Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Creación de un ambiente de verificación usando UVM para un bus AXI4-Lite para una arquitectura RISC-V de 32 bits

Informe de Proyecto de Graduación para optar por el título de

Ingeniería en Electrónica con el grado académico de Licenciatura

Irene Beatriz Rivera Arrieta

Versión de 19 de junio de 2018

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

PROYECTO DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa de Proyecto de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del proyecto de graduación denominado Creación de un ambiente de verificación usando UVM para un bus AXI4-Lite para una arquitectura RISC-V de 32 bits, realizado por la señorita Irene Beatriz Rivera Arrieta y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Ing. Esteban Baradín Méndez

Profesor lector

Ing. Aníbal Ruíz Barquero

Profesor lector

Ing. Roberto Molina Robles

Profesor asesor

Cartago, 18 de junio de 2018

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Irene Beatriz Rivera Arrieta

Cartago, 19 de junio de 2018

Céd: 1-1527-0510

Resumen

Dentro del área de desarrollo de nuevas tecnologías existe la gran necesidad de poder demostrar que el diseño se encuentra sin fallas y completamente funcional. Esto es vital y de extrema importancia cuando estas nuevas tecnologías se encuentran enfocadas en el área de la medicina, como es el caso de este proyecto. El Tecnológico de Costa Rica en conjunto con otras universidades y empresas médicas se encuentran desarrollando un microprocesador para implantes médicos. Parte del diseño del microprocesador incluye un bus de datos AXI4-Lite, por lo que se requiere de verificación. El proceso de verificación es de larga duración y extensivo, este proyecto crea los inicios de la verificación para el AXI4-Lite. Se desarrollaron el ambiente de verificación, un modelo de referencia robusto y pruebas para dar un análisis al funcionamiento de dicho bus.

Palabras clave: AXI4-Lite, metodología de verificación universal (UVM), RISC-V, systemverilog, verificación.

A mi querido padre, que desde el cielo me protege y cuida.

Agradecimientos

Este proyecto no habría sido posible sin la ayuda de Dios. Sin el apoyo de mi madre, quien sin presionarme, me sostuvo todos estos años de estudio. A mis hermanos que entre los tres nos apoyamos y ayudamos, sin importar nuestras discusiones y desacuerdos. A mis compañeros y amigos, con los cuales he recorrido gran parte de la carrera, a todos aquellos a quienes les di el apoyo que necesitaban y sobretodo a aquellos quienes me lo brindaron. Al profesor Roberto Molina por brindarme la oportunidad de hacer este proyecto y asesorarme durante su transcurso. Por último a todos aquellos que no mencioné pero estuvieron presentes en mi camino de formación.

Irene Beatriz Rivera Arrieta

Cartago, 19 de junio de 2018

Índice general

Índice de figuras iii

Índice de tablas v

1 Introducción 1
- 1.1 Contexto . . . 1
- 1.2 Problema . . . 3
  - 1.2.1 Generalidades . . . 3
  - 1.2.2 Síntesis del Problema . . . 3
- 1.3 Objetivos . . . 3
  - 1.3.1 Objetivo General . . . 3
  - 1.3.2 Objetivos específicos . . . 3

2 Marco Teórico 5
- 2.1 SystemVerilog . . . 5
- 2.2 Metodología Universal de Verificación (UVM) . . . 5
  - 2.2.1 UVM Testbench . . . 7
  - 2.2.2 UVM Test . . . 7
  - 2.2.3 UVM Environment . . . 7
  - 2.2.4 UVM Agent . . . 8
  - 2.2.5 UVM Driver . . . 9
  - 2.2.6 UVM Monitor . . . 9
  - 2.2.7 UVM Sequencer . . . 9
  - 2.2.8 UVM Sequence . . . 10
  - 2.2.9 UVM Sequence Item . . . 10
  - 2.2.10 UVM Scoreboard . . . 10
  - 2.2.11 Reference Model . . . 11
  - 2.2.12 Cobertura funcional . . . 13
  - 2.2.13 Regresión . . . 14
- 2.3 RISC-V . . . 14
- 2.4 AXI4-Lite . . . 15

3 Ambiente de Verificación para el bus de datos AXI4-Lite 23
- 3.1 Creación del ambiente . . . 23
- 3.2 Modelo de Referencia . . . 24

3.3 Pruebas . . . 26
3.3.1 Prueba manual . . . 27
3.3.2 Pruebas de Lectura . . . 27
3.3.3 Pruebas de Escritura . . . 28
3.3.4 Pruebas de Lectura y Escritura . . . 29
3.3.5 Prueba aleatoria . . . 30
3.3.6 Cobertura . . . 30

4 Resultados y Análisis **32**
4.1 Familiarización con el lenguaje . . . 32
4.1.1 Ejemplo e implementación del ambiente . . . 32
4.1.2 Interfaces . . . 34
4.2 Modelo de Referencia . . . 36
4.3 Sistema de pruebas . . . 37
4.3.1 Driver . . . 37
4.3.2 Pruebas . . . 43

5 Conclusiones y Recomendaciones **47**
5.1 Conclusiones . . . 47
5.2 Recomendaciones . . . 47

Bibliografía **48**

A Hoja de Información **49**
A.1 Información del estudiante . . . 49
A.2 Información del proyecto . . . 49
A.3 Información de la Empresa . . . 50

Índice de figuras

1.1 Diagrama de flujo del proceso de diseño y fabricación de un microprocesador. 2

2.1 Diagrama estructural UVM. Componentes del Banco de pruebas tipo UVM . 6
2.2 Microarquitectura del UVM Agent [8] 8
2.3 Conexión secuencia - secuenciador - driver[7] 10
2.4 Diagrama de funcionalidad del modelo de referencia 11
2.5 Diagrama de funcionamiento del Golden Vectors 12
2.6 Diagrama de funcionamiento del ambiente basado en transacciones. 12
2.7 Flujo de cobertura[10] 13
2.8 Mapeor de memoria para los esclavos 18
2.9 Arquitectura de lectura [14] 19
2.10 Arquitectura de escritura [14] 20
2.11 Enlace de VALID antes del READY[3] 21
2.12 Enlace de READY antes del VALID[3] 21
2.13 Enlace de VALID y READY 22

3.1 Extracto del código de instanciación de las interfaces 24
3.2 Diagrama de flujo del modelo de referencia 25
3.3 Diagrama de flujo función de lectura del maestro 1 al esclavo 1. 26
3.4 Diagrama de flujo prueba de lectura 28
3.5 Diagrama de flujo prueba de lectura 29

4.1 Esquema del DUT de ejemplo[7] 33
4.2 Mensaje de error de sintaxis generado 33
4.3 Reporte de final de compilación 34
4.4 Diagrama de flujo original del modelo de referencia 36
4.5 Mensaje de coincidencia de datos 37
4.6 Mensaje de no coincidencia de datos 37
4.7 Diagrama de flujo de la primera etapa de activación en el driver 38
4.8 Diagrama de flujo de la segunda etapa del driver, para la función de lectura . 39
4.9 Diagrama de flujo de la segunda etapa del driver, para la función de escritura 40
4.10 Diagrama de flujo de la última etapa del driver, para la función de lectura . 41
4.11 Diagrama de flujo de la última etapa del driver, para la función de escritura 42
4.12 Forma de onda de las señales del DUT para la función de lectura 43
4.13 Forma de onda de las señales del DUT para la función random(10000000) . . 44

4.14 Forma de onda de las señales del DUT para la función random(10) 44
4.15 Forma de onda de las señales del DUT para la función de escritura 45
4.16 Mensaje de que la prueba falló 45
4.17 Mensaje de que la prueba fue exitosa 46

Índice de tablas

2.1 Configuración UVM Agent[8] 9
2.2 Interoperabilidad entre el AXI y el AXI4-Lite 15
2.3 Señales del protocolo AXI4-Lite[3] 16
2.4 Parejas de protocolo de enlace por canal de transmisión[3] 20

3.1 Abreviaciones 26

4.1 Porcentajes de cobertura a 1000 pruebas 46

Capítulo 1

Introducción

1.1 Contexto

La Escuela de Ingeniería en Electrónica del Instituto Tecnológico de Costa Rica (TEC), en conjunto con la Universidad Católica del Uruguay Dámaso Antonio Larragaña (UCUDAL), se encuentra desarrollando un microprocesador para uso en dispositivos médicos implantables, tales como lo son los marcapasos o los implantes cocleares. Como parte de este desarrollo se están explorando las diferentes técnicas para diseñar el microprocesador y garantizar su eficiencia que es a prueba de errores.

Al ser una aplicación para el área de la medicina, el microprocesador que se desarrolla debe de funcionar en óptimas condiciones y no hay lugar para errores o desperfectos. Esto debido a que aún el más mínimo error puede significar la vida o muerte del paciente. Dada esta exigencia se requieren varias etapas de pruebas durante el proceso de diseño y fabricación del microprocesador. Dichas etapas se pueden observar en la figura 1.1.

Del proceso mostrado en la figura 1.1 se destacan las dos áreas de pruebas importantes, las pruebas Pre-silicio y las pruebas Post-silicio. Donde cada término se refiere a las pruebas realizadas antes y después de la fabricación del chip, respectivamente.

La primera se refiere a probar todas las posibles combinaciones de comandos e instrucciones en el microprocesador y así determinar posibles fallas, errores e incongruencias que puedan existir. La segunda área se encarga de revisar y determinar posibles fallas que hayan ocurrido durante la fabricación o que hayan sido pasadas por alto durante la verificación, esto con el fin de que el producto no salga al consumidor con errores y deba ser recogido, pues esto implica un alto costo y conlleva no sólo a grandes pérdidas, sino que puede impactar de manera negativa la imagen del fabricante.

La primera área, como se dijo anteriormente, es la encargada de encontrar la mayor cantidad de errores en el diseño antes de que éste sea enviado a fabricar, y lleva por nombre verificación. Existen tres tipos de verificación, las cuales son verificación temporal, verificación funcional y la verificación formal. Cada una de ellas examina el diseño desde diferentes enfoques.

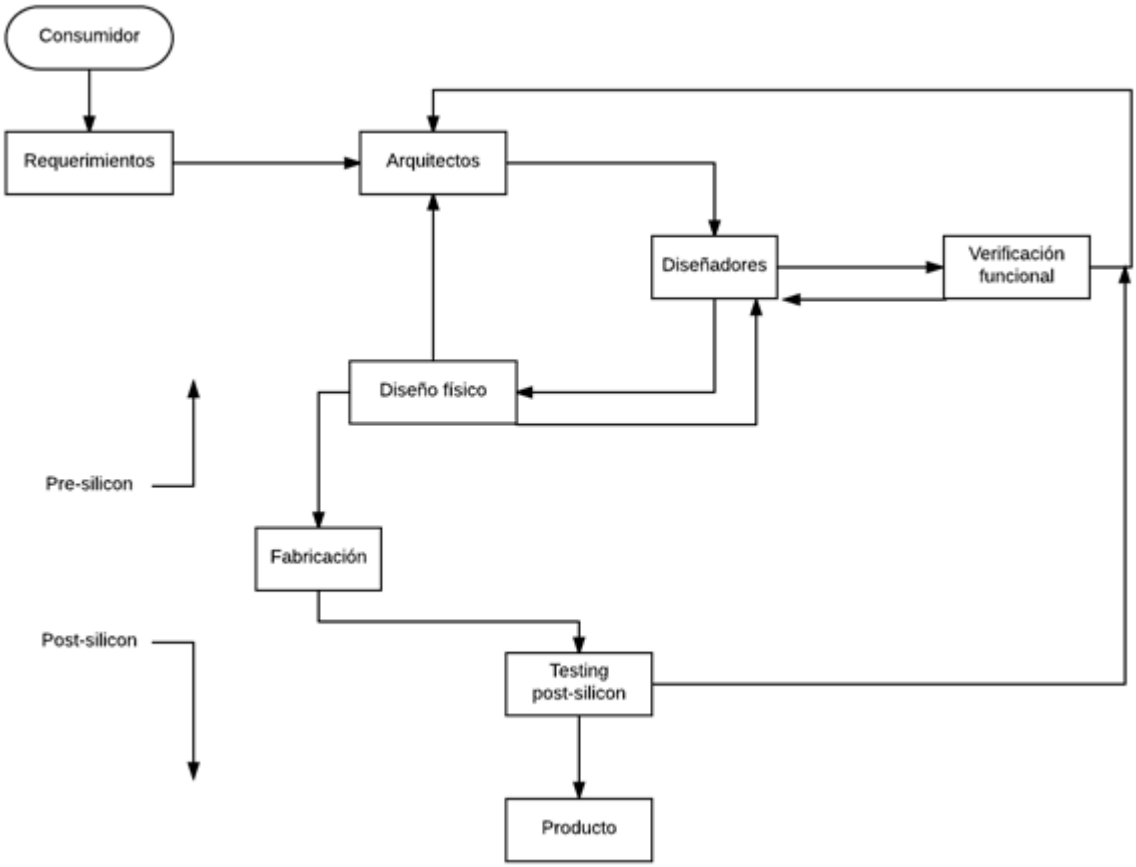

Figura 1.1: Diagrama de flujo del proceso de diseño y fabricación de un microprocesador.

- Verificación temporal: esta se encarga de revisar que el diseño cumple los tiempos de reloj y el área que el arquitecto indicó.
- Verificación funcional: este tipo es el encargado de que el diseño cumpla con todas las funciones que el arquitecto especificó.
- Verificación formal: este se enfoca en demostrar que el diseño es correcto pero utilizando una revisión con un trasfondo más matemático.

Parte del diseño del microprocesador que se encuentra en desarrollo corresponde a la creación de una interfaz que conecte el núcleo con los periféricos. Para esto fue escogido un bus de datos de protocolo AXI4-Lite. A este bus de datos se le realizó la verificación de tipo funcional, debido a que es de prioridad más alta obtener un diseño completamente funcional. Por lo tanto, existe la necesidad de crear un ambiente y una serie de pruebas que estresen este bus con el fin de determinar su correcto funcionamiento y eliminar posibles errores que conlleven a un daño en los módulos conectados a dicho bus.

1.2 Problema

1.2.1 Generalidades

Cuando se diseña un dispositivo electrónico es importante que el mismo no falle. Esta se convierte necesidad aún más importante al tratarse de microprocesadores para el área de medicina. Al ser chip manufacturados para aplicaciones médicas estos deben de ser libres de fallas, tanto a corto plazo como a largo plazo. Además, debe de ser altamente inmune a errores, es decir, en el caso de que sea detectado un error, el chip, sea capaz de reconocerlo y seguir funcionando adecuadamente.

Parte del diseño del microprocesador que se encuentra en desarrollo corresponde a la creación de una interfaz que conecte el núcleo con los periféricos. Para esto fue escogido un bus de datos de protocolo AXI4-Lite. Por lo tanto, existe además la necesidad de crear un ambiente y una serie de pruebas que estresen este bus con el fin de determinar su correcto funcionamiento y eliminar posibles errores que conlleven a un daño en los módulos conectados a dicho bus.

1.2.2 Síntesis del Problema

Desarrollar un ambiente de verificación para el bus de protocolo AXI4-Lite entre los periféricos y el controlador de memoria de un microprocesador para dispositivos médicos implantables.

1.3 Objetivos

1.3.1 Objetivo General

Obtener la verificación del bus con protocolo AXI4-Lite, entre los periféricos del microprocesador y el controlador de memoria, del microprocesador en diseño.
Indicador: El módulo cumple los porcentajes de cobertura correspondientes, los cuales se encuentran en proceso de definición y pronto serán delimitados en [8].

1.3.2 Objetivos específicos

- Diseñar el ambiente de verificación de tipo UVM para el módulo de interfaz entre los periféricos del microprocesador y el controlador de memoria.
  Indicador: La implementación de los bloques mostrados en la figura 2.2.
- Diseñar un modelo de referencia robusto para el ambiente de verificación contra el cual se realizarán las pruebas.
  Indicador: La obtención de una correcta simulación de los escenarios en relación al comportamiento de diseño bajo prueba.

- Diseño de múltiples pruebas que estresen la interfaz de manera tal que se obtengan resultados de respuesta, tanto del funcionamiento apropiado como las así llamados casos esquina.
  Indicador: La implementación de las pruebas dentro del ambiente de verificación tipo UVM obteniéndose un reporte de comparación entre el modelo de referencia y el DUT.

Capítulo 2

Marco Teórico

Para obtener una comprensión general del presente proyecto, es necesario manejar ciertos conceptos básicos, los cuales se detallarán a continuación. Se tocarán temas que abarcan desde el lenguaje de programación y el protocolo de verificación utilizado, hasta una breve explicación del diseño bajo prueba.

2.1 SystemVerilog

SystemVerilog es un lenguaje de descripción de hardware y verificación basado en Verilog[2]. Este lenguaje combina elementos de descripción de hardware de lenguajes como lo es Verilog y VHDL con elementos de lenguajes de verificación de hardware en conjunto con características de C y C++[5]. Su funcionalidad de verificación se encuentra basado en el lenguaje OpenVera, el cual fue donado por Synopsys.

SystemVerilog se encuentra basado en la orientación de objetos, por lo tanto utiliza componentes básicos como Clases, Interfaces, Heredación de clases, parametrización de clases y estructuras, entre otras.

En 2005 SystemVerilog se convirtió en el estándar P1800-2005 de IEEE. Luego, obtuvo una actualización en 2009 y en la actualidas su desarrollo se encuentra a manos del grupo Accellera.

2.2 Metodología Universal de Verificación (UVM)

La Metodología Universal de Verificación se refiere al estándar utilizado en los procesos de verificación de hardware. En esta sección se hará una breve explicación de cada una de las partes que usualmente conforman un ambiente de verificación de tipo UVM. Un ambiente de verificación UVM se encuentra compuesto por componentes de verificación reutilizables. La figura 2.1 corresponde a un diagrama estructural de un ambiente de tipo UVM compuesto por dos agentes. En este diagrama se puede apreciar que los principales componentes de un

ambiente típico corresponden a un *scoreboard* o marcador y dos agentes que se conectan a las entradas y salidas del DUT. Además, se puede determinar que dentro del *scoreboard* se encuentra el modelo de referencia y que cada agente consta de tres bloques: un secuenciador, un conductor y un monitor.

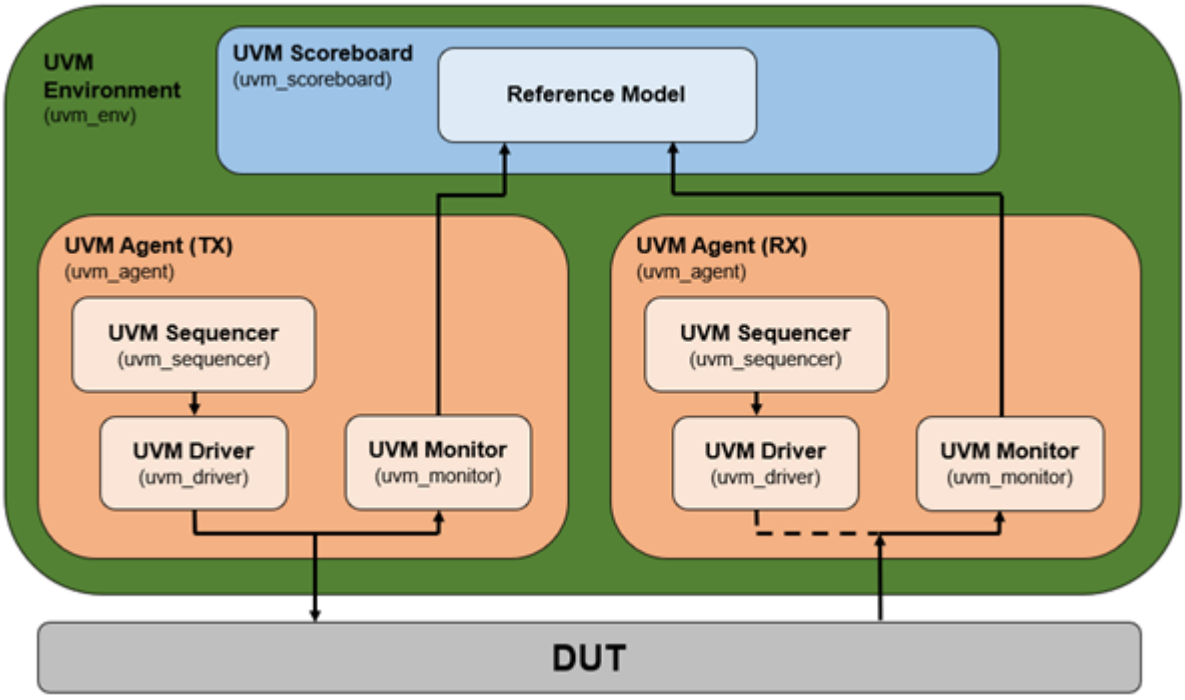

Figura 2.1: Diagrama estructural UVM. Componentes del Banco de pruebas tipo UVM

UVM consiste de tres tipos principales de clases (UVM classes)[7]:

- uvm_object
- uvm_transaction
- uvm_component

uvm_object

Métodos operacionales basados en clase central (*create, copy, clone, compare, print,...*), campo de identificación de instanciación (*name, type name, unique id, ...*) y aplicación de semillas aleatorias son definidas en él.

Todo *uvm_transaction* y *uvm_component* se derivan del *uvm_object*.

uvm_transaction

Se utiliza para la generación de estímulos y análisis.

uvm_component

Los componentes son objetos quasi-estáticos que existen durante toda la simulación.

El uvm_component define un flujo de prueba con fases, que los componentes siguen durante la simulación. Cada fase se define mediante una devolución de llamada.

2.2.1 UVM Testbench

El *Testbench* se refiere al nivel más alto de programación del UVM, este archivo típicamente instancia el UVM Test junto con el diseño bajo prueba o DUT, por sus siglas en inglés. En este archivo es donde se elige cual prueba correr y el ambiente es configurado en relación de dicho examen.

Idealmente las pruebas que se instancian deben de ser llamadas de manera dinámica para que se puedan correr varios exámenes a una sola instancia del DUT.

2.2.2 UVM Test

Este archivo se refiere a donde se instancian y configuran los archivos del ambiente, además de que aplica los estímulos que son llamados desde el *Sequencer* o secuenciador. Estos archivos deben de incluir pruebas que estimulen desde las funciones básicas del DUT hasta los casos esquina o *corner cases*.

2.2.3 UVM Environment

Este se refiere al ambiente de verificación, este es el encargado de contener a los archivos que se interrelacionan. Los archivos que normalmente se encuentran instanciados dentro del ambiente caben mencionar el agente, el *scoreboard* e inclusive se pueden encontrar otros ambientes. Según la metodología UVM, el ambiente debe de construirse de la manera más genérica posible. De esta manera, todo el ambiente pueda utilizarse para diferentes diseños con pequeñas modificaciones. Este archivo se encuentran los siguientes bloques:

- Dos agentes, uno para transmisión y otro para recepción
- Un *scoreboard* o marcador
- Interfaces virtuales

El ambiente también debe de tener una manera en la que se autoconfigure, dependiendo del tipo de prueba que se quiera correr en el DUT. Debido a esto el ambiente tiene que poseer un parámetro que corresponda al examen a correr. Cada prueba debe de tener un identificador asociado, este identificador es el encargado de configurar el ambiente de acuerdo a cada prueba específica.

2.2.4 UVM Agent

El agente es el encargado de instanciar a todos los archivos que lidean directamente con las interfaces del DUT. Son los bloques utilizados para construir el ambiente, dentro de él, usualmente, se encuentran dos instancias, las cuales serían las necesarias para monitorear y manejar el DUT. Al existir dos agentes, uno para transmisión (Tx) y uno para recepción (Rx), cada uno se encarga de diferentes funcionalidades. El agente Tx es el encargado de enviar los estímulos a las entradas del DUT para estimularlo y así provocar una reacción en sus salidas. A este agente también se le llama agente activo (Active UVM Agent). Luego, el otro agente es el encargado de observar las salidas y registrar los valores, este agente se le llama agente pasivo (Pasive UVM Agent).

La microarquitectura que corresponde al agente contiene tres bloques menores: UVM Driver, UVM Monitor y UVM Sequencer. Cada uno de estos bloques se pueden observar en la figura 2.2 y cada uno será explicado más adelante.[8]

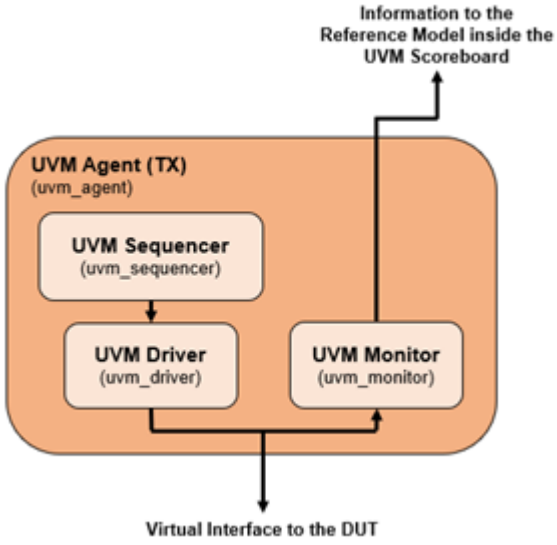

Figura 2.2: Microarquitectura del UVM Agent [8]

El agente es una parte primordial del ambiente de verificación. Como se observa en la figura 2.2 el agente interconecta al driver, al monitor y al secuenciador y los configura el comportamiento. Además, pueden tener puertos de análisis que transmiten la información tomada de la interfaz hacia el *scoreboard*.

Finalmente, el agente debe de recibir 3 parámetros como entrada desde el ambiente: el *Test ID*, la bandera TX y la bandera RX. El Test ID es un parámetro de entrada del ambiente. El ambiente debe de pasar dicho ID al agente TX para que este pueda conocer cual secuencia de prueba debe correr. La bandera Tx indica que el agente va a actuar como un agente de transmisión y va a conducir al DUT. La bandera de RX indica que el agente va correr coo agente de recepción por lo tanto solamente monitoreará las salidas del DUT la tabla 2.1 es la tabla de verdad que muestra esta configuración mencionada[8].

Tabla 2.1: Configuración UVM Agent[8]

Bandera RX	Bandera TX	Funcionalidad
0	0	El agente se encuentra apagado: UVM Driver APAGADO UVM Monitor APAGADO UVM Sequencer APAGADO
0	1	El agente se encuentra en modo TX: UVM Driver ENCENDIDO UVM Monitor ENCENDIDO UVM Sequencer ENCENDIDO
1	0	El agente se encuentra modo RX: UVM Driver APAGADO UVM Monitor ENCENDIDO UVM Sequencer APAGADO
1	1	Estado Inválido: UVM Driver APAGADO UVM Monitor APAGADO UVM Sequencer APAGADO

2.2.5 UVM Driver

El driver es el bloque encargado de conducir las señales que se generan desde el secuenciador y aplicarlas en las interfaces del DUT. Este archivo se encarga de convertir estímulos a nivel de transacción a estímulos en los pines. [1][8]

2.2.6 UVM Monitor

Este es el bloque encargado de muestrear y capturar la información para luego transferirla al resto del *testbench* con el fin de incurrir en un mayor análisis. Similar al *driver*, el monitor es responsable de convertir señales a nivel de pin a estímulos de nivel transacción.

En el monitor se recogen la información de cobertura y la comprobación de rendimiento. Opcionalmente el monitor puede imprimir información de rastreo[9].

2.2.7 UVM Sequencer

Este bloque es el encargado de controlar el flujo de transacciones que provienen de múltiples secuencias de estímulos[1]. El secuenciador es un generador de estímulos avanzado que controla los itemes que son proporcionados al driver. Por defecto, este bloque se comporta como un simple generador de impulsos y entrega datos aleatorios cuando el driver se los solicita. Esta configuración es la que permite agregar restricciones a la clase elemento de

dato y así monitorizar la distribución de dichos valores aleatorios. Entre las capacidades que el secuenciador trae incorporadas incluyen[9]:

- La habilidad de reaccionar al estado presente del DUT con cada dato generado.
- Capturar el orden entre datos en las secuencias definidas por el usuario, lo cual crea un patrón de estímulos más estructurado.
- Habilitar el modelado de tiempo en escenarios reutilizables.

2.2.8 UVM Sequence

Este se refiere al objeto que contiene la generación de estímulos[1]. La secuencia genera series de *sequence items* y los envía por medio del driver al secuenciador.

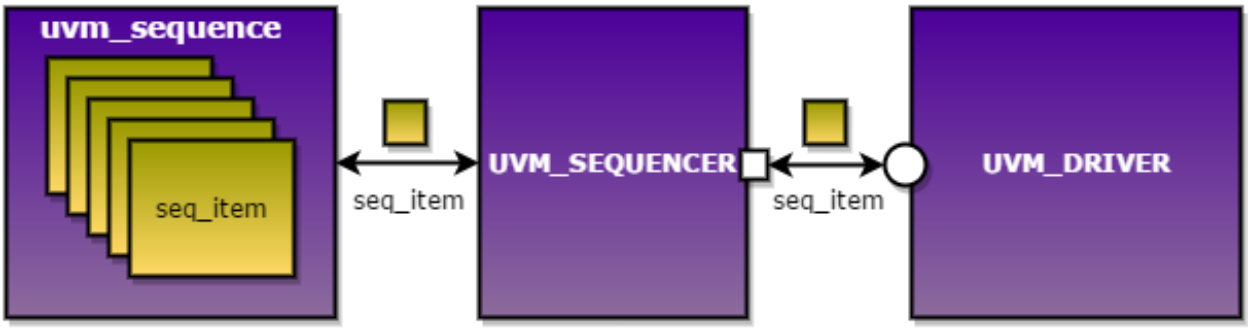

Figura 2.3: Conexión secuencia - secuenciador - driver[7]

La figura 2.3 muestra el proceso descrito anteriormente, en el primer cuadro se aprecia como dentro de la secuencia (uvm_sequence) se encuentran instanciados varios itemes de secuencia (seq_item). luego como estos salen, pasan por el secuenciador (uvm_sequencer)y llegan al conductor (uvm_driver).

2.2.9 UVM Sequence Item

El *sequence item* consiste de los campos de datos requeridos para generar los estímulos. En orden para generar dichos estímulos, los itemes de secuencia son aleatorizados en secuencias. Por consiguiente las propiedades de datos en los itemes de secuencia generalmente deben de ser declarados como *rand* y pueden ser restringidos, como es caso de las direcciones a los esclavos[7].

2.2.10 UVM Scoreboard

La principal función de este archivo es comprobar el comportamiento del DUT. Usualmente recibe las entradas y salidas del DUT, que viajan desde el agente, y las compara con las producidas por el modelo de referencia que se encuentra contenido[1][8].

Este bloque contiene, principalmente, al modelo de referencia

2.2.11 Reference Model

Se refiere a un diseño sencillo a base solamente de código, el cual emula el comportamiento del DUT. Su función principal es emitir las respuestas deseadas a los estímulos producidos por el secuenciador, para que luego estas sean comparadas dentro del *scoreboard* contra las emitidas por el DUT, las cuales se obtienen por medio del monitor. Este bloque también recibe el nombre de predictor.

Utilizando un nivel más alto de programación, el modelo de referencia calcula todas las salidas esperadas en concordancia a los estímulos que ser reciben. Estos datos de salida son actualizados con cada ciclo de reloj, ya que el modelo de referencia no necesita respuesta ni comando de ningún otro bloque que no sean los estímulos provenientes del secuenciador. Una vez que el modelo de referencia es correcto y depurado, la verificación del DUT es correcta en cada ciclo. El mayor inconveniente que presenta el modelo de referencia, es que éste debe de conocer los tiempos de respuesta del DUT, de lo contrario no coincidirán y como consecuencia el DUT no será capas de pasar prueba alguna. Lo anterior representa un gran reto, ya que, al estar programado en un nivel más alto tendrá tiempos de respuesta más rápidos que los del DUT[13].

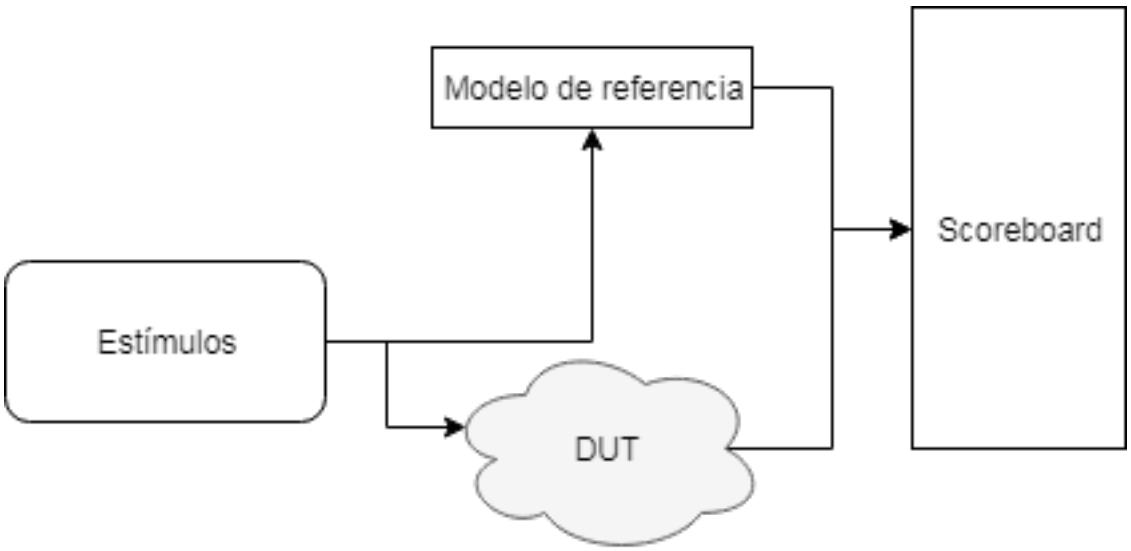

Figura 2.4: Diagrama de funcionalidad del modelo de referencia

Además del modelo de referencia existen otros dos métodos para verificar las respuestas de un DUT. Por medio de ambiente de Golden Vectors o por medio de ambientes basados en transacciones.

Un ambiente de vectores dorados es un ambiente simple en que se almacena el conocimiento base de los vectores válidos de salida y los almacena en el marcador. La ventaja de este mecanismo es que el ingeniero de verificación puede revisar todos los rastros de resultados

predichos antes de correr la simulación. Como desventaja, cada vector debe de ser creado manualmente por lo que resulta tedioso. La figura 2.5 muestra su funcionamiento[13].

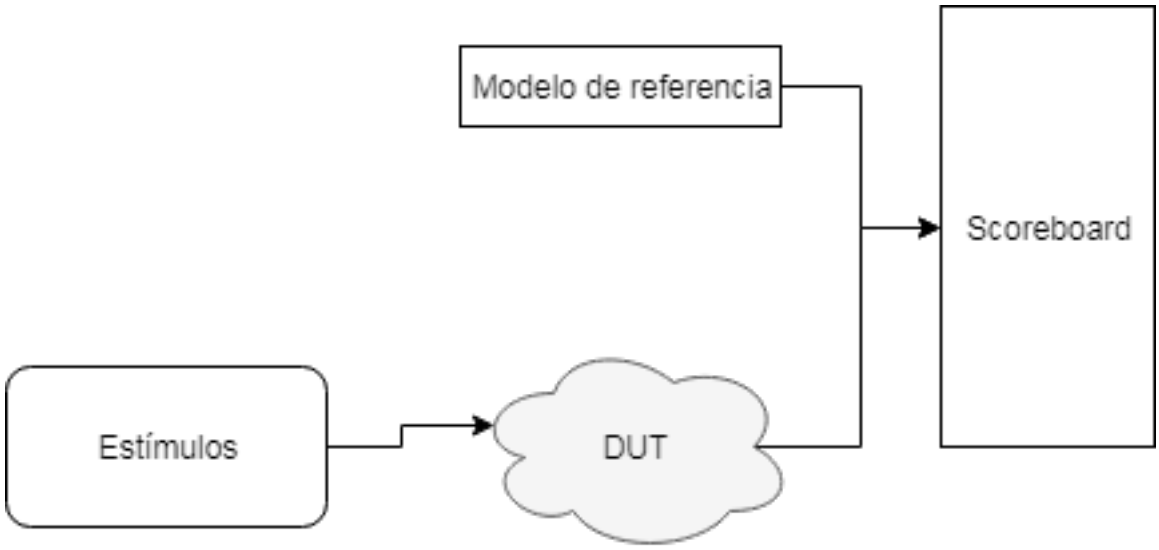

Figura 2.5: Diagrama de funcionamiento del Golden Vectors

Un ambiente basado en transacciones es utilizado para DUTs que poseen transacciones identificables donde los comandos y datos se procesan y se envían las salidas apropiadas. Esto permite al ingeniero de verificación estructurar el ambiente basándose en la naturaleza del DUT. La única desventaja y dificultad para este tipo de entorno es decidir el nivel correcto de abstracción, la cual es una decisión importante ya que influye tanto en la eficacia como en la eficiencia del entorno de verificación[13].

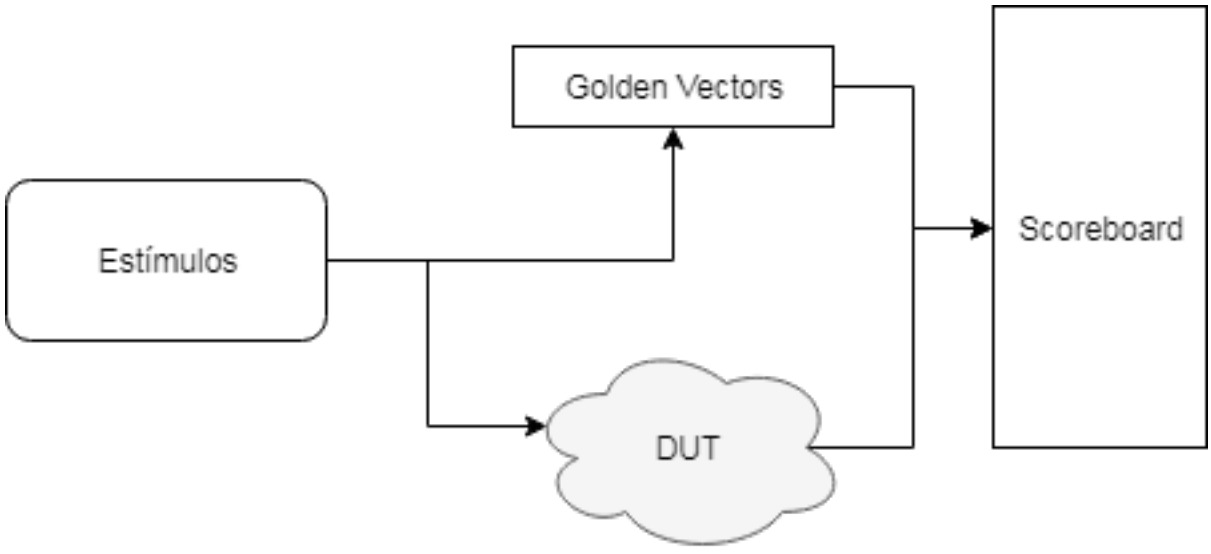

Figura 2.6: Diagrama de funcionamiento del ambiente basado en transacciones.

2.2.12 Cobertura funcional

La cobertura funcional buscar responder a la incógnita: "¿Cómo saber si la verificación del DUT ha finalizado?"[11]

En verificación de hardware, se conoce como cobertura funcional a la recopilación de estadísticas basadas en el muestreo de eventos dentro de una simulación. La cobertura se utiliza para medir cuales características del diseño han sido recorridas por las pruebas. La cobertura es utilizada para determinar el momento en el que el DUT ha sido expuesto a una variedad suficiente de estímulos por lo que existe la confianza de que el DUT funciona de manera correcta.

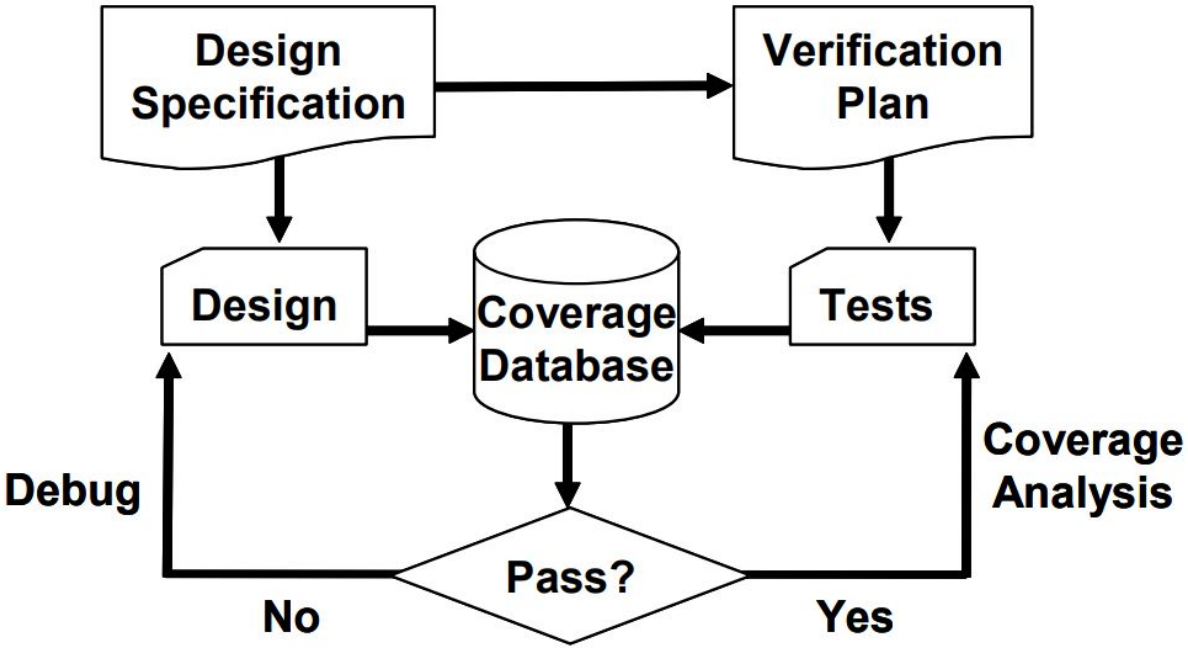

Figura 2.7: Flujo de cobertura[10]

En la figura 2.7 se muestra el proceso de obtención de cobertura. El proceso comienza con la especificación para el diseño, esta especificación da comienzo a diseño y al plan de verificación, ya que ambos deben de crearse al mismo tiempo. El plan de verificación crea las pruebas a las que se someten el DUT y el modelo de referencia, ambas respuestas entran a la base de datos de cobertura. La cobertura es quien determina si el diseño pasa las pruebas. En casa de que la prueba sea exitosa se pasa un reporte de error al diseñado para que éste comience el proceso de depuración del diseño y una vez terminada la depuración se vuelve a pasar por las pruebas continuando el ciclo hasta que el diseño complete todas los exámenes de manera exitosa. Cuando se cumple esto último se crea un análisis de cobertura para determinar la cantidad de casos se examinaron.

En cuanto los diseños se vuelven cada vez más complejos, la manera más efectiva de verificarlos es mediante pruebas aleatorias restringidas o CRT por sus siglas en inglés. Estas pruebas aleatorias son generadas mediante semillas, estas semillas se refieren a archivos dentro de las librerías del UVM que contienen combinaciones predefinidas de señales. Con la

utilización de CRT, se elimina la elaboración manual de cada cadena de estímulos, pero crea la necesidad de crear un código que rastree la efectividad de la prueba respecto al plan de verificación. Es, por lo tanto, necesario trabajar con niveles altos de abstracción. Para obtener una cobertura del 100% es necesario saber que es lo que se quiere observar y como dirigir el DUT a esos estados[10].

2.2.13 Regresión

Regresión es la ejecución continua de pruebas definidas en el plan de verificación. La regresión es un paso requerido en el ciclo de verificación por dos grandes razones. La primera es porque los ambientes de verificación con frecuencia poseen elementos aleatorios, que conducen a diferentes escenarios de entrada cada vez que se corre una prueba. Y la segunda es porque se deben de seguir corriendo pruebas luego de que se le apliquen cambios al diseño.

La tasa de incidencia de fallas disminuye conforme el ciclo de verificación alcanza la etapa de regresión. Con el fin de descubrir errores difíciles de encontrar se deben de tener grandes bancos de pruebas. Cuando se descubre un error se envía el diseño a arreglar, una vez aplicados los cambios se vuelve a pasar por la misma prueba que generó el error. [13]

En otras palabras, la regresión es un archivo generado por el ambiente donde se exponen la cobertura y el informe de fallas.

2.3 RISC-V

En 2010 nació el RISC-V en la Universidad de California, Berkeley. RISC-V es una arquitectura de conjunto de instrucciones (ISA) abierta, diseñada por Berkeley Architecture Group con el objetivo de ayudar a la investigación sobre arquitecturas y a la educación. RISC-V se encuentra completamente disponible al público y como ventajas tiene variabilidad a la longitud de las instrucciones para admitir una instrucción densa opcional, soporte a implementaciones de múltiples núcleos altamente paralelas, eficiencia en energía, entre otras[6]. El RISC-V fue diseñado para tener una extensa personalización y especialización. Su integrador ISA base puede ser extendido con una o más extensiones de set de instrucciones opcionales. [12]

Al contrario del RISC-V, los vendedores de chips comerciales cobran altas tarifas para el uso de las licencias y así poder tener acceso a las patentes. Inclusive exigen acuerdos de confidencialidad antes de que el cliente pueda lanzar su producto.

2.4 AXI4-Lite

El DUT a verificar en este proyecto corresponde al AXI4-Lite. El AXI4-Lite es parte del protocolo de interfaces de nombre AXI. El AXI es parte del ARM AMBA, el cual es una familia de buses para microcontroladores. Esta familia de buses fue introducida en el año 1996. Por su parte, la primera versión del AXI fue introducida por primera vez en 2003 como parte del AMBA 3.0 y la segunda versión, AXI4, fue publicada en 2010 como parte del AMBA 4.0. [14]

La interfaz AXI4 posee tres tipos:

- AXI4: se utiliza en requerimientos de mapeo de memoria de alto desempeño.
- AXI4-Lite: se emplea para comunicaciones simples de mapeo de memoria de bajo rendimiento.
- AXI4-Stream: aplicable a la trasmisión de datos de alta velocidad.

El protocolo AXI4-Lite, al ser para aplicaciones simples, tiene un comportamiento más sencillo en comparación a las otras dos. De hecho, el AXI4-Lite es una versión simplificada del AXI4. Las claves del funcionamiento del AXI4-Lite son:

- Todas las transacciones tienen como longitud de ráfaga 1.
- Todos los accesos a datos utilizan el ancho completo del bus de datos.
- Soporta anchos de 32-bits o de 64-bits.
- Ningún acceso es transferible ni modificable.
- No soporta los accesos exclusivos.

El AXI4-Lite tiene un ancho de bus de datos fijo, el cual debe de ser de 32 bits o de 64 bits y todas las transacciones que se manejen deben de tener el mismo ancho.

El AXI y el AXI4-Lite pueden trabajar en conjunto como maestro o esclavo del otro. La tabla 2.2 muestra la interoperabilidad entre ambos.

Tabla 2.2: Interoperabilidad entre el AXI y el AXI4-Lite

Maestro	**Esclavo**	**Interoperabilidad**
AXI	AXI	Completamente operacional
AXI	AXI4-Lite	Requiere de reflexión del ID del AXI. Puede que sea necesaria una conversión
AXI4-Lite	AXI	Completamente operacional
AXI4-Lite	AXI4-Lite	Completamente operacional

El protocolo AXI4-Lite al ser una aplicación simplificada trabaja con una menor cantidad de señales, en comparación con el AXI4. En la tabla 2.3 se muestran las señales con las que trabaja el protocolo AXI4-Lite.

Tabla 2.3: Señales del protocolo AXI4-Lite[3]

Global	W_Address	W_Data	W_Response	R_Address	R_Data
ACLK	AWVALID	WVALID	BVALID	ARVLID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

Donde cada señal se refiere a lo siguiente:

Señales Globales

- ACLK: señal de reloj.
- ARESETn: señal de reinicio, activada en bajo.

W_Address

Estas señales provienen del maestro a menos que se especifique lo contrario.

- AWADDR: dirección de escritura, tiene tamaño variable de 32 o 64 bits.
- AWPROT: de tamaño de 3 bits, indica el nivel de privilegio y la seguridad de la transacción.
- AWVALID: indica si el canal contiene señales de control y datos válidos
- AWREADY: proveniente del esclavo indica que este se encuentra listo para aceptar la información.

W_Address

Estas señales provienen del maestro a menos que se especifique lo contrario.

- AWADDR: dirección de escritura tiene tamaño variable de 32 o 64 bits.
- AWPROT: de tamaño de 3 bits, indica el nivel de privilegio y la seguridad de la transacción.
- AWVALID: indica si el canal contiene señales de control y datos válidos

- AWREADY: proveniente del esclavo indica que este se encuentra listo para aceptar la información.

W_Data

Estas señales provienen del maestro a menos que se especifique lo contrario.

- WDATA: datos de escritura de tamaño variable de 32 o 64 bits.
- WSTRB: (*write strobes*) indican cuales carriles de 8 bits contienen información.
- WVALID: indica si la información de escritura y los *write strobes* se encuentran disponibles
- WREADY: proveniente del esclavo indica que este se encuentra listo para aceptar la información.

W_Response

Estas señales provienen del esclavo a menos que se especifique lo contrario.

- BRESP: Indica el estado de la transacción de escritura.
- BVALID: indica si la información de escritura y los *write strobes* se encuentran disponibles
- BREADY: proveniente del maestro indica que este se encuentra listo para recibir respuesta de escritura.

R_Address

Estas señales provienen del maestro a menos que se especifique lo contrario.

- ARADDR: dirección de lectura de tamaño variable de 32 o 64 bits.
- ARPROT: indica privilegio y seguridad en la transacción.
- ARVALID: indican que el canal contiene una dirección de lectura válida.
- ARREADY: proveniente del esclavo indica que este se encuentra listo para aceptar la dirección.

R_Data

Estas señales provienen del maestro a menos que se especifique lo contrario.

- RDATA: datos de lectura.
- RRESP: indica estado de la transacción.
- RVALID: indican que el canal contiene información válida.
- RREADY: proveniente del esclavo indica que este se encuentra listo para aceptar la información y respuesta.

De estas señales, para el ambiente, se utilizaron las señales de datos (DATA), de dirección (ADDR) y las de validación (VALID).

El AXI4-Lite a verificar fue diseñado utilizando como base una configuración de dos maestros y dos esclavos. Este diseño sí cuenta con la característica de poder variar la cantidad de maestros y esclavos, pero con el fin de generar las primeras pruebas básicas se utilizó esta configuración. Como parte de la configuración mencionada se utiliza el mapeo de memoria mostrado en la figura 2.8 en el cual se asignan direcciones desde la 32'h40000000 hasta la 32'h40001FFF.

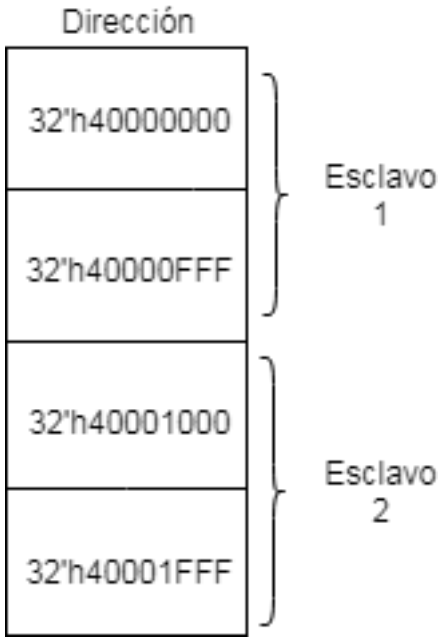

Figura 2.8: Mapeor de memoria para los esclavos

El AXI4-Lite cuenta con cinco diferentes canales, los cuales se encuentran mostrados junto a sus pines correspondientes en la tabla 2.3. Estos canales son:

- Canal de dirección de lectura (R_Address)

- Canal de dirección de escritura (W_Address)
- Canal de datos de lectura (R_Data)
- Canal de datos de escritura (W_Data)
- Canal de respuesta de escritura (W_Response)

Los datos puede moverse en ambas direcciones entre maestros y esclavo de manera simultánea, y el tamaño de transferencia puede variar. El AXI4-Lite permite solamente una transferencia de datos por transacción.

La figura 2.9 muestra como la transacción de lectura utiliza el canal de dirección de lectura y el canal de datos de escritura. Se observa que el maestro envía la instrucción de lectura y la dirección donde desear leer; luego el esclavo le responde enviando la información solicitada por medio del canal de datos de lectura, terminando ahí dicha transacción.

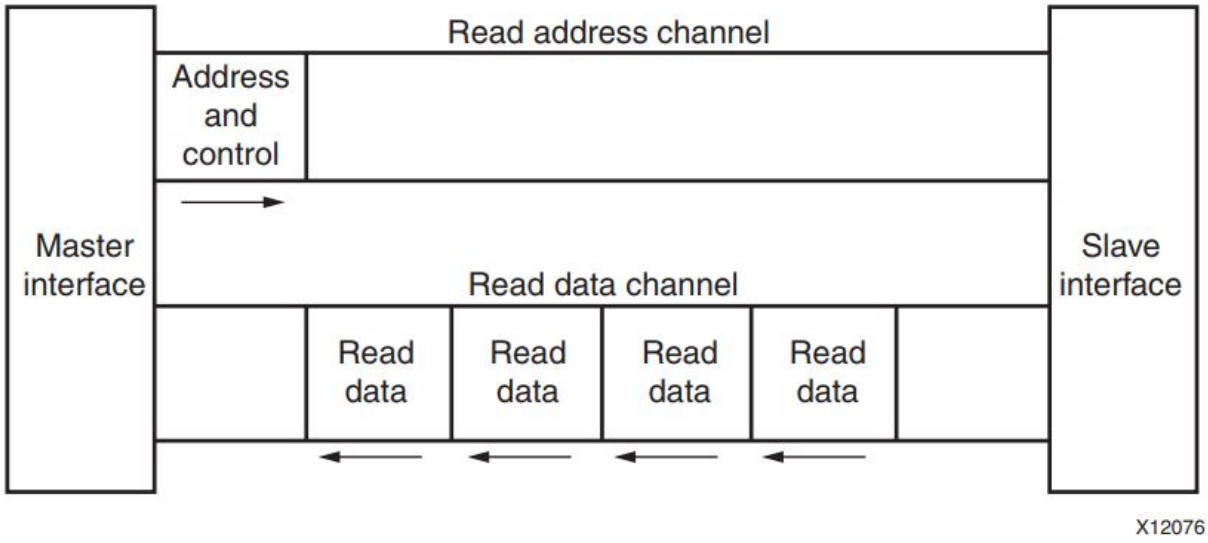

Figura 2.9: Arquitectura de lectura [14]

Por su parte, la figura 2.10 muestra la transacción de lectura. Esta transacción utiliza los otros tres canales, iniciando con el maestro utilizando el canal de dirección de escritura, donde envía la instrucción y la dirección donde desea ejecutarla. Luego, envía los datos que desea escribir y por último, el esclavo contesta con el canal de respuesta.

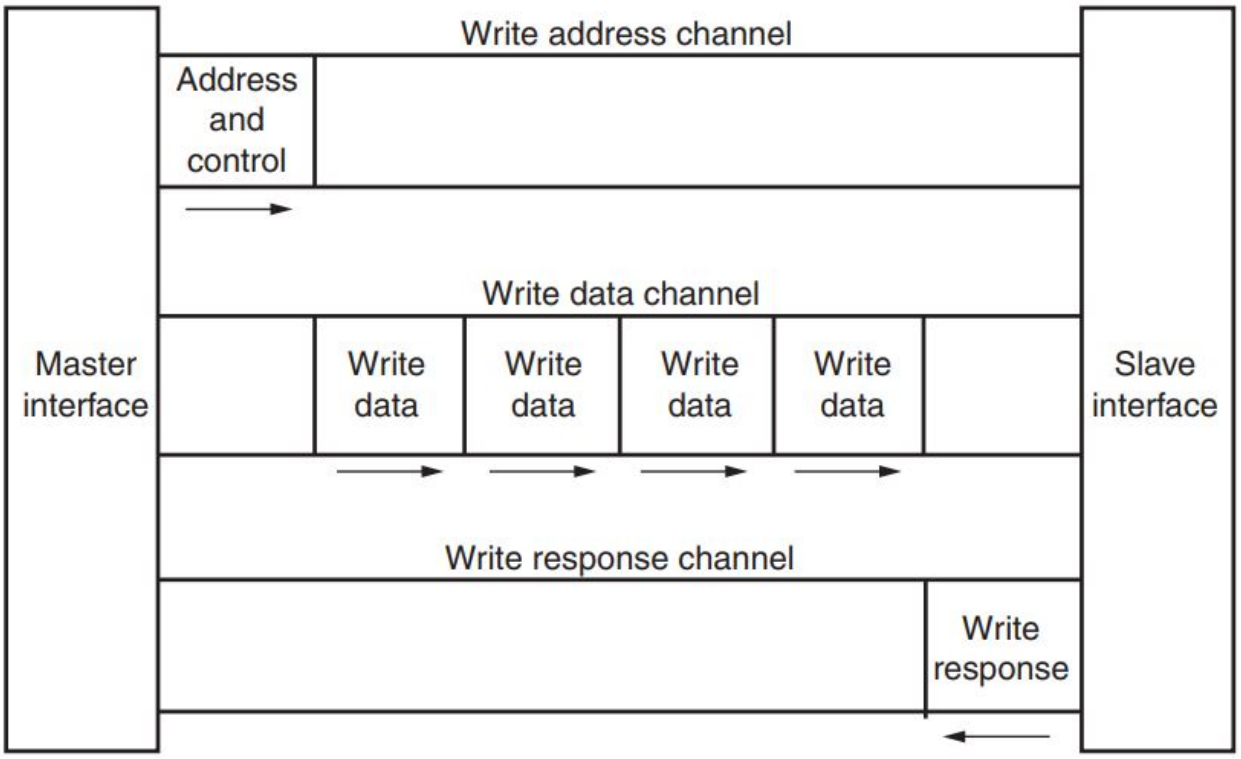

Figura 2.10: Arquitectura de escritura [14]

Como se observa en ambas figuras el proceso de lectura y escritura conllevan más de un paso y por lo tanto también más de un ciclo de reloj. Pero, ¿qué le indica al sistema que el siguiente paso debe de ejecutarse? Esto se determina por medio del protocolo de enlace o *handshake protocol*. Este protocolo define parejas de las cuales una de ellas se enciende y mantiene en alto hasta que la otra se active, una vez activadas, ambas caen e inicia el siguiente paso de operación. La tabla 2.4 muestra las parejas del AXI4-Lite.

Tabla 2.4: Parejas de protocolo de enlace por canal de transmisión[3]

Canal de transacción	Pareja de enlace
W_Address	AWVALID, AWREADY
W_Data	WVALID, WREADY
W_Response	BVALID, BREADY
R_Address	ARVALID, ARREADY
R_Data	RVALID, RREADY

El protocolo tiene tres maneras es de funcionar:

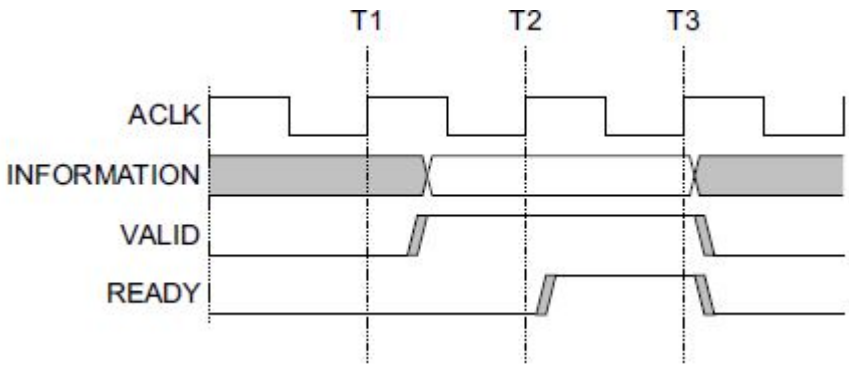

Figura 2.11: Enlace de VALID antes del READY[3]

La primera es cuando la señal de VALID se activa y hasta que la señal de READY se envíe se apaga. Esto se ilustra en la figura 2.11. Un ejemplo de esta funcionalidad son las señales de ARVALID y ARREADY provenientes de los esclavos, cuando se realiza un lectura el DUT activa a ARVALID hasta que al DUT le ingrese un alto en ARREADY.

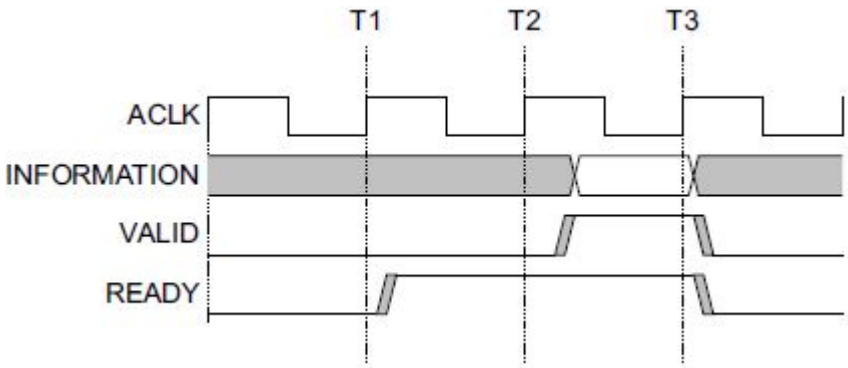

Figura 2.12: Enlace de READY antes del VALID[3]

La figura 2.12 ilustra el caso de cuando es la señal de READY que se activa hasta la aparición de la señal de VALID. Un ejemplo de este caso en el diseño que se probó fue el caso de RVALID y RREADY, cuando se realiza una lectura.

Por último se tiene el caso de cuando las señales se activan y apagan juntas. El DUT verificado en este proyecto no tenía tal caso. La figura 2.13 ilustra el caso.

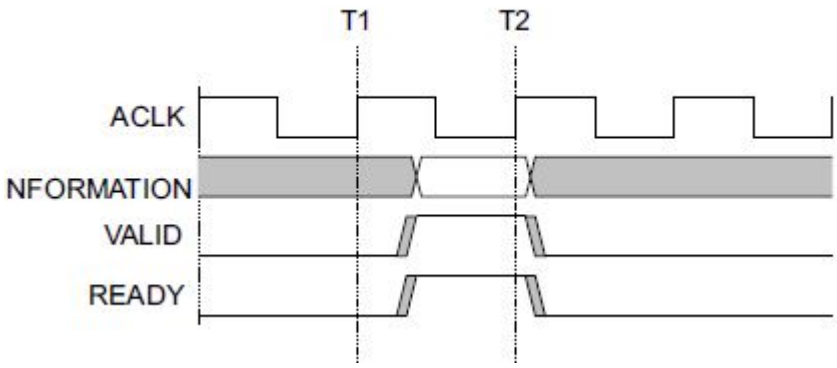

Figura 2.13: Enlace de VALID y READY

La finalidad de este proyecto es verificar que la configuración mencionada anteriormente (dos maestros y dos esclavos) trabajen de manera correcta las funciones descritas en esta sección.

Capítulo 3

Ambiente de Verificación para el bus de datos AXI4-Lite

3.1 Creación del ambiente

Se decidió utilizar la metodología UVM ya que este permite la creación de módulos base que pueden ser utilizados para diferentes aplicaciones, el UVM tiene la versatilidad de que únicamente en la generación de las pruebas y estímulos es personalizado para el DUT. Otra ventaja del UVM sobre la metodología directa es su capacidad de aceptar datos aleatorios, que permite explorar una mayor cantidad de eventos. Además, es el método que actualmente se utiliza en la industria.

Los ambientes de verificación de tipo UVM se codifican con el lenguaje *SystemVerilog*. Para trabajar con este lenguaje se requieren licencias. Por este motivo el proyecto se inició en una página abierta que permite, con ciertas limitaciones, codificar en dicho lenguaje de manera gratuita. La página tiene por nombre EDA Playground[4].

Al ser una página abierta al público, la misma contiene varios ejemplos de códigos para verificación utilizando *SystemVerilog*. La mayoría de los ejemplos disponibles son de carácter educativo, por lo que se encuentran incompletos, ya que su finalidad es ejemplificar sólo ciertas partes de un ambiente tipo UVM. Tras realizar una extensiva búsqueda en diversos sitios web, fue posible hallar un ejemplo relevante al proyecto en la página Verification Guide. El ejemplo consultado en la página antes mencionada es también de corte didáctico, sin embargo, es un ejemplo que busca explicar y demostrar el funcionamiento de cada una de las partes de un ambiente de verificación orientado a una memoria[7]. Por lo tanto, se seleccionó y utilizó este ejemplo como base para el código del proyecto. Cabe aclarar que al ser este un ejemplo orientado a una memoria, fue necesario realizar modificaciones con el fin de ajustarlo al bus de datos AXI4-Lite.

Parte de las modificaciones que se realizaron fue la implementación de las interfaces para el AXI4-Lite. Estas interfaces, luego de un proceso de prueba y error, fueron implementadas de manera matricial. Esto debido a que es la misma manera en la que se crean los pines en

el DUT. Luego, para asignar un nombre específico a cada interfaz se asignaron cables a cada una. En la siguiente figura se muestra un extracto del código ejemplificando lo dicho.

```
//-------------------------------------
//interface instance
//-------------------------------------
axi_if intf(ACLK,ARESETn);
Ports #(AW,DW,Masters)M_Ports();
Ports #(AW,DW,Slaves)S_Ports();

//-------------------------------------
//DUT instance
//-------------------------------------

AXI_Int DUT(
.ACLK(intf.ACLK),
.ARESETn(intf.ARESETn),
.M_Ports(M_Ports),
.S_Ports(S_Ports)
);
    //Bloque Master1
        //R_Address
assign M_Ports.ARADDR[0]=intf.M1_ARADDR;
assign M_Ports.ARPROT[0]=intf.M1_ARPROT;
assign M_Ports.ARVALID[0]=intf.M1_ARVALID;
assign intf.M1_ARREADY=M_Ports.ARREADY [0];
```

Figura 3.1: Extracto del código de instanciación de las interfaces

3.2 Modelo de Referencia

Como parte de la verificación se necesita hacer una revisión de las salidas del DUT y poder compararlas para determinar el correcto funcionamiento del mismo. Al utilizarse el lenguaje SystemVerilog se eligió trabajar con el ambiente con modelo de referencia debido a la flexibilidad que ofrece trabajar con niveles altos de programación, además de no necesitar una completa comprensión de la lógica dentro del DUT. Es el más adecuado para la aplicación de múltiples pruebas aleatorias, de las cuales no es posible conocer de ante mano la respuesta concreta que el DUT tendrá. Como primera opción se optó por buscar uno en línea, pero esta opción tenía varios inconvenientes. El primer inconveniente encontrado fue que la opción de buscar un modelo de referencia como tal no era posible, pues las compañías que los diseñan sólo los tienen disponibles bajo licencia. Por lo que se hubiera tenido que realizar un pago para poder acceder a ellos. Debido a esto se optó por buscar una implementación del AXI4-Lite existente, disponible para uso del público en general. Esta opción se tuvo que descartar debido a que los diseños encontrados estaban escritos en VHDL. Aunque no hubiera sido imposible utilizar otro lenguaje, porque la página EDA Playground permite el uso de varios lenguajes de programación en un mismo código, hacer esta combinación complicaba de más la lógica de la programación. Por último, se descartó de manera definitiva debido a que los diseños encontrados correspondían a modelos de solamente un maestro y un esclavo, por lo que se volvía necesario instanciarlos más de una vez, causando errores en el momento de realizar las comparaciones.

Finalmente se decidió por realizar el diseño de un modelo de referencia robusto que cumpliera las funciones del bus de datos de manera similar. La función de este modelo de referencia, descrito de mejor manera en el capítulo 2, es servir de comparación entre las respuestas que produce el DUT a las pruebas y las respuestas que debe de producir. La lógica base del modelo de referencia se puede observar en el diagrama de flujo en la figura 3.2.

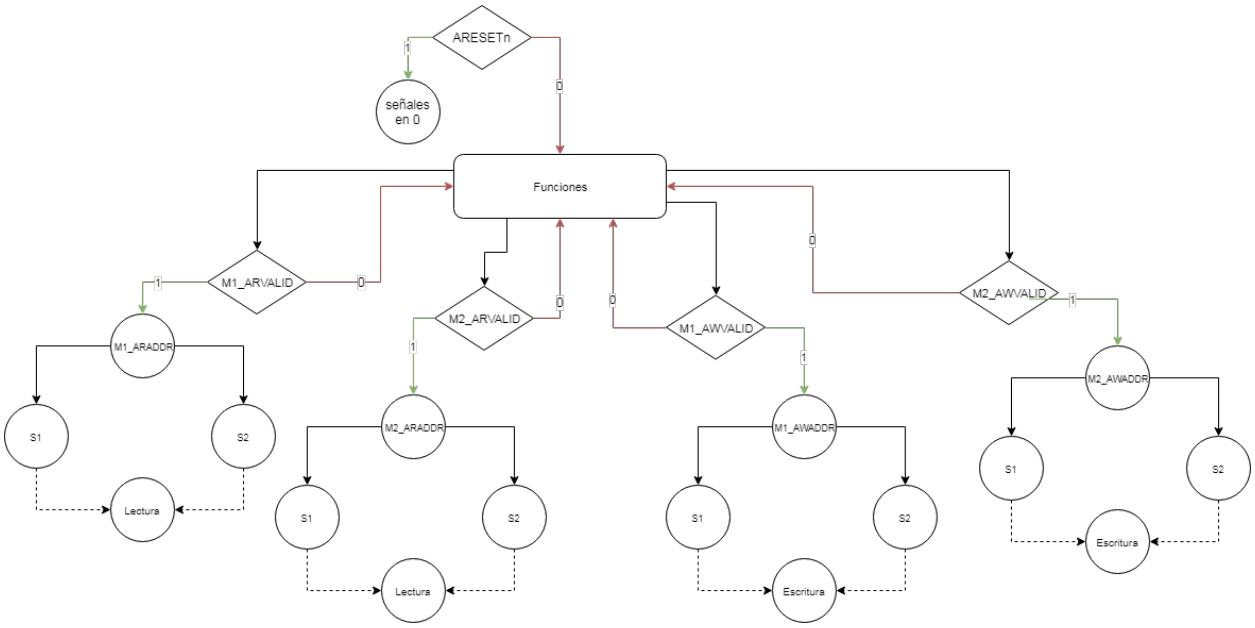

Figura 3.2: Diagrama de flujo del modelo de referencia

Cabe destacar que las *burbujas* que dicen Lectura y Escritura corresponden a la lógica detrás de las funciones. En la figura 3.3 se muestra el procedimiento que sigue el modelo de referencia para cumplir la función de lectura.

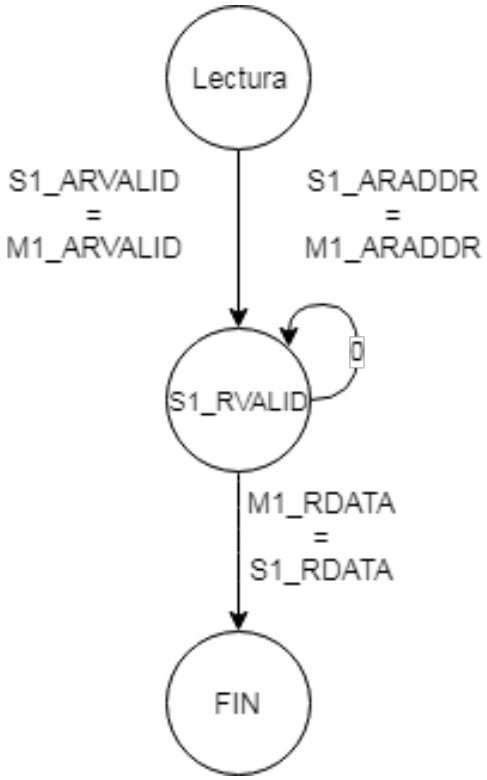

Figura 3.3: Diagrama de flujo función de lectura del maestro 1 al esclavo 1.

Por su parte, la función de escritura es homóloga a la de lectura, por lo tanto lleva en sí una lógica similar, con el único cambio en el nombre de las señales.

Debido a que el DUT trabaja por medio de la utilización de banderas para activar ciertas de sus salidas, se optó por utilizar colas de tipo FIFO (*first in first out*), ya que emular dicho comportamiento en el modelo de referencia resultaba ser de alta dificultad, además de ineficiente. Con las FIFO se almacenan los datos generados por el modelo de referencia, y cuando el DUT alza su bandera de salida correspondiente salen uno a uno. Inmediatamente salen de la cola, los datos son comparados.

3.3 Pruebas

Por tema de simplificar la nomenclatura de cada prueba se definieron las siguientes

Tabla 3.1: Abreviaciones

Abreviatura	Nombre
M1	Maestro 1
M2	Maestro 2
S1	Esclavo 1
S2	Esclavo 2

Con el objetivo de cubrir todos los casos posibles a los cuales se podría enfrentar el AXI4-Lite durante su aplicación, se diseñaron las siguientes pruebas:

3.3.1 Prueba manual

Se diseñó una manual, la cual es estática para utilizarse en casos específicos. Principalmente se utilizó para comprobar el funcionamiento de distintas partes del ambiente, tales como las interfaces y el modelos de referencia.

Esta prueba trae dentro tiempos de espera en blanco, colocando las señales en alta impedancia, tiempos de espera forzando todas las señales a cero, dos direcciones específicas para ambos esclavos. La longitud de tiempo de la prueba es lo que toma realizar la función de lectura.

3.3.2 Pruebas de Lectura

Para las pruebas de lectura se pensó en inicializar solamente la función de lectura en un solo maestro y restringiendo la dirección de lectura a un solo esclavo.*

1. Lectura del M1 al S1: en este primer caso se activa la entrada M1_ARVALID y se restringe M1_ARADDR a mantenerse dentro del rango [32'h40000000 a 32'h40000FFF] y se generan todos los casos posibles de M1_RDATA.

2. Lectura del M1 al S2: en este segundo caso se activa, también, la entrada M1_ARVALID y pero la restricción de M1_ARADDR es de [32'h40001000 a 32'h40001FFF], para trabajar con S2 y se generan todos los casos posibles de M1_RDATA.

3. Lectura del M2 al S1: similar al primer caso pero utilizando las señales M2_ARVALID, M2_ARADDR y M2_RDATA

4. Lectura del M2 al S2: de la misma naturaleza que el segundo caso pero utilizando las señales M2_ARVALID, M2_ARADDR y M2_RDATA

*Para todos los casos RVALID siempre se activará en uno luego de que la dirección sea aceptada. Y todas las señales de respuesta para aceptar o enviar los datos de lectura, también responderán en alto todos los casos.

La figura 3.4 representa la lógica que siguen las pruebas descritas anteriormente. Esta figura se hizo con un maestro genérico y con S1.

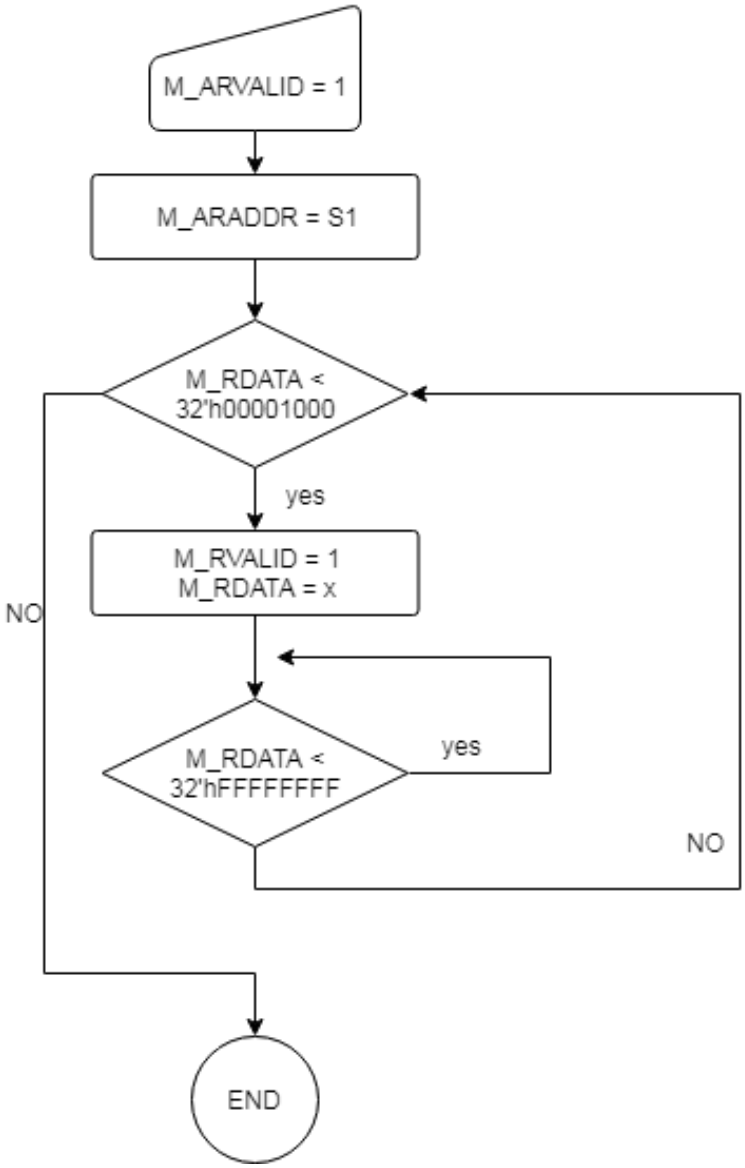

Figura 3.4: Diagrama de flujo prueba de lectura

Con el fin de tener un mayor control de las pruebas se pueden tener dos variantes, la primera puede incrementar el dato de envío uno a uno al igual que la dirección de lectura y la segunda, con ayuda de un contador, ir generando datos aleatorios sin un orden específico y una vez que se llegue a la cantidad de combinaciones esperada terminar el ciclo.

3.3.3 Pruebas de Escritura

Las pruebas de escritura buscan activar la función de escritura en un solo maestro, restringir la dirección a un solo esclavo y explorar todas las posibilidades.**

1. Escritura del M1 al S1
2. Escritura del M1 al S2

3. Escritura del M2 al S1
4. Escritura del M2 al S2

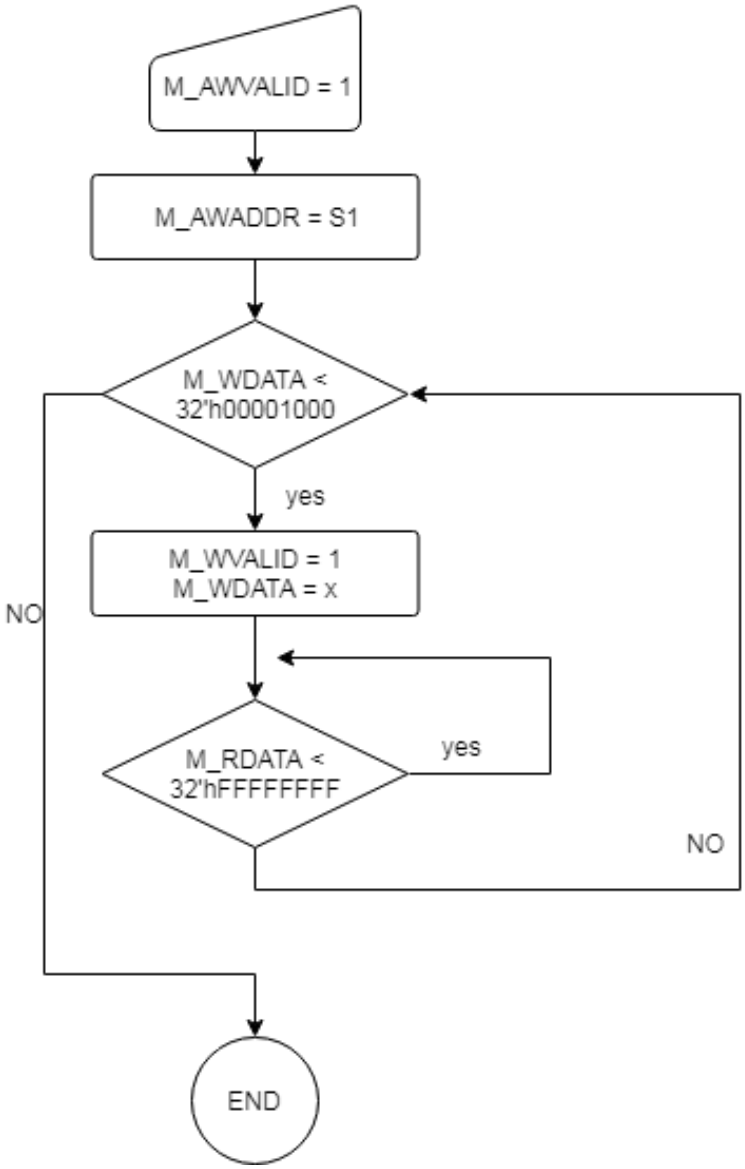

Figura 3.5: Diagrama de flujo prueba de lectura

**Para todos los casos WVALID siempre se activará en uno luego de que la dirección sea aceptada. Y todas las señales de respuesta para aceptar o enviar los datos de escritura, también responderán en alto todos los casos.

Como se puede notar en la figura 3.5, las pruebas de escritura siguen la misma lógica que la descrita para las pruebas de lectura(figura 3.4), pero para el caso de solamente de escritura.

3.3.4 Pruebas de Lectura y Escritura

1. Lectura y escritura del M1 al S1

2. Lectura del M1 al S1 y escritura del M1 al S2
3. Lectura y escritura del M1 al S2
4. Lectura y escritura del M2 al S1
5. Lectura del M2 al S1 y escritura del M2 al S2
6. Lectura y escritura del M2 al S2

En definición estas pruebas llaman de manera paralela las dos pruebas anteriores.

3.3.5 Prueba aleatoria

Se diseñó una prueba completamente aleatoria que estrese el DUT activando todas las entradas. La finalidad de esta prueba llevar al límite al DUT.

3.3.6 Cobertura

Parte muy importante del sistema de pruebas es la exploración de cada una de las posibles combinaciones de las entradas al DUT. Como primera instancia se tienen las entradas de datos, estas entradas, cada una consta de 32 bits por lo que existen:

$$32^2 = 1024$$

Y el AXI4-Lite cuenta con 2 entradas de datos, WDATA y RDATA, para cada maestro, por lo que son 4 entradas de datos en total.Por lo tanto:

$$1024^4 = 1,0995 * 10^{12}$$

Luego para las entradas a las entradas de dirección se tienen menos combinaciones, puesto que a pesar de ser entradas de 32 bits, las direcciones asignadas a cada uno de los esclavos solamente utilizan los 4 bits menos significativos, pero de estos 4 bits solamente los 3 menos significativos son los que generan las combinaciones, por lo tanto:

$$16^3 = 4096$$

Cada dirección posee 4096 combinaciones pero al ser 2 esclavos:

$$4096 * 2 = 8192$$

Son 8192 combinaciones por cada esclavo, pero existen 2 entradas de dirección (AWADDR y ARADDR) por cada maestro, por lo tanto son 4 entradas de dirección:

$$8192^4 = 4,5036 * 10^{15}$$

Por último:

$$4,5036 * 10^{15} * 1,0995 * 10^{12} = 4.9517 * 10^{27}$$

Existen $4.9517 * 10^{27}$ combinaciones disponibles para el AXI4-Lite, por lo que mínimo es necesario crear la misma cantidad repeticiones de las pruebas, para así completar al 100% la cobertura.

Capítulo 4

Resultados y Análisis

4.1 Familiarización con el lenguaje

Al inicio del proyecto uno de los mayores retos que se debió enfrentar fue la comprensión del lenguaje y del protocolo UVM, su funcionalidad y la lógica detrás del mismo. Por lo tanto, gran parte del tiempo se invirtió en el estudio de ambos, con el apoyo de tutoriales en línea y videos explicativos en la plataforma YouTube.

4.1.1 Ejemplo e implementación del ambiente

Parte del estudio realizado se enfocó en encontrar un ejemplo para utilizar como base para el proyecto, ya que al tener que seguirse un protocolo, la base de todos los ambientes es similar, gracias al mismo.

Originalmente se contaba con un ejemplo sencillo cuyo DUT es un archivo que imprime un mensaje en cada cambio del reloj, pero a pesar de tener un video explicativo, este presentaba el inconveniente de contar solamente con un secuenciador, el driver y el testbench, por lo tanto era necesario crear el monitor. En consecuencia, se tuvo que buscar ejemplos explicativos del código para un UVM Monitor. Durante la búsqueda de este código fue cuando se encontró el ejemplo de [7].

El ejemplo encontrado consistía en un ambiente de verificación para una memoria con una entrada para dirección, una entrada para datos de escritura, una para datos de lectura y los dos enables para cada función. en la figura 4.1 el esquema realizado por [7]

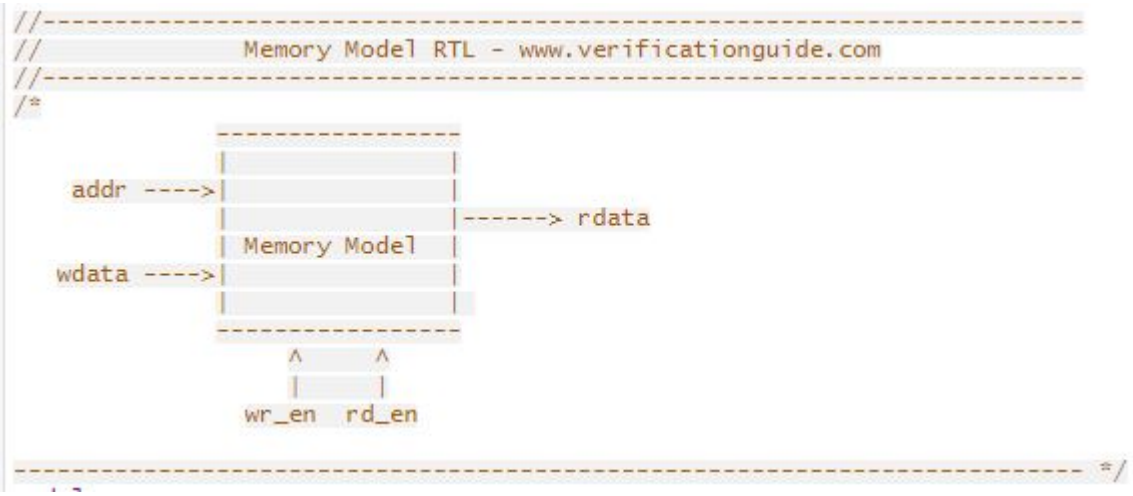

Figura 4.1: Esquema del DUT de ejemplo[7]

El ejemplo, para mantener uniformidad entre los archivos, empieza los nombres de los objetos y componentes con "mem_", por lo que se utilizó esta misma nomenclatura intercambiando el "mem" por "axi".

A cada uno de los bloques se les debió de hacer cambios para que se ajustaran a la funcionalidad del AXI. El más difícil de hacer fueron las interfaces y el que sufrió más cambios fue el driver, pero esto será explicado más adelante.

Cuando se terminaron de modificar cada uno de los bloques se hizo correr el programa y la página genera un reporte de errores o de que la compilación fue exitosa. La figura 4.2 muestra el caso en que el código no pudo compilarse debido a un error de syntax.

```
Error-[SE] Syntax error
  Following verilog source has syntax error :
  "design.sv", 49: token is '-'
      if (rd_en) rdata-1 <= mem[addr];
                         ^

1 warning
1 error
CPU time: 1.152 seconds to compile
Exit code expected: 0, received: 1
Done
```

Figura 4.2: Mensaje de error de sintaxis generado

Una vez terminadas las modificaciones en el código este se corrió varias veces y en el momento en que se corrigieron todos los errores menores existentes este compiló. La compilación se puede observar al final del informe que se muestra en la ventana inferior de la página, la figura 4.3 muestra un ejemplo de una compilación exitosa.

```
** Report counts by severity
UVM_INFO :   22
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
[axi_scoreboard]    16
[axi_test_lectura]     3

$finish called from file "/apps/vcsmx/etc/uvm-1.2/src/base/uvm_root.svh", line 527.
$finish at simulation time              155000
           V C S   S i m u l a t i o n   R e p o r t
Time: 155000 ps
CPU Time:      9.060 seconds;       Data structure size:   0.4Mb
Sun Jun 10 23:02:23 2018
Done
```

Figura 4.3: Reporte de final de compilación

Como se puede observar el reporte de compilación muestra el tiempo que le toma compilar, el tamaño del archivo y el tiempo de simulación.

Durante el desarrollo del proyecto uno de los servidores de la página se dañó completamente, por lo que por un lapso de tres a cuatro días la página se encontraba totalmente caída, durante el cual no se pudo trabajar directamente en el código. Luego de dicho lapso de tiempo la página volvió a estar disponible pero más lenta a la hora de cargar y de vez en cuando caía, dificultando el momento de programar.

4.1.2 Interfaces

El ambiente de verificación, para su funcionamiento, requiere de la creación de interfaces, las cuales conectan el DUT con el ambiente. Estas interfaces son creadas y conectadas directamente al DUT dentro del *testbench*. Luego estas son redefinidas dentro de *seq_item.sv* como señales de tipo aleatorias (*random bit*) o de tipo no aleatorias (*bit*), esto quiere decir se separan por primera vez en entradas y salidas. De las señales aleatorias dos de ellas no pueden ser completamente aleatorias, debido a que se saldrían del rango de trabajo del DUT, estas son las señales de AWADDR y de ARADDR. Dentro del DUT existe una mapeo de memoria, el cual define que la dirección para el esclavo 1 se encuentra entre [*32'h 00000000*] y [*32'h 00000FFF*], y el esclavo 2 [*32'h 00001000*] y[*32'h 00001FFF*].

Inicialmente se pensó que cada una de las señales se encontrarían instanciadas una a una por lo que las interfaces se diseñaron de esa manera. Entonces cada señal fue creada una a una colocando el prefijo $M1\_$ para las señales provenientes del primer maestro, $M2\_$ para el segundo, $S1\_$ para el primer esclavo y $S2\_$ para el segundo. Pero cuando el DUT fue

entregado para ser evaluado esto resultó no ser de dicha manera. Por lo tanto se probaron distintas maneras para conectar el DUT con las interfaces creadas en el ambiente, sin éxito alguno. Finalmente se lograron crear formando matrices de la misma manera en la en la que el DUT las crea dentro de si y asignando cada posición de la matriz a cada uno de los nombres que se habían creado para cada pin. En la figura 3.1 se muestra lo mencionado.

Con el fin de probar las interfaces se realizó una prueba sencilla al DUT, la cual consistió en hacer una lectura sencilla. La prueba activaba la bandera de lectura del maestro 1 (M1_ARVALID) y indicaba que se quería leer en la dirección *M1_ARADDR=32'h40000123*, por lo tanto a la salida se debía mostrar la misma dirección en el puerto S1_ARADDR. Pero, inicialmente, el DUT no respondió a las señales provenientes del ambiente, es decir, no habían señales de respuesta. Para poder obtener señales a la salida del DUT se tuvo que poner en alto a M1_ARVALID por más de un ciclo, incumpliendo así el protocolo que exigen que dicha señal permanezca en alto solamente por un ciclo de reloj. Una vez obtenidas señales en las salidas se notó que estas estaban respondiendo a las señales futuras, en vez de a la señal anterior.

Por lo tanto se comenzaron a revisar una a una las señales que provenían del DUT para poder determinar de donde provenía el error. Se notó que las señales entraban al FIFO de lectura pero no se mostraban a la salida. Luego, por recomendación del diseñador se revisó el flip-flop de dicha FIFO y se observó que el *enable* del flip-flop no se activaba en el momento indicado y que también presentaba el mismo problema mencionado, este se encontraba reaccionando de acuerdo al estado futuro, por lo que manteniendo el ARVALID por dos ciclos de reloj si lograba activar el enable y la información cruzaba por el FIFO como debía.

Luego de analizar y buscar la posible causa del error se decidió que como solución temporal se modificó el ambiente para que trabajara en el flanco negativo del reloj, dándole un *delay* de medio tiempo.

Con el objetivo de determinar la causa del error descrito anteriormente se intentó recrear una prueba que realizó el diseñador para determinar el funcionamiento básico del AXI4-Lite. Al ser una prueba con la que ya se contaba con los resultados hacía más sencilla la búsqueda del error, ya que se podía descartar problemas en el envío de las señales. Replicar la prueba realizada por el diseñador resulto en vano, ya que los tiempos de activación de cada señal no afectaba el resultado erróneo.

Por casualidad, luego de continuar con el trabajo, se encontró que el origen del error se hallaba en un comando de *posedge* mal ubicado. Una vez reubicada dicha línea en el código el sistema respondía como debía.

4.2 Modelo de Referencia

Lo primero que se tenía que tener en consideración para realizar el diseño del modelo de referencia son las señales del DUT que cumplen una lógica dentro del módulo, es decir, cuáles señales son necesarias para activar la lectura y cuáles son necesarias para la escritura. Esta señales son AWVALID, ARVALID, AWADDR, ARADDR, WVALID, RVALID, WDATA, RDATA.

Inicialmente se tenía pensado que el modelo de referencia cumpliera el comportamiento del diagrama en la figura 4.4. Pero este presentaba el inconveniente de que eliminaba por completo la funcionalidad de hacer lecturas y escrituras en ambos esclavos por parte de los maestros. La razón por la cual este modelo eliminaba la paralelidad de las funciones es porque para poder ejecutar una función diferente a la lectura por parte del maestro 1, debía de no cumplirla, y el AXI4-Lite es capaz de hacer varias funciones al mismo tiempo. Otra de las razones por las cual se descartó este diseño, es porque es un diseño muy ineficiente.

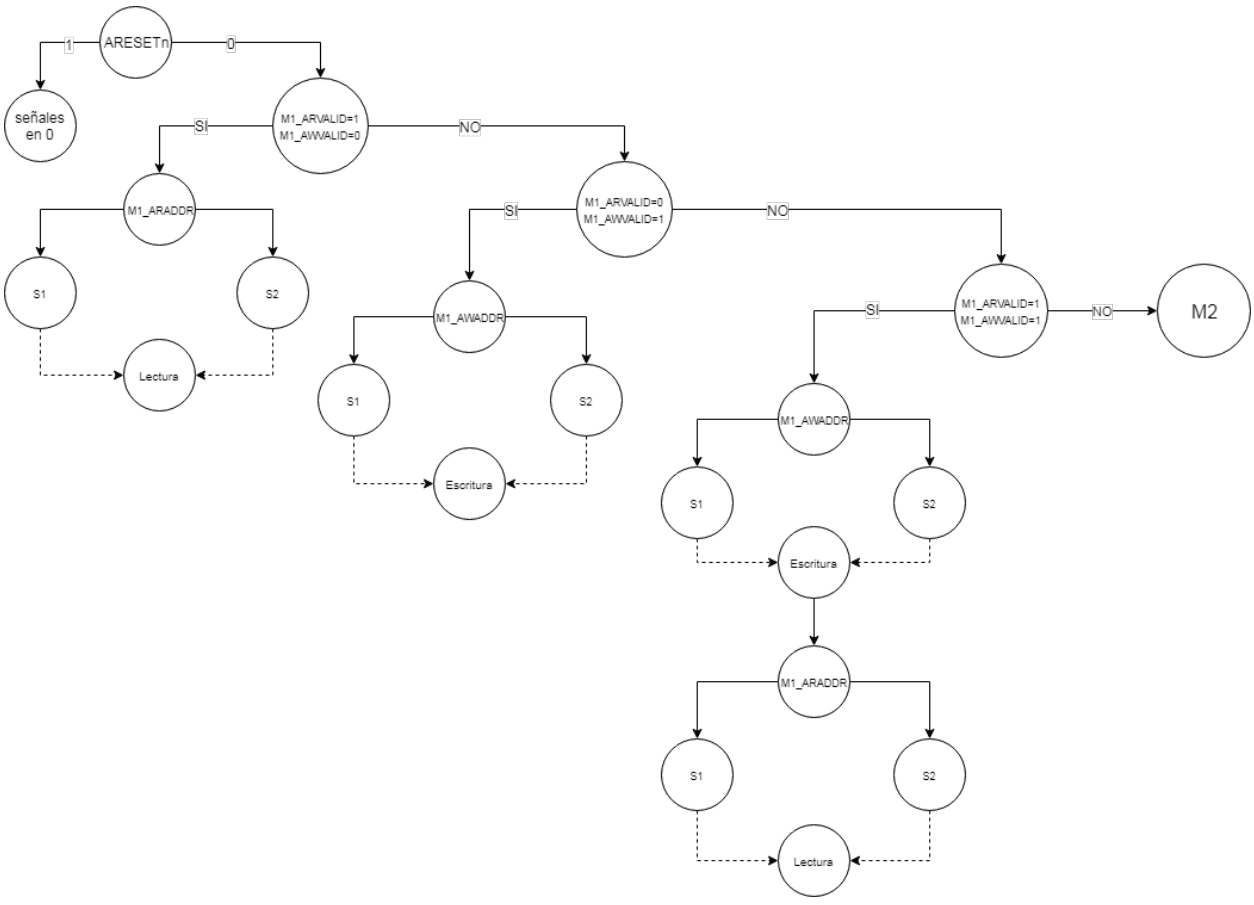

Figura 4.4: Diagrama de flujo original del modelo de referencia

Eventualmente se modificó de manera tal que cumpliera la deseada paralelidad con las funciones, como se deseaba. El resultado de estas modificaciones resultó en la funcionalidad descrita en el diagrama de flujo de la figura 3.2.

Otra parte importante del modelo de referencia es la comparación entre los datos generados

por él y los datos generados por el DUT. Esta comparación se realizó por medio del comando *a.compare(b)*, esta operación compara los valores en a con los de b y dependiendo del resultado imprime dos mensajes. En la figura 4.5 se muestra el mensaje que se imprime en el reporte cuando los datos en comparación coinciden y la figura 4.6 muestra el caso contrario.

```
UVM_INFO @ 105000: uvm_test_top.env.axi_scb [axi_scoreboard] Sent packet and received packet matched
```

Figura 4.5: Mensaje de coincidencia de datos

```
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_comparer.svh(351) @ 145000: reporter [MISCMP] Miscompare for axi_seq_item.S2_ARVALID: lhs = 'h1 : rhs = 'h0
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_comparer.svh(382) @ 145000: reporter [MISCMP] 1 Miscompare(s) for object req@755 vs. axi_seq_item@411
UVM_ERROR @ 145000: uvm_test_top.env.axi_scb [axi_scoreboard] Sent packet and received packet mismatched
```

Figura 4.6: Mensaje de no coincidencia de datos

Para probar el funcionamiento del modelo de referencia y el sistema de comparaciones se utilizó la misma prueba descrita en la sección anterior. Esto porque ya se había demostrado con anterioridad que con esta prueba se cuenta con conocimiento previo de las respuestas del DUT a dichos estímulos. La primera vez que se corre la prueba y ningún dato del modelo de referencia coincide con el del DUT, lo cual no es completamente inesperado, pues ningún diseño es 100% funcional desde el principio. La prueba arroja que los datos del modelo de referencia tienen un valor de 0. Se hacen pequeñas pruebas para determinar el error, estas consiten en colocar $\$display()$; con números para encontrar a cual ciclo *if* no se está entrando. El primer error encontrado fue que la comparación de las direcciones contaba con los símbolos de $>$ y $<$ invertidos por lo tanto nunca se cumplía la condición y no se podía entrar a la siguiente comparación. Luego, otro error encontrado fue que no se estaban agregando los datos a la cola de manera correcta.

Otro error fue que el modelo de referencia estaba guardando los datos por más tiempo que el DUT y por lo tanto, la comparación resultaba daba error.

4.3 Sistema de pruebas

El proceso de verificación no existe sin la inclusión de pruebas para el DUT.

4.3.1 Driver

Debido a que para poder correr las funciones por completo es necesario esperar a que el sistema responda para continuar, el driver fue modificado con la finalidad de que antes de recibir dichas respuestas no transmita datos entre el DUT y el ambiente. Para poder tener un control de la inicialización de las funciones se crearon ocho variables de inicio, las cuales se inicializaron en 5. Estas variables se definieron para que ellas determinen cuales señales del DUT son las siguientes en encender.

Debido a la característica del AXI4-Lite de realizar varias funciones simultáneamente, las modificaciones del driver se programaron de igual manera. El driver se seccionó en tres partes para cada función La figura 4.7 muestra el comportamiento de inicial de cada maestro para leer o escribir en el esclavo 1, se omitió el esclavo 2, pues leer y escribir en él cumple la misma lógica mostrada y el único cambio es en el prefijo de los nombres de las señales y las variables de inicialización.

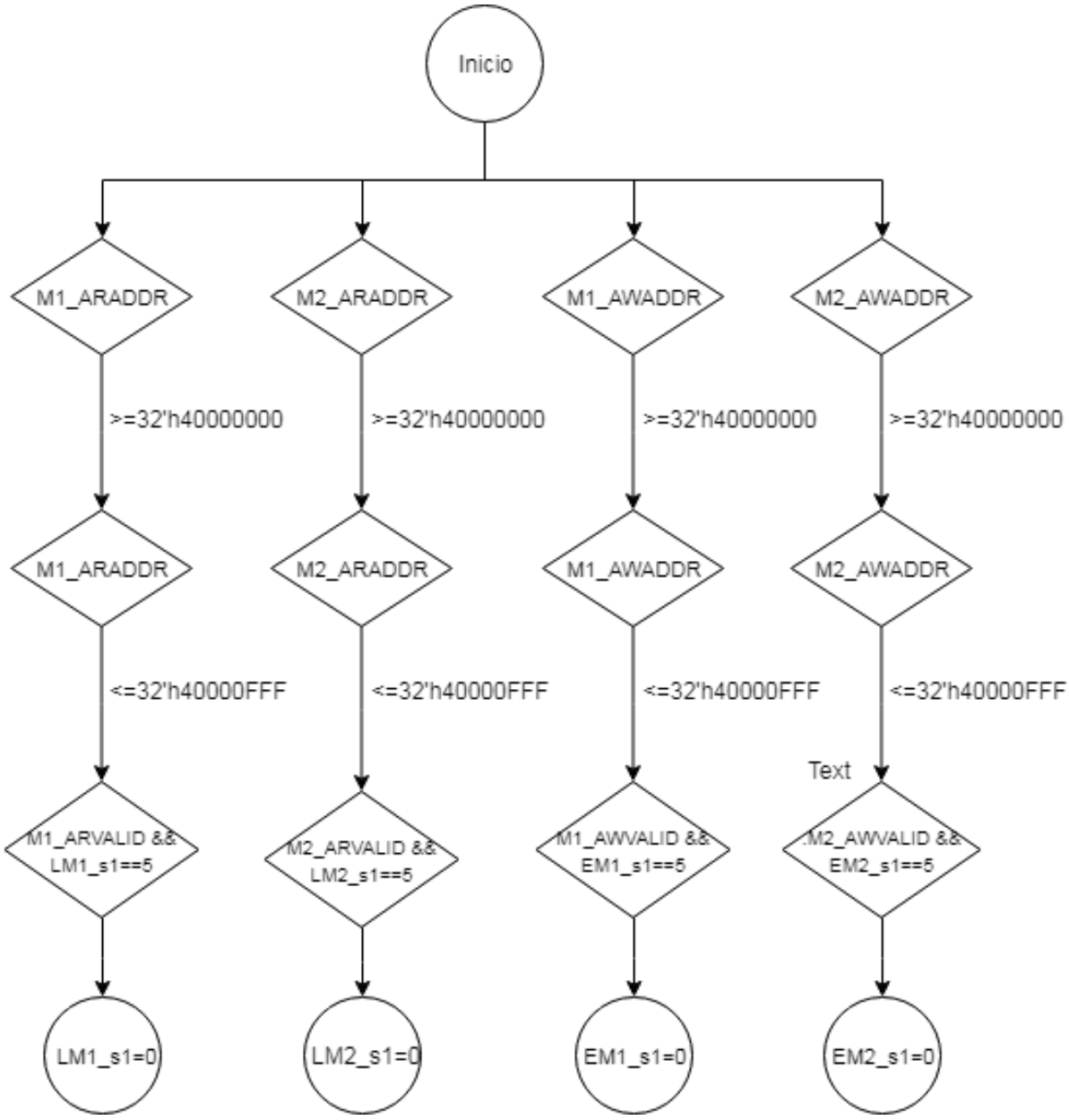

Figura 4.7: Diagrama de flujo de la primera etapa de activación en el driver

Como se puede observar la primera etapa corresponde al momento en el que se determina cual es la función que se desea ejecutar y a cual esclavo. Como las señales de VALID y la de ADDR (dirección) entran simultáneamente, se evalúa primero a cual esclavo se desea leer o escribir, y luego se verifica que el VALID si se haya activado y que la variable de inicio se encuentre en 5, cumplidas estas condiciones se pasa la variable a 0, para entrar a la etapa

de lectura o escritura, las cuales se muestran en las figuras 4.8 y 4.9.

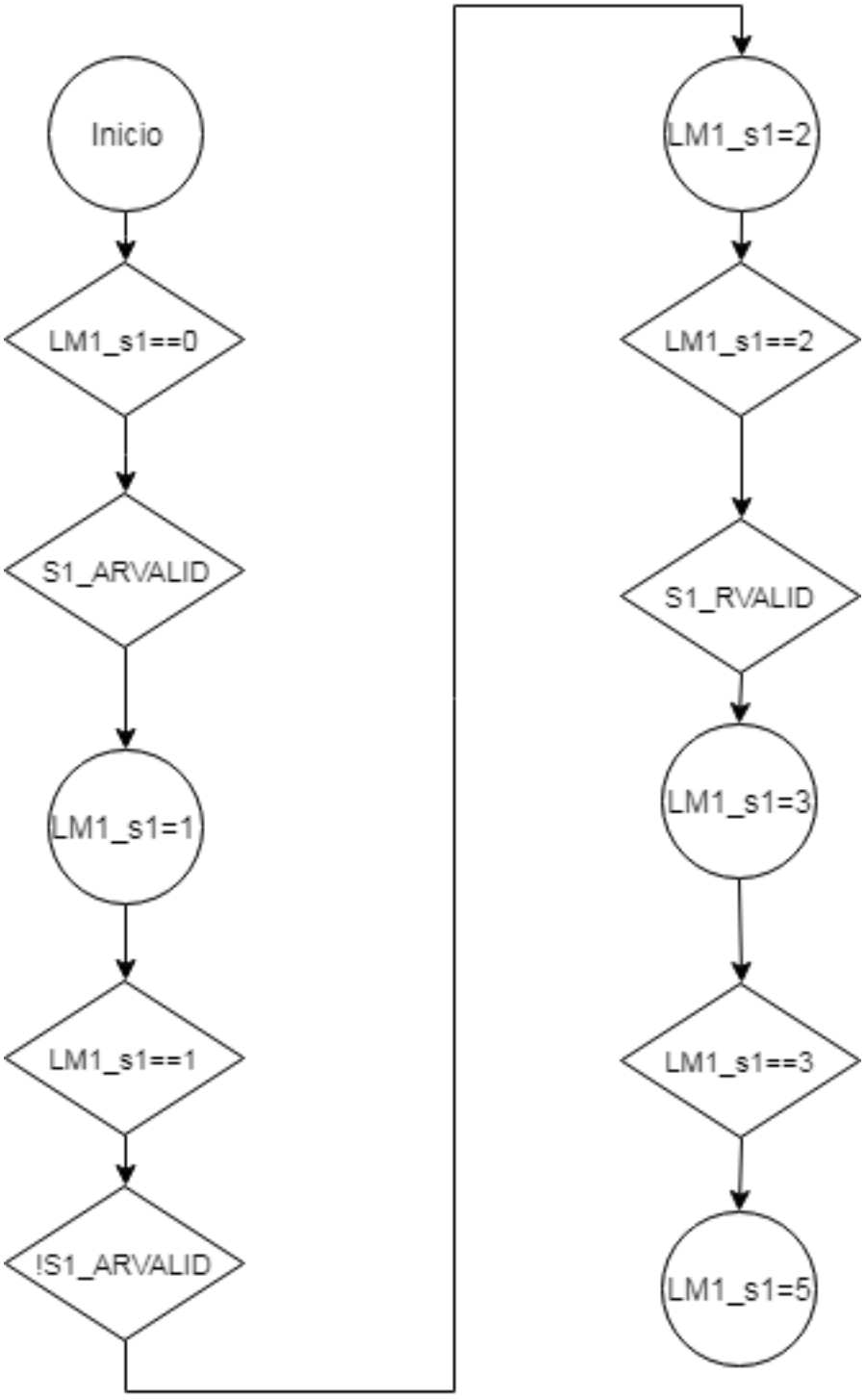

Figura 4.8: Diagrama de flujo de la segunda etapa del driver, para la función de lectura

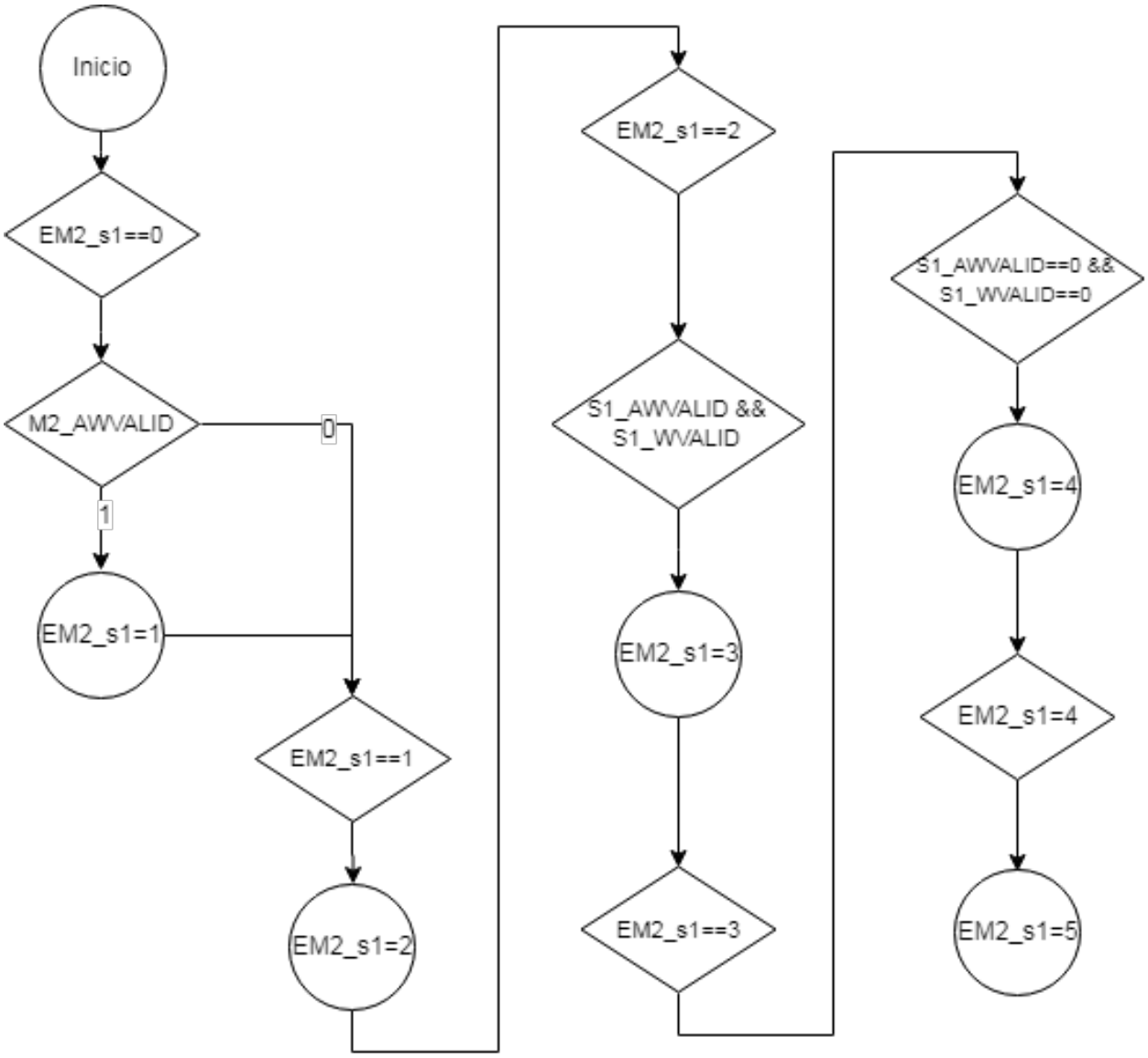

Figura 4.9: Diagrama de flujo de la segunda etapa del driver, para la función de escritura

Ambos diagramas siguen una línea de lógica similar, la única diferencia apreciable es que a la lógica de escriturase agrega un "*else*" para obtener un tiempo de espera puesto que la función de escritura requiere de ingresar datos un ciclo después de los primeros. En general ambos diagramas lo que indican son las señales de salida que deben de activarse antes de activar las siguientes señales de entrada. Por último en las figuras 4.10 y 4.11 se muestran cuáles señales deben de activarse en que momento.

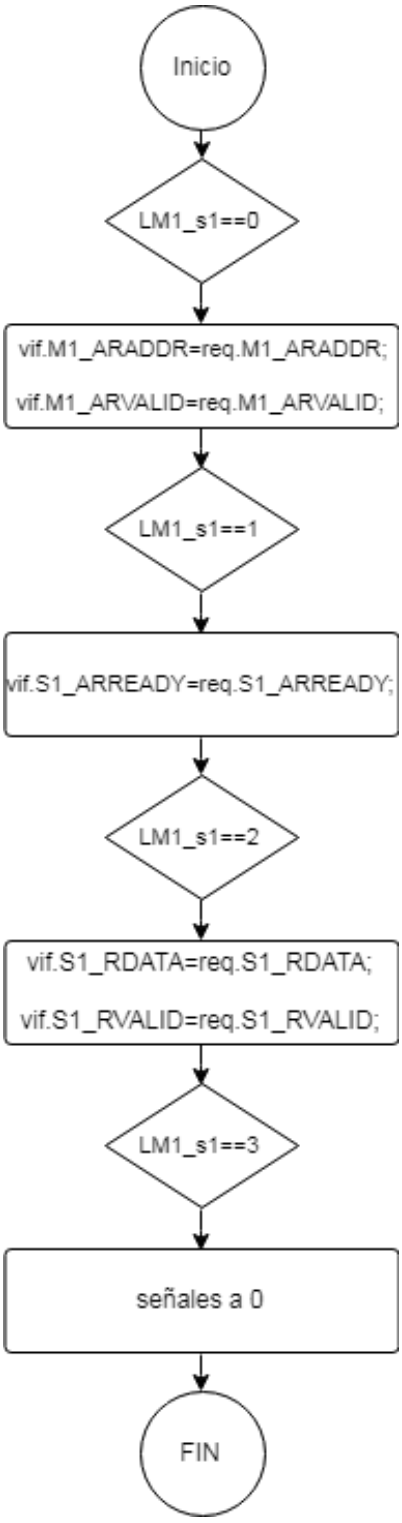

Figura 4.10: Diagrama de flujo de la última etapa del driver, para la función de lectura

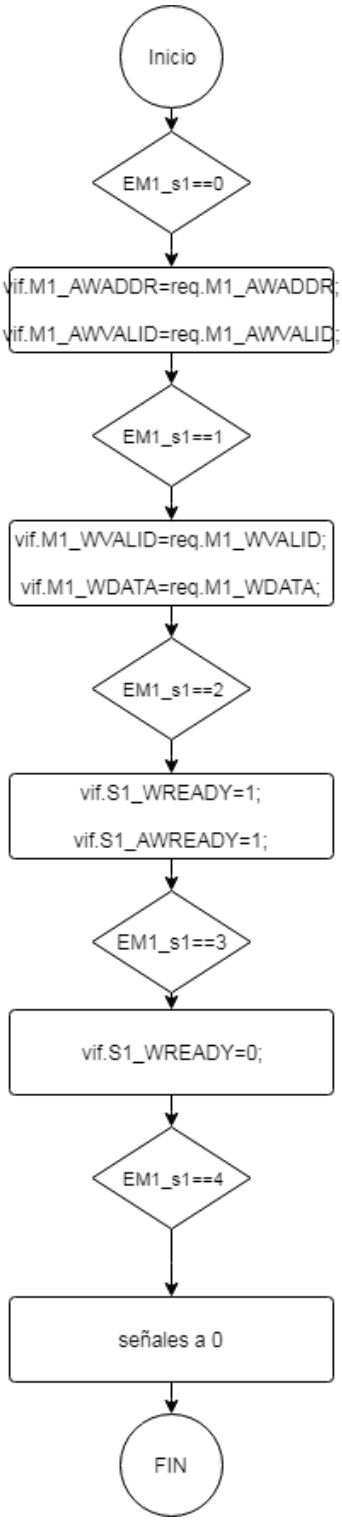

Figura 4.11: Diagrama de flujo de la última etapa del driver, para la función de escritura

4.3.2 Pruebas

El ambiente cuenta con dos pruebas principales, una de ellas que se varía de manera manual y prueba una sola iteración de la función que se desee revisar y la otra que es completamente aleatoria. La prueba manual inició siendo la prueba mencionada previamente para probar el funcionamiento de las interfaces. Pero, a lo largo del desarrollo del proyecto, fue modificada con el fin de examinar el comportamiento del DUT y el modelos de referencia a combinaciones específicas pero sencillas, por ejemplo, ambos maestros leyendo el esclavo 1 o ambos maestros escribiendo el esclavo 2. En una inicio ambas pruebas estaban definidas dentro de la misma secuencia, pero para poderlas llamar sin la necesidad de poner en comentario a la otra, se decidió colocar cada una como una secuencia aparte.

Ambas pruebas para funcionar necesitan la creación de un archivo aparte que las llame. Para la prueba aleatoria el archivo *axi_random_test.sv* es quien la llama a correr y para la prueba manual es el archivo *axi_test_lectura.sv*. El segundo tiene dicho nombre debido a que originalmente la prueba manual solo corría la función de lectura.

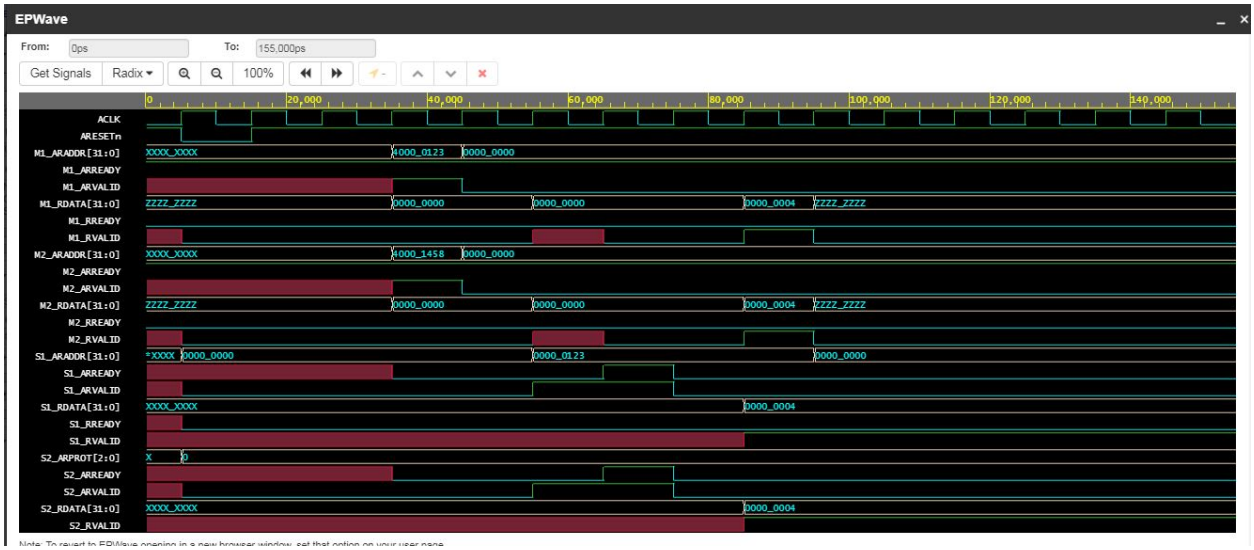

Figura 4.12: Forma de onda de las señales del DUT para la función de lectura

En la figura 4.12 se muestran las señales que viajan por las interfaces entre el DUT y el ambiente de verificación. La prueba mostrada es la prueba manual en función lectura. La prueba se configuró de manera que el maestro 1 realizara una lectura en el esclavo 1 y el maestro 2 leyera al esclavo 2. Luego, cada esclavo responde con un 5 y un 4, respectivamente. Las señales que se observan en rojo son señales que no están conectadas al sistema por el momento, una vez necesarias se conectan para que pasen por ellas la información solicitada. Por ejemplo, los RVALID de los esclavo son necesarios hasta el envío de información a los maestros, por lo tanto no se conectan hasta dicho momento. Las señales de escritura no se muestran en pantalla ya que al tratarse de lecturas solamente dichas señales se mantendrían desconectadas (en rojo).

Para obtener una visualización de las señales cuando se programa la prueba random, ésta se

hizo correr, primero 10000000 veces y luego solamente 10 veces. La figura 4.13 muestra la primera corrida y la figura 4.14 muestra la corrida de 10 veces.

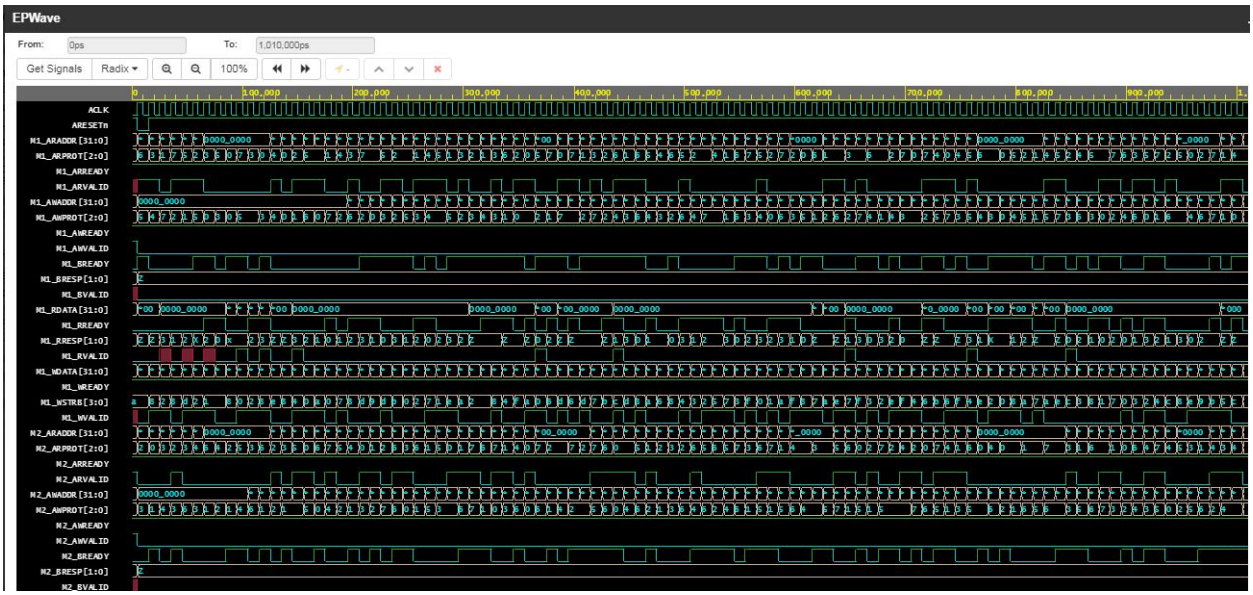

Figura 4.13: Forma de onda de las señales del DUT para la función random(10000000)

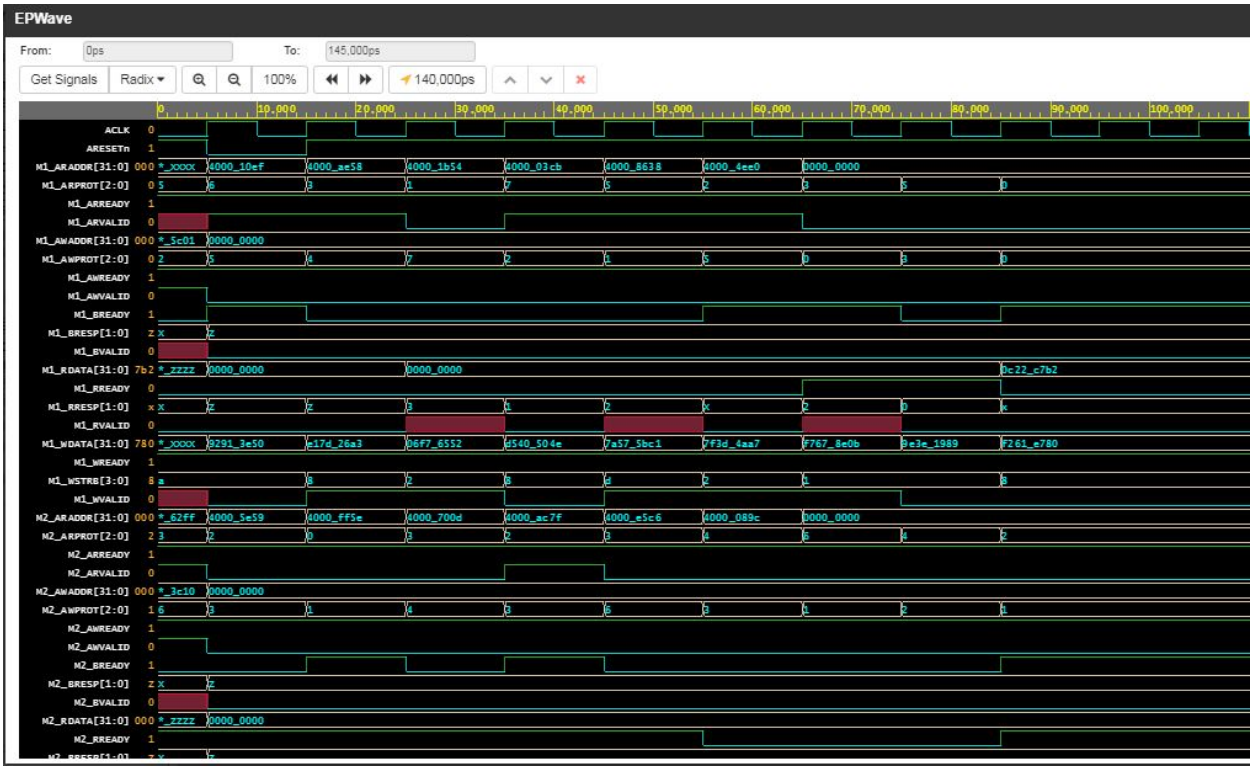

Figura 4.14: Forma de onda de las señales del DUT para la función random(10)

Por último también se corrió una prueba de solamente escritura, la figura 4.15 ilustra el resultado de dicha prueba.

La prueba consistió en, al igual que la lectura, generar una escritura por cada maestro a un esclavo, con la diferencia que se escribió el mismo dato por parte de los maestros.

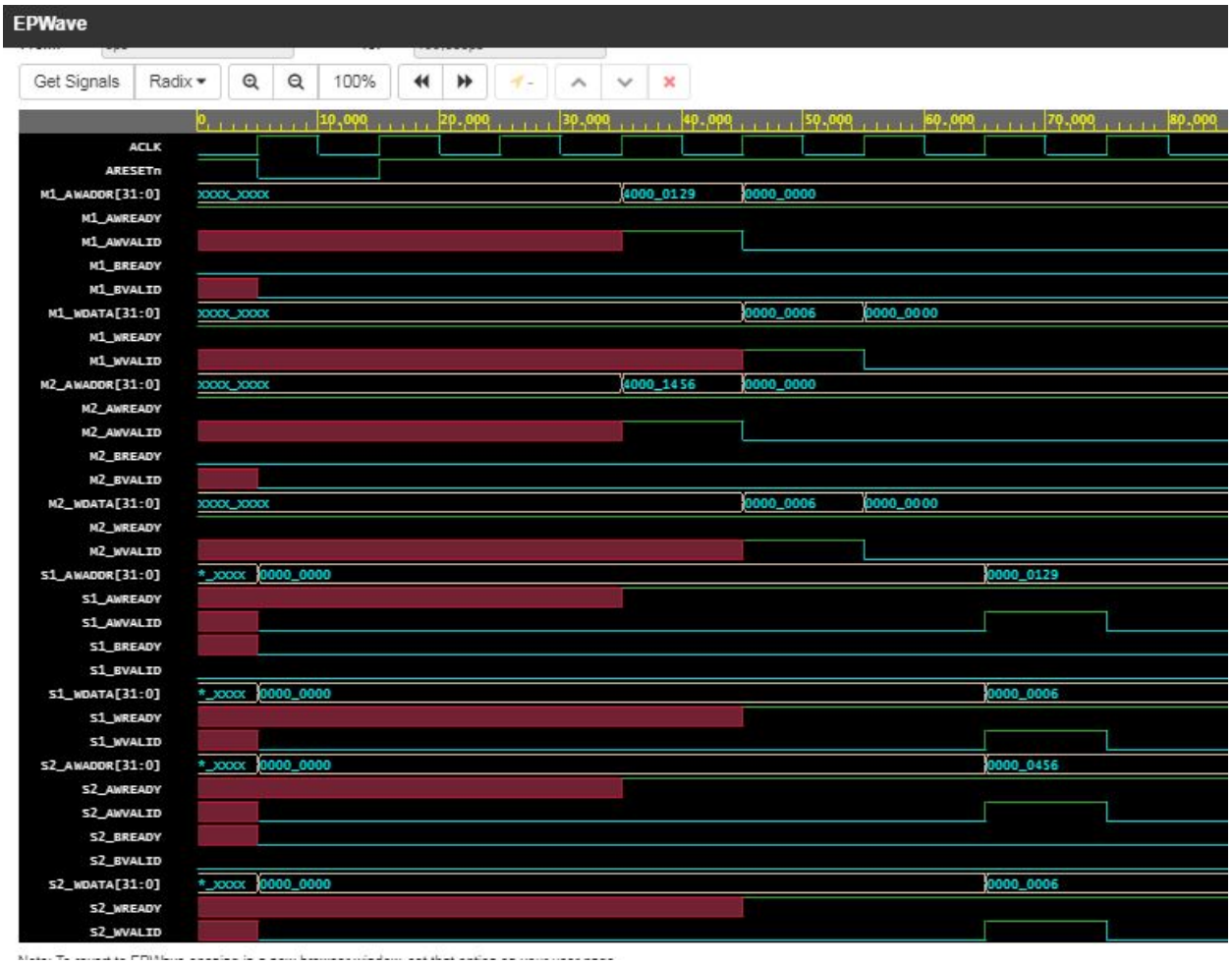

Figura 4.15: Forma de onda de las señales del DUT para la función de escritura

En las pruebas de lectura y escritura solamente se colocaron las señales relevantes a la función.

Al momento de crear el archivo para la segunda prueba, este no fue instanciado dentro del testbench, por lo que cuando se intentó correr el sistema tiraba error fatal. Antes de determinar que el problema recaía principalmente a la falta de instanciación, se notó que el archivo creado no contaba con la extensión correcta.

En el momento en el que se genera un error el programa da por fallida la prueba y por lo tanto muestra el mensaje de la figura 4.16. Los errores en caso de que no exista coincidencia entre los datos del DUT y del modelo de referencia generan este mensaje de error.

```
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_objection.svh(1270) @ 155000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO axi_tests.sv(49) @ 155000: uvm_test_top [axi_test_lectura] ----------------------------------------
UVM_INFO axi_tests.sv(50) @ 155000: uvm_test_top [axi_test_lectura] ----           TEST FAIL         ----
UVM_INFO axi_tests.sv(51) @ 155000: uvm_test_top [axi_test_lectura] ----------------------------------------
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_report_server.svh(847) @ 155000: reporter [UVM/REPORT/SERVER]
```

Figura 4.16: Mensaje de que la prueba falló

En cambio cuando no se detecta ningún error durante la compilación y simulación, el mensaje es de que la prueba se realizó con éxito y que el DUT pasó dicha prueba. Este mensaje es el mostrado en la figura 4.17

```
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_objection.svh(1270) @ 155000: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO axi_tests.sv(54) @ 155000: uvm_test_top [axi_test_lectura] ------------------------------------
UVM_INFO axi_tests.sv(55) @ 155000: uvm_test_top [axi_test_lectura] ----          TEST PASS          ----
UVM_INFO axi_tests.sv(57) @ 155000: uvm_test_top [axi_test_lectura] ------------------------------------
UVM_INFO /apps/vcsmx/etc/uvm-1.2/src/base/uvm_report_server.svh(847) @ 155000: reporter [UVM/REPORT/SERVER]
```

Figura 4.17: Mensaje de que la prueba fue exitosa

Las pruebas aplicadas deberían de cubrir todas las posibles combinaciones de entradas, esto con el fin de establecer el funcionamiento correcto del DUT. Esto busca hallar la existencia de una posible combinación que genere una falla en el código y pueda ser perjudicial en su aplicación futura. Se corrieron 1000 pruebas aleatorias con el objetivo de obtener un porcentaje de cobertura preliminar. Es un dato preliminar debido a que aún no se han establecido en [8] cual es el porcentaje de cobertura al que debe de llegar el proceso de verificación del AXI4-Lite. De estas 1000 pruebas se obtuvieron los siguientes porcentajes de covertura.

Tabla 4.1: Porcentajes de cobertura a 1000 pruebas

Entrada	Porcentaje de cobertura
S1_ARADDR	0.001079
S1_AWADDR	0.001079
S1_RDATA	0.004316
S1_WDATA	0.001079
S2_ARADDR	0.002158
S2_AWADDR	0.001079
S2_RDATA	0.004316
S2_WDATA	0.001079

Los porcentajes obtenidos son muy bajos, se encuentran muy por debajo del 1%. Esto se debe a que la cantidad de pruebas aplicadas es extremadamente bajo en comparación a la cantidad de combinaciones posibles. Al ser todos datos de 32 bits la cantidad de combinaciones es de $4.951 * 10^{27}$ por lo que mínimo es necesario correr la misma cantidad de pruebas, lo cual es imposible debido que la página no es capaz de soportar dicha cantidad. Para correr dicha cantidad es necesario de un servidor que pueda correr pruebas por días.

Capítulo 5

Conclusiones y Recomendaciones

5.1 Conclusiones

Los reportes de compilación demuestran una buena implementación de un ambiente de verificación tipo UVM para un bus de datos AXI4-Lite.

Se logró obtener un modelo de referencia robusto para el AXI4-Lite que emula el comportamiento básico de dicho bus de datos.

Se diseñaron e implementaron dos pruebas base para la prueba de funcionamiento del AXI4-Lite.

Debido a la alta cantidad de pruebas necesarias para poder obtener una buena cobertura en la verificación del AXI4-Lite es necesaria la migración a un servidor con la capacidad de correr el programa por lapsos de días.

5.2 Recomendaciones

Se recomienda parametrizar el ambiente, con el objetivo de probar la característica de parametrización del AXI4-Lite.

Se aconseja diseñar y aplicar una mayor cantidad de pruebas.

Se recomienda depurar y parametrizar el modelo de referencia.

Es necesaria la ampliación de la cobertura por parte del ambiente al AXI4-Lite, por lo que es necesario continuar con las pruebas y de esta manera obtener una buena cobertura y generar una regresión a base de ella.

Se recomienda realizar una migración al servidor, con el fin de generar una regresión, con las pruebas que se apliquen y los *bugs* que se encuentren a partir de ellas.

Bibliografía

[1] accellera. Universal Verification Methodology (UVM) 1.2 User's Guide [online]. 2015. URL http://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[2] P. Aggarwal. Introduction To System Verilog [online]. URL http://www.asicguru.com/system-verilog/tutorial/introduction/1/.

[3] ARM. MAMBA AXI and ACE Protocol Specification, 2013.

[4] Doulos. EDA Playground [online]. URL https://www.edaplayground.com/.

[5] Doulos. What is SystemVerilog? [online]. URL https://www.doulos.com/knowhow/sysverilog/whatissv/.

[6] C. Duran et al. A 32-bit risc-v axi4-lite bus-based microcontroller with 10-bit sar adc, 2016. URL https://ezproxy.itcr.ac.cr:2474/stamp/stamp.jsp?tp=&arnumber=7451073.

[7] Verification Guide. Uvm testbench example [online]. URL http://www.verificationguide.com/p/uvm-testbench-example.html.

[8] R. Molina. CR_TEC_RISCV [online]. URL https://docs.google.com/document/d/1gcME_wfzyNn1Q1ZkhdJeJmecnod1TrHKlhpaqSYdueA/edit#.

[9] S. Rosenberg and K. A. Meade. *A Practical Guide to Adopting the Univerdal Verification Methodology(UVM)*. Cadence Design Systems, Inc., 2010.

[10] C. Spear. *SYSTEMVERILOG FOR VERIFICATION*. Springer, 2006.

[11] Tech Design Forum. Verification coverage [online]. URL http://www.techdesignforums.com/practice/guides/verification-coverage/.

[12] A. Waterman and K. Asanovic. pecifications Risc-V Foundation: The RISC-V Instruction Set ManualVolume I: User-Level ISA Version 2.2, 2017.

[13] B. Wile, J. C. Goss, and W. Roesner. *Comprehensive Functional Verification*. Morgan Kaufmann Publishers, 2005.

[14] Xilinx. AXI Reference Guide [online]. 2011. URL https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

Apéndice A

Hoja de Información

A.1 Información del estudiante

Nombre: Irene Beatriz Rivera Arrieta

Cédula: 115270510

Carné ITCR: 201121803

Dirección de su residencia en época lectiva: Guadalupe, Goicoechea, San José. Urbanización Villa Capri Casa # 1

Dirección de su residencia en época no lectiva: Guadalupe, Goicoechea, San José. Urbanización Villa Capri Casa # 1

Teléfono en época lectiva: 8981-2235

Teléfono en época no lectiva: 8981-2235

Email: betz.93@gmail.com

Fax: N/A

A.2 Información del proyecto

Nombre del proyecto: Creación de un ambiente de verificación usando UVM para un bus AXI4-Lite para una arquitectura RISC-V de 32 bits.

Profesor asesor: Roberto Molina Robles

Horario de trabajo del estudiante: L-M, V: 9:00am a 12md, 1pm a 4:30pm.

A.3 Información de la Empresa

Nombre: Escuela de Ingeniería en Electrónica

Zona: N/A

Dirección: Calle 15, Avenida 14. 1 km Sur de la Basílica de los Ángeles

Teléfono: 2552-5333

Fax: N/A

Apartado: 159-7050

Actividad Principal: Educación.