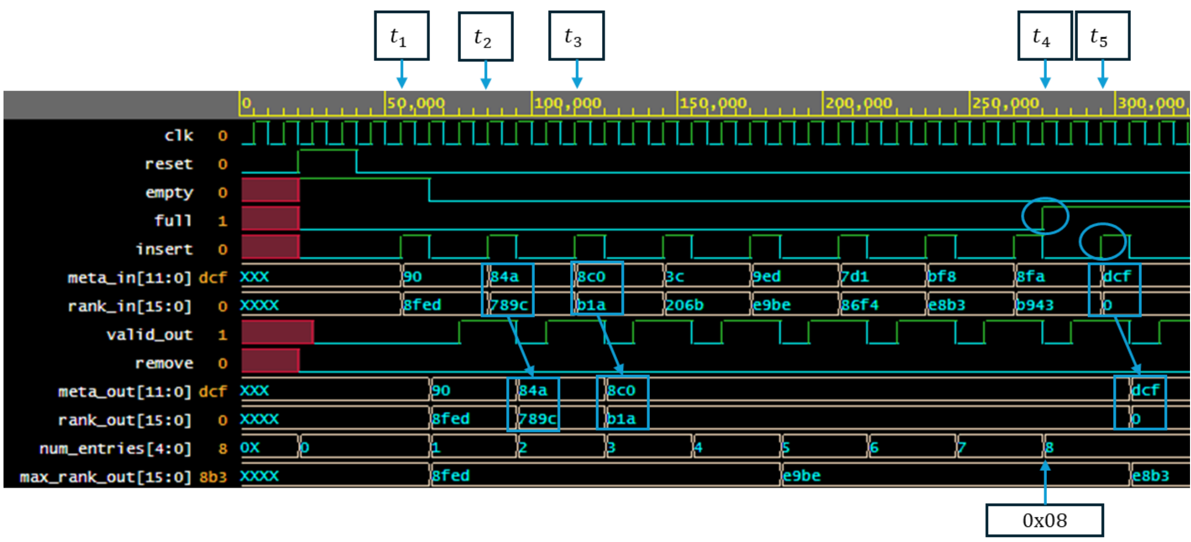


Figura 29 Fase de ejecución del test `t_i04`

- **Test `t_r02`:** El Código 45 muestra la fase de ejecución del test `t_r02`. Antes de su ejecución, se definen dos secuencias, una secuencia `insert_into_full` (línea 2) que llena la PIFO con datos aleatorios y una secuencia de inserción de datos `remove_into_empty`, línea 3 que elimina los datos de la PIFO. El comienzo de la tarea se señaliza activando el mecanismo de `objections`, línea 5, lo que se consigue con la llamada al método `raise_objections` de la clase `phase`.

  En las líneas 6 y 7 se crean las secuencias mediante el mecanismo de *Factory*, las cuales se denominan `seq_1`, para atacar la interfaz de entrada, y `seq_2`, para la interfaz de salida. Finalmente, se envían dichas secuencias al componente *driver* a través del componente *sequencer* correspondiente. Esto se muestra en las líneas 8 y 9.

```
1   virtual task run_phase(uvm_phase phase);
2     insert_into_full seq;
3     remove_into_empty seq_r;
4     super.run_phase(phase);
5     phase.raise_objection(this);
6     seq = insert_into_full::type_id::create("seq");
7     seq_r = remove_into_empty::type_id::create("seq_r");
8     seq.start(env.pifo_insert.agent.sequencer);
9     seq_r.start(env.pifo_remove.agent.sequencer);
10    phase.drop_objection(this);
11  endtask: run_phase
```

Código 45 Fase de ejecución del test `t_r02`

En la Figura 30, tras la fase de `reset`, en el instante t_1, comienza nuevamente la secuencia encargada de las operaciones de inserción hasta llenar la PIFO, en este caso en el instante t_2 la señal `full` se activa, debido a que se configuró la variable `L2_REG_WIDTH` para asignar un tamaño de 4 a la PIFO. En el instante t_3 se valida la primera operación de extracción desactivando así la señal de `full`, a partir de este instante comienza la secuencia de operaciones de extracción hasta llegar al instante t_4, donde la señal `num_entries` toma el valor 0x0 y la señal `empty` se activa.

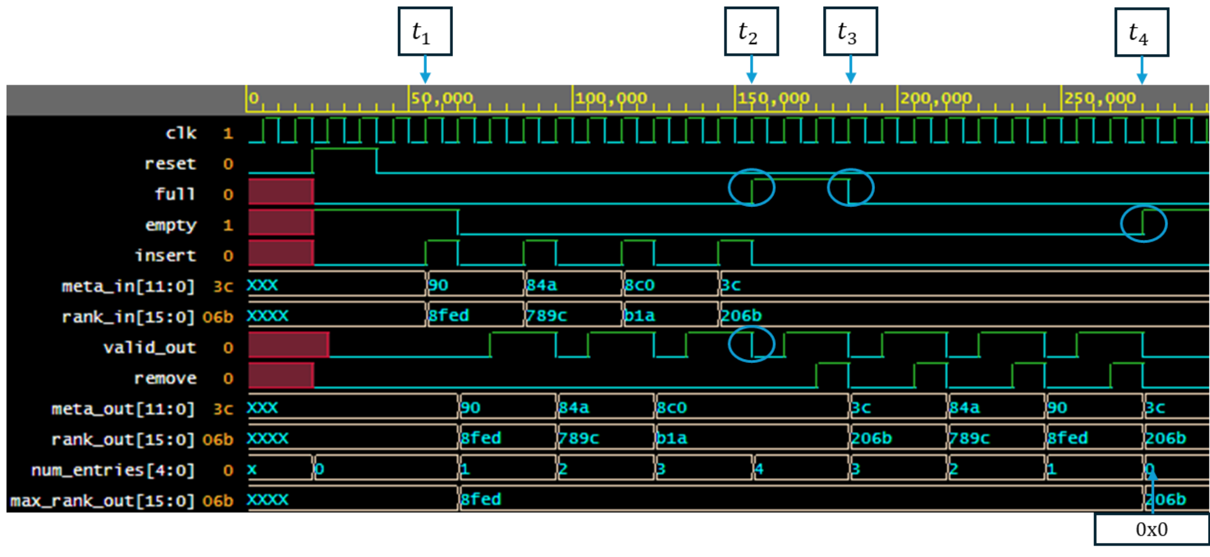


Figura 30 Fase de ejecución del test `t_r02`

Como resultado adicional, en la Figura 31 se muestra el log de la ejecución del test `t_r02` haciendo uso del mecanismo de mensajes que facilita la depuración en UVM, para ello, es necesario modificar el parámetro de configuración `+UVM_VERBOSITY` y establecer el valor en este caso a `MEDIUM`, para poder filtrar el resto de los mensajes dedicados a puramente depuración del código y mostrar el contenido de la PIFO en cada operación de inserción y extracción.

```
pifo_scoreboard] [SCB] Push_front queue_rank = '{36845} at: 65000
pifo_scoreboard] [SCB] Insert queue_rank = '{30876, 36845} at: 95000
[pifo_scoreboard] [SCB] Insert queue_rank = '{2842, 30876, 36845} at: 125000
[pifo_scoreboard] [SCB] Insert queue_rank = '{2842, 8299, 30876, 36845} at:
[pifo_scoreboard] [SCB-PASS!] DatoEsperado = 2842, DatoEliminado = 2842
[pifo_scoreboard] [SCB] Remove queue_rank = '{8299, 30876, 36845} at: 185000
[pifo_scoreboard] [SCB-PASS!] DatoEsperado = 8299, DatoEliminado = 8299
[pifo_scoreboard] [SCB] Remove queue_rank = '{30876, 36845} at: 215000
[pifo_scoreboard] [SCB-PASS!] DatoEsperado = 30876, DatoEliminado = 30876
[pifo_scoreboard] [SCB] Remove queue_rank = '{36845} at: 245000
[pifo_scoreboard] [SCB-PASS!] DatoEsperado = 36845, DatoEliminado = 36845
[pifo_scoreboard] [SCB] Remove queue_rank = '{} at: 275000
```

Figura 31 Resultados ejecución test t_r02 mecanismo mensajes UVM

Capítulo 3. Arquitectura RISC-V

Como se ha mencionado al comienzo del documento, RISC-V es una especificación ISA abierta que utiliza el paradigma de diseño de arquitecturas RISC. Fue creada con objetivos educativos, para el estudio de la arquitectura de computadores. El principal esfuerzo de diseño trata de cumplir con las ocho grandes ideas en el diseño de arquitecturas de computadores:

- **Ley de Moore**. Establece que los recursos de los circuitos integrados se duplican cada 18 a 24 meses. Como el diseño de una nueva arquitectura de procesador pueden llevar años, los recursos disponibles por chip pueden duplicarse o cuadriplicarse fácilmente entre el inicio y el final del proyecto. Situación que obliga a los diseñadores a anticiparse y ubicarse en el estado de la tecnología[18].
- **Abstracción.** Tanto a nivel de diseño como de programación, ha sido necesario desarrollar técnicas para incrementar la productividad. En caso contrario, el incremento de la capacidad de integración, prevista por la Ley de Moore, incrementa de forma drástica el tiempo de diseño. Una técnica importante de productividad para hardware y software es usar abstracciones para caracterizar el diseño en diferentes niveles de representación; los detalles de nivel inferior están ocultos para ofrecer un modelo más simple a niveles más altos.
- **Incremento de prestaciones del caso común rápido.** Es la idea de enfocarse en el caso común en vez de tratar de optimizar los casos más raros, Irónicamente, el caso común suele ser más simple que el caso raro y, por lo tanto, suele ser más fácil de mejorar. Esta idea de sentido común es posible con una cuidadosa experimentación y medición.

- **Incremento mediante la utilización de paralelismo.** Como su nombre indica, aplicar técnicas de paralelismo en los diseños llevan a mejores resultados de rendimiento.
- **Incremento de rendimiento a través de la utilización de técnicas de segmentación (*pipelining*).** Como resultado del paralelismo, una de las ideas fundamentales para el incremento de las prestaciones en un procesador es la utilización de técnicas de *pipelining* Básicamente, la idea de segmentar el cauce de procesamiento para incrementar el rendimiento del sistema. Esta técnica implica tanto la utilización un control detallado en el diseño hardware como el desarrollo de un compilador optimizado.
- **Predicción.** En algunos casos, en promedio, puede ser más rápido predecir y comenzar a trabajar en lugar de esperar hasta haber realizado el cálculo de las condiciones de ejecución, suponiendo que el mecanismo para recuperarse de una predicción errónea no sea demasiado costoso y que su predicción sea relativamente precisa.
- **Jerarquía de memoria.** En general, un programador requiere que la memoria sea rápida, grande y barata, ya que la velocidad de la memoria a menudo determina el rendimiento, la capacidad limita el tamaño de los problemas que se pueden resolver y el costo de la memoria hoy en día suele ser la mayor parte del costo de la computadora.
- **Fiabilidad mediante redundancia.** El procesador no solo necesita ser rápido; necesita ser confiable. Dado que cualquier dispositivo físico puede fallar, los diseñadores hacen que los sistemas sean confiables al incluir componentes redundantes que pueden tomar el control cuando ocurre un fallo y ayudar a detectar estos fallos.

3.1. Variantes de RISC-V

RISC-V incluye una colección de opciones derivadas de la arquitectura ISA básica, donde se utiliza una convención de nomenclatura asociado a un hardware concreto, además es posible añadir extensiones a la ISA base. Para comprender esta nomenclatura se analiza el siguiente ejemplo: **RV32IM**

- **RV:** Las dos primeras letras son la abreviación de RISC V y todos los nombres comienzan de esta forma
- **M:** Extensión que indica que el hardware soporta la multiplicación y la división.

- **32:** Este número indica el número de bits que tiene de ancho el banco de registros, en este ejemplo es de 32 bits. Las opciones disponibles son 32, 64 y 128 bits.
- **I:** Indica que se soporta la aritmética de números enteros, forma parte del ISA base.

Las extensiones **estándar** disponibles son:

- M para multiplicación y división,
- A para instrucciones atómicas,
- F para soporte de punto flotante de precisión simple (32 bits)
- D para soporte de punto flotante de precisión doble (64 bits) [19].

Se puede configurar un ISA con todas las extensiones de forma que el nombre sería RV32IMAFD, existiendo la posibilidad de sustituir IMAFD por la letra "G" de forma que se sobreentiende que se soportan todas las extensiones estándar. Además de estas extensiones existen otras más adicionales:

- S para trabajar en modo supervisor,
- Q para soporte de punto flotante de precisión *quad*,
- C para trabajar con instrucciones comprimidas de 16 bits mediante el seguimiento de una serie de reglas.

3.2. Características técnicas de RISC-V

Tabla 4 Características técnicas RISC-V [20]

Origen	Universidad de California, Berkeley
Gestión	RISC-V Foundation.
Licencia	BSD (abierta y gratuita).
Tamaño de palabra	32 bits, 64 bits, 128 bits (con extensiones SIMD o vectoriales).
Tipo	RISC, load/store.
Núcleo	RV32I, RV64I, RV128I (no cambia, tiene extensiones modulares).
Extensiones modulares	M (instrucciones para multiplicar y dividir). A (operaciones atómicas). F (aritmética de punto flotante de precisión simple). D (aritmética punto flotante de precisión doble). Q (aritmética punto flotante de precisión cuádruple). L (aritmética punto flotante decimal). C (instrucciones comprimidas). B (manipulación de bits). J (Lenguajes traducidos dinámicamente). P (para instrucciones SIMD). V (vectoriales). E (aplicaciones de sistemas empotrados) y G (conjunto M, A, F y D).
Tipos de registros	De propósito general (16 o 32, registros x0: siempre a valor 0x0).
Codificación	Variable.
Branching	Comparación y saltos.
Endianness	Little-endian.
ISA modular	Se pueden ir agregando módulos a partir del ISA fijo.

3.3. Formato de instrucciones RISC-V

Para resumir los contenidos se explican a continuación las instrucciones de RISC-V basado en el ISA base RV32I, más adelante se mostrarán detalles para el *core* elegido.

En RV32I las instrucciones están codificadas en palabras de 32 bits y se almacenan por defecto de forma alineada en la memoria (4 bytes) con el formato *Little-endian*, aunque es posible adaptar el hardware para trabajar en el modo *big-endian.* Existen seis formatos básicos de instrucciones Figura 32:

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	Tipo R
imm[11:0]				rs1	funct3	rd		opcode	Tipo I
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	Tipo S
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	Tipo B
imm[31:12]						rd		opcode	Tipo U
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	Tipo J

Figura 32 Formato de instrucciones [21]

Los seis formatos básicos de instrucción RV32I, ISA base, son los siguientes [21]:

- **Tipo R**: instrucciones para operaciones entre registros. Se leen los valores contenidos en dos registros de 32 bits y se escribe el resultado en el registro destino.
- **Tipo I**: instrucciones para inmediatos cortos y *loads*.
- **Tipo S**: instrucciones para *stores* (acceso a memoria).
- **Tipo B**: instrucciones de control de flujo (*branches*).
- **Tipo U**: instrucciones para inmediatos largos.
- **Tipo J**: instrucciones para saltos incondicionales.

Como se puede ver en la Figura 32**,** la codificación de la instrucción se ubica desde el bit ‘0’ hasta el bit ‘6’, es decir, los 7 primeros bits pertenecen al *opcode*, que indica el tipo de instrucción. En caso de guardar algún tipo de información en un registro de destino, éste viene indicado en los bits (7-11), es decir, se necesitan 5 bits ya que tenemos 32 registros. Lo siguiente es el campo funct3, que determina que operación se va a realizar dentro del tipo de instrucción, por ejemplo, en caso de un *opcode* que indica tipo R, funct3 determinará si se trata de un ADD, SUB, SLT, etc.

En cuanto a los registros de origen, correspondiendo el primer registro a los bits (15-19) y el segundo, si está presente, estará siempre en los 5 bits que van del 20 al 24. También, si el tipo de instrucción necesita un dato inmediato, estos vendrán en sus respectivos campos. Para dar una visualización más amplia de todas las configuraciones posibles se muestra la Figura 33, donde se presenta las 47 instrucciones que ofrece RV32I con la información acerca de la estructura de la instrucción, *opcodes*, tipo de formato y nombres.

3.3.1. Computación Entera

Las instrucciones aritméticas sencillas (add, sub), instrucciones lógicas (and, or, xor) e instrucciones de desplazamiento (sll, srl, sra) realizan la función que se espera de cualquier tipo de ISA. Leen los valores procedentes de cualquiera de los 32 registros de origen y escriben el resultado al registro destino. Además, RISC-V cuenta con versiones inmediatas de estas instrucciones (addi, slli, slti, sltiu, xori, srli, srai, ori, andi).

3.3.2. Load y Store

En RV32I se dispone de instrucciones de *load* y *store* de palabras de 32 bits (lw, sw) y también *bytes* y *halfwords*, tanto en versión *signed* o *unsigned* (lb, lbu, lh, lhu, sb, sh). *Bytes* y *halfwords* con signo hacen *sign-extensión* a 32 bits y son escritos en el registro destino. Esta extensión de datos permite que, aunque el dato sea más corto de 32 bits, las operaciones aritméticas posteriores operen correctamente. *Bytes* y *halfwords* sin signo, se extienden con cero a 32 bits.

3.3.3. Saltos Condicionales

Los saltos condicionales se gestionan mediante instrucciones de tipo B. En RV32I se puede comparar dos registros y saltar si el resultado es igual (beq), distinto (bne), mayor o igual (bge), o menor (blt). Además, también están disponibles las versiones sin signo (bgeu, bltu).

31 25	24 20	19 15	14 12	11 7	6 0	
imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw
imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	R sltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and
0000 pred succ		00000	000	00000	0001111	I fence
0000 0000 0000		00000	001	00000	0001111	I fence.i
000000000000		00000	000	00000	1110011	I ecall
000000000001		00000	000	00000	1110011	I ebreak
csr		rs1	001	rd	1110011	I csrrw
csr		rs1	010	rd	1110011	I csrrs
csr		rs1	011	rd	1110011	I csrrc
csr		zimm	101	rd	1110011	I csrrwi
csr		zimm	110	rd	1110011	I csrrsi
csr		zimm	111	rd	1110011	I csrrci

Figura 33. Mapa de opcodes de RV32I [21]

3.3.4. Saltos Incondicionales

La instrucción *jump* and *link* (`jal`) en llamadas a funciones, almacena la dirección de la siguiente instrucción PC+4 en el registro destino, normalmente el registro `ra`. Para los saltos incondicionales utiliza el registro cero (`x0`) en lugar de `ra` ya que no cambia.

En RV32I está también la instrucción *jump* and *link* con registro (`jalr`) que puede hacer una llamada a función a una dirección de memoria calculada dinámicamente o retornar de la función usando `ra` como registro origen, y el registro cero como destino. La instrucción `jalr` es útil para operaciones de *switch* o *case*, que calculan la dirección a saltar.

3.3.5. Otras instrucciones

Las instrucciones de *control* y *status register* (`csrrc`, `csrrs`, `csrrw`, `csrrci`, `csrrsi`, `csrrwi`) proveen acceso fácil a registros que ayudan a medir el rendimiento de un programa. Comportándose como contadores de 64 bits, que pueden ser leídos en 32 bits midiendo el tiempo, ciclos ejecutados y número de instrucciones retiradas.

Por otro lado, la instrucción `ecall` hace solicitudes al entorno de ejecución, tales como llamadas al sistema. Los depuradores utilizan la instrucción `ebreak` para transferir el control al entorno de depuración. La instrucción `fence` coordina accesos a dispositivos de I/O.

3.4. Ejecución de instrucciones

A continuación, se presenta un ejemplo práctico de la ejecución de una instrucción sobre el flujo de datos que sigue la arquitectura RISC-V de un procesador mínimo (Figura 34). Para ello se hace uso de emulsiV [22], un simulador online desarrollado para mostrar el funcionamiento de la arquitectura y el diseño digital de un procesador RISC-V.

La interfaz gráfica que muestra el programa es intuitiva, presentando el flujo de datos en las diferentes etapas de la ejecución de la instrucción.

La instrucción a ejecutar es: `addi x1, x0, 32,` es decir, se quiere realizar una suma entre el valor contenido en el registro `x0`, que como ya se ha mencionado siempre tiene el valor 0x0 y el valor inmediato, en este caso 32, y almacenar el resultado 32 en el registro destino `r1`.

La primera etapa del ciclo de ejecución de la instrucción es la etapa de *Fetch* donde determina la siguiente instrucción a ejecutar. Para ello se consulta al registro PC, que en este caso al ser la primera instrucción tiene el valor “00000000” por lo tanto se busca en la memoria la

dirección apuntada por PC y se carga la instrucción en el bus de datos de la memoria de instrucciones. Dicha instrucción se almacena en el registro de instrucciones (IR).

La siguiente etapa es la decodificación (Figura 35) de la instrucción, que gracias a la lógica utilizada y la normalización de las instrucciones es capaz de decodificar dicha instrucción, donde el registro de instrucciones interpreta esa serie de bits y extrae la información sobre qué tipo de instrucción es, cual es el registro de destino y origen si lo hubiera (no es el caso) y extrae el valor inmediato.

El siguiente paso es realizar la operación en la ALU (*arithmetic logic unit*) (Figura 36). Para ello, la ALU entiende la operación a realizar y recibe del banco de registros el valor que tiene almacenado en este caso `x0` y recibe del registro de instrucciones el valor inmediato 32 y realiza la operación obteniendo un resultado.

La siguiente etapa es la de memoria ya que en este ejemplo no hay comparaciones que realizar, simplemente se guarda el valor del resultado en el registro destino `r1`.

Por último, se procede a la etapa de PC (Figura 37) donde se incrementa el valor de PC para apuntar a la siguiente instrucción (Figura 38).

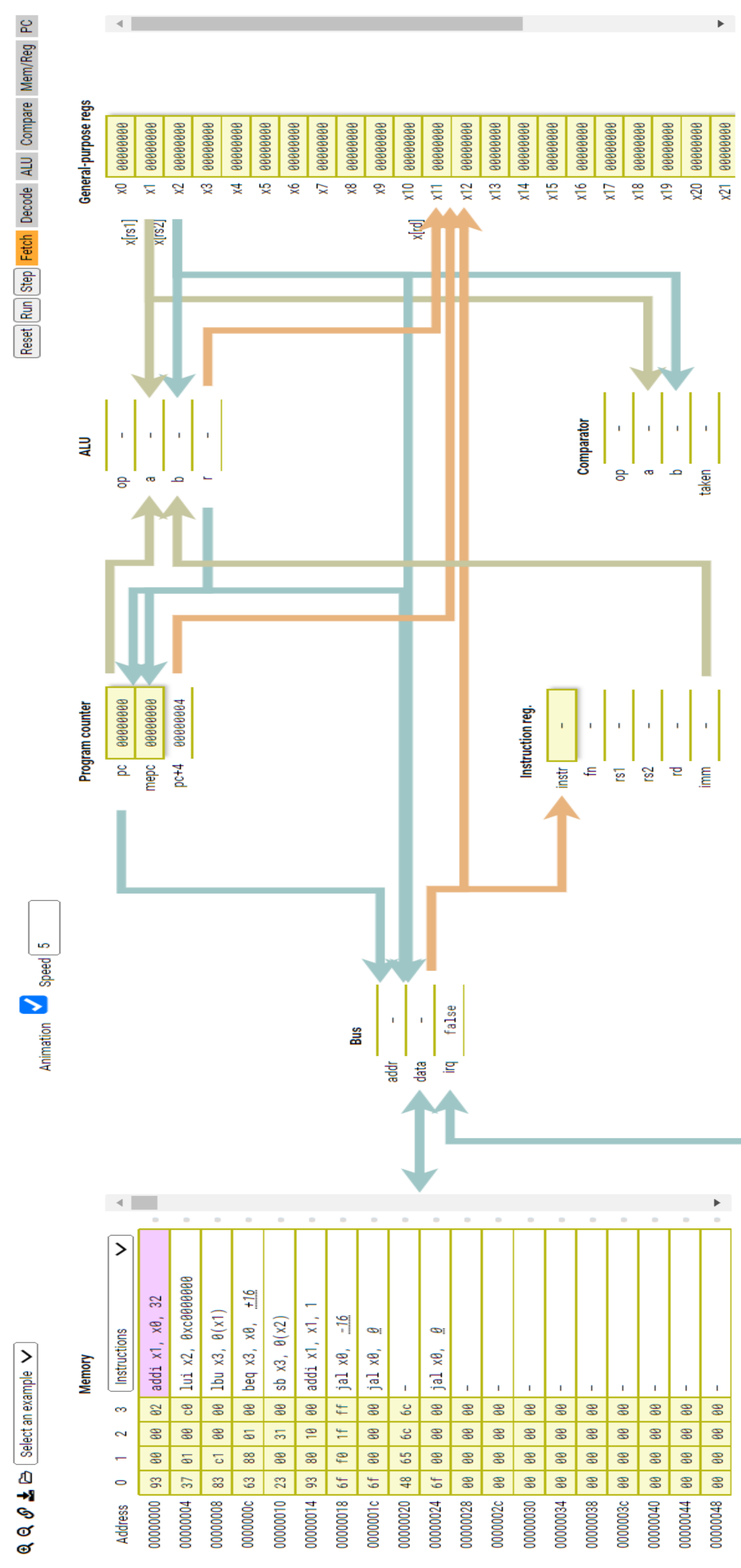


Figura 34. Etapa de Fetch del caso práctico addi x1, x0, 32

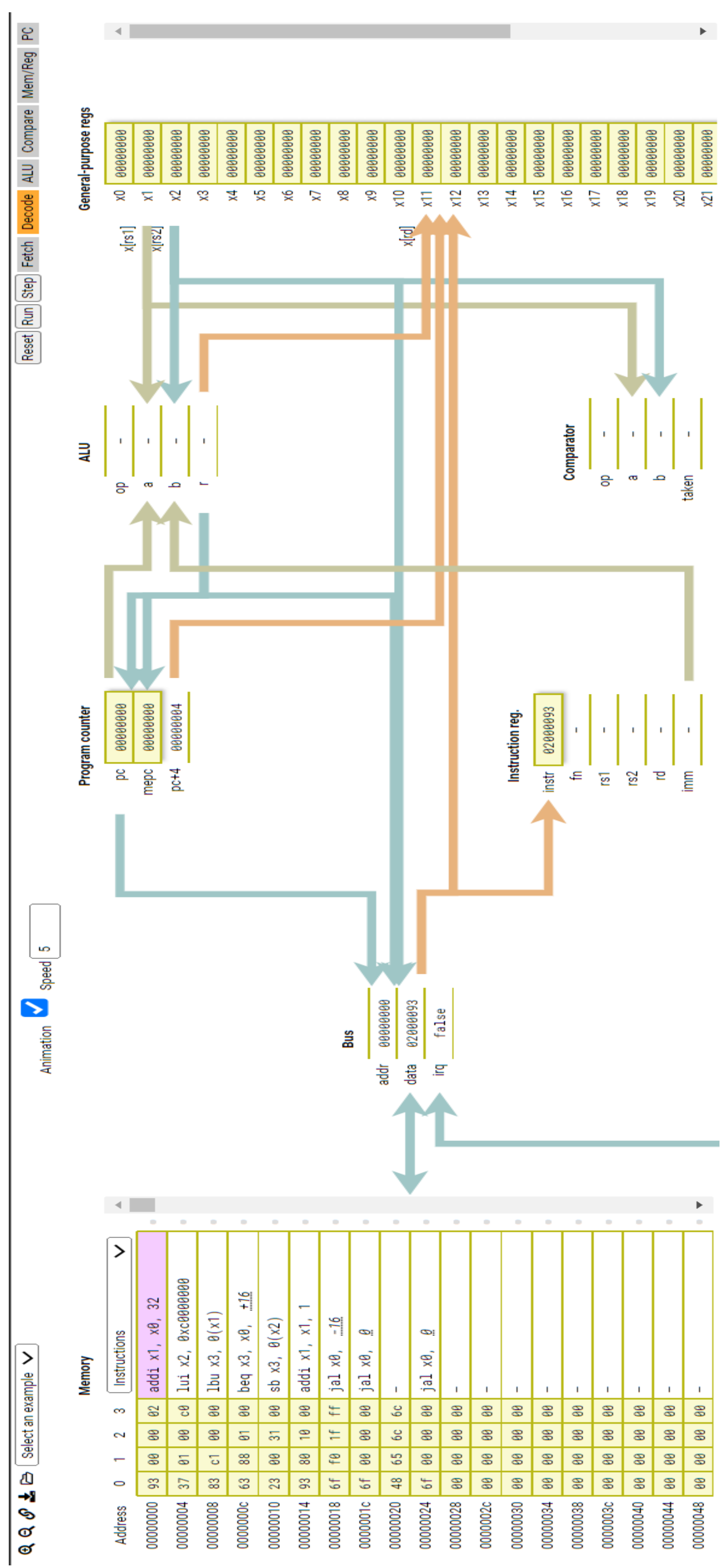


Figura 35. Etapa de DECODE

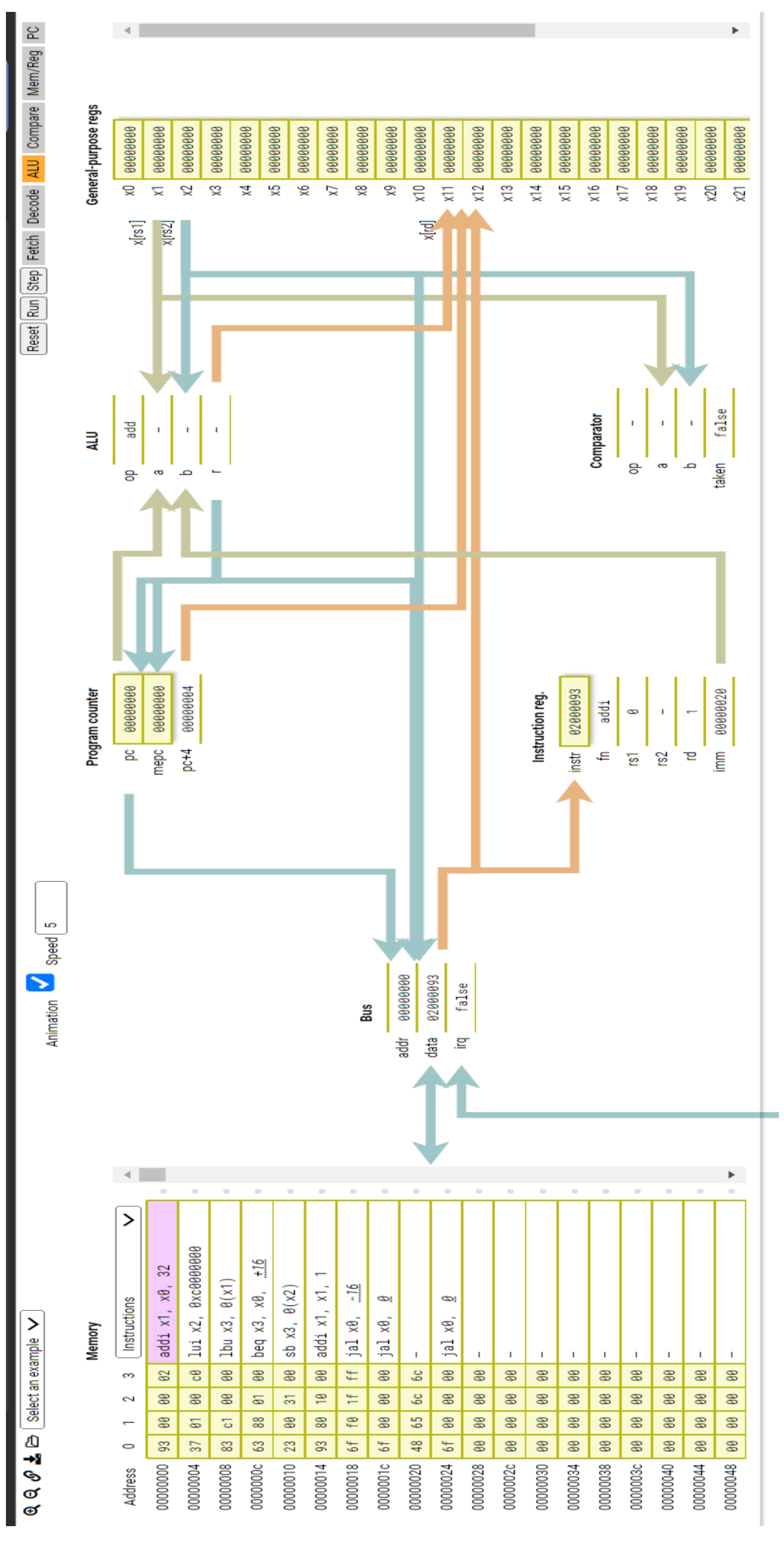


Figura 36. Etapa de ALU (Execute)

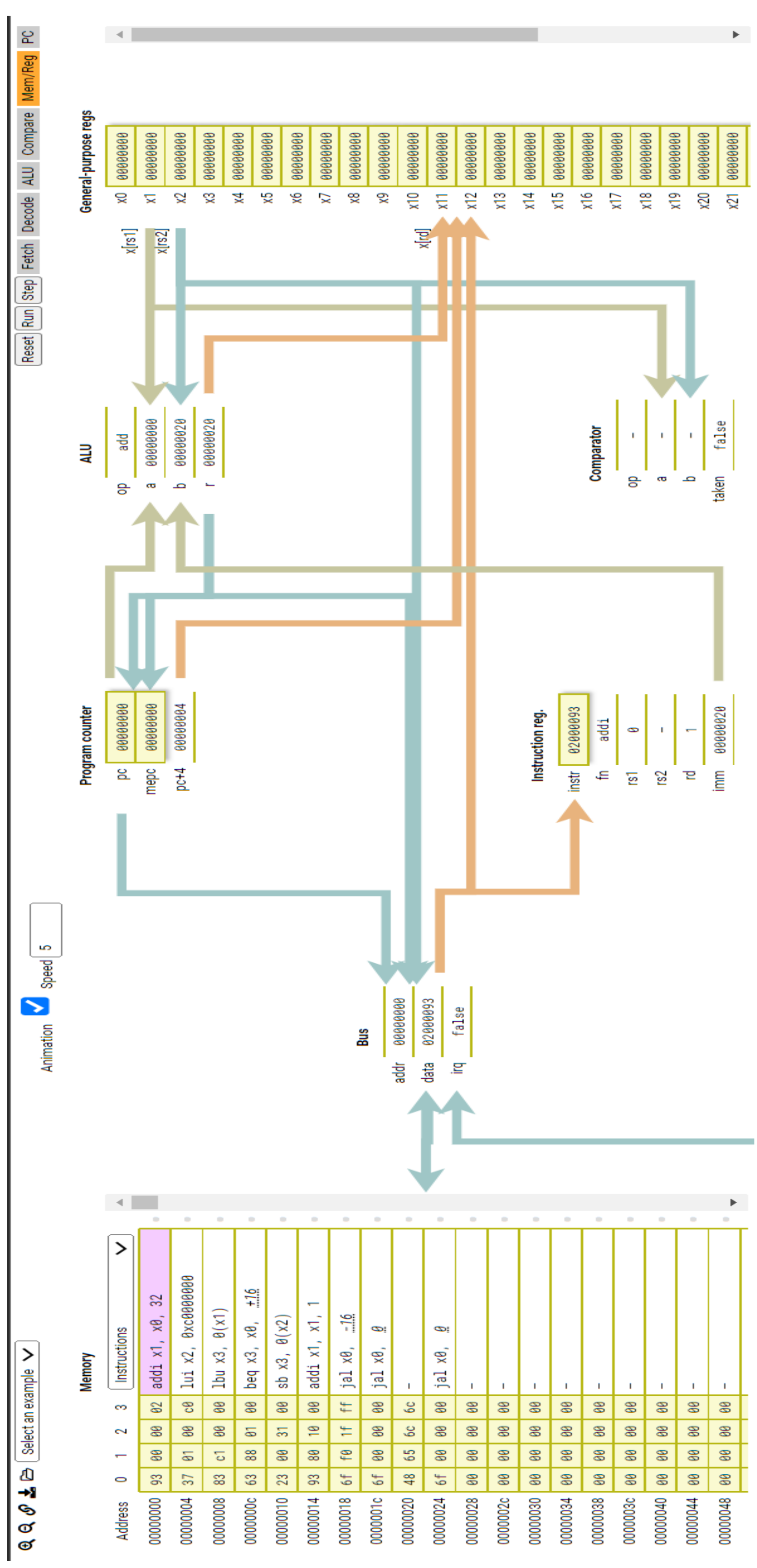


Figura 37. Etapa de escritura en memoria (Mem/Reg) y PC

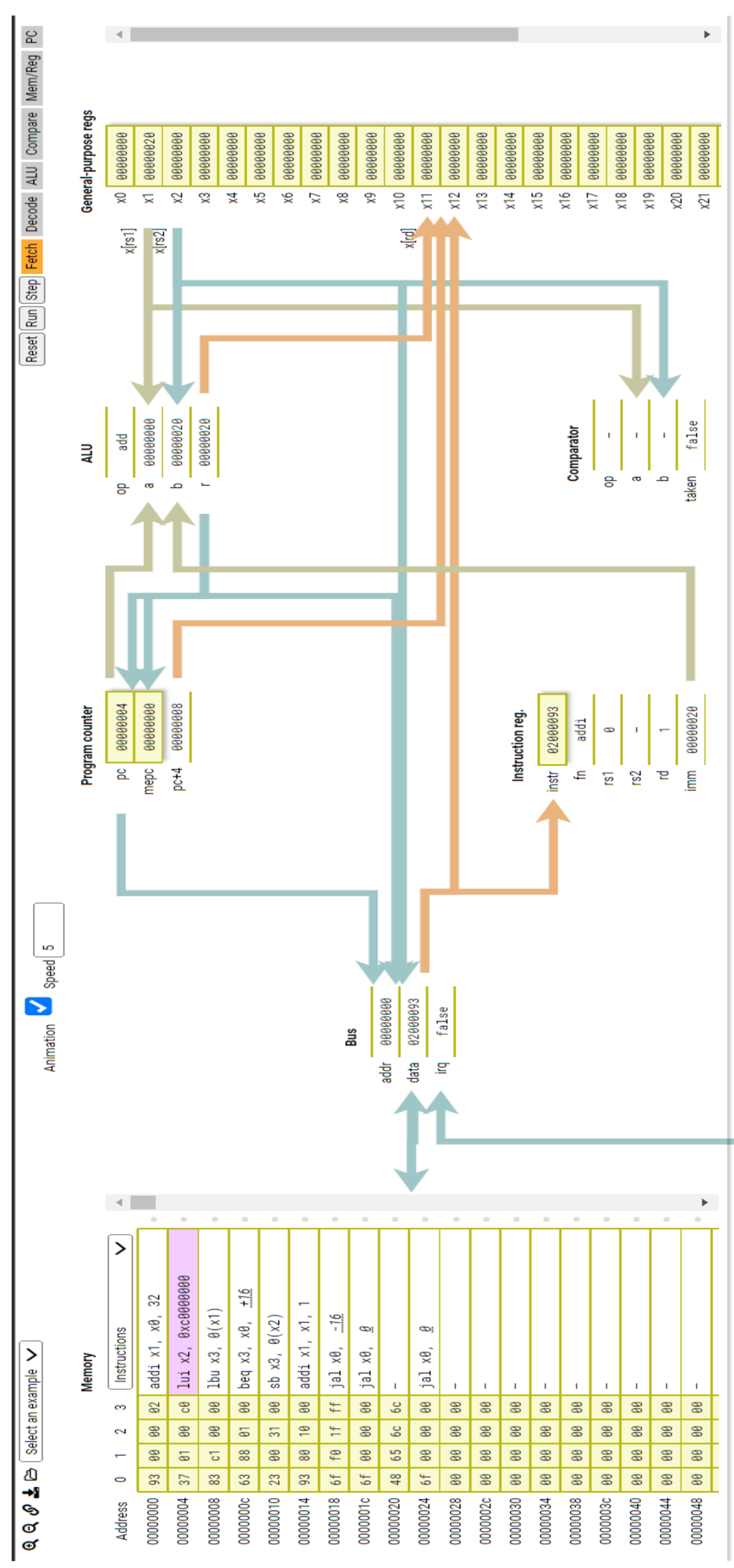


Figura 38. Fetch de la siguiente instrucción

3.5. Registros

En RISC-V todas las operaciones se realizan entre registros, salvo *load/store* que sí utilizan accesos a memoria. Tanto para RV32 como RV64 y RV128 se utilizan siempre 32 registros. RISC-V soporta una ABI (*Application Binary Interface)* común que facilita la coherencia entre las distintas implementaciones, manteniendo la misma nomenclatura y propósitos entandares para cada registro con el fin de facilitar las tareas tanto a los programadores como a los compiladores. La notación que se sigue es muy sencilla "xN" siendo N el número de registro. Por ejemplo, el registro x1 es el registro número 1.

El registro `x0` siempre contiene el valor constante 0x0. No se le puede asignar otro valor, se utiliza principalmente para inicializar otros registros a 0, esto tiene un impacto tremendo a la hora de simplificar el ISA de RISC-V.

El registro PC (*Program counter),* contiene la dirección de la siguiente instrucción a ejecutar. En la Tabla 5 se representa los diferentes registros con su nomenclatura, su nombre y el uso dedicado a cada uno de ellos.

Tabla 5. Convención de registros en RISC-V [23]

Registro	Nombre	Uso
x0	zero	Valor constante 0
x1	ra	*Return Address:* Dirección de retorno de la función
x2	sp	*Stack pointer:* puntero del *stack*
x3	gp	*Global Pointer:* Puntero global a los datos
x4	tp	*Thread Pointer:* Puntero de hilo
x5-x7	t0-t2	*Temporaries:* Registros de propósito general que no conservan el valor entre funciones.
x8	s0/fp	*Saved Register/Frame Pointer:* Registro seguro que conserva el valor entre funciones. /Contiene la dirección anterior a la llamada de una función.
x9	s1	*Saved Register*: Registro seguro de propósito general que conserva el valor entre funciones.
x10-x11	a0-a1	*Function arguments/Return values:* Registros utilizados para pasar argumentos a las funciones o como valor de retorno de las funciones.
x12-x17	a2-a7	*Function arguments:* Registros utilizados para pasar argumentos a las funciones.
x18-x27	s2-s11	*Saved Registers:* Registros seguros de propósito general que conservan el valor entre funciones.
x28-x31	t3-t6	*Temporaries:* Registros de propósito general que no conservan el valor entre funciones.

3.6. Modos de direccionamiento

Los modos de direccionamiento son los diferentes procedimientos que determinan la ubicación de un operando o una instrucción en memoria. Las arquitecturas de computadores varían en cuanto al número de modos de direccionamiento que ofrecen desde el hardware. Se ha comprobado que cuando los modos de direccionamiento son simples, el diseño de CPUs segmentadas es mucho más eficiente [[24]].

Cuando existen pocos modos de direccionamiento, van codificados directamente dentro de la propia instrucción (IBM/390, y en la mayoría de los RISC); pero cuando hay demasiados, ocurre, al contrario, ya que a menudo tiene un campo específico en la propia instrucción para especificar dicho modo de direccionamiento [21].

En RISC-V nos encontramos principalmente con cuatro modos de direccionamiento:

- **Indirecto**. El único modo de direccionamiento para *loads* y *stores* consiste en sumar un valor inmediato de 12 bits al contenido de un registro.
- **Direccionamiento relativo a PC.** Se usa en las instrucciones de saltos. Dado que las instrucciones de RISC-V deben ser múltiplos de dos bytes el modo de direccionamiento de saltos multiplica el valor inmediato de 12 bits por 2, le extiende el signo y lo suma al PC. En la instrucción `jal` se usa este mismo modo, multiplicando la dirección de 20 bits por 2, se extiende el signo y se suma el resultado al PC para obtener la dirección a saltar.
- **Directo a registro.** El operando se encuentra en un registro.
- **Inmediato.** El operando está especificado directamente en la instrucción.

3.7. Modos privilegiados del procesador

RISC-V incluye distintos modos de funcionamiento. En primer lugar, da soporte al funcionamiento de propósito general, es decir, la ejecución de aquellas instrucciones que se encuentran en el modo usuario, donde usualmente ejecuta las aplicaciones. En segundo lugar, cuenta con dos modos privilegiados: el modo máquina que ejecuta el código más fiable y el modo supervisor que provee soporte para sistemas operativos como Linux, FreeBSD y Windows. Ambos modos son más privilegiados que el modo usuario y por tanto incluyen todas las características de sus modos menos privilegiados añadiendo funcionalidades adicionales tales como la habilidad de manejar interrupciones y controlar I/O.

Cuando un hilo hardware (*hart*) se ejecuta en un nivel de privilegio determinado, este es codificado en el CSR correspondiente. Actualmente hay definidos cuatro niveles diferentes de privilegio en la ISA RISC-V, como se muestra en Tabla 6

Tabla 6 Niveles de privilegio de RISC-V

Nivel	Codificación	Nombre	Abreviatura
0	00	Usuario	U
1	01	Supervisor	S
2	10	Hipervisor	H
3	11	Máquina	M

Los niveles de privilegio se usan para proporcionar protección entre diferentes componentes del *stack* software; los intentos de operación que no estén permitidos en el nivel de ejecución causarán una excepción.

3.7.1. Modo Máquina

Se trata del modo más privilegiado. En este modo los *harts (*hardware *threads:* hilo de ejecución en hardware), tienen acceso completo a la memoria, I/O y funcionalidades necesarias para configurar el sistema. Por tanto, se trata del único modo de privilegio que se implementa en todos los procesadores RISC-V estándar.

La característica más importante del modo máquina es la capacidad de interceptar y manejar las excepciones mostradas en la Figura 39 donde el bit más significativo de *mcause* es puesto a '1' para interrupciones o en '0' para excepciones síncronas.

Interrupción / $\overline{\text{Excepción}}$ mcause[XLEN-1]	Código de Excepción mcause[XLEN-2:0]	Descripción
1	1	Interrupción de software de Supervisor
1	3	Interrupción de software de Máquina
1	5	Interrupción de temporizador de Supervisor
1	7	Interrupción de temporizador de Máquina
1	9	Interrupción externa de Supervisor
1	11	Interrupción externa de Máquina
0	0	Dirección de instrucción desalineada
0	1	Fallo de acceso en instrucción
0	2	Instrucción ilegal
0	3	Breakpoint
0	4	Dirección de Load desalineada
0	5	Fallo de acceso en Load
0	6	Dirección de Store desalineada
0	7	Fallo de acceso en Store
0	8	Llamada al entorno desde modo U
0	9	Llamada al entorno desde modo S
0	11	Llamada al entorno desde modo M
0	12	Fallo de página en instrucción
0	13	Fallo de página en Load
0	15	Fallo de página en Store

Figura 39. Causas de excepciones e interrupciones en RISC-V [21]

3.7.1.1. `mhartid` (registro ID del *hart*)

Este registro tiene un tamaño que depende de la implementación RISC-V. El registro contiene un número que identifica el ID del hilo hardware en ejecución del código. Además, se debe cumplir la condición de que la lectura de este registro se pueda realizar en cualquier implementación. No hace falta que los hilos de hardware estén enumerados de manera contigua, pero si debe haber un hilo cuyo ID sea 0.

3.7.1.2. `mstatus` (Registro de estado de máquina)

La longitud del registro `mstatus` depende de la implementación RISC-V, para RV32 el registro `mstatus` tiene un XLEN = 32, mientras que para RV64 XLEN = 64. Este registro además de presentar el valor de una gran cantidad de estados realiza una de las funciones más importantes dentro del control de interrupciones y es que contiene el habilitador global de interrupciones, la estructura del registro se muestra en la Figura 40.

XLEN-1	XLEN-2 … 23	22	21	20	19	18	17
SD	*Reservado*	TSR	TW	TVM	MXR	SUM	MPRV
1	XLEN-24	1	1	1	1	1	1

16 15	14 13	12 11	10 9	8	7	6	5	4	3	2	1	0
XS	FS	MPP	*Res.*	SPP	MPIE	*Res.*	SPIE	*Res.*	MIE	*Res.*	SIE	*Res.*
2	2	2	2	1	1	1	1	1	1	1	1	1

Figura 40. Estructura del registro mstatus del nivel de máquina

Cuando se ejecuta el procesador en modo **M**, las interrupciones solo se toman si el bit de habilitación de interrupciones global, `mstatus.MIE`, es 1. Además, cada interrupción tiene su propio bit de habilitación en el CSR `mie`. Las posiciones de los bits en `mie` corresponden a los códigos de interrupción de la Figura 39: por ejemplo, `mie`[7] corresponde a la interrupción de temporizador en modo **M**. El CSR `mip` tiene la misma estructura e indica cuáles interrupciones se encuentran pendientes. Juntando los tres CSRs, una interrupción de temporizador de máquina puede tomarse si `mstatus.MIE=1`, `mie`[7]=1 y `mip`[7]=1. Todos estos registros mencionados son explicados en los siguientes apartados.

3.7.1.3. `mip` y `mie` (registros de interrupciones de máquina)

Por un lado, el registro `mip` lista las interrupciones que están actualmente pendientes por procesar, mientras que el registro `mie` realiza una lista de cuáles de dichas interrupciones puede

tomar el procesador y cuál debe ignorar. Para ello, la estructura que sigue ambos registros es similar como se muestra en la Figura 41

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0	
Reservado	MEIP	*Res.*	SEIP	*Res.*	MTIP	*Res.*	STIP	*Res.*	MSIP	*Res.*	SSIP	*Res.*	mie
Reservado	MEIE	*Res.*	SEIE	*Res.*	MTIE	*Res.*	STIE	*Res.*	MSIE	*Res.*	SSIE	*Res.*	mip
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1	

Figura 41. Estructura registros mie y mip

Son registros de escritura y lectura que contienen los bits de interrupciones pendientes (`mip`) y habilitantes de interrupciones (`mie`). Solo es posible escribir en los bits correspondientes a las interrupciones del menor privilegio (`SSIP`), interrupciones de temporizador (`STIP`) e interrupciones externas (`SEIP`). El resto de los bits son solo de lectura.

3.7.1.4. `mcause` (Registro de causa de excepción de máquina)

Consiste en un registro de escritura y lectura cuya estructura se muestra en Figura 42. El tamaño dependerá de la implementación de la especificación ISA.

XLEN-1	XLEN-2 0
Interrupción	Código de Excepción
1	XLEN-1

Figura 42. Estructura del registro mcause

En caso de detectar una excepción y si fuese provocada por una interrupción el bit dedicado a ello se pondrá a 1. Dejando el resto de *bits* a 0 para futuras extensiones de la ISA.

3.7.1.5. `mtvec` (registro de direcciones de vectores de *trap* de la máquina)

Este registro contiene la dirección a la cual salta el procesador cuando ocurre una excepción, es decir, guarda la dirección del vector del *trap* que está siendo atendido por la máquina (Figura 43).

XLEN-1 2	1 0
BASE[XLEN-1:2]	MODE
XLEN-2	2

Figura 43. Estructura del registro mtvec

En cualquier implementación, el registro `mtvec` debe existir. Ya que el PC de la instrucción que causó la excepción es preservado en otro registro llamado `mepc`, y `mtvec` es escrito al PC.

3.7.1.6. `mepc` (PC de excepción de máquina)

Al igual que el resto de registro, el tamaño varía según la implementación RISC-V. Este registro apunta a la instrucción donde ocurrió la excepción. El bit LSB de este registro siempre vale 0, mientras que, en implementaciones sin instrucciones alineadas en 16 bits, deben ser los dos primeros bits menos significativos los que tomen el valor 0.

3.7.2. Modo usuario

Hay situaciones en las que no es adecuado confiar en todo el código de una aplicación, por lo que es necesario restringir el acceso del código que no se considera seguro a las instrucciones privilegiadas, con el fin de evitar al programa tener el control del sistema. Para conseguir estas restricciones tenemos disponible el modo usuario, que niega el acceso generando una excepción de instrucción ilegal cuando se intenta usar una instrucción o CSR (*Control and Status Registers*) de modo máquina.

Por lo demás, el modo **U** y el modo **M** se comportan muy similarmente. El software de modo **M** puede entrar al modo **U** poniendo `mstatus.MPP` en **U** (como muestra la Tabla 6, codificado como 0), luego ejecutando una instrucción mret. Si una excepción ocurre en modo **U**, el control es devuelto al modo **M.**

También es necesario restringir el acceso del código no confiable a su propia memoria. Para ello, el procesador tiene implementado la funcionalidad PMP (*Physical Memory Protection:* Protección física de memoria*)*, que permite al modo máquina especificar a que direcciones de memoria puede acceder el modo usuario.

31 24	23 16	15 8	7 0	
PMP3	PMP2	PMP1	PMP0	pmpcfg0
PMP7	PMP6	PMP5	PMP4	pmpcfg1
PMP11	PMP10	PMP9	PMP8	pmpcfg2
PMP15	PMP14	PMP13	PMP12	pmpcfg3

Figura 44. Estructura de configuraciones de PMP en los CSRs pmpcfg.

PMP consiste de varios registros de dirección (usualmente de ocho a dieciséis) y sus registros de configuración correspondientes, los cuales otorgan o niegan permisos de lectura, escritura y ejecución. Cuando un procesador en modo **U** intenta hacer *fetch* de una instrucción, o

ejecuta un load o store, la dirección es comparada contra todos los registros de dirección PMP. Si la dirección es mayor o igual que la dirección PMP i, pero menor que la dirección PMP i+1, entonces el registro de configuración de PMP i+1 decide si ese acceso puede proceder; de lo contrario, levanta una excepción de acceso. La Figura 44 muestra la estructura de un registro de dirección y configuración de PMP.

3.7.3. Modo supervisor

El esquema de PMP descrito en la sección anterior es atractivo para sistemas embebidos porque provee protección de memoria a un costo relativamente bajo, pero tiene varios inconvenientes que limitan su uso en computación de propósito general. Dado que PMP solo soporta una cantidad fija de regiones de memoria, no escala a aplicaciones complejas. Y como estas regiones deben ser contiguas en memoria física, el sistema puede sufrir de fragmentación de memoria.

Procesadores RISC-V más sofisticados tratan estos problemas de la misma manera que casi todas las arquitecturas de propósito general: usando memoria virtual basada en páginas. Esta funcionalidad forma el núcleo del modo supervisor (modo **S**), un modo de privilegio opcional diseñado para soportar sistemas operativos modernos similares a *Unix*, tales como *Linux*, *FreeBSD* y *Windows*. El modo **S** es más privilegiado que el modo **U**, pero menos privilegiado que el modo **M**. Al igual que en el modo **U**, el software del modo S no puede usar CSRs ni instrucciones del modo **M**, y está sujeto a restricciones de PMP.

RISC-V provee un mecanismo de delegación de excepciones, por el cual las interrupciones y excepciones síncronas pueden ser delegadas al modo **S** selectivamente, evitando software de modo **M** por completo.

3.7.3.1. `medeleg` (Registro delegación de interrupción de máquina)

Registro encargado de controlar cual de las interrupciones son delegadas al modo **S**. Al igual que `mip` y `mie`, cada bit en `mideleg` corresponde al código de excepción de la misma manera que ocurría anteriormente. Cualquier interrupción delegada al modo **S** puede ser enmascarada por software en modo **S**.

3.7.3.2. `sie` y `sip` (Registros de interrupciones de máquina)

El registro `sie` se encarga de habilitar la interrupción de supervisor y `sip` de listar las interrupciones de supervisor pendientes. Son CSRs de modo **S** y subconjuntos de los CSRs `mie` y

`mip`. Tienen la misma estructura que en el modo **M**, pero solo en los bits correspondientes a interrupciones que han sido delegadas en `mideleg` es posible leer y escribir con `sie` y `sip`.

3.7.3.3. `sstatus` (Registro de estado)

Este registro lleva a cabo la misma función que su contraparte de modo **M**, al ser un subconjunto de `mstatus`, la estructura del registro es similar. El tamaño también dependerá del tipo de implementación RISC-V Figura 45.

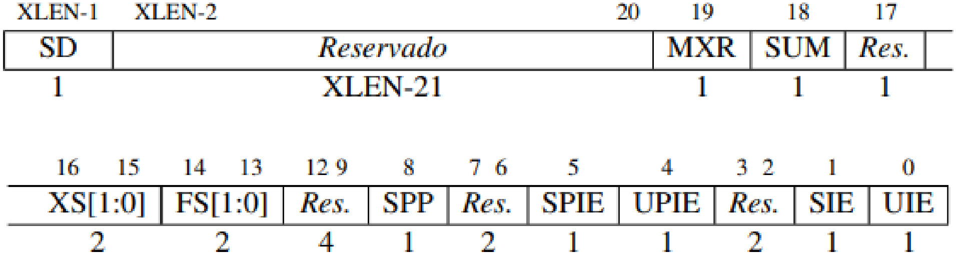


Figura 45. Estructura del registro sstatus del nivel de supervisor

3.8. Memoria virtual

El modo S provee un sistema convencional de memoria virtual que divide la memoria en páginas de tamaño fijo con los propósitos de traducción de direcciones y protección de memoria. Cuando la paginación está habilitada, la mayoría de las direcciones (incluyendo las direcciones efectivas de load, store y el PC) son direcciones virtuales que deben ser traducidas a direcciones físicas para tener acceso a la memoria física. Las direcciones virtuales son traducidas a direcciones físicas por medio del recorrido de un árbol, conocido como tabla de páginas.

Los esquemas de paginación de RISC-V son nombrados SvX, donde X es el tamaño de una dirección virtual en bits. El esquema de paginación de RV32, Sv32, soporta un espacio virtual de direcciones de 4 GiB, el cual está dividido en 2^{10} *megapáginas* de 4 MiB. Cada megapágina está subdividida en 2^{10} páginas base (unidad fundamental de paginación) cada una de 4 KiB.

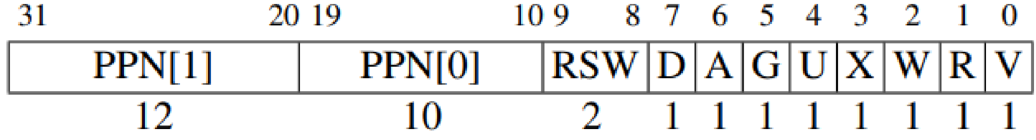


Figura 46. Una entrada de la tabla de páginas (PTE) de RV32 Sv32

La Figura 46 muestra la estructura de una PTE (Page Table Element: Entrada de la Tabla de Páginas) Sv32, la cual tiene los siguientes campos:

- **Bit V**: indica si el resto de esta PTE es válido (V=1). Si V=0, cualquier traducción de direcciones virtuales que pase por esta PTE resulta en un fallo de página.
- **Bits R, W y X**: indican si la página tiene permisos de lectura, escritura y ejecución, respectivamente. Si los tres bits son 0, esta PTE es un puntero al siguiente nivel de la tabla de páginas; de lo contrario, es una hoja del árbol.
- **Bit U**: indica si esta página es una página de usuario. Si U=0, el modo U no puede acceder a esta página, pero sí el modo S. Si U=1, El modo U puede acceder a esta página, pero el modo S no.
- **Bit G**: indica que este mapeo existe en todos los espacios virtuales de direcciones, información que el hardware puede usar para mejorar el rendimiento de la traducción de direcciones. Típicamente se emplea solo para páginas que pertenecen al sistema operativo.
- **Bit A**: indica si una página ha sido accedida desde la última vez que el bit A fue borrado.
- **Bit D**: indica si una página ha sido ensuciada (i.e., escrita) desde la última vez que el bit D fue borrado.
- **RSW**: reservado para uso del sistema operativo; el hardware lo ignora
- **PPN**: Número de página física - a. Si esta PTE es una hoja, el PPN es parte de la dirección física traducida. De lo contrario, el PPN da la dirección del siguiente nivel de la tabla de páginas

Como ya se ha resaltado varias veces, el objetivo de RISC-V es que sea una ISA flexible y fácil para añadir o quitar prestaciones, para ello, en caso de querer implementar la función de memoria virtual se hace uso del CSR `satp`, cuya estructura se muestra en la Figura 47.

31	30 22	21 0	
MODE	ASID	PPN	RV32
1	9	22	

Figura 47. CSR satp

Este registro controla el sistema de paginación y contiene tres campos. El campo MODE habilita la paginación y selecciona la profundidad de la tabla siguiente, en caso de valer este campo 0, entonces la implementación no cuenta con traducción o protección, en caso de valer 1 se tiene RV32 Sv32, es decir, direccionamiento virtual de 32 bits basado en página.

El campo ASID (identificador de espacio de direcciones) es opcional y puede ser usado para reducir el costo de cambio de contextos, finalmente el campo PPN contiene la dirección n física de

la tabla de páginas raíz, dividido entre el tamaño de página 4 KiB. Típicamente, el software de modo M escribirá cero a `satp` antes de ingresar al modo S por primera vez, deshabilitando paginación, luego el software de modo S lo escribirá nuevamente después de configurar las tablas de páginas. Cuando la paginación está habilitada en el registro `satp`, las direcciones virtuales de los modos S y U son traducidas a direcciones físicas por un recorrido de la tabla de páginas, iniciando en la raíz. La Figura 48 ilustra este proceso:

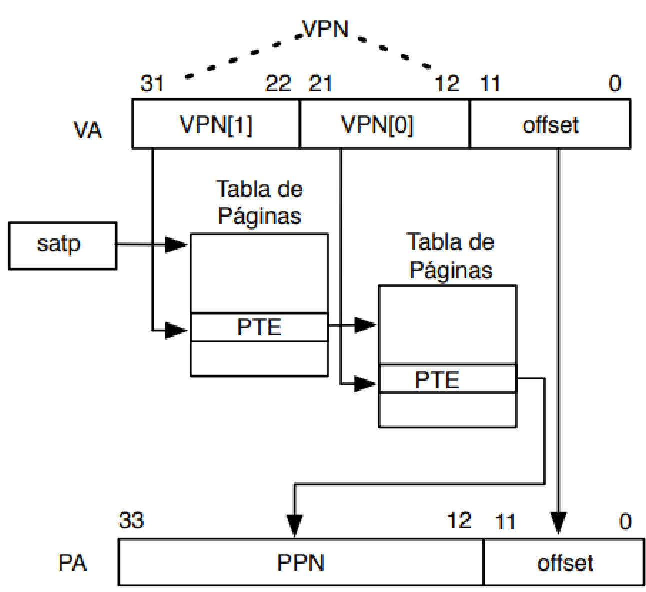


Figura 48 Diagrama del proceso de traducción de direcciones Sv32

3.9. Comparación de RISC-V con otras arquitecturas

Hoy en día la mayoría de los dispositivos están basados en arquitecturas x86 (como los fabricados por Intel y AMD) o en la arquitectura ARM (Qualcomn, MediaTek o Apple). Sin embargo, hay alternativas prometedoras en el mercado, y una de las más llamativas es RISC-V.

La característica principal por la que está tomando protagonismo es por ser *Open Source*, pero también por muchas otras características que hacen que sea una ISA totalmente modular.

Tanto Intel como AMD o fabricantes basados en ARM basan sus procesadores en aplicaciones específicas, mientras que RISC-V apuesta por una modularidad en los diseños que aporta gran flexibilidad en sus procesadores, lo que implica que es posible añadir componentes y sensores a medida que además ayudan a minimizar costes y a reducir el número de transistores necesarios, mejorando por tanto la disipación de calor y haciendo que RISC-V ofrezca procesadores personalizados pero con la capacidad de implementar módulos adicionales.

Por otro lado, a la hora de implementar el ISA, RISC-V se ha basado en las anteriores arquitecturas existentes. A continuación, se presenta una lista con las lecciones aprendidas por RISC-V [25].

- **Costo.** Tanto en ARM-32, MIPS-32 y x86-32 las instrucciones de multiplicación y división son obligatorias, en RISC-V son opcionales añadiendo la extensión "M".
- **Simplicidad.** ARM-32 no tiene registro cero, modos de direccionamiento complejos al igual que x86-32. En RISC-V existe el Registro x0 dedicado a proporcionar el valor 0, y no incluye instrucciones complejas.
- **Rendimiento.** En MIPS-32 los registros origen y destino varían en el formato de instrucción, en x86-32 como máximo hay 2 registros por instrucción, en RISC-V los registros origen y destino están fijos en la instrucción, y tiene hasta 3 registros por instrucción.
- **Espacio para crecer.** Espacio limitado disponible para el *opcode* tanto en ARM-32 como en MIPS-32, mientras que en RISC-V existe un espacio adicional disponible para el *opcode*.
- **Tamaño del programa.** ARM-32 utiliza solamente instrucciones de 32 bits menos en +Thumb-2, definida como un ISA aparte. En MIPS-32 sucede algo similar, pero con el ISA aparte +microMIPS para 16 bits. En x86-32 las instrucciones sí son variables. En RISC-V las instrucciones normalmente son de 32 bits, pero presenta la extensión "C" que facilita la implementación en 16 bits.

3.10. Conclusiones

En este capítulo se realiza un estudio de la arquitectura RISC-V a nivel ISA, sus características y diferentes registros. Además, se ha descrito sus modelos de direccionamiento de memoria y la ruta de datos y unidad de control. Por último, se ha estudiado el soporte que presenta la arquitectura para el Sistema Operativo y se han mostrado diferentes comparaciones entre RISC-V y otras especificaciones ISA como ARM o x86.

Capítulo 4. Procesador RISC-V a verificar

En este capítulo se presenta la descripción del procesador RISC-V a verificar. De entre las múltiples implementaciones existentes se ha decidido usar una implementación de fácil estudio y que cumpliera con el estándar descrito en cuanto a juego de instrucciones y *pipeline*. Sin embargo, la elección del procesador ha estado marcada, a su vez, por el tiempo disponible para su estudio y verificación. Este último aspecto se traduce es que su elección debía cumplir con los siguientes requisitos:

- Ser una implementación de código abierto y escrita en lenguaje *Verilog*.
- Tener, a ser posible, disponible un *testbench* que actuara como modelo de referencia.
- Ser compatible con el ISA del procesador RISC-V.

En este punto se decidió por usar el procesador descrito en [26]. Esta implementación se adaptaba a los requisitos planteados, aunque no implementa la totalidad del juego de instrucciones del estándar. Este último aspecto no se considera un obstáculo para el propósito de este TFM.

4.1. Descripción del procesador

En este punto se describe la estructura y las principales características de la implementación a verificar. La Figura 49 muestra el diseño del procesador RISC-V de 32 bits elegido. Este cuenta con un *pipeline* de 5 etapas que soporta las instrucciones básicas de un procesador RISC-V, concretamente las instrucciones: `and`, `or`, `add`, `sub`, `mul`, `addi`, `lw`, `sw` y `beq`.

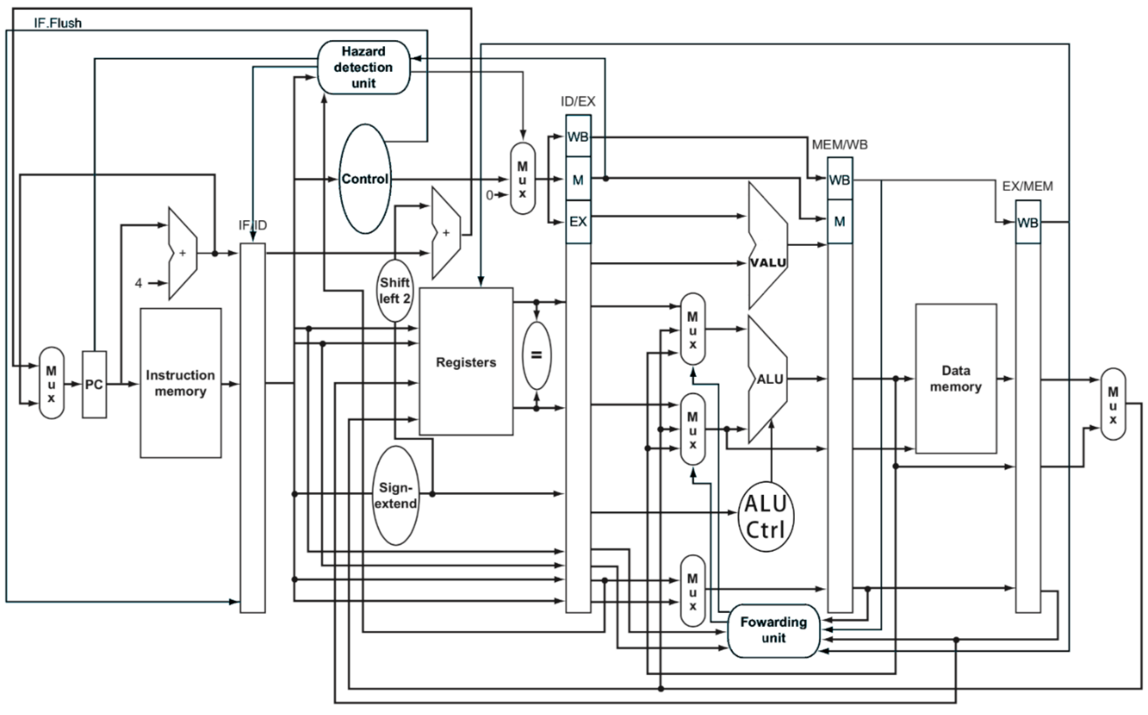


Figura 49 Diagrama bloques del diseño RISC-V a verificar

A continuación, se describe cada una de las etapas del *pipeline* y los componentes principales involucrados.

- ***Fetch*:** En esta etapa se accede al contador de programa PC que almacena la dirección de la próxima instrucción. Con esta dirección, se accede a la memoria de instrucciones donde se almacena las instrucciones del programa. Cuenta con un multiplexor que selecciona la siguiente dirección del PC ya sea por incremento de la dirección actual (+4) o se actualiza con la dirección de una instrucción de salto.
- **Decodificación:** Esta etapa maneja la unidad de control que genera las señales de control necesarias para las siguientes etapas basadas en la instrucción decodificada. Se accede al banco de registros, donde se leen los operandos de la instrucción. Cuenta, además, con un extensor de signo que convierte valores inmediatos de 16 bits a 32 bits. Hace uso de comparadores y desplazamiento a la izquierda de 2 posiciones que ayudan en el cálculo de direcciones de salto. Este procesador RISC-V también cuenta con una unidad *Hazard Detection* que detecta amenazas para el pipeline y que se explicará más en detalle posteriormente.
- **Ejecución:** En esta etapa se lleva a cabo la operación, donde interviene la ALU para realizar las operaciones aritméticas y lógicas procesadas en esta etapa. La selección de los operandos hace uso de multiplexores, que seleccionan los operandos de entre los valores provenientes del banco valores de registros, datos inmediatos, resultados anteriores, etc.

Dispone de una unidad de control que genera las señales de control específicas de la ALU y, además, controla el *Forwarding Unit*, que permite manejar el reenvío de datos para evitar añadir etapas de espera o *stalls* en el *pipeline* y que se detallará más adelante.

- **Acceso a memoria:** Esta etapa cuenta con la memoria de datos, donde se almacena l leen los operandos involucrados en las operaciones `load` y `store`. También incluye un multiplexor que selecciona, a la salida de la etapa, el valor a escribir en el banco de registros. Esta selección se realiza de entre el dato de memoria y el resultado de la ALU.
- **Escritura de resultados:** Esta etapa, cuenta con un multiplexor que selecciona el valor que se escribirá de vuelta en el banco de registros y el propio banco de registros donde se escribe el resultado final de la instrucción.

El procesador RISC-V utilizado implementa una interfaz adaptada, como muestra la Figura 50. A pesar de que el procesador cuenta con una memoria de instrucciones de 32 bits, los diseñadores han simplificado la interfaz con un bus de instrucciones de entrada (`instr_i`) de 8 bits, debido a que la finalidad es implementarlo físicamente ya sea en una FPGA o ASIC. De esta manera se reduce la complejidad de la conectividad con otros componentes del sistema como memoria y periféricos. Por este motivo, el diseño original de este RISC-V realiza cuatro lecturas de 8 bits para ensamblar internamente la instrucción completa de 32 bits. Por este motivo, antes de realizar el proceso de verificación, se modificó el código fuente para adaptar las interfaces de manera que se facilite la integración del entorno de verificación UVM a diseñar.

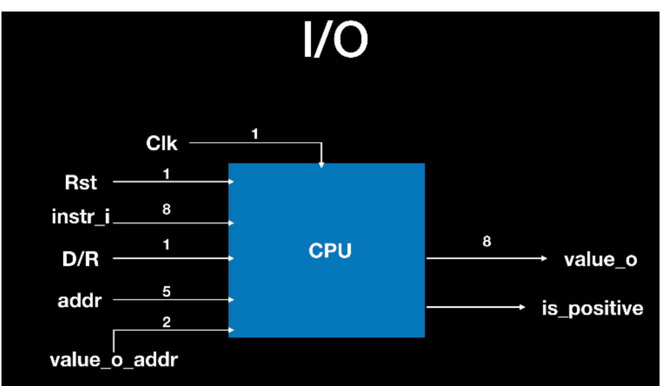


Figura 50 Interfaz de E/S del procesador RISC-V original

La Tabla 7 muestra las señales de la interfaz original de E/S correspondiente a la Figura 50.

Tabla 7 Descripción señales E/S RISC-V original

Señal	I/O	Descripción
instr_i	Input	Bus de instrucciones de 8 bits.
D/R	Input	Señal de control. A nivel alto indica que el valor de salida se guarda en la memoria de datos. A nivel bajo provoca que este, se guarde en el banco de registros.
addr	Input	Dirección de acceso de 5 bits, ya que el tamaño original de la memoria de datos, así como el número de registros, es de 32.
value_o_addr	Input	Señal de control de 2 bits qué bye, de la palabra de 32 bits, se debe leer.
value_o	Output	Bus de 8 bits con el byte de salida.
is_positive	Output	Señal que indica casos de desbordamiento para la interpretación de la señal `value_o`.

4.2. *Testbench* original del procesador RISC-V

En este apartado se detallará el código del *testbench* original y se mostrarán las diferentes partes de que consta el *testbench*. El *testbench* original está descrito a nivel de *bare metal.* Es decir, se trata de un test donde se ejecutan las operaciones básicas del ISA propio de RISC-V. Además de ejecutar este test, se crearon otros tests diferentes siguiendo la estructura original del *testbench* con el fin de verificar las distintas funcionalidades del procesador.

En el Código 46 se muestra el módulo `top`, denominado `testfixture`. En primer lugar se definen los parámetros y directivas indicando el periodo del reloj `CYCLE_TIME` y el número máximo de ciclos de simulación `End_CYCL`.

Entre las líneas 5 a 20 se definen las señales y registros temporales necesarios para el funcionamiento del *testbench* y la comunicación con la CPU. Estas señales son principalmente las ya mencionadas anteriormente. Además, se definen dos registros importantes. El registro `instr_store`, cuya función es cargar el contenido de memoria con el programa a ejecutar, como se explicará más adelante, y el registro `Golden`, donde se cargará el resultado esperado del programa ejecutado. El test original va comparando este registro con el resultado a la salida para verificar la funcionalidad del procesador. Por último, se define la señal `Start`, esta señal se activa cuando se detecta una determinada secuencia dentro del programa de instrucciones con el objetivo de indicar al procesador cuando debe comenzar a ejecutar las instrucciones. Esto posibilita mantener la CPU inactiva mientas se carga la memoria de instrucciones a través del interfaz de E/S

de 8 bits. El verificador solo tiene que añadir una determinada secuencia en el programa a cargar para hacer que esta señal se active y habilite el funcionamiento de la CPU.

En la línea 22, se genera la señal de reloj con el periodo definido al comienzo. Por último, entre las líneas 24-34 se referencia el módulo `CPU` y se conectan las señales correspondientes al interfaz de esta `CPU`.

```
1  `timescale 1ns/10ps
2  `define CYCLE_TIME 10.0        // Modify your clock period here
3  `define End_CYCLE  300         // Modify cycle times
4  module testfixture;
5  reg                  Clk;
6  reg                  Start;
7  reg                  DataOrReg;
8  reg [4:0]            address;
9  reg [7:0]            instr_i;
10 reg                  reset;//used to initalize memorys and registers
11 reg [7:0]            instr_store[0:(64*4+1)];
12 reg [1:0]            vout_addr;
13 wire[7:0]            value_o;
14 wire                 is_positive;
15 wire [2:0]           easter_egg;
16 integer              i, outfile, counter;
17 integer              stall, flush,idx;
18 integer              j,k;
19 integer              err;
20 reg  [7:0]           golden [0:63];
21
22 always #(`CYCLE_TIME/2) Clk = ~Clk;
23
24 CPU CPU(
25     .clk_i   (Clk),
26     .DataOrReg(DataOrReg),
27     .address(address),
28     .instr_i(instr_i),
29     .reset(reset),
30     .vout_addr(vout_addr),
31     .value_o(value_o),
32     .is_positive(is_positive),
33     .easter_egg(easter_egg)
34 );
```

Código 46 Módulo top del *testbench* original

El *testbench* se simplifica en un único proceso `initial` en el cual se realizan las siguientes operaciones.

- Se inicializan las variables temporales para ejecutar el *testbench* tal y como se muestra en las líneas 36 a 44 del Código 47. Posteriormente, entre las líneas 52 a 57 se realiza la generación de un pulso de `reset` para inicializar la `CPU` con los valores predeterminados.

- En la línea 47 se utiliza el comando `$readmemb`. Este comando busca en el directorio especificado el fichero con el código binario correspondiente al programa a ejecutar. En este caso se carga directamente el programa original en la memoria de instrucciones. En la línea 48, y de la misma forma se carga el fichero con los resultados esperados en el registro `golden` para su posterior comparación.

```
35 initial begin
36     counter = 0;
37     stall = 0;
38     flush = 0;
39     idx = 0;
40     DataOrReg = 1;
41     address = 5'd8;
42     vout_addr = 2'b11;
43     err = 0;
44     instr_i = 0;
45     for(k=0;k < (64*4+1) ;k=k+1) instr_store[k] = 0;
46     // Load instructions into instruction memory
47     $readmemb("../dat/instruction2.txt", instr_store);
48     $readmemh("../dat/golden.dat",golden);
49     // Open output file
50     outfile = $fopen("../dat/output.txt") | 1;
51 
52     Clk = 1;
53 
54     reset = 0;
55     reset = 1;
56     #(`CYCLE_TIME)
57     reset = 0;
58 end
```
Código 47 Inicialización y carga de instrucciones del *testbench* original

4.3. Ejemplo de test del procesador RISC-V

Debido a que los tests originales del procesador son muy básicos. Aunque existía un test que realizaba el cálculo de la serie de **Fibonacci**, este no está completo y su ejecución daba errores. Por este motivo, se decidió escribir un programa en ensamblador para el cálculo de un determinado elemento de dicha serie y probarlo en el procesador. De esta manera, con este programa, se consigue tener unos resultados de simulación fiables, con una mayor cobertura de verificación y poder así tener muchos más datos para poder comparar una vez finalizado el entorno de verificación UVM.

La serie de Fibonacci es una secuencia infinita de números enteros en la que cada número es la suma de los dos números precedentes. La serie comienza con los números 0 y 1. A partir de estos dos primeros números, cada número subsiguiente se obtiene sumando los dos números anteriores. La secuencia de Fibonacci se puede definir formalmente de la siguiente manera:

- $F_0 = 0$
- $F_1 = 1$
- $F_n = F_{n-1} + F_{n-2}$; para n ≥ 2

En el Código 48 se implementa el código en ensamblador para el cálculo del número 10 de la serie de Fibonacci.

Entre las líneas 4-13 se realiza la inicialización de registros, haciendo uso de instrucciones *load* inmediatos `li`. El registro `x7` almacena el límite que viene a ser el número de la serie Fibonacci que se quiere calcular, en este caso el número 10 (línea 10). Los registro `x2` y `x3` se utilizan para la comparación con los valores `0` y `1` respectivamente para los casos en que `n` sea igual a `0` o `1`, esto es, los casos especiales $F_0 = 0$ y $F_1 = 1$. Las líneas 12 y 13 realizan dichas comparaciones con el uso de la instrucción `beq`. Los registros `x5` y `x6` almacenan el valor de los primeros números de la serie `0` y `1`. Por último, el registro `x8` será utilizado como contador de la serie comenzando en 1 (línea 10).

En caso de que el número a calcular (asignado al registro `x7`) sea diferente de F_0 y F_1, el programa continúa hacia el bucle principal `LOOP`. En la línea 17 si el contador `x8` es igual a `x7`, es decir, si se ha calculado hasta el décimo término, el programa salta a `EXIT`. En caso contrario, realiza la suma de los registros `x5` y `x6`, para seguir avanzando en la serie, y almacena el resultado en `x4` mediante una instrucción `add` (línea 18). En la línea 19 se actualiza el primer número de Fibonacci moviendo el valor del registro `x4` a `x6` con una instrucción `ori`. Lo mismo ocurre en la línea 20, pero para el segundo número de Fibonacci. La línea 21 incrementa el contador `x8` en 1 con una instrucción `addi` y, finalmente, en la línea 22 se vuelve al `LOOP` principal mediante la instrucción `beq x0, x0, LOOP`. Como ya se explicó en el capítulo Arquitectura RISC-V, el registro `x0` siempre mantiene el valor `0`, por lo tanto, esta instrucción en RISC-V es un salto incondicional ya que la comparación siempre será verdadera.

El programa seguirá dentro del bucle hasta que ambos registros, `x7` y `x8` coincidan, para saltar a la salida `EXIT` en la línea 24: donde el resultado final que se ha ido almacenando en `x4` se mueve al registro `x10`, ya que este es el registro típico utilizado como valor de retorno de funciones. En la línea 26 nuevamente se realiza un salto incondicional a la etiqueta `HALT`, ubicada en la línea 37, donde se almacena el resultado del cálculo del décimo número Fibonacci en la dirección de memoria `0x100`, para ello, en la línea 38 se guarda esta dirección en el registro `x7`, usando la instrucción `li`, y en la línea 39 finalmente se almacena el valor de `x10` con el resultado en la dirección `0x100` mediante la instrucción `sw` y en la siguiente y última línea se finaliza la ejecución.

```
 1  Text
 2  .globl main
 3
 4  main:
 5              li x7, 10             # Limit
 6              li x2, 0              # Used to determine if n (x7) equals 0
 7              li x3, 1              # Used to determine if n (x7) equals 1
 8              li x5, 0              # First number
 9              li x6, 1              # Second number
10              li x8, 1              # Counter
11
12              beq x7, x2, DO        # If n == 0 then jump to DO and print 0
13              beq x7, x3, WRITE     # if n == 1 then jump to WRITE and print 1
14
15
16  LOOP:
17              beq x8, x7, EXIT      # if counter == limit then jump to EXIT
18              add x4, x5, x6        # Add x5 to x6 and store in x4
19              ori x5, x6, 0         # Assign x6 value to x5 (x5=x6)
20              ori x6, x4, 0         # Assign x4 value to x6 (x6=x4)
21              addi x8, x8, 1        # Add 1 to my counter
22              beq  x0, x0, LOOP     # Jump to loop always
23
24  EXIT:
25              add x10,x4,x0         # Move x4 result to x10
26              beq x0, x0, HALT      # Jump to halt always
27
28  DO:
29              li x4, 0              # load 0 in x4
30              add x10,x4,x0         # move 0 to x10
31              beq x0, x0, HALT      # Jump to halt always
32
33  WRITE:
34              li x4, 1              # load 1 in x4
35              add x10,x4,x0         # move 1 to x10
36
37  HALT:
38              li x7, 0x100          # load addr 0x100 into x7
39              sw x10, 0(x7)         # store x10 value into addr 0x100
40              beq x0, x0, HALT      # Jump to halt always
```

Código 48 Código ensamblador para el cálculo de un determinado número de la serie Fibonacci

Para poder obtener el código hexadecimal de la compilación, fue necesario instalar las *Toolchain* de GNU para el RISC-V [27]. Si bien esta instalación no es un proceso trivial, no se ha querido añadir a la memoria del presente TFM por motivos de extensión del mismo. Tras su instalación, se pudo ensamblar el código haciendo uso del comando:

```
riscv64-unknown-elf-as -o fibo10.o fibo10.s
```

La opción -o especifica el nombre del archivo binario de salida, en este caso `fibo10.o` y seguidamente se nombra el archivo de código ensamblador de entrada que contiene el programa RISC-V `fibo10.s`

Una vez obtenido el código binario, se hace uso del comando `riscv64-unknown-elf-objdump`, la ejecución de este comando es algo más compleja:

```
riscv64-unknown-elf-objdump --disassemble-all --disassemble-zeroes --section=.text --section=.data fibo10.o > fibo10.dump
```

Este comando permite desensamblar código para la arquitectura RISC-V. La opción `--disassemble-all` desensambla todas las secciones del archivo seleccionado. La opción `--disassemble-zeroes` incluye las instrucciones que son ceros en la salida, también conocidas como no operaciones, utilizado para mostrar un análisis completo y realizar una depuración detallada de la conversión. La opción `--section=.text` especifica el nombre de la sección en este caso `text` que limita el desensamblado al código ejecutable del programa, mientras que `--section=.data` contiene los datos estáticos. Por último, se indica el archivo binario y se redirecciona la salida, en este caso, al fichero `fibo10.dump`.

Tras su ejecución, se obtiene el código hexadecimal para poderlo cargar en memoria a través del comando `$readmemh`. A continuación, se presenta el código hexadecimal (Código 49) correspondiente al código ensamblador anteriormente presentado.

```
 1  00a00393     //   li    t2,10
 2  00000113     //   li    sp,0
 3  00100193     //   li    gp,1
 4  00000293     //   li    t0,0
 5  00100313     //   li    t1,1
 6  00100413     //   li    s0,1
 7  02238463     //   beq   t2,sp,40 <DO>
 8  02338863     //   beq   t2,gp,4c <WRITE>
 9  00740c63     //   beq   s0,t2,38 <EXIT>
10  00628233     //   add   tp,t0,t1
11  00036293     //   ori   t0,t1,0
12  00026313     //   ori   t1,tp,0
13  00140413     //   addi  s0,s0,1
14  fe0006e3     //   beqz  zero,20 <LOOP>
15  00020533     //   add   a0,tp,zero
16  00000c63     //   beqz  zero,54 <HALT>
17  00000213     //   li    tp,0
18  00020533     //   add   a0,tp,zero
19  00000663     //   beqz  zero,54 <HALT>
20  00100213     //   li    tp,1
21  00020533     //   add   a0,tp,zero
22  10000393     //   li    t2,256
23  00a3a023     //   sw    a0,0(t2)
24  fe000ce3     //   beqz  zero,54 <HALT>
```

Código 49 Código hexadecimal correspondiente al programa Fibonacci descrito en ensamblador

Se puede comprobar el correcto funcionamiento de la *toolchain* de RISC-V de la siguiente manera. Para comprobar que, efectivamente, el código hexadecimal corresponde con el mismo programa descrito en ensamblador, basta con seguir la estructura de instrucciones propia de esta arquitectura.

Por ejemplo, en la línea 10 del Código 49, la instrucción `add tp, t0, t1` que corresponde a la instrucción `add x4, x5, x6` siguiendo la *Application Binary Interface* propia de RISC-V [28], si se realiza la conversión a binario y se codifica manualmente la instrucción, se observa que, efectivamente, cumple con una instrucción tipo R concretamente la suma entre dos registros como muestra la Figura 51.

funct7	rs2	rs1	funct3	rd	opcode
0000000	00110	00101	000	00100	0110011
	x6	x5		x4	add

Figura 51 Codificación de la instrucción `add x4, x5, x6`

4.4. Simulación del test en Questasim

Para probar el correcto funcionamiento del sistema, se realizó la simulación del mismo usando el entorno EDA (*Electronic Design Automation*) de Siemens, Questasim. Este entorno es un entorno profesional que permite realizar la simulación de un sistema descrito en lenguaje HDL y que, además, da soporte al estándar de *SystemVerilg*. Esto implica que reconoce todas las estructuras que proporciona el estándar, lo que facilita el cálculo de cobertura funcional, integra de facto la cobertura de código, la cobertura de propiedades y que, además, da soporte a la metodología UVM en todas sus versiones.

A continuación, se presentan algunas formas de ondas que detallan la ejecución de un test básico, concretamente el test que se muestra en el Código 50, para explicar en detalle el funcionamiento del procesador a nivel de señales y *pipeline* y cómo gestiona la dependencia de datos mediante la unidad de *Forwarding.* Esta dependencia de datos aparece entre la instrucción `add` y las dos instrucciones `addi` anteriores al proporcionar estos los operandos fuentes de aquella.

```
1 //Initialization
2 00200113   //PC = 0  addi $x2, $x0, 2    $x2 = 2
3 00500193   //PC = 4  addi $x3, $x0, 5    $x3 = 5
4 003100b3   //PC = 8  add $x1, $x2, $x3   $x1 = 7
5 00002003   //PC = 12 lw $x0, 0(x0)       No operation (FIN programa)
6 11111111
```

Código 50 Test básico con dependencia de datos

A continuación, se comentan los resultados de las simulaciones para este caso concreto, siguiendo el orden del *pipeline* y comentando detalladamente el funcionamiento del procesador atendiendo a las señales más relevantes. Por último, se presentará y comentará los resultados del

test para el programa del cálculo del décimo número de la serie Fibonacci, en este caso, sin atender al detalle, si no, a nivel de ejecución correcta del programa.

- **Etapa *Fetch* y *Decode***: La ejecución de las instrucciones por el procesador comienza, como ya se comentó, al recibir la señal `start` a nivel activo en el instante t_1. En este instante comienza la etapa de *fetch* con el valor inicial del contador de programa a '0' La primera instrucción almacenada en la dirección '0', es `32'h00200113` (línea 2, Código 50). El ciclo finaliza en el instante t_2, donde la instrucción se almacena en el registro de instrucciones del procesador y se actualiza el contador de programa a '4', direccionando la siguiente instrucción, `32'h00500193` en este caso. Al paralelizar la etapa de *Fetch*, la etapa de decodificación ya tiene localizado para la primera instrucción el registro destino `x2` y el valor inmediato 2. Durante esta etapa se decodifica el tipo de operación mediante el bus `ALUOp`. En este caso se trata de una instrucción tipo R, donde el valor '3' indica una operación `addi` (suma con dato inmediato). En el instante t_3 la nueva etapa de *Fetch* vuelve a actualizar PC a '8' y se busca la tercera instrucción `32'h003100b3`. Mientras se realiza esta etapa, de forma paralela se ejecuta la etapa decodificación de la segunda instrucción. En esta etapa se detecta el registro destino `x3` y el dato inmediato 5, manteniendo el mismo tipo de operación '3' en `ALUOp`. Por último, en el siguiente ciclo se decodifica la tercera y última instrucción decodificando el registro de destino `x1`, y los registros fuente `x2` y `x3`. En este caso, el bus de control `ALUOp` pasa a tomar el valor '2' indicando una operación `add` (suma entre registros).

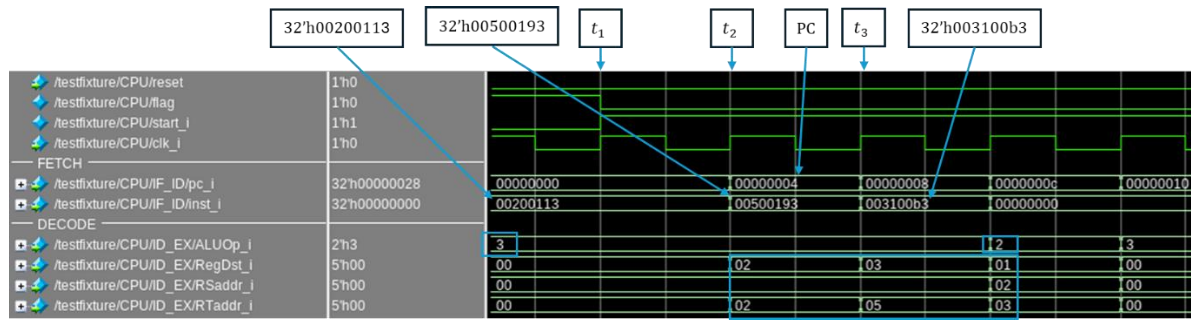


Figura 52 Resultados de simulación para la etapa *Fetch*

- **Etapa *Execute***: Siguiendo el *pipeline*, en el instante t_3 comienza la etapa de ejecución de la primera instrucción. En esta etapa, las señales más interesantes son las propias de la `ALU` encargada de realizar las operaciones aritméticas. Así, en el instante t_3 de la Figura 53, el bus `ALUCtrl` vale '1', indicando a la `ALU` que la operación debe ser una suma, como indica el siguiente extracto de código del fichero `ALU.v`:

```
 1  module ALU (data1_i, data2_i, ALUCtrl_i, data_o,Zero_o);
 2  input [31:0] data1_i, data2_i;
 3  input [2:0] ALUCtrl_i;
 4  output reg[31:0] data_o;
 5  output reg Zero_o;
 6  parameter SUM = 3'b001;
 7  parameter SUB = 3'b010;
 8  parameter AND = 3'b011;
 9  parameter OR  = 3'b100;
10  parameter XOR = 3'b101;
11  parameter MUL = 3'b110;
12  always@(*)begin
13  Zero_o    = (data1_i - data2_i)?0:1;
14  case(ALUCtrl_i)
15    SUM : begin
16      data_o = data1_i + data2_i;
17    end
```

Código 51 Extracto módulo ALU.v

En ese mismo instante, realiza la suma entre los operandos de entrada `data1` = '0' y `data2` = '2' y el resultado lo envía al resto del *pipeline* a través de la señal `data_o` con valor '2'. En el instante t_4 se realiza el mismo procedimiento, pero para la segunda instrucción siguiendo la paralelización de la CPU gracias a la segmentación. En este caso, la suma entre `0` y `5` igual `5`. Por último, el instante t_5 se ejecuta la última instrucción donde se suma ambos resultados anteriores `2` y `5` con resultado `7`.

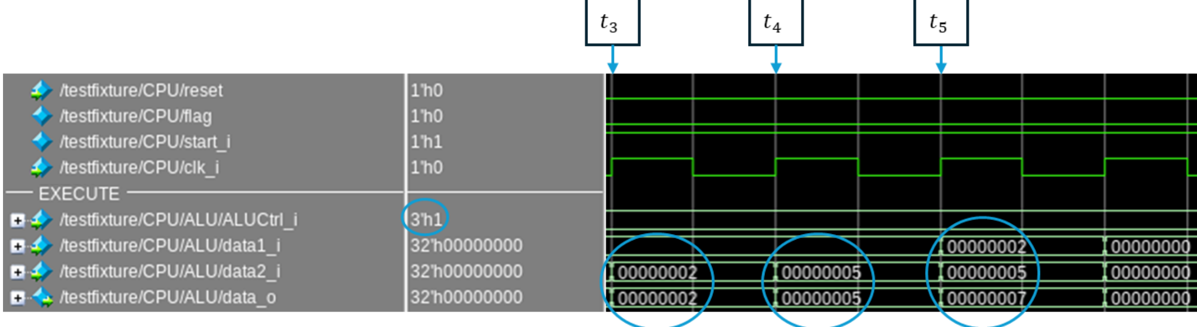


Figura 53 Resultados de simulación para la etapa *Execute*

- **Etapa acceso a memoria:** La cuarta etapa del *pipeline* para instrucciones de este tipo realizan un simple *passthrough* ya que no se realiza operaciones de *load* o *store* para interactuar con la memoria.
- **Etapa de escritura de resultados:** En la quinta y última etapa del pipeline se escribe el resultado de las instrucciones en el banco de registros. La etapa 5ª comienza en el instante t_5, cuando la señal de control `RegWrite` se activa indicando que se va a realizar una escritura en el banco de registros. La operación sobre la memoria se hace efectiva en el siguiente flanco de bajada de la señal de reloj. En este caso particular se escribe el valor '2' en el registro `x2`. En el instante t_6 comienza la etapa de escritura de la siguiente instrucción. De la misma manera, la señal `RegWrite` se mantiene a nivel

alto y en el siguiente flanco de bajada se escribe el valor 5 en el registro x3. Por último, en el instante t_7 se escribe en el registro x1 el resultado de la suma '7', como resultado de la etapa de escritura de la tercera instrucción.

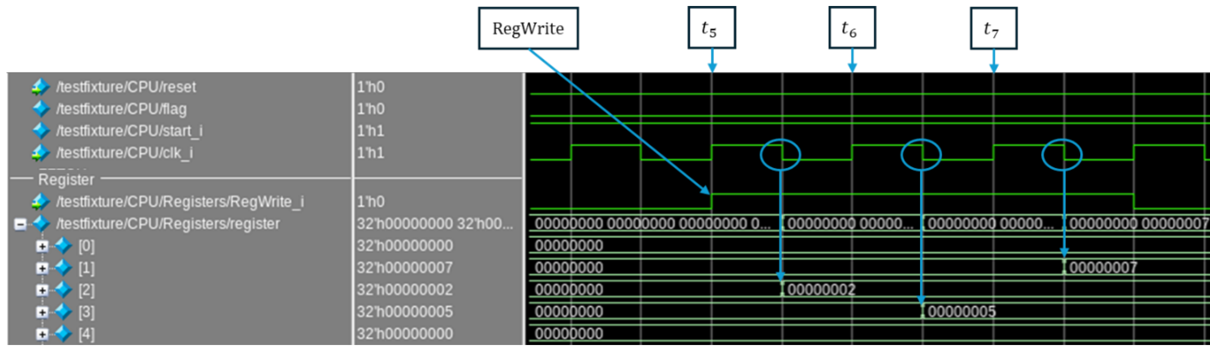


Figura 54 Resultados de simulación de la etapa escritura de resultados

Unidad *Forwarding*: Como ya se ha comentado, el procesador RISC-V a verificar cuenta con una unidad de *forwarding*, unidad indispensable en procesadores segmentados debido al peligro de romper el pipeline por las instrucciones dependientes de instrucciones anteriores. En la simulación de las instrucciones anteriores existe dependencia de datos, ya que la última instrucción hace uso de los registros x2 y x3 que previamente, han sido utilizados como registros destino en instrucciones para almacenar un valor inmediato. Por este motivo; llegar a la etapa de ejecución de la última instrucción, si no se cuenta con esta unidad, el procesador haría uso de los valores no actualizados de ambos registros, ya que estos no tendrían aún el resultado de las instrucciones anteriores. Una solución para este problema es la inclusión de unidad *forwarding* que permita redirigir estos resultados sin necesidad de terminar el recorrido por todo el *pipeline,* tal y como muestra la Figura 55.

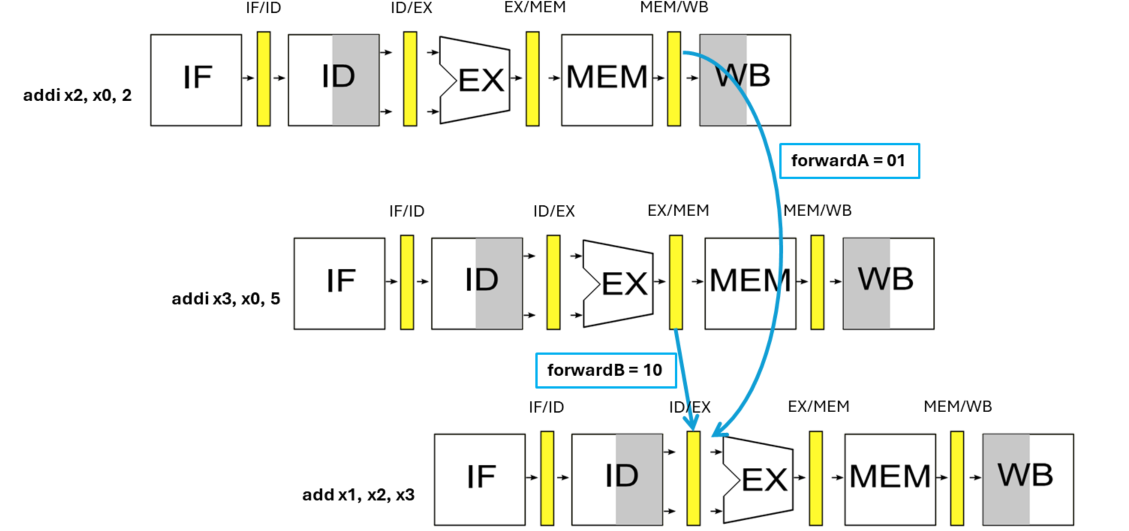


Figura 55 Esquema de funcionamiento de la unidad *forwarding* RISC-V a verificar

La unidad de *forwarding* está íntimamente ligada a la presencia de los multiplexores de entrada a la ALU que se muestra en la Figura 49.

Las señales de control `forwardA` y `forwardB` controlan directamente estos multiplexores de la `ALU`, de esta manera selecciona qué valor va a recibir la `ALU`, siendo las posibilidades:

- 00 -> ID/EX: el valor seleccionado proviene del banco de registros
- 10 -> EX/MEM: el valor seleccionado proviene de la etapa de acceso a memoria
- 01 -> MEM/WB: el valor seleccionado proviene de la etapa de escritura de resultados.

Por lo tanto, en la tercera instrucción se hace uso directamente de los resultados de las instrucciones anteriores sin necesidad de añadir burbujas o ciclos de *stall* al *pipeline*. El resultado de la simulación se muestra a continuación una vez entendido el concepto del significado de cada señal.

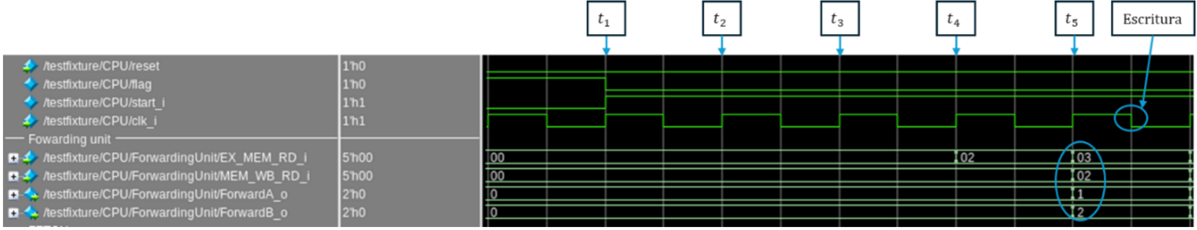


Figura 56 Simulación de la ejecución unidad forwarding

Como se puede ver, justo en el instante t_5, la unidad *forwarding* avisa mediante las señales `ForwardA` y `ForwardB` que se envíe directamente los resultados almacenados en `MEM/WB` y `EX/MEM`, respectivamente, hacia la `ALU` a través de los multiplexores de entrada. Además, los desarrolladores, para que esta acción no consuma una etapa más, como ya se comentó en la etapa de escritura, ésta se hace justamente en el flaco de bajada de la señal de reloj, aprovechando al máximo el rendimiento.

4.4.1. Ejecución de un test completo en el procesador RISC-V

En este punto, se ha podido comprobar el funcionamiento del procesador RISC-V para ejecutar un test básico. Esto ha permitido conocer los aspectos más relevantes del diseño seleccionado. En este apartado se muestra el resultado de la simulación la ejecución de un test que realiza el cálculo del décimo número de la serie Fibonacci (Código 49). En este caso, y a diferencia del test básico, se comenta los resultados generales sin entrar en detalles internos.

En la Figura 57 se muestra la inicialización de los registros. En el instante t_1, tras activarse la señal `start`, el procesador comienza a ejecutar el programa, comenzando por la etapa de *fetch* de la primera instrucción, hasta llegar a la escritura del primer registro en el instante t_2. Este instante correspondo a la etapa de escritura de la primera instrucción. A partir de ese instante, en cada ciclo se inicializan los registros `t2`, `sp`, `gp`, `t0`, `t1` y `s0` respectivamente.

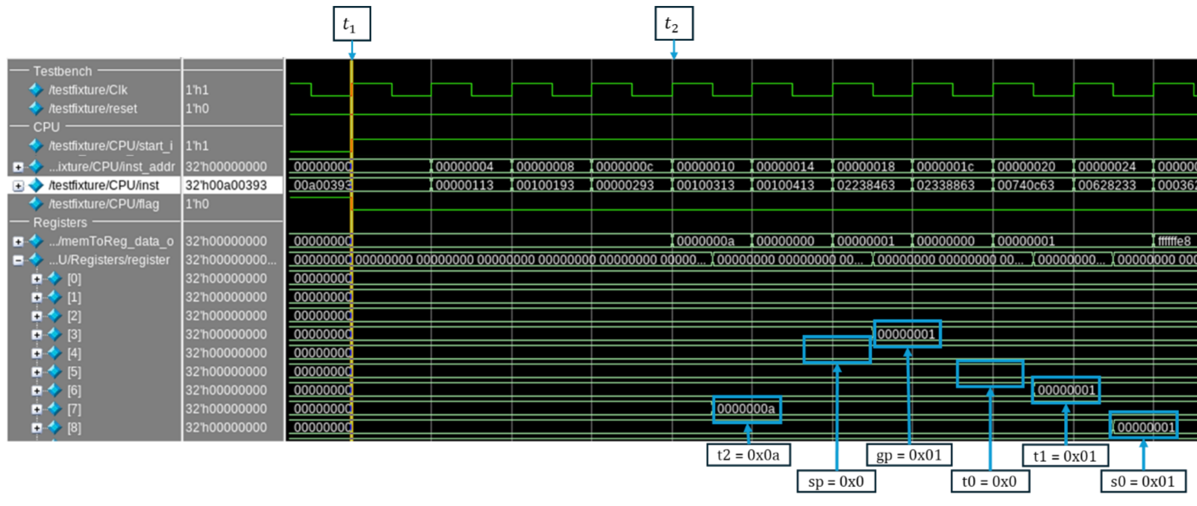


Figura 57 Resultado de la simulación del test serie Fibonacci - Inicialización de los registros

En la Figura 58, se observa la actualización de los registros correspondiente a la ejecución del bucle `LOOP` del (Código 48). En el instante t_1, se ejecuta la instrucción `beq` correspondiente al caso del cálculo F_0 (Código 48). En el siguiente ciclo de reloj se ejecuta la instrucción para el cálculo del caso F_1(Código 48) En el siguiente ciclo, se realiza el ciclo de *Fetch* correspondiente al tercer `beq`. Su ejecución se lleva a cabo en caso de que el registro `x8` (contador) llegue al límite indicado por el registro `x7`. En las tres instrucciones anteriores la señal `isBranch` se activa indicando que se trata de una instrucción de salto. Sin embargo, en este caso particular, no debe saltar, ya que en ningún caso se cumple la condición. A partir del instante t_2, comienza la actualización de los registros `x4`, `x5`, `x6` y `x8` para el cálculo de los diferentes elementos de la serie Fibonacci. Por último, en el instante t_3, se introduce en el *pipeline* la última instrucción de salto incondicional del bucle `LOOP`. En este caso, la condición se cumple y la señal `PC_Branch_Select` se activa, provocando el salto al comienzo del bucle `LOOP` donde se pasa a la instrucción en la dirección `0x00000020` como se observa en el instante t_4. Al cumplirse la condición, se produce un `flush` del pipeline, para eliminar del mismo la instrucción posterior al salto, que no se debe ejecutar.

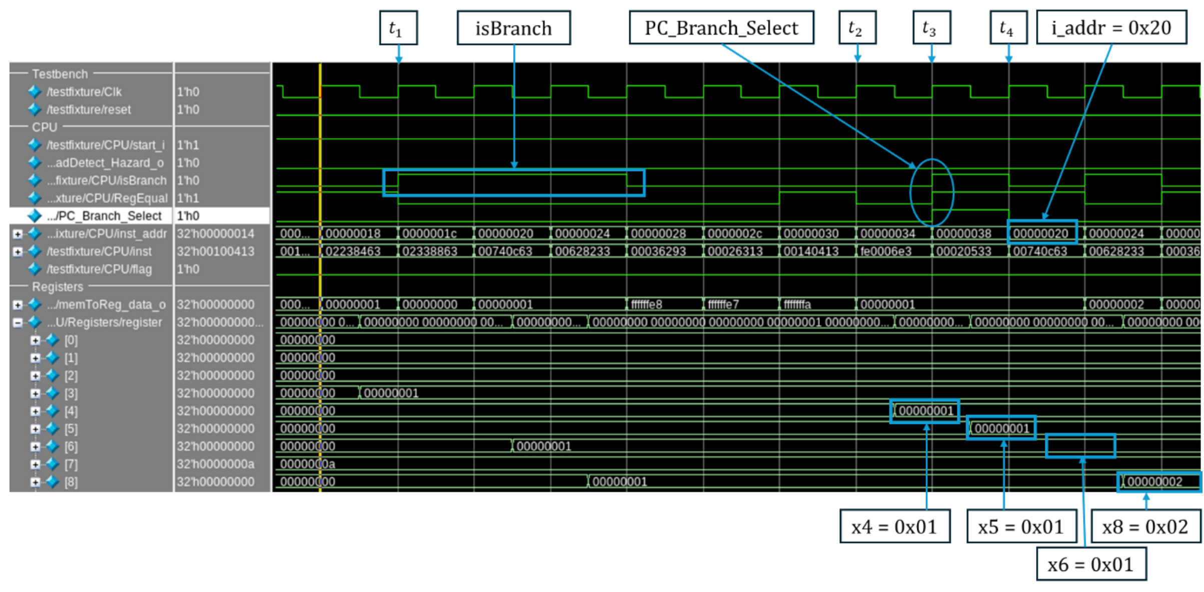


Figura 58 Resultado de la simulación del test serie Fibonacci - Actualización de los registros

En la Figura 59 se muestra el último ciclo de ejecución del bucle principal. En el instante t_1 se produce el cálculo del décimo número de la serie, que se deposita correctamente en el registro `x6` con el valor obtenido `0x37`. En el instante t_2 se actualiza el contador `x8`, que toma finalmente el valor `0x0a`, igualando el límite almacenado en el registro `x7`. Este hecho produce que la señal de salto `PC_Branch_Select` se active, de manera que la siguiente instrucción se encuentra en la dirección cuya etiqueta se nombra `EXIT`. La dirección de salto contiene una instrucción que deposita el valor del resultado obtenido en el registro `x10`. En el instante t_3 se realiza el *Fetch* de dicha instrucción y se lee la instrucción de salto incondicional que lleva el programa a ejectuar el tramo final del código, bajo la etiqueta `HALT`. Cinco ciclos más tarde, en el instante t_4 (lo que tarda el propio *pipeline*), se registra en `x10` el valor obtenido `0x37`.

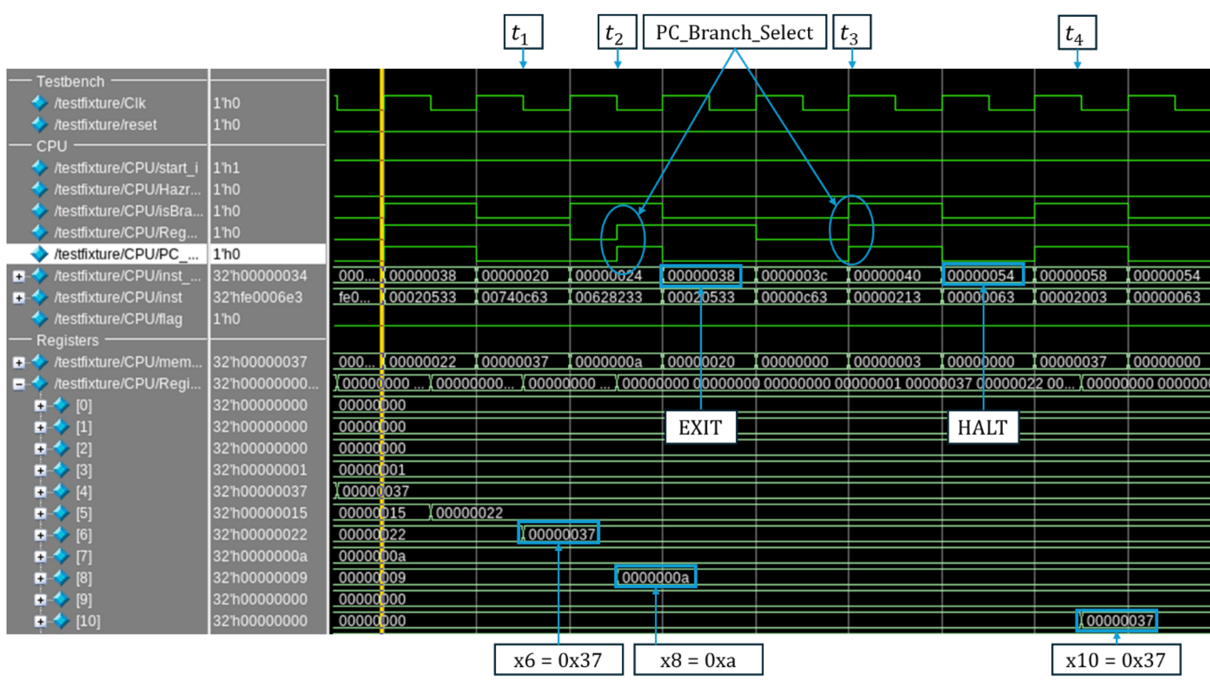


Figura 59 Resultado de la simulación test serie Fibonacci - Cálculo del décimo número

En la Figura 60 se muestra la ejecución de una instrucción de memoria. Finalizado el cálculo del elemento de la serie Fibonacci, se ejecuta el código etiquetado como `HALT`. En este código se produce un acceso a memoria para escritura. En este caso, se almacena el valor calculado (registro `x10`) en la posición de memoria `0x100`. Para su ejecución, se usa como registro base el registro `x7`. En primer lugar, se inicializa dicho registro con el valor 0x100 y, posteriormente, se ejecuta la instrucción `sw x10, 0(x7)`.

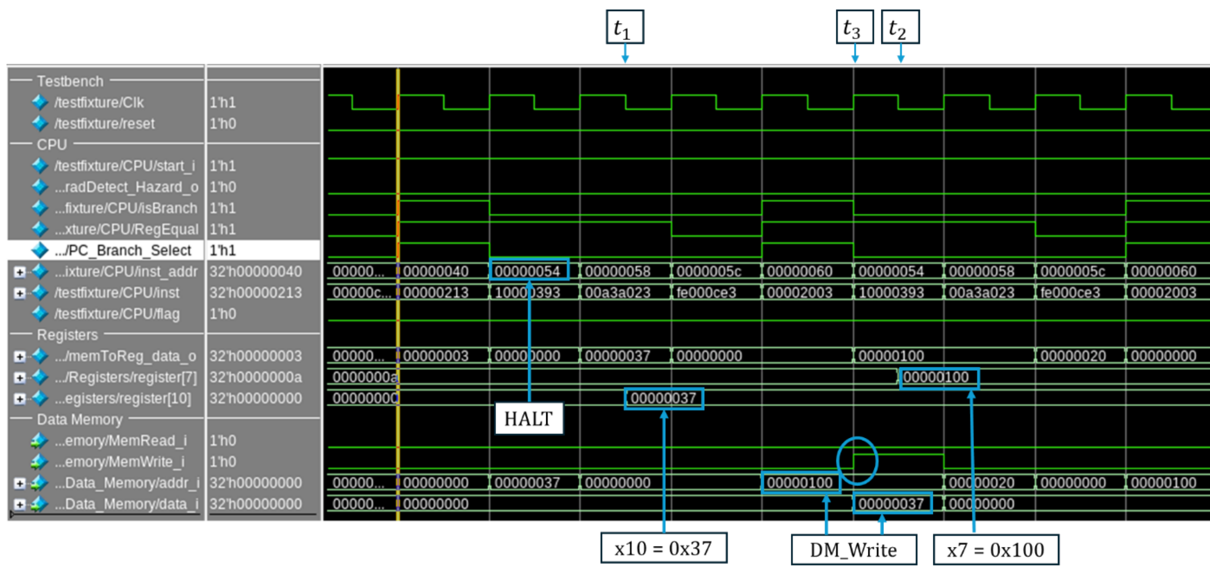


Figura 60 Resultado de la simulación del test serie Fibonacci - Almacenamiento en memoria de datos

El instante t_1 corresponde al ciclo de escritura del resultado en el registro x10. En el instante t_2 se carga el registro x7 con la dirección de acceso a memoria, línea 38 del Código 48. Finalmente, en el instante t_3, se realiza el acceso a memoria, escribiendo el dato 0x37 en la posición 0x100. Un dato a tener en cuenta es que la dirección 0x100 aparece en el bus de direcciones antes de ser almacenado en el registro x7. Esto es debido a la dependencia de datos que hay entre la línea 38 y la línea 39 del Código 48 debido al uso del registro x7. La correcta ejecución del código muestra cómo la unidad de avance de datos (*Data Forwarding Unit*), previamente explicada, ha funcionado correctamente, adelantando el valor a depositar en el registro x7, evitando así la necesidad de parar el *pipeline*.

Capítulo 5. Creación del entorno verificación UVM para RISC-V

Una vez finalizada la descripción de todos los componentes que forman un entorno de verificación UVM; descrito el diseño del procesador con arquitectura RISC-V bajo prueba; y mostrado los resultados esperados de la funcionalidad de este, en el presente capítulo se describirá el entorno de verificación UVM propuesto para validar el funcionamiento del mismo. En primer lugar, se explicará porqué ha sido necesario adaptar las interfaces del diseño con el fin de posibilitar la comunicación entre las memorias de instrucciones y datos con el entorno de verificación a través de las interfaces virtuales. En un segundo apartado, se explicará la implementación de un nuevo componente, el componente *subscriber*. El componente *susbcriber* es un componente cuya función es dar un grado de cobertura al ingeniero de verificación para poder realizar una verificación basada en cobertura. El diseño de este componente es una novedad dentro de los trabajos previos realizados en la división dsi del IUMA, donde se enmarca este TFM. Este componente está íntimamente ligado a los mecanismos de cobertura que aporta *SystemVerilog*. Estos mecanismos son, por ejemplo, la definición de grupos de cobertura, *covergroups* y puntos de cobertura *coverpoint*, entre otros.

5.1. Adaptación previa del diseño

Como se ha explicado, el procesador RISC-V a verificar tiene las memorias de instrucciones y datos integradas internamente en la CPU. Sin embargo, para ciertos objetivos de diseño y verificación, como en un entorno de verificación UVM, es beneficioso extraer estas memorias y

conectarlas externamente ya que facilita el acceso a las señales que nos permite tanto estimular el DUV como manejar las señales de control. Para ello, se ha redefinido las interfaces de entrada y salida de la CPU con el fin de extraer ambas memorias.

En primer lugar, tras hacer el estudio del diseño a nivel RTL, se han identificado las interfaces involucradas en las transacciones de memoria dentro del procesador. Estas incluyen el bus de direcciones, que especifica la dirección de memoria a la que se accede; el bus de datos, que transporta los datos hacia y desde la memoria, y aquellas señales de control que son las responsables de indicar las operaciones de lecturas y/o escrituras. En el Código 52 se muestra la redefinición de los puertos de entrada y salida del procesador. Esta nueva interfaz se ajusta al hecho de haber eliminado las memorias del módulo top CPU. Las señales internas del conexionado de las memorias pasan a ser señales de interconexión del nuevo interfaz que pasará a definirse en el *testbench* al mismo nivel de jerarquía que la propia CPU. Entre las líneas 12 y 18 se encuentra los nuevos puertos dedicados a la conexión con las memorias externas. Es importante conocer el nuevo conexionado que se muestra de manera ilustrativa en la Figura 61, para definir los nuevos puertos considerando sus direcciones, *input* o *output.*

```
1  module CPU
2  (
3      clk_i,
4      DataOrReg,
5      address,
6      instr_i,
7      reset,
8      vout_addr,
9      value_o,
10     is_positive,
11     easter_egg,
12     inst_addr,              //Dirección instruccón -> Instruction_Memory
13     inst,                   //Instrucción -> Instruction_Memory
14     aluToDM_data_o,         //Dirección dato -> DataMemory
15     EX_MEM_RDData_o,        //Dato entrada -> DataMemory
16     Data_Memory_data_o,     //Dato salida  -> DataMemory
17     EX_MEM_MemWrite_o,      //Escritura dato -> DataMemory
18     EX_MEM_MemRead_o        //Lectura dato -> DataMemory
19 );
20 //------------------------- I/O Ports ----------------------------//
21 input                    clk_i;
22 input                    DataOrReg;
23 input  [4:0]             address;
24 input  [7:0]            instr_i;
25 input                    reset;
26 input  [1:0]             vout_addr;
27 output reg[7:0]          value_o;
28 output                  is_positive;
29 output   reg[2:0]        easter_egg;
30 output wire [31:0]       inst_addr;
31 input   [31:0]           inst;
32 output wire [31:0]       aluToDM_data_o;
33 output wire [31:0]       EX_MEM_RDData_o;
34 output wire              EX_MEM_MemWrite_o;
35 output wire              EX_MEM_MemRead_o;
36 input  [31:0]            Data_Memory_data_o;
```

Código 52 Adaptación de la Interfaz E/S módulo CPU

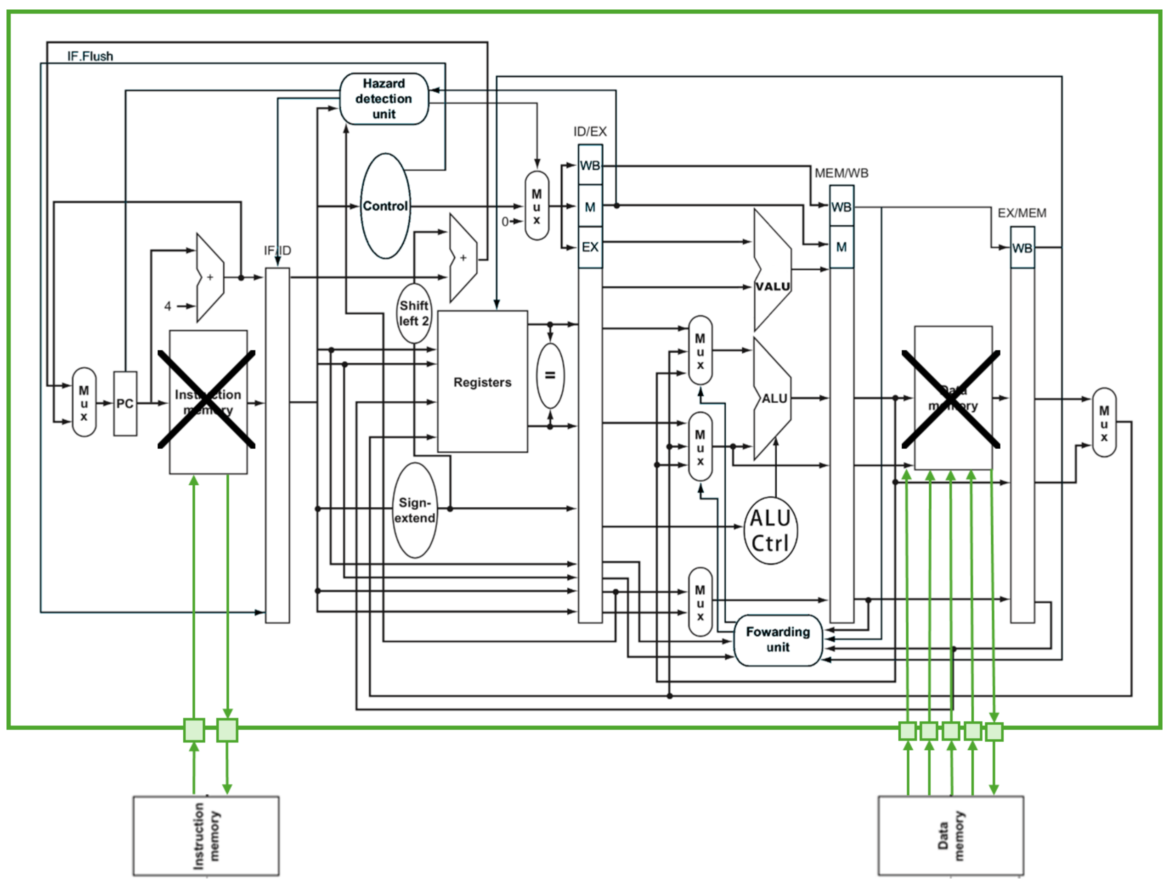


Figura 61 Adaptación de la interfaz E/S del módulo CPU

5.2. Estructura del entorno de verificación del RISC-V en UVM

A continuación, se procede a explicar la estructura del entorno UVM planteado para la verificación del módulo RISC-V elegido, mostrado en la Figura 62. También se describirán los componentes que intervienen en su implementación.

Este entorno sigue, en esencia, la estructura ya mostrada en la Figura 24, con la salvedad de que en este caso se incluyen dos componentes UVM *Agent* pasivos. El primero de ellos se centrará en la interfaz que gestiona la memoria de instrucciones, mientras que el segundo se hará cargo de la interfaz conectada con la memoria de datos. Por lo tanto, también se tienen dos tipos de interfaces y de monitores diferenciadas. Un tipo para cada componente *Agent.* Finalmente, se han añadido el componente *Scoreboard* y el componente *Subscriber*. El primero de ellos no se implementa en este TFM al no tener como objetivo el uso de un modelo de referencia. El segundo, componente *subscriber* se encarga de recibir las transacciones enviadas por la memoria de instrucciones y detectadas por el componente *monitor* de la interfaz de instrucciones. Con las transacciones recibidas, este componente realiza una cobertura para indicar el grado de diferentes instrucciones que se han cubierto en la simulación.

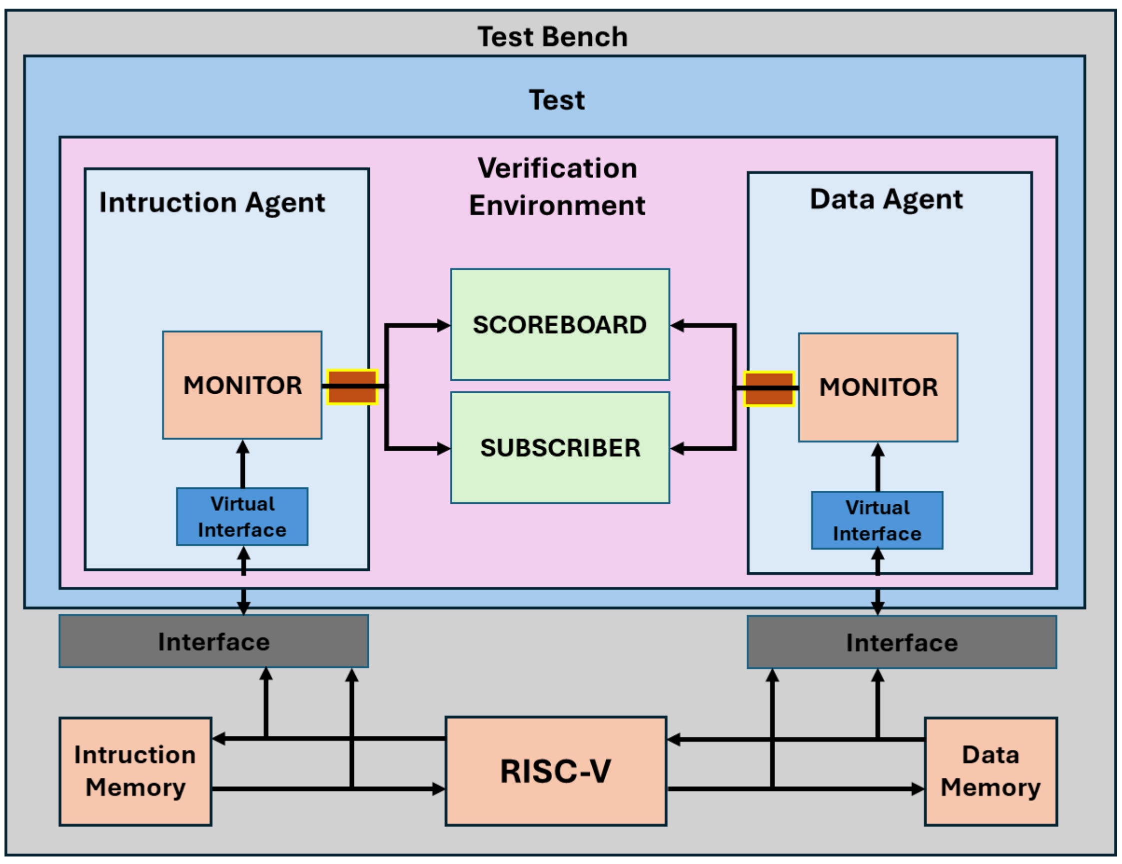


Figura 62 Estructura del entorno UVM creado para la verificación del procesador RISC-V

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 que intervienen. Para ello, se seguirá una metodología *down-top*, partiendo desde los componentes de menor nivel de abstracción en la jerarquía del entorno de verificación UVM, como se ha ido realizando durante el transcurso de este trabajo.

5.2.1. Transacción de instrucción y Transacción de datos

Para abordar el entorno propuesto, es necesario implementar dos tipos de transacciones o paquetes. Por un lado, la transacción `inst_packet` es la utilizada por el monitor de la interfaz de instrucciones, donde almacenará las señales presentes en la interfaz que conecta el RISC-V con la memoria de instrucciones y que enviará mediante los mecanismos de TLM a los componentes *Scoreboard* y *Subscriber*. En la línea 2 del Código 53, se declara la variable asociada al campo `inst`, con los 32 bits que forman la instrucción a ejecutar. En la línea 2, se declara la variable asociada al campo `inst_addr`, con la dirección de la instrucción. Entre las líneas 5 a 11, se definen las variables que se utilizará para decodificar los distintos campos que forma una instrucción con arquitectura RISC-V. Estos campos se ajustan al estándar definido por RISC-V.

Entre las líneas 12 a 22 se registra la clase `inst_packet` dentro de la *Factory* y, seguidamente, en la línea 23, se define el constructor de la clase padre `uvm_sequence_item`. En la línea 27 se define la función `decode()`, que se encarga de decodificar la instrucción para que el componente *subscriber*, al recibir las transacciones `inst_packet`, tenga acceso directo a los campos requeridos para cometer su objetivo como se explicará más adelante. Por último, en la línea 36, la tarea `displayAll()` mostrará por pantalla, con el fin de depurar el código, la instrucción y dirección de cada transacción.

```
1   class inst_packet extends uvm_sequence_item;
2         rand bit [31:0] inst;
3         bit [31:0] inst_addr;
4         //Propiedades para extraer campos específicos
5         bit [6:0]  opcode;
6         bit [3:0]  funct3;
7         bit [11:0] funct7;
8         bit        sign;
9         bit [4:0]  rs1;
10        bit [4:0]  rs2;
11        bit [4:0]  rd;
12        `uvm_object_utils_begin(inst_packet)
13              `uvm_field_int(inst, UVM_DEFAULT)
14              `uvm_field_int(inst_addr, UVM_DEFAULT)
15              `uvm_field_int(opcode, UVM_DEFAULT)
16              `uvm_field_int(funct3, UVM_DEFAULT)
17              `uvm_field_int(funct7, UVM_DEFAULT)
18              `uvm_field_int(sign, UVM_DEFAULT)
19              `uvm_field_int(rs1, UVM_DEFAULT)
20              `uvm_field_int(rs2, UVM_DEFAULT)
21              `uvm_field_int(rd, UVM_DEFAULT)
22        `uvm_object_utils_end
23        function new(string name = "inst_packet");
24              super.new(name);
25        endfunction: new
26        //Decodificación de la instrucción
27        function void decode();
28              opcode = inst[6:0];
29              funct3 = inst[14:12];
30              funct7 = inst[31:20];
31              sign = inst[31];
32              rs1 = inst[19:15];
33              rs2 = inst[24:20];
34              rd  = inst[11:7];
35        endfunction: decode
36        virtual task displayAll();
37        `uvm_info("DP", $sformatf("inst = %0h inst_addr = %0h", inst,
38  inst_addr), UVM_LOW)
39        endtask: displayAll
40  endclass: inst_packet
```

Código 53 Transacción `inst_packet`

El Código 54 muestra la implementación del paquete asociado a la interfaz de datos, denominado `data_packet`. En este caso, las variables, definidas entre las líneas 2 a 6 son: `data_in`, con el dato a insertar en memoria; `data_out`, con el dato de salida en caso de instrucción de lectura en la memoria de datos; `data_addr`, con la dirección de memoria y, finalmente, `data_R` y `data_W` que indican el tipo de operación a realizar en memoria, lectura o escritura respectivamente. Entre las líneas 8 a 14 se registra el paquete en la *Factory*. En la línea 16 se define el constructor de la clase padre `uvm_sequence_item` y, por último, se describe en la línea 20 la tarea `displayAll()`, encargada de imprimir por pantalla el contenido del paquete.

```
1  class data_packet extends uvm_sequence_item;
2        bit [31:0] data_in;        //EX_Mem_RData_o
3        bit [31:0] data_out;       //Data_Memory_Data
4        bit [31:0] data_addr;      //aluToDM_data_o
5        bit data_R;                 //EX_Mem_MemRead
6        bit data_W;                 //EX_Mem_MemWrite
7
8        `uvm_object_utils_begin(data_packet)
9              `uvm_field_int(data_in, UVM_DEFAULT)
10             `uvm_field_int(data_out, UVM_DEFAULT)
11             `uvm_field_int(data_addr, UVM_DEFAULT)
12             `uvm_field_int(data_R, UVM_DEFAULT)
13             `uvm_field_int(data_W, UVM_DEFAULT)
14       `uvm_object_utils_end
15
16       function new(string name = "data_packet");
17             super.new(name);
18       endfunction: new
19
20       virtual task displayAll();
21       `uvm_info("DP", $sformatf("data_in = %0h data_out = %0h
22 data_addr = %0h data_R = %0h data_W = %0h", data_in, data_out,
23 data_addr, data_R, data_W), UVM_LOW)
24       endtask: displayAll
25
26 endclass: data_packet
```

Código 54 Transacción `data_packet`

5.2.2. Interfaces Virtuales

Como se ha explicado, el entorno de verificación UVM propuesto, define dos interfaces virtuales. 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.

En el Código 55, se muestra la implementación de la interfaz involucrada en la comunicación entre el entorno de verificación y la memoria de instrucciones. Esta interfaz tiene como puertos de entrada, las señales de `clk` y `rst`, tal y como se refleja en la línea 1.

```
1  interface inst_if(input logic clk, rst);
2
3        logic [31:0] inst;
4        logic [31:0] inst_addr;

endinterface: inst_if
```
Código 55 Interfaz de instrucciones `inst_if`

El Código 56, por su parte, muestra la implementación de la interfaz asociada a la intervención con la memoria de datos. En este caso, la interfaz contiene los campos asociados a dicha memoria, que corresponde a los campos, explicados dentro del componente transacción.

```
1  interface data_if(input logic clk,rst);
2
3        logic [31:0] data_in;
4        logic [31:0] data_out;
5        logic [31:0] data_addr;
6        logic data_R;
7        logic data_W;

endinterface: data_if
```
Código 56 Interfaz de datos `data_if`

5.2.3. Componente *Monitor*

En primer lugar, el componente UVM *Monitor* de instrucciones se implementa en la clase `inst_monitor`. En la descripción de este componente, las principales tareas que realiza son las siguientes:

- **build_phase:** Al igual que se ha explicado con el ejemplo durante todo este Trabajo Fin de Máster sobre la PIFO, en esta fase, se efectúa la extracción de la interfaz virtual de instrucciones, inicializando con ello el puntero definido en la línea 17. En caso de no encontrar alguno de estos parámetros, emite un mensaje de tipo `UVM_FATAL`, deteniendo la simulación, tal y como se muestra en la línea 18.
- **run_phase:** A diferencia de un entorno con agentes activos, donde se utiliza el mecanismo de `objections` en el envío de secuencias mientras se mantenga el protocolo de *handshake* entre los componentes *sequencer* y el componente *driver*, en este caso el componente *monitor,* mantiene su fase `run_phase` activa siempre que se produzca envíos de secuencias. Sin embargo, al no contar en este entorno con un agente activo que implemente el mecanismo de `objection`, el monitor empezaría y terminaría su fase de ejecución en el instante 0 de simulación. Para resolver este problema, en la línea 29 del Código 57, se activa un `objection` con la función `phase.raise_objection()`, para indicar al resto de componentes que la fase *run* no puede finalizar hasta que el monitor avise mediante la ejecución de la orden

phase.drop_objection(). Esto permite, ejecutar la tarea collect_data(), definida en la línea 32, que es la encargada de almacenar en cada flanco de reloj las señales presentes en la interfaz de instrucciones dentro del paquete inst_packet. Esta operación se realiza en las líneas 38 a 40. Por último, el componente envía el paquete a través del puerto de análisis item_collected_port. De esta manera, los componentes *Scoreboard* y *Subscriber* podrán acceder a dichos paquetes y realizar las operaciones pertinentes.

```
1  class inst_monitor extends uvm_monitor;
2     virtual inst_if ivif;
3     string monitor_intf_i;
4     int num_pkts;
5
6     uvm_analysis_port #(inst_packet) item_collected_port;
7     inst_packet inst_collected;
8
9     `uvm_component_utils(inst_monitor)
10
11    function new(string name, uvm_component parent);
12      super.new(name, parent);
13    endfunction: new
14
15    function void build_phase(uvm_phase phase);
16      super.build_phase(phase);
17      if(!uvm_config_db#(string)::get(this,"","monitor_intf_i", monitor_intf_i))
18     `uvm_fatal("NOSTRING", {"Need intf:",get_full_name(), ".monitor_intf_i"})
19     `uvm_info(get_type_name(),$sformatf("INTF=%0s", monitor_intf_i),UVM_HIGH)
20
21      if(!uvm_config_db#(virtual inst_if)::get(this,"",monitor_intf_i,ivif))
22     `uvm_fatal("NOVIF", {"intF must be set for: ", get_full_name(), ".ivif"})
23
24      item_collected_port = new("item_collected_port", this);
25      inst_collected = inst_packet::type_id::create("inst_collected");
26
27      `uvm_info(get_full_name(), "Build stage complete.", UVM_LOW)
28     endfunction: build_phase
29
30    virtual task run_phase(uvm_phase phase);
31      phase.raise_objection(this);
32      collect_data();
33      phase.drop_objection(this);
34    endtask: run_phase
35
36    virtual task collect_data();
37      forever begin
38           @(posedge ivif.clk);
39           inst_collected.inst <= ivif.inst;
40           inst_collected.inst_addr <= ivif.inst_addr;
41           item_collected_port.write(inst_collected);
42           num_pkts++;
43      end
44    endtask: collect_data
45
46    virtual function void report_phase(uvm_phase phase);
47      `uvm_info(get_type_name(),$sformatf("CollectINST=%0d",num_pkts), UVM_HIGH)
48    endfunction: report_phase
49 endclass: inst_monitor
```

Código 57 Componente *Monitor* - inst_monitor

Por otro lado, el componente *monitor* responsable de muestrear la interfaz de datos resulta muy similar. La principal diferencia radica en la tarea `collect_data()`, como se muestra en el Código 58. En este caso, en cada flanco de reloj, se debe comprobar que la señal `data_R` o la señal `data_W` (línea 10) se encuentren activas en la interfaz de datos indicando que se ha producido una operación de escritura o lectura, que son las señales que validan las transacciones de datos. Cuando se detecte una transacción, se almacena en el paquete `data_packet` las señales que intervienen en la interfaz (líneas 11 a 16). Finalmente, en la línea 16, se envía el paquete de la misma manera que ocurre con el monitor de instrucciones, usando el método `write` asociado a su puerto TLM.

```
1   virtual task run_phase(uvm_phase phase);
2         phase.raise_objection(this);
3         collect_data();
4         phase.drop_objection(this);
5   endtask: run_phase
6
7   virtual task collect_data();
8     forever begin
9      @(posedge dvif.clk);
10     wait(dvif.data_R || dvif.data_W)
11     data_collected.data_R <= dvif.data_R;
12     data_collected.data_W <= dvif.data_W;
13     data_collected.data_in <= dvif.data_in;
14     data_collected.data_out <= dvif.data_out;
15     data_collected.data_addr <= dvif.data_addr;
16     item_collected_port.write(data_collected);
17     num_pkts++;
18    end
19  endtask: collect_data
```

Código 58 Componente *Monitor* - `data_monitor`

5.2.4. Componente *Agent*

Siguiendo la estructura del entorno de verificación, el componente UVM *Agent* de instrucciones se implementa en la clase `inst_agent` y el componente UVM *Agent* de datos se implementa en la clase `data_agent`. Como se ha mencionado, ambos agentes se crean como tipo pasivos, por lo que cada uno contiene un único componente *monitor*. En el Código 59 se muestra la implementación del agente responsable de la interfaz de instrucciones, mientras que el responsable de datos se desarrolla de la misma manera, pero teniendo en cuenta los componentes correspondientes a cada interfaz.

En la línea 2, se indica el comportamiento del agente a través de la variable `is_active`, iniciando este valor a `UVM_PASSIVE.` De esta forma, cuando se ejecute la fase `build_phase`, en la línea 13 y se compruebe el valor de dicha variable no se crearán los componentes *sequencer* ni *driver*. En la línea 18 se crea el componente *monitor* mediante el método `create`. Y, como siempre, se registra el componente dentro de la *Factory* en las líneas 5 a 7.

```
1  class inst_agent extends uvm_agent;
2        protected uvm_active_passive_enum is_active = UVM_PASSIVE;
3
4        inst_monitor  imonitor;
5        `uvm_component_utils_begin(inst_agent)
6        `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_ALL_ON)
7        `uvm_component_utils_end
8
9        function new(string name, uvm_component parent);
10             super.new(name, parent);
11       endfunction
12
13       function void build_phase(uvm_phase phase);
14             super.build_phase(phase);
15       if(is_active == UVM_ACTIVE) begin
16       end
17
18       imonitor = inst_monitor::type_id::create("imonitor",this);
19       `uvm_info(get_full_name(),"Build stage complete.", UVM_LOW)
20       endfunction: build_phase
21 endclass: inst_agent
```
Código 59 Componente Agent - `inst_agent`

5.2.5. Componente *Subscriber*

Durante el desarrollo de este TFM, se ha decidido utilizar herramientas de cobertura funcional para identificar qué partes del diseño han sido ejercitadas y cuáles no. Todo este proceso depende de las instrucciones que se ejecuten. El objetivo es conocer la cobertura de todos los tipos de instrucciones posibles que soporta el procesador RISC-V; el acceso a memoria tanto lecturas como escrituras; los accesos a direcciones altas y bajas de memoria; el uso de los diferentes registros, tanto utilizados como fuente o como destino, y el uso de datos inmediatos positivos o negativos.

Para ello, se ha optado por hacer uso de las estructuras ***covergroups*** de *SystemVerilog* que permite recolectar datos de cobertura durante la simulación. La implementación de los *covergroups* se hará dentro del componente *Subscriber*, ya que este componente permite recibir y procesar las transacciones enviadas por el componente *monitor*. Cada vez que el componente monitor detecta una transacción, ejecuta el método `write`, tal y como se ha descrito. El componente *subscriber* implementa este método, por lo que resulta ideal para activar y mantener la cobertura del sistema. De esta manera se diferencian las funciones de monitorización y de verificación en componentes separado. Además, se centraliza la cobertura en un solo componente, facilitando la gestión y análisis de los datos de cobertura. El Código 60 muestra la cabecera del componente UVM *subscriber.* Este componente se implementa en la clase `subscriber` que deriva de la clase base `uvm_subscriber`, La línea 1 muestra la definición de la clase, que tiene como parámetro el paquete, en este caso se trata de la transacción asociada a la interfaz de instrucciones, ya que el

interés es extraer la información de cada instrucción recibida por dicha interfaz y registrar la cobertura cubierta.

```
1  class subscriber extends uvm_subscriber #(inst_packet);
2        inst_packet ipacket;
3        int count;
4        `uvm_component_utils(subscriber)
```

Código 60 Componente UVM `Subscriber`

A continuación, se explica la estructura de los diferentes *covergroups* utilizados para recolectar datos de cobertura sobre los diferentes tipos de instrucciones que soporta el procesador RISC-V a verificar. Cada *covergroup* define varios *coverpoints* que especifican qué campos de la instrucción se van a monitorizar.

El Código 61 muestra el *covergroup* para las instrucciones de tipo R. En la línea 2, al activar la opción `option.per_instance`, se indica que la cobertura se recolectará por cada unidad creada del *covergroup*. De esta manera, cada instancia maneja sus propios datos de cobertura de manera independiente. Entre las líneas 3 a 5 se define el primer *coverpoint* o punto de cobertura denominado `Op_code`. Este *coverpoint* monitoriza el campo `opcode` de la instrucción recibida por parte del componente *monitor* en la función `write()`, como se explicará más adelante. La definición de un punto de cobertura, *coverpoint*, permite el uso de los denominados *bins*. Un *bin* no es más que la definición de un valor, o rango de valores, que se pretenden monitorizar. En la línea 4 se define el *bin* denominado `Op_Tipo_R` que captura cuando el *opcode* sea `7'h33`, que corresponde con el *opcode* para instrucciones tipo R en RISC-V. Además, para indicar que un *bin* específico no debe contribuir al cálculo de la cobertura global cuando se evalúa ese *coverpoint* se utiliza la opción `option.weight` con el valor zero, tal y como se ve en la línea 5. Esto es debido a que el cálculo de la cobertura real para cada instrucción se hará mediante la combinación de varios puntos de cobertura. Esto es posible gracias al método `cross` (línea 62) que combina todos los *coverpoints* para obtener una cobertura cruzada completa, estableciendo un peso de cobertura igual al peso relativo a cada uno de ellos.

Entre las líneas 7 a 11 se define el *coverpoint* denominado `Funct3`, que monitoriza el campo `funct3` de la instrucción. Este punto de cobertura define *bins* para los diferentes valores que corresponde a las distintas operaciones dentro de las instrucciones tipo-R. En este caso se han definido las operaciones `add`, `and` y `or`. Además, siguiendo la codificación de instrucciones del estándar RISC-V, tanto las operaciones `add` como las operaciones `sub`, tienen el mismo *opcode* y también el mismo campo `funct3`, por lo que es necesario distinguirlas a través del campo `funct7`. Por este motivo, en la línea 13, se añade un nuevo *coverpoint,* `Funct7`, para monitorizar este campo con *bins* para valores que diferencian entre `add` y `sub`.

Entre las líneas 18 a 61, se define los *coverpoints* responsables de monitorizar los registros de los operandos fuente `rs1` y `rs2`, y el registro de destino `rd`. La estrategia seguida para determinar el grado de cobertura en cuanto al uso de los diferentes registros consiste en considerar la finalidad de los mismos. Teniendo en cuenta que el procesador dispone de 32 registros, en vez de usar un *bin* para cada uno, se ha optado por agrupar aquellos registros cuyo uso sea el mismo. Por ejemplo, los registros `x18` a `x27` (*Temporaries*) corresponden a registros de propósito general que no conservan el valor entre funciones. Destacar, que el caso de `rd`, se descarta como parte de cobertura la detección del registro x0. Esta estrategia permite reducir drásticamente el número de instrucciones para obtener una cobertura aceptable.

Por último, en la línea 62, se combina todos los *coverpoints* anteriores para obtener una cobertura cruzada completa, asignando el peso relativo con la opción `weight` como el producto de los pesos de los *coverpoints* individuales.

```
1  covergroup cg_R;
2         option.per_instance = 1;
3         Op_code:   coverpoint ipacket.opcode{
4                              bins Op_Tipo_R = {7'h33};
5                              option.weight=0;
6         }
7         Funct3:    coverpoint ipacket.funct3{
8                              bins add_f3 = {3'h0};
9                              bins and_f3 = {3'h7};
10                             bins or_f3  = {3'h6};
11                             option.weight=0;
12        }
13        Funct7:    coverpoint ipacket.funct7{
14                             bins add_f7 = {3'h0};
15                             bins sub_f7 = {3'h20};
16                             option.weight=0;
17        }
18        Rs1:       coverpoint ipacket.rs1{
19                             bins x0 = {5'b00000};
20                             bins x1 = {5'b00001};
21                             bins x2 = {5'b00010};
22                             bins x3 = {5'b00011};
23                             bins x4 = {5'b00100};
24                             bins x5_x7 = {[5:7]};
25                             bins x8 = {5'b01000};
26                             bins x9 = {5'b01001};
27                             bins x10_x11 = {[10:11]};
28                             bins x12_x17 = {[12:17]};
29                             bins x18_x27 = {[18:27]};
30                             bins x28_x31 = {[28:31]};
31                             option.weight=0;
32        }
33        Rs2:      coverpoint ipacket.rs2{
34                             bins x0 = {5'b00000};
35                             bins x1 = {5'b00001};
36                             bins x2 = {5'b00010};
37                             bins x3 = {5'b00011};
38                             bins x4 = {5'b00100};
39                             bins x5_x7 = {[5:7]};
40                             bins x8 = {5'b01000};
41                             bins x9 = {5'b01001};
42                             bins x10_x11 = {[10:11]};
43                             bins x12_x17 = {[12:17]};
```

```
44                          bins x18_x27 = {[18:27]};
45                          bins x28_x31 = {[28:31]};
46                          option.weight=0;
47        }
48        Rd:          coverpoint ipacket.rd{
49                          bins x1 = {5'b00001};
50                          bins x2 = {5'b00010};
51                          bins x3 = {5'b00011};
52                          bins x4 = {5'b00100};
53                          bins x5_x7 = {[5:7]};
54                          bins x8 = {5'b01000};
55                          bins x9 = {5'b01001};
56                          bins x10_x11 = {[10:11]};
57                          bins x12_x17 = {[12:17]};
58                          bins x18_x27 = {[18:27]};
59                          bins x28_x31 = {[28:31]};
60                          option.weight=0;
61        }
62        Total_cg_R: cross Op_code, Funct3, Rs1, Rs2, Rd{
63                     option.weight = 1*4*12*12*11;
64        }
65 endgroup: cg_R
```
Código 61 Componente *Subscriber* - covergroup para instrucciones Tipo-R

En el Código 62, se implementa el *covergroup* para la cobertura de instrucciones tipo-I. En la línea 1 se declara el *covergroup* `cg_I`, y de la misma manera que ocurrirá con el resto de *covergroups*, se activa la opción `option.per_instance` para asegurar que la cobertura se recopile por cada *covergroup* definido y no globalmente a través de todas las instancias de este *covergroup*.

En el caso de las instrucciones tipo-I, es necesario añadir un nuevo *bin* para el *coverpoint* `Op_code`, ya que en este tipo se distingue mediante el *opcode*, por un lado, las instrucciones de operaciones entre registros con datos inmediatos y por otro, las instrucciones de carga de memoria. En este caso se ha definido el *bin* `Op_Tipo_I_imm` como, el *bin* encargado de capturar todas las instrucciones con *opcode* igual a `7'h13` y el *bin* `Op_Tipo_I_load` como, el *bin* encargado de capturar las instrucciones con *opcode* igual a `7'h03`. En la línea 6, se fija la opción `weight` a cero, indicando que las ocurrencias de estos *bins* no contribuirán a la cobertura global.

El *coverpoint* `Funct3` (línea 8), monitoriza el campo `funct3` de la instrucción almacenado en el paquete de instrucciones. En este caso se requiere la definición de 3 *bins* para poder diferenciar entre las operaciones `addi`, `ori` y `load`. El caso de esta última, se codifica la instrucción *load Word* (`lw`) ya que es la que soporta el procesador RISC-V a verificar.

Para las instrucciones tipo-I se añade un nuevo *coverpoint* que monitoriza el signo del campo que codifica el dato inmediato. Para ello se crea el *bin* `pos`, que captura valores inmediatos positivos y el *bin* `neg` que captura valores inmediatos negativos. Destacar que el valor recibido en la línea 15 mediante `ipacket.sign` es el bit más significativo del campo de dato inmediato,

conociendo que los valores negativos se representan en formato complemento a dos, si este vale '1', implica que el dato sea negativo.

Entre las líneas 20 a 23 se crea el *covergroup* `Funct7` que monitoriza el campo `funct7` de las instrucciones. A diferencia de las instrucciones tipo-R, este *covergroup* donde se utilizó para diferenciar entre operaciones `add` y `sub`, para las instrucciones tipo-I, cuando se trate de una operación `lw`, el campo `funct7` indica la dirección de lectura de memoria. Por este motivo, se propone la definición de un *bin* denominado `low_f7` para capturar accesos a direcciones bajas de memoria y el bin `high_f7` para la captura de accesos a direcciones altas de memoria. Los valores propuestos para este caso particular son `0:2048` y `2048:4096` respectivamente.

Finalmente, indicar que los *covergroups* relacionados con los campos registro fuente `Rs1` y registro destino `Rd`. Se mantienen igual a los explicados para instrucciones tipo-R. En este caso, solo se cuenta con un registro fuente en el campo `rs1` de la instrucción, por lo que el *covergroup* para `rs2` desaparece.

```
1  covergroup cg_I;
2          option.per_instance = 1;
3          Op_code:   coverpoint ipacket.opcode{
4                       bins Op_Tipo_I_imm = {7'h13};
5                       bins Op_Tipo_I_load = {7'h3};
6                       option.weight=0;
7          }
8          Funct3:    coverpoint ipacket.funct3{
9                       bins addi_f3 = {3'h0};
10                      bins ori_f3 = {3'h6};
11                      //L_WIDTH indica Load Word
12                      bins lw_f3 = {3'h2};
13                      option.weight=0;
14         }
15         I_imm:      coverpoint ipacket.sign{
16                      bins pos = {1'b0};
17                      bins neg = {1'b1};
18                      option.weight=0;
19         }
20         Funct7:     coverpoint ipacket.funct7{
21                      bins low_f7  = {[0:2048]};
22                      bins high_f7 = {[2048:4096]};
23                      option.weight=0;
24         }
25         Rs1:         coverpoint ipacket.rs1{
26                      bins x0 = {5'b00000};
27                      bins x1 = {5'b00001};
28                      bins x2 = {5'b00010};
29                      bins x3 = {5'b00011};
30                      bins x4 = {5'b00100};
31                      bins x5_x7 = {[5:7]};
32                      bins x8 = {5'b01000};
33                      bins x9 = {5'b01001};
34                      bins x10_x11 = {[10:11]};
35                      bins x12_x17 = {[12:17]};
36                      bins x18_x27 = {[18:27]};
37                      bins x28_x31 = {[28:31]};
38                      option.weight=0;
39         }
40         Rd:          coverpoint ipacket.rd{
```

```
41                  bins x1 = {5'b00001};
42                  bins x2 = {5'b00010};
43                  bins x3 = {5'b00011};
44                  bins x4 = {5'b00100};
45                  bins x5_x7 = {[5:7]};
46                  bins x8 = {5'b01000};
47                  bins x9 = {5'b01001};
48                  bins x10_x11 = {[10:11]};
49                  bins x12_x17 = {[12:17]};
50                  bins x18_x27 = {[18:27]};
51                  bins x28_x31 = {[28:31]};
52                  option.weight=0;
53          }
54          Total_cg_I: cross Op_code, Funct3, I_imm, Funct7, Rs1, Rd{
55                  option.weight = 2*3*2*2*12*11;
56          }
57  endgroup: cg_I
```

Código 62 Componente *Subscriber* - covergroup para instrucciones Tipo-I

Para el caso de las instrucciones tipo-S (Código 63), se crea el *covergroup* `cg_S`. De la misma manera que en los casos anteriores, en la línea 4 se define el *coverpoint* `Op_code` que monitoriza el campo *opcode* de la instrucción. En este caso se define el *bin* `Op_Tipo_S` cuya captura es verdadera cuando el *opcode* sea el propio de una instrucción tipo-S, `7'h23`.

En la línea 8 el *covergroup* `Funct3` monitoriza el campo `funct3` de la instrucción. Únicamente se define el *bin* `sw_f3` que solo valida cuando este campo toma el valor `3'h2` indicando operación *Store Word* ya que es el único tipo de operación que soporta el RISC-V a verificar. Finalmente, entre las líneas 12 a 41 se define los *covergroups* encargados de la cobertura funcional para los registros. En este tipo de instrucciones se usan tanto el registro fuente `rs1`, que contiene la dirección base, como el registro `rs2`, que contiene el dato a almacenar.

```
1   covergroup cg_S;
2           option.per_instance = 1;
3           Op_code:  coverpoint ipacket.opcode{
4                   bins Op_Tipo_S = {7'h23};
5                   option.weight=0;
6           }
7           Funct3:   coverpoint ipacket.funct3{
8                   bins sw_f3 = {3'h2};
9                   option.weight=0;
10          }
11          Rs1:       coverpoint ipacket.rs1{
12                  bins x0 = {5'b00000};
13                  bins x1 = {5'b00001};
14                  bins x2 = {5'b00010};
15                  bins x3 = {5'b00011};
16                  bins x4 = {5'b00100};
17                  bins x5_x7 = {[5:7]};
18                  bins x8 = {5'b01000};
19                  bins x9 = {5'b01001};
20                  bins x10_x11 = {[10:11]};
21                  bins x12_x17 = {[12:17]};
22                  bins x18_x27 = {[18:27]};
23                  bins x28_x31 = {[28:31]};
24                  option.weight=0;
25          }
26          Rs2:       coverpoint ipacket.rs2{
27                  bins x0 = {5'b00000};
```

```
28                      bins x1 = {5'b00001};
29                      bins x2 = {5'b00010};
30                      bins x3 = {5'b00011};
31                      bins x4 = {5'b00100};
32                      bins x5_x7 = {[5:7]};
33                      bins x8 = {5'b01000};
34                      bins x9 = {5'b01001};
35                      bins x10_x11 = {[10:11]};
36                      bins x12_x17 = {[12:17]};
37                      bins x18_x27 = {[18:27]};
38                      bins x28_x31 = {[28:31]};
39                      option.weight=0;
40          }
41          Total_cg_S: cross Op_code, Funct3, Rs1, Rs2{
42                      option.weight = 1*1*12*12;
43          }
44  endgroup: cg_S
```

Código 63 Componente *Subscriber* - covergroup para instrucciones Tipo-S

Por último, la cobertura de las instrucciones tipo-B se implementa en el Código 64 mediante el *covergroup* denominado `cg_B`.

De manera similar al resto de *covergroups*, en la línea 3 se define el *coverpoint* `Op_code` que monitoriza el campo *opcode* de la instrucción y se captura mediante el *bin* `Op_Tipo_R`. Cuando el código de operación tome el valor propio al de una instrucción tipo-R `7’h63` se producirá un acierto.

El *coverpoint* `Funct3`, definido en la línea 7 del Código 64 para el caso de instrucciones tipo-B, se define un solo *bin* denominado `beq_f3`, que capture el valor `3’h0`. Esto es debido a que este tipo de instrucciones en el procesador RISC-V seleccionado, solo implementa la instrucción `beq`. Finalmente, entre las líneas 11 a 40, se definen los *coverpoints* encargados de monitorizar los campos de registros fuentes. Cada uno de ellos define todas las agrupaciones de registros mencionados anteriormente, de manera que se cubren todas las posibilidades de comparación entre registros cuando la cobertura para esta instrucción sea máxima.

```
1   covergroup cg_B;
2           option.per_instance = 1;
3           Op_code:   coverpoint ipacket.opcode{
4                       bins Op_Tipo_B = {7'h63};
5                       option.weight=0;
6           }
7           Funct3:    coverpoint ipacket.funct3{
8                       bins beq_f3 = {3'h0};
9                       option.weight=0;
10          }
11          Rs1:       coverpoint ipacket.rs1{
12                      bins x0 = {5'b00000};
13                      bins x1 = {5'b00001};
14                      bins x2 = {5'b00010};
15                      bins x3 = {5'b00011};
16                      bins x4 = {5'b00100};
17                      bins x5_x7 = {[5:7]};
18                      bins x8 = {5'b01000};
19                      bins x9 = {5'b01001};
20                      bins x10_x11 = {[10:11]};
21                      bins x12_x17 = {[12:17]};
```

```
22                          bins x18_x27 = {[18:27]};
23                          bins x28_x31 = {[28:31]};
24                          option.weight=0;
25          }
26          Rs2:        coverpoint ipacket.rs2{
27                          bins x0 = {5'b00000};
28                          bins x1 = {5'b00001};
29                          bins x2 = {5'b00010};
30                          bins x3 = {5'b00011};
31                          bins x4 = {5'b00100};
32                          bins x5_x7 = {[5:7]};
33                          bins x8 = {5'b01000};
34                          bins x9 = {5'b01001};
35                          bins x10_x11 = {[10:11]};
36                          bins x12_x17 = {[12:17]};
37                          bins x18_x27 = {[18:27]};
38                          bins x28_x31 = {[28:31]};
39                          option.weight=0;
40          }
41          Total_cg_B: cross Op_code, Funct3, Rs1, Rs2{
42                          option.weight = 1*1*12*12;
43          }
44  endgroup: cg_B
```

Código 64 Componente Subscriber - covergroup instrucciones tipo-B

Por último, en el Código 65, se muestra la implementación de las operaciones para cada uno de los *covergroups* previamente explicados. En primer lugar, las líneas 45 a 51 define el constructor de la clase *Subscriber*. Este constructor en primer lugar, llama al constructor de la clase base `uvm_subscriber` y, en segundo lugar, crea los objetos asociados a los *covergroups* desarrollados anteriormente.

En la línea 53, comienza la función `write()`. Hay que recordar que esta función la invoca el componente monitor cada vez que detecta una instrucción y que, además, envía esta transacción como parámetro de esta función. El primer paso es decodificar el paquete que contiene la instrucción llamando al método `decode()`, desarrollado en la clase `inst_packet`. Tras la llamada, se almacena la instrucción decodificada en el paquete `ipacket`. En la línea 57, se incrementa el contador de instrucciones procesadas por el componente *Subscriber*. En la línea 59, se evalúa el campo *opcode* de la instrucción para determinar su tipo y tomar una muestra mediante el método `sample()` del *covergroup* correspondiente. De esta manera, cuando se recibe una instrucción determinada, ninguno de los *coverpoints* del resto de *covergroups* incrementará su cobertura.

Finalmente, en la línea 67, se ejecuta la fase `extract_phase`, donde se imprimirá toda la información de depuración sobre el número de paquetes de cobertura recolectados y la cobertura actual de cada *covegroup*. Es importante resaltar la necesidad de realizar este volcado una vez hayan finalizado las fases de ejecución. En este caso se ha utilizado la fase `extract`, por ser una fase cuya finalidad compatibiliza con este propósito.

```
45  function new(string name, uvm_component parent);
46        super.new(name, parent);
47        cg_R = new();
48        cg_I = new();
49        cg_S = new();
50        cg_B = new();
51   endfunction: new
52
53  function void write(inst_packet t);
54        //decodificamos la inst
55        t.decode();
56        ipacket = t;
57        count++;
58        //Filtro-instrucción no aporte al coverage de las demás
59        case (t.opcode)
60              7'h33: cg_R.sample();  // R-Type
61              7'h13, 7'h03: cg_I.sample();  // I-Type
62              7'h23: cg_S.sample();  // S-Type
63              7'h63: cg_B.sample();  // B-Type
64        endcase
65  endfunction: write
66
67  virtual function void extract_phase(uvm_phase phase);
68  `uvm_info(get_type_name(),$sformatf("pakcets collected=%0d",count), UVM_LOW)
69  `uvm_info(get_type_name(),$sformatf("coverage= %f",cg_R.get_coverage()),UVM_LOW)
70  `uvm_info(get_type_name(),$sformatf("coverage= %f",cg_I.get_coverage()),UVM_LOW)
71  `uvm_info(get_type_name(),$sformatf("coverage= %f",cg_S.get_coverage()),UVM_LOW)
72  `uvm_info(get_type_name(),$sformatf("coverage= %f",cg_B.get_coverage()),UVM_LOW)
73  endfunction: extract_phase
74  endclass: subscriber
```
Código 65 Componente Subscriber - Función `write()`

5.2.6. Componente *Environment*

Una vez desarrollados el agente de instrucciones, el agente de datos y el componente *subscriber* se englobarán en un mismo componente, el componente UVM *environment*. Este componente se denomina `riscv_env` y la implementación se muestra en el Código 66.

En primer lugar, se define la clase riscv_env, que hereda de la clase base uvm_env (línea 1), en segundo lugar, se declaran los componentes *agent* que van a estar dentro del entorno, en este caso, el encargado de la interfaz de instrucciones `inst_agent` y el de la interfaz de datos `data_agent` respectivamente (líneas 2 y 3). Además; se referencia el componente *subscriber* que también se incluye dentro del entorno (línea 4).

Como siempre, en la línea 5, se hace uso de la macro `` `uvm_component_utils `` para registrar el componente en la *Factory*. Entre las líneas 7 y 9, se declara el constructor new. Luego, entre las líneas 11 a 22, se declara la función principal de este componente, la función correspondiente a la fase `build_phase`. En la línea 12, se llama al método `build_phase` de la clase padre. Seguidamente, en las líneas 17 y 18, se crean ambos agentes y en la línea 19 se crea el componente *subscriber* mediante el método `create` tras su registro previo en la *Factory*.

Por último, en la línea 23 se ejecuta la fase `connect_phase`, que se encarga de comunicar los puertos TLM desde el agente responsable de la interfaz de instrucciones con el componente *subscriber* mediante el método `connect`.

```
1  class riscv_env extends uvm_env;
2         inst_agent  iagent;
3         data_agent  dagent;
4         subscriber  sb;
5         `uvm_component_utils(riscv_env)
6
7         function new(string name, uvm_component parent);
8               super.new(name, parent);
9         endfunction
10
11        function void build_phase(uvm_phase phase);
12              super.build_phase(phase);
13
14        uvm_config_db#(string)::set(this,
15 "iagent.imonitor","monitor_intf_i","i_intf");
16
17        iagent = inst_agent::type_id::create("iagent",this);
18        dagent = data_agent::type_id::create("dagent",this);
19        sb     = subscriber::type_id::create("sb",this);
20        `uvm_info(get_full_name(), "Build stage complete.", UVM_LOW)
21        endfunction: build_phase
22
23        function void connect_phase(uvm_phase phase);
24        iagent.imonitor.item_collected_port.connect(sb.analysis_export);
25        `uvm_info(get_full_name(), "Connect phase complete.", UVM_LOW)
26        endfunction: connect_phase
27 endclass: riscv_env
```

Código 66 Componente Environment

5.2.7. Módulo TOP

Como se explicó anteriormente en este TFM, se ha realizado una extracción de las memorias de instrucciones y datos para facilitar el acceso a las señales de las mismas. Para ello ha sido necesario eliminar ambas memorias de la CPU y ubicarlas en un nivel de jerarquía superior. Por lo tanto, es en el módulo Top donde se referencia los 3 módulos; la CPU, la memoria de instrucciones y la memoria de datos al mismo nivel de jerarquía.

En el Código 67 se muestra el primer fragmento de la implementación del módulo top denominado `riscv_top` (línea 1). La descripción comienza importando el paquete UVM y las macros de esta metodología en las líneas 3, 4 y 5. Antes de referenciar el DUV y las interfaces virtuales se generan los nodos que se utilizarán para conectar sus señales. Para ello, entre las líneas 7 a 24, se definen todas y cada una de las variables asociadas a cada una de las señales de la interfaz de instrucciones y de la interfaz de datos.

El segundo paso es definir las interfaces de instrucción y datos virtuales `ivif` y `dvif` respectivamente y, por último, en la línea 29 se genera la señal de reloj.

```
1  `define CYCLE_TIME 10.0
2  module riscv_top;
3          import uvm_pkg::*;
4          `include "uvm_macros.svh"
5          import riscv_pkg::*;
6
7          bit             clk;
8          bit             rst;
9          reg                Start;
10         reg                DataOrReg;
11         reg [4:0]          address;
12         reg [7:0]          instr_i;
13         reg [7:0]          instr_store[0:(64*4+1)];
14         reg [1:0]          vout_addr;
15         wire[7:0]          value_o;
16         wire               is_positive;
17         wire [2:0]         easter_egg;
18         wire [31:0]     inst_addr, inst;
19         wire [31:0]     aluToDM_data_o;
20         wire [31:0]     EX_MEM_RDData_o;
21         wire            EX_MEM_MemWrite_o;
22         wire            EX_MEM_MemRead_o;
23         wire [31:0]     Data_Memory_data_o;
24         wire [31:0]     data_mem_o;
25
26         inst_if ivif(.clk(clk), .rst(rst));
27         data_if dvif(.clk(clk), .rst(rst));
28
29         always #5 clk = ~clk;
```

Código 67 Primera parte del módulo `riscv_top`

En el Código 68 se muestra el segundo fragmento de la implementación del módulo top, donde se referencia en primer lugar y entre las líneas 30 a 36, la memoria de instrucciones con sus respectivas conexiones con las señales de la interfaz virtual de instrucciones, así como las señales comunes `clk` y `rst` y las señales de interconexión que no requieren ser enviadas a la interfaz virtual, pero que sí son necesarias para su correcto funcionamiento.

De la misma manera, entre las líneas 38 a 48, se referencia la memoria de datos. Su interfaz integra las conexiones internas que permiten el correcto funcionamiento y las conexiones con el entorno de verificación UVM. Por último, se referencia la CPU entre las líneas 50 a 67. La interfaz de la CPU integra tanto las conexiones internas entre las señales de salida de la CPU y las memorias, así como sus correspondientes conexiones con las interfaces virtuales.

```
30         Instruction_Memory Instruction_Memory(
31                 .clk         (clk),
32                 .reset       (rst),
33                 .addr_i      (ivif.inst_addr),
34                 .instr_i     (instr_i),
35                 .instr_o     (ivif.inst)
36         );
37
38         Data_Memory Data_Memory(
39                 .clk_i       (clk),
40                 .reset       (rst),
41                 .op_addr     (address),
42                 .addr_i      (dvif.data_addr),
43                 .data_i      (dvif.data_in),
44                 .MemWrite_i  (dvif.data_W),
```

```
45                 .MemRead_i   (dvif.data_R),
46                 .data_o      (dvif.data_out),
47                 .data_mem_o  (data_mem_o)
48         );
49 
50         CPU riscv_top(
51                 .clk_i(clk),
52                 .reset(rst),
53                 .inst(ivif.inst),
54                 .inst_addr(ivif.inst_addr),
55                 .EX_MEM_RDData_o(dvif.data_in),
56                 .Data_Memory_data_o(dvif.data_out),
57                 .aluToDM_data_o(dvif.data_addr),
58                 .EX_MEM_MemRead_o(dvif.data_R),
59                 .EX_MEM_MemWrite_o(dvif.data_W),
60                .DataOrReg(DataOrReg),
61                 .address(address),
62                 .instr_i(instr_i),
63                 .vout_addr(vout_addr),
64                 .value_o(value_o),
65                .is_positive(is_positive),
66                .easter_egg(easter_egg)
67         );
```
Código 68 Segunda parte del módulo `riscv_top`

Por último, en el Código 69 se muestra el tercer fragmento de la implementación del módulo top, donde se describen los procesos `initial` utilizadas para su verificación. En primer lugar, entre las líneas 6 a 72, se inicializan las diferentes variables necesarias para el correcto funcionamiento del procesador. Antes de comenzar con la simulación, en la línea 74, se carga directamente en la memoria de instrucciones el programa que se quiere simular. Este primer proceso `initial` se encarga, además, de generar el pulso de la señal *reset*, donde se inicializa la señal *reset* a uno y, tras un periodo de tiempo definido por la variable `CYCLE_TIME`, se vuelve a desactivar la señal de *reset*. Tras el proceso de *reset*, en las línea 80 y 81, se indica al procesador que comience la ejecución de instrucciones asignando los valores `8’hFE` y `8’hFF` a la señal `instr_i`, tal y como se ha explicado anteriormente.

Por último, entre las líneas 84 a 89, se describe el segundo proceso `initial`. En este proceso se asocia los agentes creados por el componente *Environment* a sus correspondientes interfaces virtuales, haciendo uso del método `set` que proporciona la base de datos de UVM `uvm_config_db` y, seguidamente, en la línea 88, se ejecuta el método `run_test()`. De esta forma se inicia el mecanismo de fases de UVM, lo que deriva en la ejecución del test seleccionado, en este caso particular, el test base `riscv_base_test`.

Además, con el fin de detener la simulación, se ha implementado un tercer proceso. Este proceso, descrito entre las líneas 91 a 96, comprueba en cada ciclo de reloj el valor de la señal `riscv_top.inst`. En caso de que esta señal tome el valor `32'h00002003`, se detendrá la simulación tras una serie de ciclos mediante la macro `$finish`. Por este motivo, los programas

que se inserten en la memoria de instrucciones deben tener una última instrucción codificada como `32'h00002003`.

```
68          initial begin
69          DataOrReg = 1;
70          address = 5'd8;
71          vout_addr = 2'b11;
72          instr_i = 8'h00;
73
74          $readmemh("../dat/inst32hex.covergr0.txt", Instruction_Memory.memory);
75          clk = 1;
76          rst = 0;
77          rst = 1;
78          #(`CYCLE_TIME)
79          rst = 0;
80          @(posedge clk) instr_i <= 8'hFE;
81          @(posedge clk) instr_i <= 8'hFF;
82          end
83
84          initial begin
85  uvm_config_db#(virtual inst_if)::set(uvm_root::get(),"*.iagent.*","i_intf", ivif);
86  uvm_config_db#(virtual data_if)::set(uvm_root::get(),"*.dagent.*","data_intf", dvif);
87  uvm_config_db#(virtual inst_if)::set(uvm_root::get(),"*.imonitor.*","i_intf", ivif);
88          run_test("riscv_base_test");
89          end
90
91          always@(posedge clk) begin
92                  if(riscv_top.inst == 32'h00002003) begin
93                          repeat(20) @(posedge clk);
94                          $finish;
95                  end
96          end
97  endmodule
```

Código 69 Tercera parte del módulo `riscv_top`

5.2.8. Componente Test

Como se ha explicado, el entorno de verificación UVM está compuesto por dos agentes pasivos, por lo que no se hará uso de componentes *sequencers*, ni *drivers*. Sin embargo, la implementación del componente test sigue siendo necesario y útil, aunque su papel puede ser menos complejo que en un entorno con agentes activos. Dentro de sus funciones sigue estando, a pesar de contar únicamente con agentes pasivos, la inicialización del sistema y la configuración de los distintos componentes del entorno como los monitores, *subscriber* y *scoreboard*. Además, este componente incluye la conexión de las interfaces y la configuración de cualquier parámetro necesario para la simulación. La metodología UVM aconseja implementar un test base y, a partir de este, todos aquellos test de mayor complejidad para la verificación del diseño. El componente base test es quien maneja el control del flujo de verificación, determinando cuándo se inicia y se detiene la simulación.

Para este caso en particular, se ha implementado el componente test denominado `riscv_base_test` que hereda de la clase `uvm_test`, tal y como se muestra en la línea 1 del Código 70. Esta clase contiene los métodos y fases necesarias para ejecutar el entorno de

verificación correctamente. En la línea 2, se registra el componente en el mecanismo de *Factory.* En las líneas 4 y 5 se declaran las variables `riscv_env` y `uvm_table_printer` respectivamente. Posteriormente, entre las líneas 7 a 9 se define el constructor de la clase base `uvm_test`. Entre las líneas 11 a 16 se ejecuta la fase `build_phase`. En la línea 12 de esta fase, y mediante la ejecución de la macro `super` se asegura la herencia de todos los métodos de la clase base. Finalmente, en la línea 13, se crea el componente *environment* utilizando el método `create` de la fábrica. Las líneas 14 y 15, como se explicó anteriormente, se configura el objeto `printer` para imprimir por pantalla el esquema jerárquico del entorno formado por los distintos componentes.

La fase de `end_of_elaboration_phase` se define entre las líneas 18 a 21 y se ejecuta simplemente para llamar al método `sprint` que muestre la jerarquía del entorno previamente creado. Por último, en la línea 23, se define la fase de ejecución `run_phase`, estableciendo el tiempo de drenaje en la línea 24.

```
1  class riscv_base_test extends uvm_test;
2          `uvm_component_utils(riscv_base_test)
3
4          riscv_env env;
5          uvm_table_printer printer;
6
7          function new(string name, uvm_component parent);
8                  super.new(name, parent);
9          endfunction: new
10
11         function void build_phase(uvm_phase phase);
12                 super.build_phase(phase);
13                 env = riscv_env::type_id::create("env", this);
14                 printer = new();
15                 printer.knobs.depth = 5;
16         endfunction:build_phase
17
18         virtual function void end_of_elaboration_phase(uvm_phase phase);
19         `uvm_info(get_type_name(), $sformatf("Printing the test topology
20 :\n%s", this.sprint(printer)), UVM_LOW)
21         endfunction: end_of_elaboration_phase
22
23         virtual task run_phase(uvm_phase phase);
24                 phase.phase_done.set_drain_time(this, 50000000);
25                 $display("test!! run_phase ");
26         endtask: run_phase
27
28 endclass: riscv_base_test
```

Código 70 Componente Test

5.3. Resultados del Test Cobertura Fibonacci

A continuación, se mostrarán los resultados de cobertura obtenidos para el caso particular del programa Fibonacci, aunque es importante mencionar, que, si bien este test es de utilidad para realizar un test funcional mediano, debido a su limitado número de instrucciones no producirá unos reportes de resultados de cobertura muy extensos.

En la Figura 63 se muestra los distintos porcentajes de cobertura de cada grupo de cobertura. De todos los tipos, son las instrucciones tipo-R, con un 40,69%, las que menor porcentaje de cobertura han obtenido tras la simulación del test Fibonacci. Esto es lógico sabiendo además que de las 24 instrucciones totales que forman el programa Fibonacci, tan solo 4 son del tipo-R.

Name	Coverage	Goal	% of Goal	Status	Included
/riscv_pkg/subscriber	49.43%				
TYPE cg_R	40.69%	100	40.69%		✓
TYPE cg_I	61.32%	100	61.32%		✓
TYPE cg_S	43.47%	100	43.47%		✓
TYPE cg_B	52.22%	100	52.22%		✓

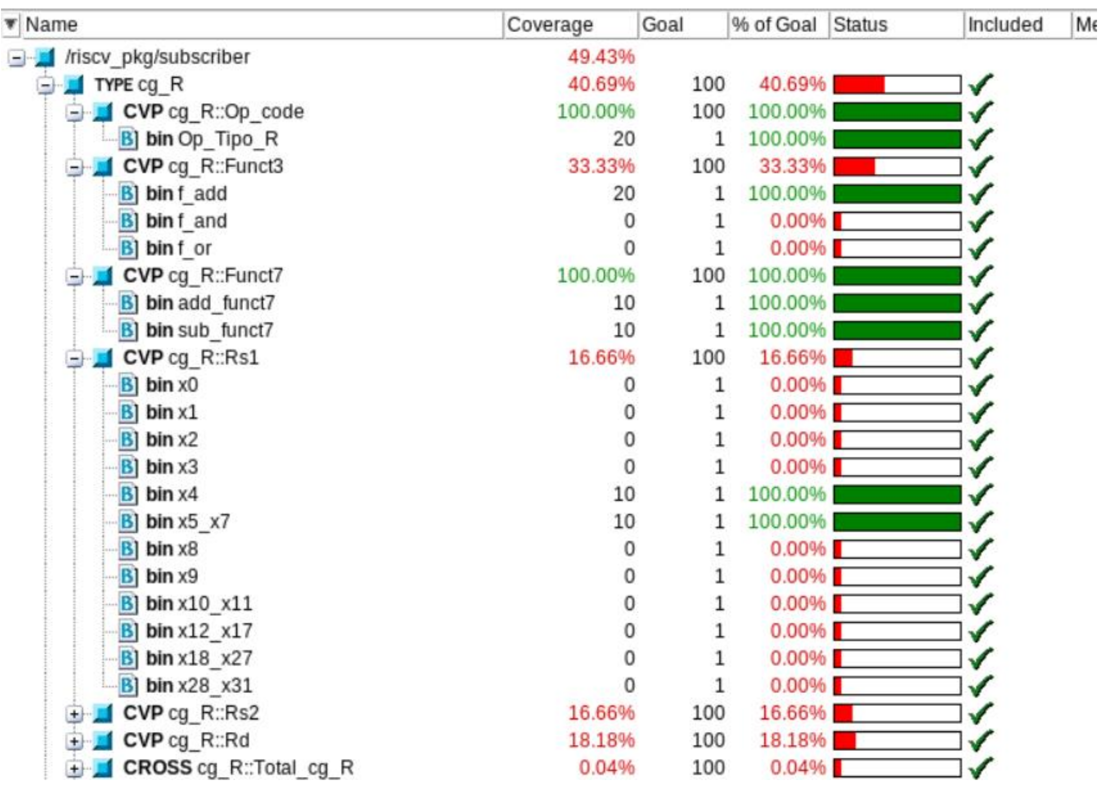
Figura 63 Porcentaje de cobertura de cada *covergroup* del Test Fibonacci

Ante estos resultados es importante conocer la información más detallada acerca de cuáles son los *coverpoint* que más han aportado en el porcentaje de cobertura, así como qué *bin* han sido cubiertos. De esta manera, es posible realizar un nuevo test que compruebe los casos de menor cobertura conociendo más en detalle cuáles son estos casos. En la Figura 64 se muestra de manera desglosada el caso particular de la cobertura para instrucciones Tipo-R.

Se comprueba, en primer lugar que el *coverpoint* `Op_Code` tiene un total del 100% de la cobertura; el *coverpoint* `Funct3` que representa en este caso las tres posibles operaciones tiene un 33.33%, ya que en el programa Fibonacci solo se hace uso de una de las operaciones, concretamente las operaciones `add`; se observa además que los *coverpoint* con menor cobertura son los relacionados a los registros. Esto es debido a una cuestión meramente matemática, ya que existe un número mayor de registros que, por ejemplo, tipos de operaciones. En este caso particular, el *coverpoint* `Rs1` tiene un total de 16.66% de cobertura, debido a que de los 12 posibles grupos de registros, se han utilizado como registros fuentes 1, los grupos de registros `x4` y `x5_x7` presentes en el programa Fibonacci.

Sin embargo, el dato más relevante acerca de la información del grado de cobertura para las instrucciones Tipo-R y el resto de las instrucciones es el dato mostrado por el `cross` que, como se ha explicado, indica el grado de cobertura total teniendo en cuenta todas las posibles

combinaciones que se pueden dar en la codificación de la instrucción, asignando un peso para cada uno de los *coverpoint*. En este caso particular, el nivel de cobertura es de 0.04%. Este resultado indica un porcentaje muy bajo pero que concuerda debido a la inmensa cantidad de posibilidades siendo el test un simple programa Fibonacci. Este dato de porcentaje es lógico teniendo en cuenta que si se multiplica los diferentes pesos relativos: $Posibilidades = 1 * 4 * 12 * 12 * 11 = 6.336$, esto quiere decir que para obtener el 100% de cobertura para las instrucciones Tipo-R se necesita de las 6.336 instrucciones posibles y, que para el test Fibonacci, se ha cubierto el 0.04% de las 6.336 diferentes instrucciones.

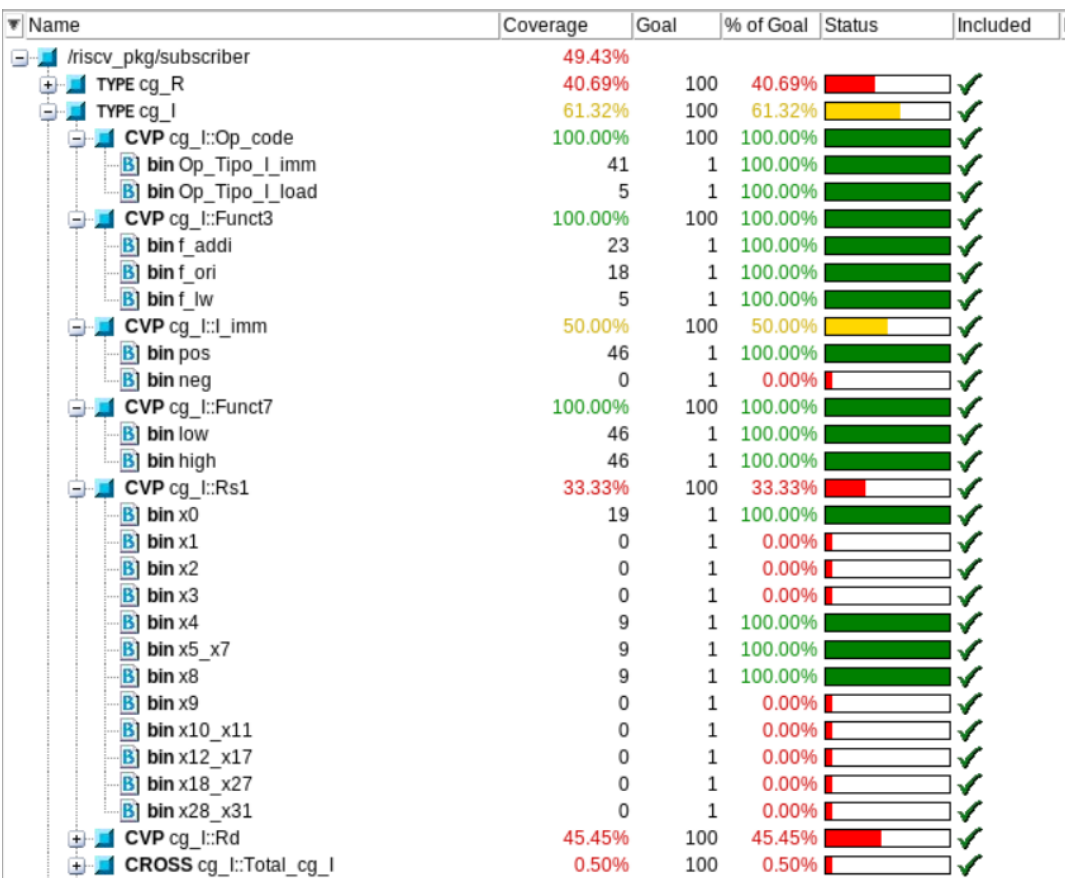


Name	Coverage	Goal	% of Goal	Status	Included	Me
/riscv_pkg/subscriber	49.43%					
TYPE cg_R	40.69%	100	40.69%		✓	
CVP cg_R::Op_code	100.00%	100	100.00%		✓	
bin Op_Tipo_R	20	1	100.00%		✓	
CVP cg_R::Funct3	33.33%	100	33.33%		✓	
bin f_add	20	1	100.00%		✓	
bin f_and	0	1	0.00%		✓	
bin f_or	0	1	0.00%		✓	
CVP cg_R::Funct7	100.00%	100	100.00%		✓	
bin add_funct7	10	1	100.00%		✓	
bin sub_funct7	10	1	100.00%		✓	
CVP cg_R::Rs1	16.66%	100	16.66%		✓	
bin x0	0	1	0.00%		✓	
bin x1	0	1	0.00%		✓	
bin x2	0	1	0.00%		✓	
bin x3	0	1	0.00%		✓	
bin x4	10	1	100.00%		✓	
bin x5_x7	10	1	100.00%		✓	
bin x8	0	1	0.00%		✓	
bin x9	0	1	0.00%		✓	
bin x10_x11	0	1	0.00%		✓	
bin x12_x17	0	1	0.00%		✓	
bin x18_x27	0	1	0.00%		✓	
bin x28_x31	0	1	0.00%		✓	
CVP cg_R::Rs2	16.66%	100	16.66%		✓	
CVP cg_R::Rd	18.18%	100	18.18%		✓	
CROSS cg_R::Total_cg_R	0.04%	100	0.04%		✓	

Figura 64 Resultados de cobertura para instrucciones Tipo-R - test Fibonacci

De manera similar, en la Figura 65 se muestra la información de grado de cobertura de forma desglosada para el caso de las instrucciones Tipo-I. Se observa que el *coverpoint* `Op_code` obtiene el 100% de cobertura ya que en el programa Fibonacci se detecta las instrucciones con dato inmediato e instrucciones *load,* hecho reflejado en los *bins* `Op_Tipo_I_imm` y `Op_Tipo_I_load` respectivamente; el *coverpoint* `Funct3` también obtiene el 100% de cobertura ya que en el programa se utilizan las diferentes operaciones `addi`, `ori` y `lw`; el *coverpoint* `I_imm`, sin embargo, obtiene el 50% de cobertura, esto es debido a que el programa Fibonacci, hace uso de datos inmediatos positivos, en ningún momento negativos como así lo refleja el bin `neg` con un 0% de cobertura, que sumado al *bin* `pos`, con un 100%, de la media para el *coverpoint* `I_imm` del 50%.

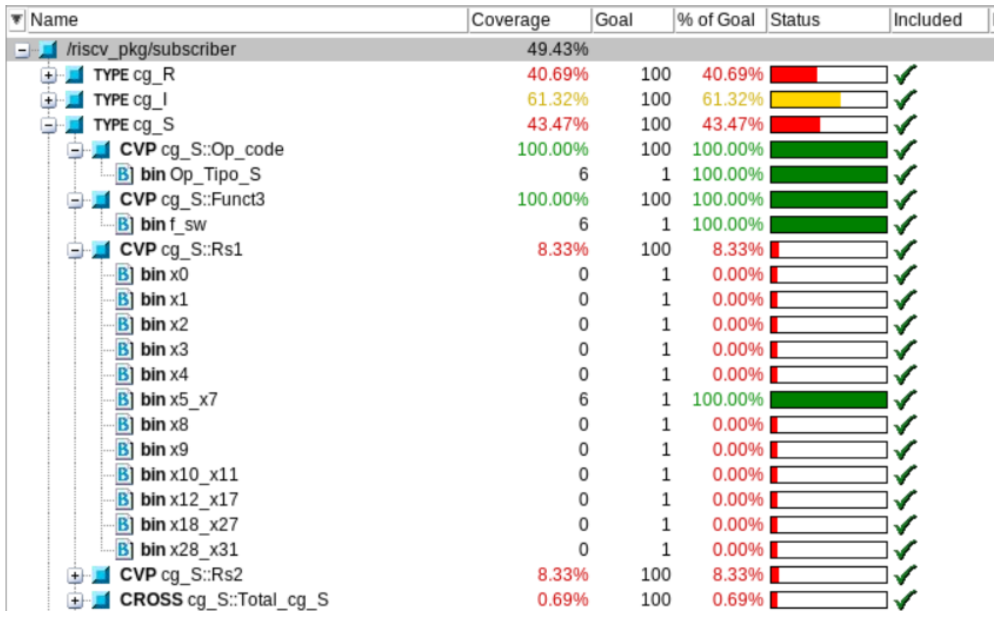
Nuevamente, los *coverpoint* con menor porcentaje de cobertura son aquellos que más posibilidades tienen, que son los definidos para los registros fuente y destino, cuyos porcentajes son respectivamente 33.33% y 45.45%. Finalmente, el `cross` para el caso de instrucciones Tipo-I en el test Fibonacci se ha cubierto un 0.50%.



Name	Coverage	Goal	% of Goal	Status	Included
/riscv_pkg/subscriber	49.43%				
TYPE cg_R	40.69%	100	40.69%		✓
TYPE cg_I	61.32%	100	61.32%		✓
CVP cg_I::Op_code	100.00%	100	100.00%		✓
bin Op_Tipo_I_imm	41	1	100.00%		✓
bin Op_Tipo_I_load	5	1	100.00%		✓
CVP cg_I::Funct3	100.00%	100	100.00%		✓
bin f_addi	23	1	100.00%		✓
bin f_ori	18	1	100.00%		✓
bin f_lw	5	1	100.00%		✓
CVP cg_I::I_imm	50.00%	100	50.00%		✓
bin pos	46	1	100.00%		✓
bin neg	0	1	0.00%		✓
CVP cg_I::Funct7	100.00%	100	100.00%		✓
bin low	46	1	100.00%		✓
bin high	46	1	100.00%		✓
CVP cg_I::Rs1	33.33%	100	33.33%		✓
bin x0	19	1	100.00%		✓
bin x1	0	1	0.00%		✓
bin x2	0	1	0.00%		✓
bin x3	0	1	0.00%		✓
bin x4	9	1	100.00%		✓
bin x5_x7	9	1	100.00%		✓
bin x8	9	1	100.00%		✓
bin x9	0	1	0.00%		✓
bin x10_x11	0	1	0.00%		✓
bin x12_x17	0	1	0.00%		✓
bin x18_x27	0	1	0.00%		✓
bin x28_x31	0	1	0.00%		✓
CVP cg_I::Rd	45.45%	100	45.45%		✓
CROSS cg_I::Total_cg_I	0.50%	100	0.50%		✓

Figura 65 Resultados de cobertura para instrucciones Tipo-I - test Fibonacci

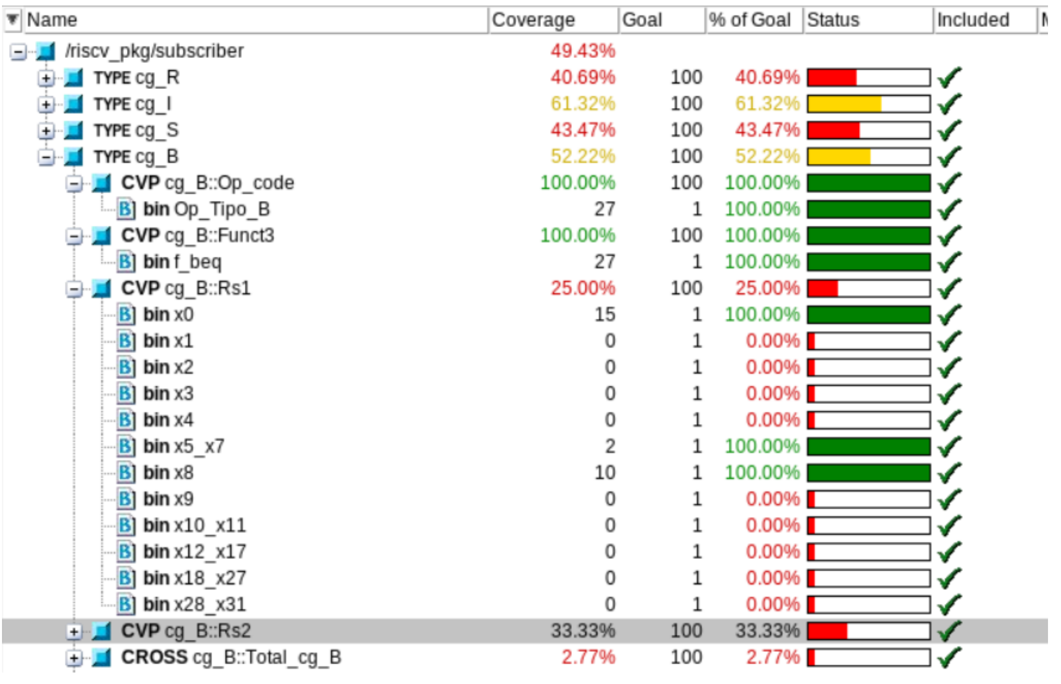
En la Figura 66 se muestra la información de grado de cobertura de forma desglosada para el caso de las instrucciones Tipo-S. En este caso, el *coverpoint* `Op_code` es del 100% ya que en el programa se hace uso de instrucciones *store* y el *coverpoint* `Funct3`, al existir una única operación, si la codificación de la instrucción es correcta, este *coverpoint* debería ser siempre igual al anterior; los *coverpoint* `Rs1` y `Rs2` en este caso son más bajos, concretamente del 8.33%, ya que en el test Fibonacci tan solo existe una única instrucción de este tipo, reduciendo así mucho el grado de cobertura para este *coverpoint*. Sin embargo el `cross` indica un total de 0.69%, siendo el grado de cobertura mayor con respecto a las instrucciones anteriores. Esto es debido al decremento de posibilidades. Por ejemplo, para las instrucciones Tipo-R existe un total de 6.336 posibilidades mientras que para las instrucciones Tipo-S las posibilidades son 144, por lo tanto, a pesar de que el programa Fibonacci ejecuta menos instrucciones Tipo-S, el grado de cobertura es mayor para este tipo de instrucciones.



Name	Coverage	Goal	% of Goal	Status	Included
/riscv_pkg/subscriber	49.43%				
TYPE cg_R	40.69%	100	40.69%		✓
TYPE cg_I	61.32%	100	61.32%		✓
TYPE cg_S	43.47%	100	43.47%		✓
CVP cg_S::Op_code	100.00%	100	100.00%		✓
bin Op_Tipo_S	6	1	100.00%		✓
CVP cg_S::Funct3	100.00%	100	100.00%		✓
bin f_sw	6	1	100.00%		✓
CVP cg_S::Rs1	8.33%	100	8.33%		✓
bin x0	0	1	0.00%		✓
bin x1	0	1	0.00%		✓
bin x2	0	1	0.00%		✓
bin x3	0	1	0.00%		✓
bin x4	0	1	0.00%		✓
bin x5_x7	6	1	100.00%		✓
bin x8	0	1	0.00%		✓
bin x9	0	1	0.00%		✓
bin x10_x11	0	1	0.00%		✓
bin x12_x17	0	1	0.00%		✓
bin x18_x27	0	1	0.00%		✓
bin x28_x31	0	1	0.00%		✓
CVP cg_S::Rs2	8.33%	100	8.33%		✓
CROSS cg_S::Total_cg_S	0.69%	100	0.69%		✓

Figura 66 Resultados de cobertura para instrucciones Tipo-S - test Fibonacci

Por último, en la Figura 67 se muestra la información de grado de cobertura de forma desglosada para el caso de las instrucciones Tipo-B. En este caso ocurre exactamente lo mismo que para las instrucciones Tipo-S, donde el hecho de que el *coverpoint* para los registros destino desaparece decrementa drásticamente el número de posibilidades. Esto implica que el grado de cobertura total sea en este caso el mayor de todas las instrucciones, debido también a que en el test Fibonacci se utiliza hasta 7 instrucciones de este tipo. Además estas instrucciones están involucradass en los diferentes bucles del programa repitiendo así la misma instrucción. Por estas razones, el `cross` para las instrucciones Tipo-B es del 2.77%.



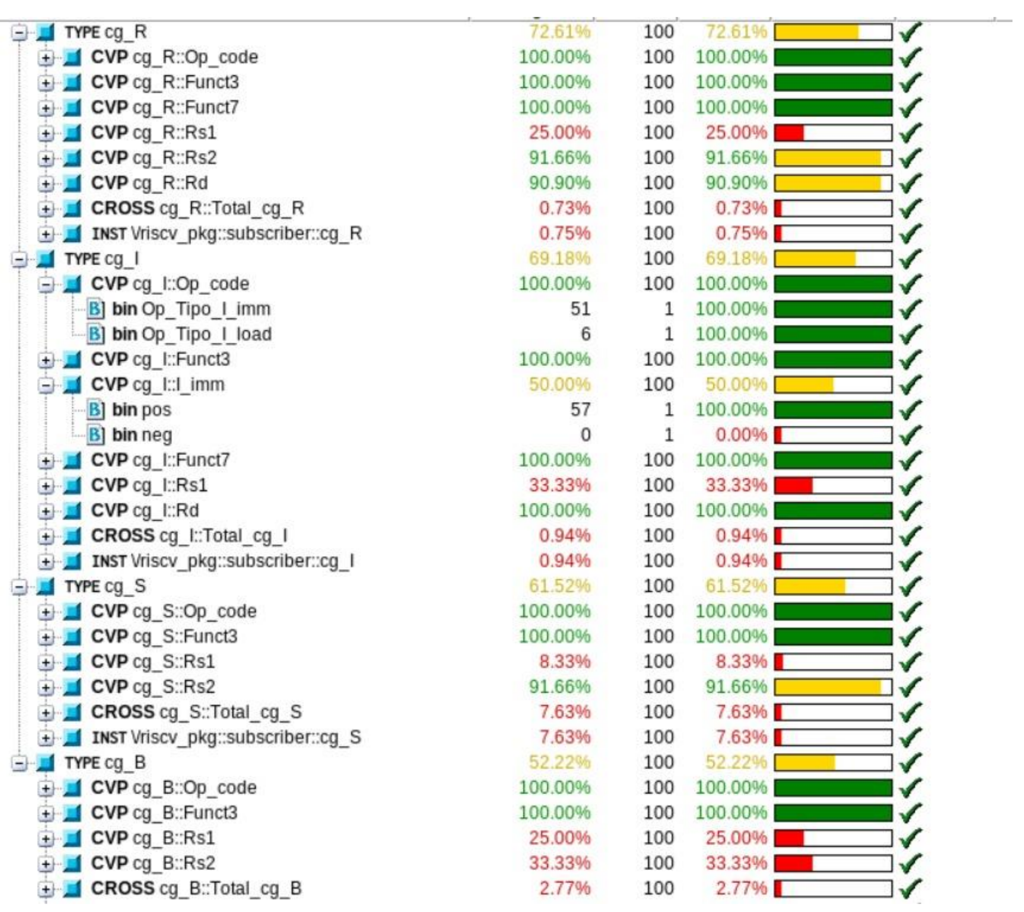
Name	Coverage	Goal	% of Goal	Status	Included
/riscv_pkg/subscriber	49.43%				
TYPE cg_R	40.69%	100	40.69%		✓
TYPE cg_I	61.32%	100	61.32%		✓
TYPE cg_S	43.47%	100	43.47%		✓
TYPE cg_B	52.22%	100	52.22%		✓
CVP cg_B::Op_code	100.00%	100	100.00%		✓
bin Op_Tipo_B	27	1	100.00%		✓
CVP cg_B::Funct3	100.00%	100	100.00%		✓
bin f_beq	27	1	100.00%		✓
CVP cg_B::Rs1	25.00%	100	25.00%		✓
bin x0	15	1	100.00%		✓
bin x1	0	1	0.00%		✓
bin x2	0	1	0.00%		✓
bin x3	0	1	0.00%		✓
bin x4	0	1	0.00%		✓
bin x5_x7	2	1	100.00%		✓
bin x8	10	1	100.00%		✓
bin x9	0	1	0.00%		✓
bin x10_x11	0	1	0.00%		✓
bin x12_x17	0	1	0.00%		✓
bin x18_x27	0	1	0.00%		✓
bin x28_x31	0	1	0.00%		✓
CVP cg_B::Rs2	33.33%	100	33.33%		✓
CROSS cg_B::Total_cg_B	2.77%	100	2.77%		✓

Figura 67 Resultados de cobertura para instrucciones Tipo-B - test Fibonacci

5.4. Resultados Test de Cobertura Simple

Para finalizar, en este apartado se ha decidido comprobar el incremento de la cobertura a través de un test que incluye una combinación elevada de instrucciones y registros, este test consiste en un programa cuya finalidad no tiene otra que aumentar la cobertura, por lo que no es de interés explicar el código ensamblador de dicho programa.

En la Figura 68 se muestra los resultados sobre el grado de cobertura para las diferentes instrucciones tras simular el test mencionado. Se observa como los diferentes *covergroups* han aumentado sus porcentajes de cobertura significativamente. Por el ejemplo, la cobertura para las instrucciones Tipo-R ha pasado de un 40.69% a un 72.61%, siendo los mayores responsables de este aumento los *coverpoint* `Funct3` , `Rs2` y `Rd`, que han pasado del 33% al 100%, del 16% al 91% y del 18% al 90% respectivamente cada uno.



TYPE cg_R	72.61%	100	72.61%
CVP cg_R::Op_code	100.00%	100	100.00%
CVP cg_R::Funct3	100.00%	100	100.00%
CVP cg_R::Funct7	100.00%	100	100.00%
CVP cg_R::Rs1	25.00%	100	25.00%
CVP cg_R::Rs2	91.66%	100	91.66%
CVP cg_R::Rd	90.90%	100	90.90%
CROSS cg_R::Total_cg_R	0.73%	100	0.73%
INST Vriscv_pkg::subscriber::cg_R	0.75%	100	0.75%
TYPE cg_I	69.18%	100	69.18%
CVP cg_I::Op_code	100.00%	100	100.00%
bin Op_Tipo_I_imm	51	1	100.00%
bin Op_Tipo_I_load	6	1	100.00%
CVP cg_I::Funct3	100.00%	100	100.00%
CVP cg_I::I_imm	50.00%	100	50.00%
bin pos	57	1	100.00%
bin neg	0	1	0.00%
CVP cg_I::Funct7	100.00%	100	100.00%
CVP cg_I::Rs1	33.33%	100	33.33%
CVP cg_I::Rd	100.00%	100	100.00%
CROSS cg_I::Total_cg_I	0.94%	100	0.94%
INST Vriscv_pkg::subscriber::cg_I	0.94%	100	0.94%
TYPE cg_S	61.52%	100	61.52%
CVP cg_S::Op_code	100.00%	100	100.00%
CVP cg_S::Funct3	100.00%	100	100.00%
CVP cg_S::Rs1	8.33%	100	8.33%
CVP cg_S::Rs2	91.66%	100	91.66%
CROSS cg_S::Total_cg_S	7.63%	100	7.63%
INST Vriscv_pkg::subscriber::cg_S	7.63%	100	7.63%
TYPE cg_B	52.22%	100	52.22%
CVP cg_B::Op_code	100.00%	100	100.00%
CVP cg_B::Funct3	100.00%	100	100.00%
CVP cg_B::Rs1	25.00%	100	25.00%
CVP cg_B::Rs2	33.33%	100	33.33%
CROSS cg_B::Total_cg_B	2.77%	100	2.77%

Figura 68 Resultados de cobertura para el test de cobertura simple

Los resultados obtenidos con estos test de cobertura muestran la validez del entorno creado, así como la descripción de grupos de cobertura (covergoups), de puntos de cobertura (coverpoints), y de rangos de valores (bins). Lejos de los resultados numéricos en sí mismo, en los que los porcentajes de cobertura son muy bajos, lo que sobresale es el hecho de que según se incremente el número de combinaciones en las instrucciones utilizadas, estos porcentajes se han visto incrementados en las cantidades esperadas.

Hay que indicar que en la actualidad existen varios proyectos que permiten la generación de instrucciones con propósito de cobertura. Uno de los que más proyección tiene es FORCE-RISCV [29]. FORCE-RISCV es un generador de secuencias de instrucciones (ISG) para la arquitectura del conjunto de instrucciones RISC-V que puede ser utilizado para generar pruebas para la verificación de diseño de procesadores RISC-V. Este utiliza aleatorización para elegir instrucciones, registros, direcciones y datos para los tests. Por otro lado, existe un proyecto llamado RISCV-DV [30] que consiste en un conjunto de herramientas y entornos de verificación desarrollados por la compañía Google para la verificación de procesadores RISC-V aplicando la metodología UVM y ampliamente utilizado para la generación de *test* y secuencias dirigidos a procesadores basados en esta arquitectura.

Capítulo 6. Conclusiones y trabajos futuros

En este capítulo se expondrán las conclusiones obtenidas a lo largo del desarrollo del presente Trabajo Fin de Máster y sus posibles continuaciones o líneas futuras.

6.1. Conclusiones

Una vez finalizado el Trabajo Fin de Máster denominado “Diseño de un *testbench* UVM para la verificación de un procesador RISC-V”, se comprueba que se han cumplido los objetivos prefijados en el anteproyecto. El objetivo principal de este TFM consiste en el diseño y desarrollo de un *testbench* UVM usando como dispositivo a verificar un procesador con arquitectura RISC-V. Este procesador en primer lugar consistía en el procesador lowRISC. Que se trata del primer entorno de verificación UVM relacionado con RISC-V en la División de Diseño de Sistemas Integrados y que algunas de las herramientas utilizadas para la verificación del procesador lowRISC, como es el caso de RISCV-DV han sido imposible de instalar ya que, si bien son *OpenSource*, hacen uso de un simulador con soporte UVM y *SystemVerilog* sobre plataformas no disponibles en la división. Por tal motivo, se decidió optar por otro procesador más simple pero que fuera compatible con las características y funcionalidades fundamentales de la arquitectura RISC-V y optar por hacer uso de las *toolchains* de RISC-V (en este caso las proporcionadas por GNU) con el fin de obtener los códigos hexadecimales de los ejemplos a usar durante la verificación.

La verificación del comportamiento del módulo RISC-V, así como la simulación se llevó a cabo siguiendo un enfoque *white-blox,* debido a que fue necesario conocer en detalle las diferentes señales internas para interpretar el funcionamiento del procesador como el caso del manejo de las

dependencias de datos o a la hora de extraer las memorias de instrucciones y datos conociendo exactamente cuáles son los puertos de entrada y salida en todo momento.

Respecto a los objetivos parciales establecidos inicialmente, se considera haber logrado la totalidad de ellos. Los conocimientos adquiridos en el estudio de la metodología UVM han sido suficientes para el desarrollo de un entorno de verificación UVM basado en dos agentes pasivos responsables de ambas interfaces, de instrucción y de datos.

El procedimiento seguido en la realización de este Trabajo Fin de Máster ha sido transcrito a la propia redacción del presente documento memoria, siendo la secuencia definida por los capítulos similar a la seguida en el desarrollo real. De esta manera, en primer lugar, se llevó a cabo el estudio del lenguaje de programación *SystemVerilog* por ser la base principal que soporta la metodología UVM, elaborando un entorno de verificación completo con el fin de asentar los conocimientos sobre un módulo PIFO. El siguiente paso fue estudiar los aspectos generales de la metodología UVM y elaborar un nuevo entorno de verificación del módulo PIFO de manera que se conocieron las principales diferencias y ventajas que aporta UVM. Una vez finalizada esta etapa, el siguiente paso fue conocer en detalle el DUV a verificar en este trabajo.

Cabe destacar que en gran parte del tiempo destinado a la realización del TFM se invirtió en el estudio de la metodología, al tratarse de conocimientos complejos y a la vez muy abstractos, que solo tienen sentido al tener una concepción global de ellos.

Finalizada la etapa de estudio, se adaptó el IP a verificar para su implementación en un entorno de verificación UVM, así como distintos cambios dentro del diseño debido a que la finalidad del RISC-V a verificar es para su implementación física en ASIC y el objetivo de este TFM es comprobar el correcto funcionamiento. Finalmente, se elaboró el entorno de verificación UVM para el procesador RISC-V elegido y además se propuso incorporar el componente *subscriber* que permitió, tras elaborar un plan de verificación y de cobertura, obtener los resultados esperados.

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

6.2. Líneas Futuras

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

Con el *testbench* desarrollado, hacer uso de un procesador completo, tal como lowRISC, y desarrollar un generador de instrucciones y secuencias, tal vez en Python, que permita la generación de todo tipo de instrucciones y, lo más importante, de todo tipo de secuencia de instrucciones. Estas últimas deberían de contemplar los diferentes riesgos de un procesador segmentado, esto es, dependencias de datos y combinación de instrucciones de control.

Integrar un modelo de referencia en lenguaje de alto nivel que permita la verificación automática del sistema. Hoy en día es posible encontrar modelos de referencia *OpenSource* del procesador RISC-V. Debido a que el uso de estos modelos reduce la etapa de verificación a una simple comprobación de dos ficheros de salida, no se debe descartar la posibilidad de añadir una verificación híbrida en la que, además del modelo de referencia, el componente *Scoreboard* haga uso de un modelo interno en *SystemVerilog* que permita realizar comprobaciones intermedias.

Bibliografía

[1] "Part 8: The 2022 Wilson Research Group Functional Verification Study - Verification Horizons." Accessed: Feb. 10, 2024. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/

[2] "Part 10: The 2020 Wilson Research Group Functional Verification Study - Verification Horizons." Accessed: Feb. 10, 2024. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2021/01/20/part-10-the-2020-wilson-research-group-functional-verification-study/

[3] "Engineer - Consultant Jobs - ASIC Verification." Accessed: Feb. 10, 2024. [Online]. Available: https://www.chipright.com/consultant-jobs/asic-verification/

[4] "RISC-V International – RISC-V: The Open Standard RISC Instruction Set Architecture." Accessed: Feb. 10, 2024. [Online]. Available: https://riscv.org/

[5] "About RISC-V – RISC-V International." Accessed: Feb. 10, 2024. [Online]. Available: https://riscv.org/about/

[6] R. von Hanxleden, Institute of Electrical and Electronics Engineers, IEEE Council on Electronic Design Automation, and IFIP WG 10.5, *Proceedings of the 2020 Forum for Specification & Design Languages (FDL) : Kiel (Germany), 15-17 September 2020*.

[7] A. Waterman, “Design of the RISC-V Instruction Set Architecture,” 2016. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html

[8] “RISC-V: Rhea el primer microprocesador europeo fruto de EPI - ArchiTecnologia.” Accessed: Feb. 10, 2024. [Online]. Available: https://architecnologia.es/risc-v-el-nuevo-chip-europeo-rhea-y-primer-fruto-de-epi

[9] “El BSC e INTEL anuncian un laboratorio conjunto para el desarrollo de los supercomputadores del futuro a zettascale | BSC-CNS.” Accessed: Feb. 10, 2024. [Online]. Available: https://www.bsc.es/es/noticias/noticias-del-bsc/el-bsc-e-intel-anuncian-un-laboratorio-conjunto-para-el-desarrollo-de-los-supercomputadores-del

[10] “Zettascale computing - Wikipedia.” Accessed: Feb. 15, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Zettascale_computing

[11] “lowRISC: Collaborative open silicon engineering.” Accessed: Feb. 15, 2024. [Online]. Available: https://lowrisc.org/

[12] “PIFO_tb_simple - EDA Playground.” Accessed: Jun. 15, 2024. [Online]. Available: https://edaplayground.com/x/MJQM

[13] “Introduction To System Verilog | System Verilog Tutorial | System Verilog.” Accessed: May 07, 2024. [Online]. Available: http://www.asicguru.com/system-verilog/tutorial/introduction/1/

[14] “SystemVerilog TestBench.” Accessed: May 07, 2024. [Online]. Available: https://www.chipverify.com/systemverilog/systemverilog-simple-testbench

[15] “PIFO_tb_systemverilog2agentes - EDA Playground.” Accessed: Jun. 14, 2024. [Online]. Available: https://edaplayground.com/x/fEXw

[16] “SystemVerilog Race Condition Challenge Responses - Verification Horizons.” Accessed: Jun. 15, 2024. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2020/08/13/systemverilog-race-condition-challenge-responses/

[17] "OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL Software version: TLM 2.0.1 Document version: JA32," 2009.

[18] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design The Hardware/Software Interface: RISC-V Edition."

[19] A. Waterman, K. A. Asanovi´c, and J. Hauser, "The RISC-V Instruction Set Manual," 2021.

[20] Josep Roca, "RISC-V, especificaciones y características de la ISA más libre." Accessed: Apr. 30, 2022. [Online]. Available: https://hardzone.es/reportajes/que-es/risc-v/

[21] David Patterson and Andrew Waterman, "Elogios para La Guía Práctica de RISC-V."

[22] "emulsiV - Simulator for Virgule, a minimal processor based on the RISC-V architecture." Accessed: Aug. 06, 2022. [Online]. Available: http://tice.sea.eseo.fr/riscv/

[23] WikiChip, "Registers - RISC-V - WikiChip." Accessed: Apr. 30, 2022. [Online]. Available: https://en.wikichip.org/wiki/risc-v/registers

[24] "Tipos de modos de direccionamiento en computadoras." Accessed: Oct. 12, 2022. [Online]. Available: https://techlandia.com/tipos-modos-direccionamiento-computadoras-lista_548496/

[25] Isaac, "RISC-V: similitudes y diferencias con otras ISAs tipo RISC - ArchiTecnologia." Accessed: Apr. 30, 2022. [Online]. Available: https://architecnologia.es/risc-v-similitudes-y-diferencias-con-otras-isas-tipo-risc

[26] "GitHub - jasonlin316/RISC-V-CPU: A RISC-V 5-stage pipelined CPU that supports vector instructions. Tape-out with U18 technology." Accessed: Jun. 20, 2024. [Online]. Available: https://github.com/jasonlin316/RISC-V-CPU

[27] "GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC." Accessed: Jul. 16, 2024. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain

[28] “ABI – Hazlo bien – RISC-V es todo tuyo. – Diseño del sistema VLSI.” Accessed: Jul. 16, 2024. [Online]. Available: https://www.vlsisystemdesign.com/abi-get-this-one-right-risc-v-is-all-yours/

[29] “GitHub - openhwgroup/force-riscv: Instruction Set Generator initially contributed by Futurewei.” Accessed: Jul. 16, 2024. [Online]. Available: https://github.com/openhwgroup/force-riscv

[30] “GitHub - chipsalliance/riscv-dv: Random instruction generator for RISC-V processor verification.” Accessed: Jul. 16, 2024. [Online]. Available: https://github.com/chipsalliance/riscv-dv

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 8 y en la Tabla 9, respectivamente.

Tabla 8 Condiciones Hardware

Equipo/dispositivo	Modelo	Fabricante/Comerciante
Ordenador personal	LENOVO ideapad Y700: • CPU *Intel Core i7.* • GPU *GeForcce* GTX 960M. • 16 GB RAM. • 256 GB SSD. • 2 puertos USB 3.0 • 1 puerto USB.	Lenovo Group Limited.
Estación de trabajo	Acer Predator G3: • CPU Intel Core i7-4790. • 8 GB RAM. • 1 TB HDD. • 4 puertos USB	Hacer Inc.

Tabla 9 Condiciones Software

Equipo/dispositivo	Modelo	Fabricante/Comerciante
Sistema Operativo	*Microsoft Windows 10 Home 64 bits*	*Microsoft Corporation*
Microsoft Office	*2021*	*Microsoft Corporation*
Mendeley Desktop	*Versión 1.19.2*	*Elsevier*
Simulador y compilador Questasim	*Versión 2019.04*	*Mentor Graphics*, a *Siemens Business*
Librería UVM	*Versión 1.1d*	*Mentor Graphics*, a *Siemens Business*
Adobe Reader	*Versión 2023*	*Adobe Systems Inc.*

PRESUPUESTO

PRESUPUESTO

Para realizar el presente Trabajo Fin de Máster, ha sido imprescindible emplear diversos recursos de distinta naturaleza. Con base en estos recursos, es posible calcular la cantidad de cada uno según su tipo y su contribución al proyecto. Por ello, en este capítulo se especificarán todos los recursos materiales y humanos utilizados. Así pues, el presupuesto total para la ejecución de este Trabajo Fin de Grado se desglosará en los siguientes conceptos:

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

6.3. Recursos humanos

Para calcular el coste asociado a las horas de trabajo dedicadas al desarrollo del Trabajo Fin de Máster, si bien no existe una obligación, se toma como referencia y se emplea una ecuación recomendada por el Colegio Oficial de Ingenieros de Telecomunicación (COIT). Esta ecuación calcula los honorarios en función de las horas trabajadas, tanto durante como fuera de la jornada laboral convencional. Por lo tanto, la fórmula para calcular los costes de los Recursos Humanos (RRHH) es la siguiente:

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

Donde las incógnitas corresponden a los siguientes conceptos:

- El parámetro **Hn** define el número de horas trabajadas dentro de la jornada laboral convencional.
- El término **He** hace referencia a las horas especiales o extras realizadas fuera de la jornada laboral convencional.
- La incógnita **Ct** representa el factor de corrección que se aplica en función de la totalidad de horas invertidas en el proyecto.

Los posibles valores del factor de corrección **Ct** se encuentran en la Tabla 10, según el procedimiento de cálculo establecido por el COIT:

Tabla 10 Factor de corrección según horas trabajadas

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

De acuerdo con el Plan de Estudios del Máster en Ingeniería en Tecnologías de la Telecomunicación, las horas destinadas a la asignatura Trabajo Fin de Máster son de 450 horas (18 ECTS). Por esta razón, en el caso particular de este TFM, el factor de corrección será de 0,55. Así, el resultado de la ecuación mencionada anteriormente es el siguiente:

$$Honorarios(€) = 0{,}55 * (74{,}88 * 450 + 96{,}72 * 0) = 18.532{,}8€$$

El coste asociado a las horas de trabajo dedicadas durante los últimos meses asciende a dieciocho mil quinientos treinta y dos euros con ochenta céntimos (18.532,8 €). Cabe destacar que a dicha cantidad no se le han aplicado impuestos ni retenciones.

6.4. Recursos Materiales

Tal como se mencionó en el Pliego de Condiciones, se han empleado herramientas de hardware y software durante la realización de este Trabajo Fin de Máster. Al calcular el coste de cada uno de estos recursos, se relaciona su coste total con su tiempo útil. Para realizar este cálculo, se utilizará la siguiente fórmula:

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

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

6.5. Recursos *hardware*

Es fundamental considerar la amortización de los recursos de hardware que se han utilizado a lo largo del desarrollo de este Trabajo Fin de Máster. En la Tabla 11 se presentan los resultados del coste total de los recursos de hardware.

Tabla 11 Coste total de los recursos hardware

Equipo/dispositivo	Valor de adquisición	Amortización	Coste mensual	Tiempo de uso	Importe
Ordenador personal – Lenovo ideapad Y700	700 €	36 meses	19,4 €	6 meses	116,4 €
Estación de trabajo - Hacer predator	950 €	36 meses	26,38 €	6 meses	158,28 €
Coste total					274,68 €

Como se visualiza en la tabla adjuntada, el coste total de los recursos *hardware* es de doscientos setenta y cuatro euros con sesenta y ocho céntimos (274,68 €).

6.6. Recursos *software*

En esta sección se calcula el coste total de los recursos de software utilizados durante el Trabajo Fin de Máster. Es importante mencionar que muchas de estas herramientas han sido proporcionadas de manera gratuita por la Universidad de Las Palmas de Gran Canaria (ULPGC). En la Tabla 12 se especifican los diferentes costes de todos estos recursos de software.

Tabla 12 Amortización de los recursos Software

Recurso	Valor de adquisición	Coste mensual	Tiempo de uso	Importe
Sistema Operativo Microsoft Window 10 Home – 64 bits	145 €	4,03 €	4 meses	16,12 €
Microsoft Office	0,00 € (Licencia ULPGC)	0,00 €	4 meses	0,00 €
Mendeley Desktop	0,00 € (software libre)	0,00 €	4 meses	0,00 €
Simulador y compilador Questasim	2.000,00 €	166,66 €	4 meses	666,64 €
Librería UVM	0,00 € (librería de libre distribución)	0,00 €	4 meses	0,00 €
Adobe Reader	0,00 € (software libre)	0,00 €	4 meses	0,00 €
Coste total:				682,76 €

Por lo tanto, el coste total de los recursos materiales asociados a los recursos *software* asciende hasta los seiscientos ochenta y dos con setenta y seis céntimos (682,76 €).

6.7. Material fungible

Los costes que derivan del material fungible utilizados en el presente Trabajo Fin de Máster se reduce al coste de impresión y encuadernación de la memoria del TFM. Estos costes ascienden a 20,00 €.

6.8. Redacción de la Memoria

También se debe considerar los gastos asociados a la redacción de la memoria de este Trabajo Fin de Máster a partir de la siguiente fórmula:

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

Donde los términos que forman la ecuación tienen el siguiente significado:

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

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

$$P = 18.532{,}8 + 274{,}68 + 682{,}76 = 19.490{,}24 \text{ €}.$$

El Colegio Oficial de Ingenieros de Telecomunicación, indica que el coeficiente de corrección Cn para valores de P inferiores a 30.000 € tendrá por valor la unidad. Por lo tanto, la ecuación de los gastos derivados a la redacción de la memoria se muestra a continuación.

$$R = 0{,}07 * 19.490{,}24 * 1 = 1.364{,}31 \text{ €}$$

Los gastos relativos a la redacción de la memoria del presente Trabajo Fin de Máster quedan en mil trescientos sesenta y cuatro euros con treinta y un céntimos (1.364,31 €).

6.9. Derechos de visado del COIT

El Colegio Oficial de Ingenieros de Telecomunicación, anualmente, proporciona valor a los costes derivados de los derechos de visado por la ejecución de proyectos técnicas de carácter general. De esta forma, para el año 2024 se calcula de la siguiente manera:

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

Donde los términos que forman la ecuación tienen el siguiente significado:

- El parámetro V es el coste final del visado del COIT.
- El término P1 hace referencia al presupuesto del proyecto.
- El parámetro C1 representa un coeficiente reductor en función del presupuesto.
- La incógnita P2 se refiere al presupuesto de ejecución material correspondiente a la obra civil.
- El término C2 es un coeficiente de corrección en función del presupuesto P2.

Para el presente Trabajo Fin de Máster, teniendo en cuenta su naturaleza, se determina que el coste derivado del término P_2 es igual a 0,00 €, por lo tanto, no se aplica el coeficiente C_2. Respecto al coeficiente anterior C_1, ya que el presupuesto no es superior a los 30.000,00 €, nuevamente toma el valor de unidad, dando como resultado de la ecuación:

$$V = 0{,}006 * 19.490{,}24 * 1 = 116{,}94\ €$$

Finalmente, el coste derivado a los derechos de visado del presupuesto es de ciento dieciséis euros con noventa y cuatro céntimos (116,94 €).

6.10. Gastos de tramitación y envío

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

6.11. Coste total del proyecto

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

Tabla 13 Coste total del Trabajo Fin de Máster

Concepto	Importe
Honorarios por tiempo empleado	18.532,8€
Recursos *hardware*	274,68 €
Recursos *software*	682,76 €
Material fungible	20,00 €
Costes de redacción	1.364,31 €
Derechos del visado del COIT	116,94 €
Costes de tramitación y envío	6 €
Subtotal	20.997,49 €
IGIC (7%)	1.469,82 €
Coste total	22.467,31 €

Finalmente, el coste para el presupuesto del Trabajo Fin de Máster "Diseño de un *testbench* UVM para la verificación de un procesador RISC-V" es de veintidós mil cuatrocientos sesenta y siete euros con treinta y un céntimos (22.467,31 €).

Fdo.: D. Francisco Duque El Ayachi

En Las Palmas de Gran Canarias a 18 de Julio de 2024