INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERIA ELECTRÓNICA

Diseño de un ambiente de verificación basado en la metodología UVM para un microprocesador RISC-V 32I

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

Daniel Rojas Chacón

Cartago, Setiembre 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 Diseño de un ambiente de verificación basado en la metodología UVM para un microprocesador RISC-V 32I, realizado por el señor Daniel Rojas Chacón 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 Ruiz Barquero

Profesor lector

Ing. Roberto Molina Robles

Profesor asesor

3 de setiembre de 2018

Declaración de Autenticidad

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

En los casos en que he utilizado material bibliográfico, he procedido a indicar las fuentes mediante citas.

En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Daniel R.

Daniel Rojas Chacón

Cédula: 115560668

Resumen

La escuela de ingeniería electrónica del Instituto Tecnológico de Costa Rica actualmente posee un proyecto de investigación junto con otras universidades ubicadas en Sur América y Europa, que consiste en el diseño de un microprocesador basado en la arquitectura RISC-V que se plantea usar en aplicaciones médicas para dispositivos implantables como marcapasos o prótesis. Debido a esto es necesario la implementación de pruebas que verifiquen el funcionamiento de cada parte o bloque que posee el microprocesador en su interior antes de la fabricación de este.

Para hacer esas pruebas se requiere de un ambiente de verificación funcional, el cual realiza una simulación donde se envían señales de excitación al modelo RTL y a un modelo de referencia del dispositivo bajo prueba, posteriormente se compara ambas respuestas y se chequea en cuales de los casos ocurre un error y así determinar la falla en el diseño que ocasiona esos problemas.

Para realizar el ambiente de verificación fue necesario utilizar una metodología que permitiera la estandarización en el diseño y la implementación por lo que se utilizó la metodología UVM (Universal Verification Methodology), por lo tanto en este proyecto se describen las distintas etapas de diseño y pruebas hechas al ambiente.

Palabras Clave: *Ambiente de Verificación, Backdoor, DUV, RISC-V, Scoreboard, Testbench, Testing, UVM, Verificación Funcional.*

Abstract

The electronics engineering school in the Technology Institute of Costa Rica has now an investigation project with other universities in South America and Europe, they have been doing a design of a microprocessor wich is based on RISC-V architecture and its proposed to be used in medical applications to implanted devices as pacemakers or protesis. For that is necessary the implementation of test that verify the performance of any part inside of the microprocessor prior the fabrication.

A functional verification environment should exist to send signals as needed to RTL and Reference Models of the device under test, later on it does the comparison between both responses and determines if ther is an error or not, so then if there is a design fault that is causing the issues.

To make the verification environment it was necessary to use a methodology that allows the standardization in the design and the implementation, so it's used the UVM methodology (Universal Verification Methodology) so in this project is described the stages of the design and the test do in the environment

Keywords: *Verification Environment, Backdoor, DUV, RISC-V, Scoreboard, Testbench, Testing, UVM, Functional Verification.*

Dedicatoria

Este proyecto de graduación lo dedico principalmente a mi Dios por cumplir su propósito en mi vida y le agradezco por permitirme concluir con éxito esta meta. En segundo lugar, dedico este trabajo a mis papás (Wilberth Rojas y Helen Chacón) y a mis hermanas (Laura Rojas y Mariela Rojas) por motivarme a seguir adelante cuando hubieron pruebas difíciles en mi vida, por orar por mí cuando tuve necesidad y por creer en mi cuando otros no lo hicieron.

Por último, quiero dedicar y hacer un agradecimiento especial a mis abuelos: Edwin Chacón quien me ayudo en gran manera durante sus últimos años de vida y estoy seguro de que estará orgulloso en el cielo al ver que logré cumplir esta meta que él vio iniciar, también a mi abuela Noemy Bolaños por ayudarme durante todo mi tiempo universitario cuando más lo necesité y por abrirme las puertas de su casa cuando no tuve donde estar.

Salmos 119:49-50 Nueva Traducción Viviente (NTV)

Recuerda la promesa que me hiciste; es mi única esperanza.

Tu promesa renueva mis fuerzas; me consuela en todas mis dificultades.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

La utilización de dispositivos electrónicos para aplicaciones médicas debe de asegurar que los componentes de hardware estén en perfectas condiciones, debido a que un fallo en estos puede generar peligros a la vida de las personas.

Por lo tanto, se necesita cumplir con estándares que garanticen el funcionamiento correcto durante su ciclo de vida y así evitar fallas que puedan poner en riesgo la vida o la integridad física de los usuarios. Cuando esto no se cumple es necesario hacer una intervención para cambiar el dispositivo electrónico y de esta manera evitar mayores complicaciones en la persona, sin embargo esto provoca el inconveniente de que se deban de realizar operaciones o procedimientos que aumentan el riesgo para la salud.

Por otra parte cabe mencionar, que una de las posibles causas de errores en cualquier dispositivo son las fallas que vienen desde el momento del diseño o fabricación del producto. Debido a esto es necesario la implementación de sistemas que verifiquen el funcionamiento de las partes que conforman los sistemas electrónicos digitales, circuitos integrados y demás componentes involucrados en los diseños de los dispositivos médicos.

Este proyecto se centró en la importancia de la verificación funcional de circuitos integrados utilizados para las aplicaciones médicas, ya que esto es una herramienta para conocer los errores cometidos durante el diseño RTL y así corregirlos antes de su fabricación en chip.

Específicamente la utilización de etapas de verificación tiene gran relevancia en los sistemas de gran complejidad como es el caso de los microprocesadores debido que las operaciones que ejecuta son críticas y existe mayor probabilidad de la ocurrencia de errores durante la ejecución de la tarea.

En el caso del proyecto de investigación, se puede decir que es de mucha importancia empezar con la verificación funcional de los distintos elementos del microprocesador como lo son la ALU, memorias, controladores, buses y también el full chip; ya que se facilita la comprobación del funcionamiento de cada uno de esos bloques al mismo tiempo que se va haciendo la etapa de diseño y permitiendo así acortar el tiempo para la corrección de los errores.

Es así que surge la necesidad de la realización de un ambiente de verificación que compruebe el funcionamiento del sistema en general, es decir, cuando todos los bloques funcionales del microprocesador son conectados entre sí pueden generarse problemas que afecten el desempeño general del microprocesador.

De todo lo anterior, se puede decir que una de las etapas críticas en el diseño del micro-

procesador es la realización de un ambiente de verificación que se adapte a la unidad bajo prueba, en este caso al funcionamiento general del microprocesador y actualmente no se tiene un ambiente de verificación que se pueda utilizar, por lo que es necesario desarrollarlo.

El ambiente de verificación realiza una simulación donde se compara la respuesta del modelo RTL del D.U.V., con respecto a la respuesta de un módulo de referencia, el cual debe tener el funcionamiento "ideal" del microprocesador. Con esto se busca determinar los casos de operación del dispositivo en donde ocurre un error y así se puede determinar la falla en el diseño que ocasiona esos errores.

Por otro lado, el ambiente idealmente debe de simular todos los escenarios a los cuales se puede enfrentar el D.U.V., para así detectar la falla y poder informar en que situación se dio el problema.

Cabe mencionar que existen varios niveles de abstracción o de división que puede tener un dispositivo y los cuales se pueden observar en la Figura 1, cada uno puede poseer su propio ambiente de verificación, los más importantes para el desarrollo del proyecto de investigación en el diseño del microprocesador son: el nivel de sistema, el cual permite ver el funcionamiento del dispositivo médico implantable de manera general y en donde solo importa que el comportamiento sea el correcto. El Nivel de tarjeta, permite dividir en secciones de hardware que tienen una función específica como por ejemplo la comunicación, el control, la adquisición de datos, entre otras. El nivel de chip, permite ver el sistema en componentes electrónicos que desempeñan funciones de almacenamiento, transmisión, cálculo y operaciones de datos.En el caso del proyecto, el microprocesador se puede categorizar de esta manera por ser un componente electrónico. Por último el nivel de unidades, en donde se puede ver cada componente electrónico en los bloques funcionales internos. En el caso del microprocesador se pueden mencionar: la ALU, Unidad de Control, DMA, Registros, UART, Buses, entre otros.

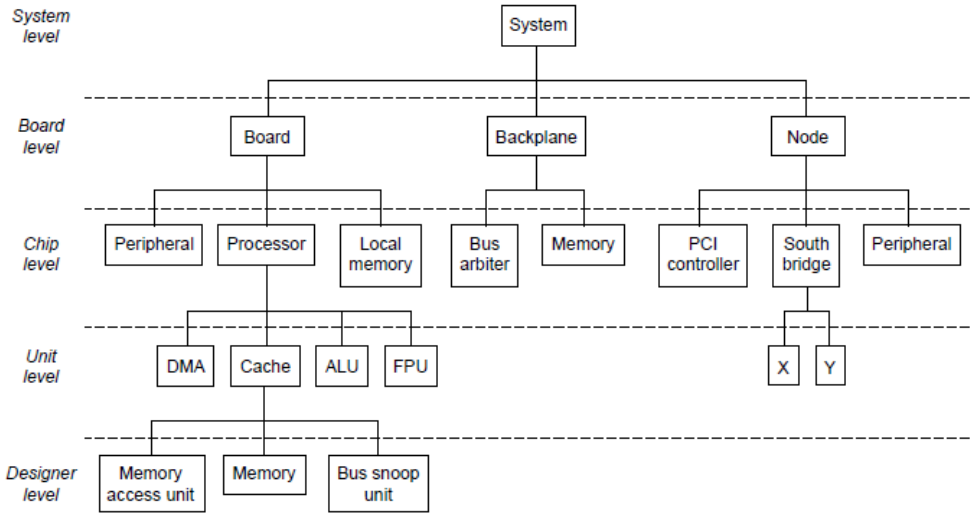

Figura 1: Niveles de abstracción para la verificación funcional

Según la imagen anterior, se puede observar que un microprocesador se encuentra en la

categoría de chip es decir en un nivel intermedio de la jerarquía, por lo tanto es recomendable empezar la verificación en niveles inferiores de la jerarquía de hardware para asegurar que cada bloque interno funcione de manera correcta.

Una vez hecho esto se sube de nivel en la jerarquía y se realiza la verificación funcional del microprocesador. La ventaja de realizar la verificación funcional primero en niveles más bajos de la pirámide y después en niveles superiores, es que permite determinar de una mejor manera el o los componentes que posee los errores de funcionamiento.

Por otra parte, existen metodologías de verificación para asegurar que se cumpla con un procedimiento establecido o estándar, sin embargo, el desarrollo del área de verificación funcional en la ingeniería electrónica es muy reciente, hace 10 años aproximadamente empezó a tener importancia esta rama del diseño VLSI y es por esto que existen pocas metodologías de verificación, entre las más utilizadas son: la OVM (Open Verification Methodology) y la UVM (Universal Verification Methodology). Se puede decir que la UVM es la "evolución" de la OVM por lo tanto es la que más se utiliza en la actualidad.

Además, la utilización de una metodología para la verificación como lo es la UVM y SystemVerilog, es de gran importancia debido a que permite la estandarización en la implementación del proyecto y con esto se puede hacer cosas que facilitan el diseño del ambiente de verificación, por ejemplo la creación de bloques genéricos que permiten ser utilizados en otros ambientes, también permite la creación de bloques específicos que permiten adaptar el ambiente al funcionamiento del D.U.V. que se quiera verificar.

Por otra parte, se pueden utilizar librerías que permitan la simplificación de la descripción e implementación de esos bloques funcionales que debe poseer el ambiente de verificación.

Capítulo 2

Meta y Objetivos

Meta:

Generar los distintos ambientes de verificación para los bloques internos y el full chip de un microprocesador de arquitectura RISC-V utilizado en aplicaciones médicas.

Indicador: Simulaciones de cada bloque funcional del microprocesador sin errores de funcionamiento.

Objetivo General:

Desarrollar un ambiente de verificación para el full chip de un microprocesador de arquitectura RISC-V utilizando la metodología UVM.

Indicador: Simulaciones de las pruebas sobre el ambiente de verificación hecho en la plataforma EDAPlayground

Objetivos Específicos:

- Construir los bloques genéricos de un ambiente de verificación basado en la metodología UVM.

  Indicador: Simulación de los bloques genéricos del ambiente de verificación con un porcentaje de éxito de 100

- Diseñar el modelo de referencia del full chip del microprocesador, instrucciones de prueba, los componentes para el envío de instrucciones hacia el dispositivo bajo verificación y el monitor para la observación de las transacciones en la interface SPI del microprocesador.

  Indicador: Simulaciones sobre el modelo de referencia, el envío de instrucciones sin retrasos en el diagrama de tiempo y envío de instrucciones al modelo de referencia.

- Diseñar el modelo de registros para la lectura de datos en el banco de registros del microprocesador.

  Indicador: Comprobación de lecturas en registros del DUV utilizando acceso por backdoor, sin la existencia de daño en los datos leídos.

Capítulo 3

Marco Teórico

3.1. Pruebas en dispositivos

Cuando un componente de hardware realiza una tarea especifica pueden existir distintos detonadores que provoquen un mal funcionamiento del sistema. Algunas veces estos errores pueden ser provocados por factores externos como lo son las condiciones a las que esta sometido el medio, otras ocasiones pueden ser generados durante la fabricación o en las etapas de diseño y es por eso que existe la necesidad de realizar pruebas para la identificación de esos problemas que vienen por defecto en el dispositivo.

Según lo anterior es de gran importancia la realización de diferentes tipos de test durante el ciclo de fabricación de chips [1], ya que estas herramientas permiten ubicar el origen de los problemas que afectan el comportamiento de los sistema y de esta manera los diseñadores o los fabricantes puedan tener los antecedentes para hacer las correcciones antes de la comercialización de los productos electrónicos.

La Figura 2 refleja que los test se pueden dividir en dos tipos según la ubicación en el ciclo de fabricación:

- El primer tipo de test se denomina pruebas pre-silicon o también llamada Verificación Funcional estas tienen el objetivo de buscar los errores debidos a fallos en el diseño RTL del dispositivo y para esto es necesario la implementación de un ambiente de verificación que modela el comportamiento de un elemento ante los eventos que le pueden ocurrir durante la ejecución de las distintas tareas para las que fue diseñado ya que dichos acontecimientos pueden provocar que exista un fallo. Según [2] "La verificación del diseño es un análisis predictivo que asegura que el diseño sintetizado realizará las funciones requeridas cuando esté fabricado.". Este proceso es iterativo por lo que los diseñadores deben hacer modificaciones cada vez que el proceso de verificación encuentre un defecto.

- El segundo tipo de test es el llamado pruebas pos-silicon, se efectúan una vez hecha la verificación del diseño y fabricado el primer prototipo. La razón de hacerlas es que durante la fabricación puede existir dos causas de fallos: la primera es la perdida de rendimiento catastrófica la cual es causada por defectos aleatorios, sin embargo esta causa se ha reducido por las detecciones de fallos durante los procedimientos de

manufactura. La segunda causa es la perdida de rendimiento paramétrico y es originada por fluctuaciones durante el proceso de la fabricación. Para poder determinar que un dispositivo no posea ningún defecto debido a esas dos causas existen las pruebas DFT, estas son diseñadas según las especificaciones del dispositivo. [2]

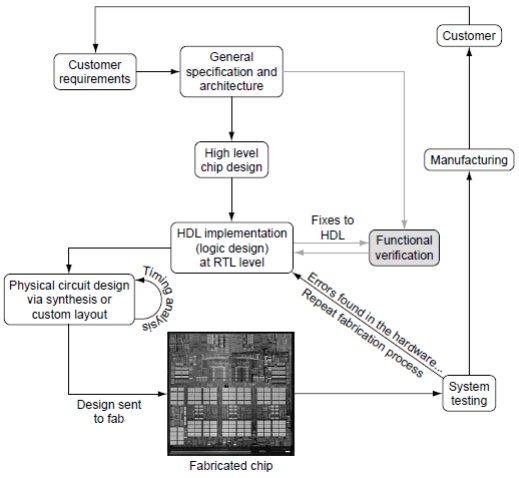

Figura 2: Ciclo de fabricación y pruebas en chips

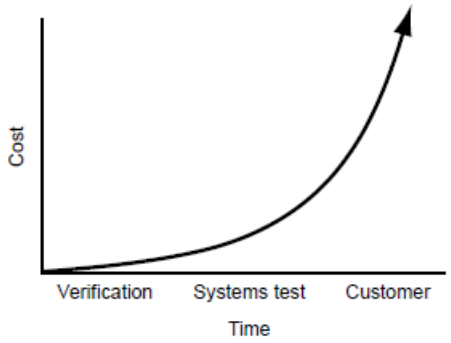

Figura 3: Costo para corregir errores

Una de las ventajas de realizar este tipo de pruebas en los circuitos electrónicos es que permite una reducción en los costos debidos a re-fabricación del componente cuando se encuentran errores [1], esto se puede observar en la Figura 3. Además según [1] "Las fuentes de los costos de verificación se dividen en tres área: costos de ingeniería, herramientas de simulación y diseño , y tiempo.".

Por otra parte, en la actualidad se ha generado la necesidad de utilizar metodologías que permitan estandarizar las pruebas que se deben de hacer tanto en las etapas de verificación funcional como en las etapas de testing pos-silicon y cabe aclarar que muchas veces las empresas tienen sus propias metodologías, sin embargo también existen las libres.

3.2. Ambiente de verificación:

Como se observó anteriormente, una etapa vital durante el ciclo de fabricación de circuitos electrónicos es la realización de pruebas durante el diseño. De ahí surge el concepto de verificación funcional, según [1] esta asegura que el diseño cumple con las tareas destinadas por la arquitectura global del sistema, es decir comprueba que se cumplen con todas las funcionalidades que se contemplaron en las especificaciones.

Para hacer la verificación funcional es necesario el diseño de un ambiente de verificación, el cual es según [1] "El conjunto de código de software y herramientas que permite a los ingenieros de verificación identificar fallas en el diseño".

La Figura 4 muestra un diagrama de un ambiente de verificación con los elementos básicos que puede contener, sin embargo esos componentes pueden variar según las necesidades del dispositivo.

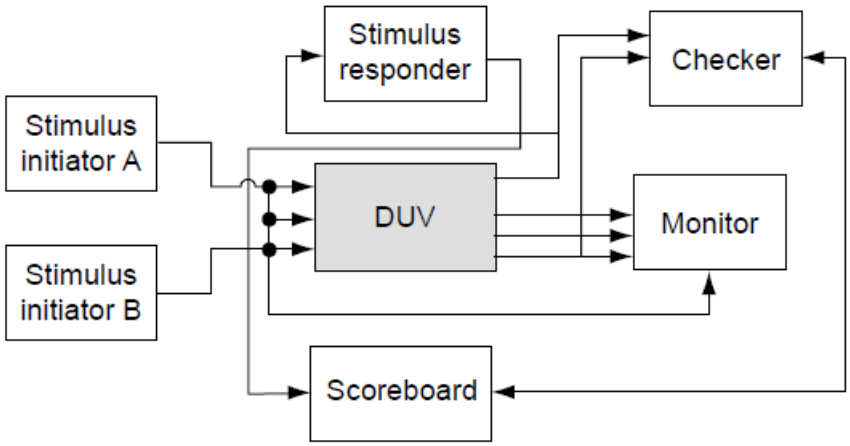

Figura 4: Esquema de un ambiente de verificación básico

Este ambiente de verificación o testbench como también puede ser llamado corresponde a un sistema de simulación escrito en un lenguaje de verificación de hardware o HVL por sus siglas en inglés que se encarga de crear, observar y chequear una secuencia predeterminada de señales que van a las entradas del dispositivo bajo verificación (DUV) y que pueden ser generadas de manera directa o de manera aleatoria, para posteriormente observar la respuesta que tiene el dispositivo debido a esas señales de excitación [1].

Otra de las funciones que tiene el ambiente es comparar la respuesta obtenida por el DUV cuando se aplican los estímulos, contra la respuesta que debería de tener cuando el dispositivo está bajo las condiciones normales de funcionamiento y para eso se obtienen por medio de un modelo de referencia que simula el comportamiento deseado, de esta manera se conocen los casos donde ocurren problemas en los resultados de las operaciones.

Según [1, pág. 74]:

> El reto para un ingeniero de verificación es crear un testbench que estimule el diseño con patrones interesantes de entrada (idealmente, estos patrones deben cubrir toda la funcionalidad del diseño de ser posible, o al menos lo más que sea posible) y calcula las respuestas esperadas de las salidas basado en esos patrones de entrada. El diseño puede decir que esta funcionando como lo previsto mediante el ejercicio de toda la funcionalidad y mediante la predicción y chequeo de todas las respuestas.

Por otra parte es necesario que el ambiente de verificación pueda realizar las pruebas al DUV sin verse limitado por el funcionamiento de los elementos que rodean al dispositivo y lo que debe hacer es llevar a la máxima capacidad que posee el DUV, esto quiere decir que al momento de crear un ambiente de verificación solo se debe tomar en cuenta las restricciones que tiene el DUV y no de los demás elementos. Es así que esto permite sobre estresar al DUV con el fin de buscar los llamados "Corner Cases" que son eventos raramente vistos que suceden luego de cientos de trillones de ciclos de simulación [1].

Según [1] "Un ingeniero de verificación expone los bugs al correr simulaciones complejas en el diseño".

3.3. Metodología UVM

Una de los problemas que han surgido después del crecimiento en el área de verificación funcional, ha sido que muchas compañías y grupos interesados en el tema han desarrollado metodologías de verificación por separado provocando así que no exista compatibilidad entre ellas y reducen la productividad en esta área [4]. A raíz de este inconveniente es que ha surgido la idea de crear metodologías universales y así se originó la Metodología Universal de Verificación (UVM), la cual según [4, pág. 1]:

> Mientras UMV es revolucionaria, siendo la primer metodología de verificación en ser estandarizada, esta también es evolucionaría, como esta construida sobre la Metodología de Verificación Abierta (OVM), mientras combinó la Metodología de Verificación Avanzada (AVM) con la Metodología Universal de Reuso (URM) y conceptos de la Metodología eReuso (eRM). Además, UVM también infunde conceptos y código de la Metodología de Verificación Manual (VMM), más la experiencia colectiva y conocimiento de mas de 300 miembros del Grupo de Trabajo de Acellera de la Metodología de Verificación Universal (UVMWG) para ayudar a estandarizar la metodología de verificación.

Eso quiere decir que UVM vino a estandarizar la verificación funcional al unir las distintas metodologías aplicadas por los diferentes grupos existentes. Además de eso UVM se rige bajo la norma [5] que fue hecha con el fin de dar a las áreas de: herramientas de Diseño Electrónico Automático (EDA), semiconductores y diseño de sistemas; un estandar IEEE bien definido que unifica el diseño de hardware, especificaciones y verificación en un lenguaje.

Por otra parte es importante decir que la metodología UVM permite al diseñador crear ambientes de verificación configurables a las necesidades que se tengan. La arquitectura

que se utiliza en esta metodología posee varios elementos que conforman un ambiente de verificación y los cuales se presenta en la imagen 5

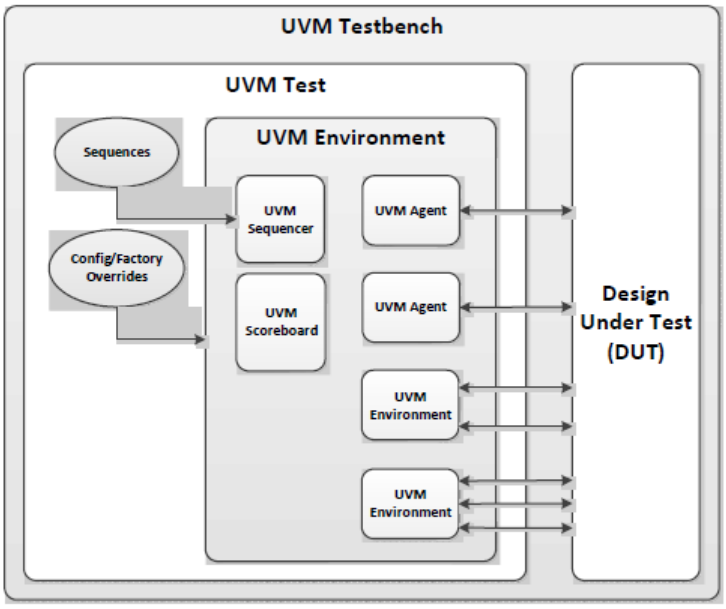

Figura 5: Arquitectura de la medotología UVM

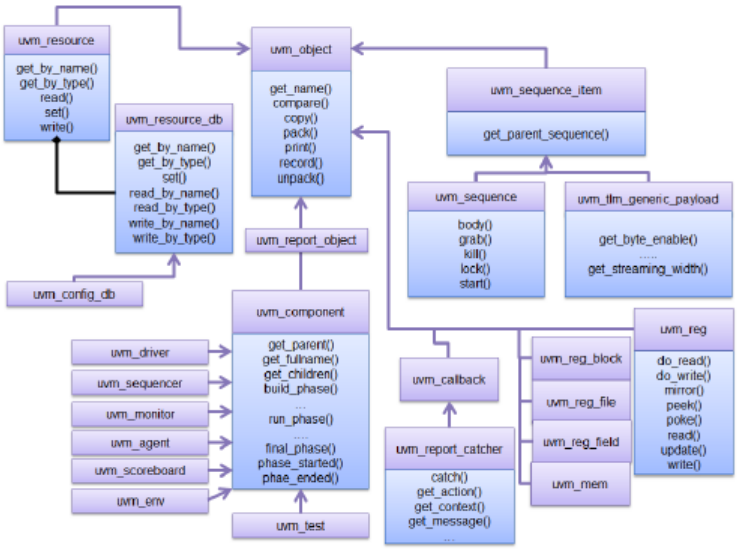

Figura 6: Jerarquía de clases en UVM

Un ambiente de verificación hecho bajo la metodología UVM utiliza distintas librerías de clases que permiten la construcción de los diferentes bloques necesarios, esas clases poseen una jerarquía que se representa en la Figura 6, se puede observar que cada clase tiene ligadas funciones especificas para que cada bloque realice la función que debe cumplir.

La metodología UVM puede crear confusión entre dos conceptos y para eso es necesario hacer una distinción entre el ambiente de verificación y el bloque llamado *ambiente*. De ahi la importancia de conocer los elementos que componen un ambiente de verificación UVM:

3.3.1. Testbench UVM:

El *Testbench* es el bloque de más alto nivel ya que es aquí donde se instancia el DUV para hacer las pruebas, también se instancia la clase de *test* que se va a realizar y por último se hace la interconexión entre ambos módulos utilizando una *interface virtual* [3]. Se puede decir que el *testbench* es el ambiente de verificación según el concepto dado por [1].

Dentro del *testbench* el *test* se instancia de manera dinámica durante el tiempo de ejecución permitiendo así que la compilación sea una única vez y se pueda realizar pruebas diferentes durante la simulación [3].

3.3.2. Test UVM:

El *Test* se encarga de ejecutar 3 funciones: la primera es instanciar el *Ambiente* de más alto nivel ya que al haber una jerarquía puede existir dependencia con *ambientes* mas pequeños, también se encarga de hacer las configuraciones al ambiente por medio de lo que se llama "factory overrides"que son cambios en las configuraciones de la creación. Por último se encarga de hacer la aplicación de los estímulos invocando las *secuencias* a través del *ambiente* y las envía hacia el DUV. Siempre que se realiza una ejecución del *testbench* existe un *test* base y otros que lo modifican o lo extienden cambiando las *secuencias* de datos que este envía al dispositivo [3].

3.3.3. Ambiente UVM:

El *Ambiente* es un bloque en la jerarquía que se encarga de agrupar y contener en su interior otros componentes que están interrelacionados, por ejemplo el *agente*, el *scoreboard* o otros *ambientes* que están abajo en la jerarquía. En el caso de sistemas grandes se utilizan ambientes distintos para los diferentes bloques funcionales que posee el sistema [3].

3.3.4. Scoreboard UVM:

El *Scoreboard* es el elemento que se encarga de chequear el comportamiento de un determinado DUV. Para esto se reciben las señales de entradas y salidas del DUV a través de los puertos de análisis que tiene el *agente*. Luego las entradas que recibe las envía a un modelo de referencia que puede estar dentro del *scoreboard*, también se se le puede llamar predictor, para así obtener las respuestas esperadas del dispositivo y con eso comparar las salidas actuales del DUV con las obtenidas por el modelo de referencia [3].

3.3.5. Agente UVM:

El *Agente* es un componente que contiene otros elementos que interactúan directamente con el DUV. Este puede actuar en dos modos a la vez: el modo activo permite generar los estímulos al DUV y el modo pasivo permite monitorear las interfaces con el DUV sin controlarlas. Esto quiere decir que para el modo pasivo solo se necesita un *monitor* que se encarge de observar el flujo de datos através de la *interface* y de hacer el coverage de la simulación; mientras que el modo activo necesita además del *monitor*, un *sequencer* y un *driver* para enviar datos [3].

La Figura 7 muestra los bloques internos y las conexiones que existen dentro y al rededor del *agente* respectivamente. [3]

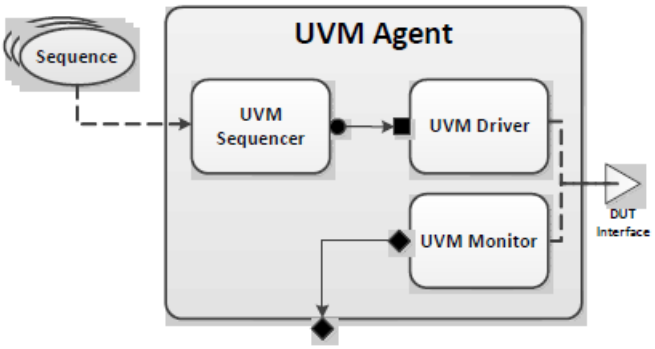

Figura 7: Agente UVM

Sequencer UVM:

El *Sequencer* se encarga de controlar el flujo de búsquedas y respuestas de las *secuencias* entre el *driver* y el *sequence*. Este bloque utiliza el modelo de interfaces TLM (Modelo de Nivel de Transacciones) para el envío y recepción de estas [3].

Sequence UVM:

Es un objeto que contiene el comportamiento para generar los estímulos que van al DUV, pero no es parte de la jerarquía [3], también pueden controlar la ejecución de otro tipo de *secuencias* [4].

Sequence Item UVM:

Un *Sequence Item* consiste en los espacios necesarios de datos para formar un estímulo y la generación de estos tiene la posibilidad de ser aleatoria al declarar cada espacio como rand siguiendo ciertas restricciones que el diseñador determine, también puede ser de manera estática y se hace en los métodos de creación de las secuencias. Las propiedades de estos datos se especifican en la clase creada para formar cada secuencia [3, pág. 8].

Monitor UVM:

El *Monitor* recoge la información que pasa a través de la *interface virtual*, esta se utiliza para corroborar el protocolo de comunicación y hacer el coverage de la simulación. Además de eso la información que es observada se envía a través de un puerto de análisis que comunica con el *scoreboard* y es ahí donde se realiza la comparación de los datos obtenidos por el DUV y los obtenidos por el modelo de referencia [3].

Driver UVM:

El *Driver* se encarga de recibir las secuencias provenientes del *Sequencer* y las envía a través de la *interface virtual* al DUV siguiendo el protocolo de comunicación requerido. Por lo tanto se encarga de la conversión del nivel de transacción al nivel de señal y posteriormente aplica esos estímulos al DUV [3].

3.3.6. Modelo de Registros:

En la metodología UVM se utilizan una capa de clases destinadas para lo que se denomina RAL (Register Abstraction Layer) con el fin de crear un modelo que permite simular el comportamiento de los registros, archivos de registros o memorias que posee un dispositivo bajo verificación en su diseño [3, pág. 75], en la Figura 8 se muestra la incorporación de un *modelo de registros* en la arquitectura del ambiente de verificación.

A través de un modelo de registros se puede utilizar los métodos de acceso *Back-door* y *Front-door* con el fin de verificar el contenido de información que tienen los registros y las posiciones de memoria así como verificar que el dispositivo haga una correcta decodificación de las rutas de acceso a esos registros para guardar o a acceder a la información [3, pág. 75], ya que la función principal de esta clase es reflejar los valores que tienen los registros y las memorias en el hardware [3, pág. 79].

Los valores reflejados en los *modelos de registros* deben ser actualizados cada cierto periodo debido a que estos pueden ser modificados en cualquier momento que el hardware realice una operación [3].

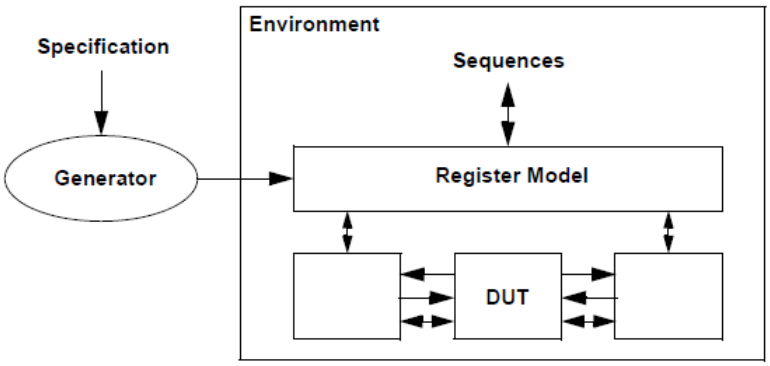

Figura 8: Implementación del modelo de registros en el ambiente de verificación

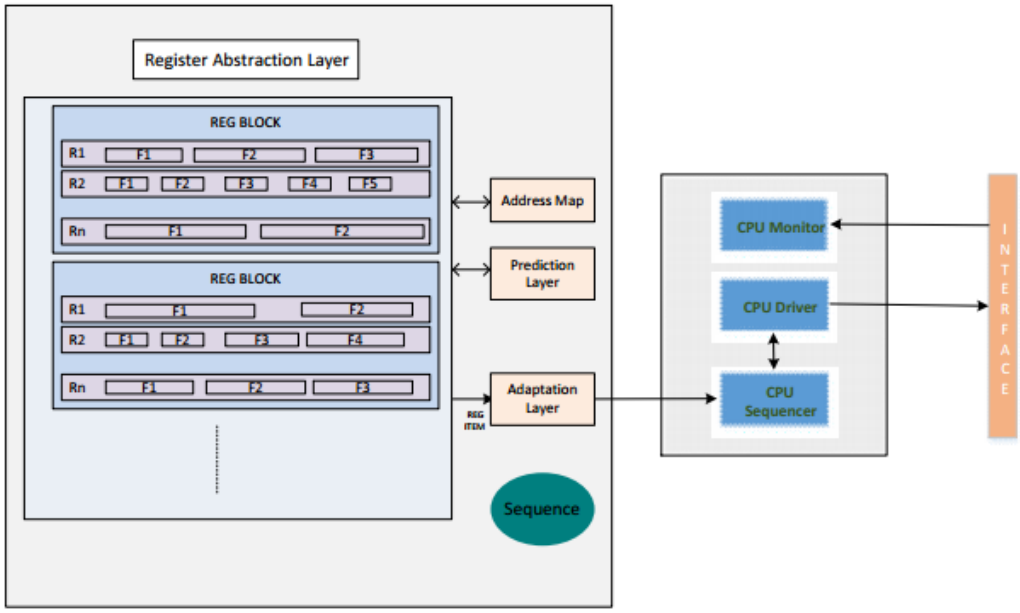

Figura 9: Capas de los modelos de registros [8]

Los *modelos de registros* poseen varias capas que los conforman y que se pueden observar en la Figura 9, los más importantes son los siguientes:

1. Register Field: Consiste en secciones de los registros que almacenan un trozo de la información. Además de eso en algunos casos pueden servir como elementos que almacenan información necesaria para la configuración de funcionamiento de un dispositivo o también secciones que indican un elemento importante para el procesamiento de la información. Es necesario describir algunas características importantes como lo es el tamaño, el nombre, la posición del bit menos significativo, entre otras cosas. [3]

2. Register: Son los modelos que simulan los registros que tiene el dispositivo bajo verificación, también al igual que los register field se indican algunas de sus características importantes. [3]

3. Register Block: Los register block son elementos que contienen cierta cantidad de registros en su interior, funciona de una manera similar a otro elemento llamado Register File. Para poder acceder a los registros internos es necesario la utilización de un mapeo de direcciones que permite revisar cada registro identificándolo de una manera específica.[3]

4. Predictores: Son elementos que permiten predecir y actualizar el contenido de la información que poseen el modelo de registros [8].

5. Adaptadores: Son elementos que permiten modificar la información que esta en el DUV utilizando el contenido en el modelo de registros [8].

3.3.7. Otros conceptos y elementos de la metodología UVM:

Algunos de los conceptos y elementos que permiten completar y entender el funcionamiento de un testbench son los siguientes:

- Una **transacción** es una clase que encapsula la información usada para la comunicación entre dos o más componentes [4]. Por ejemplo las secuencias de información son enviadas al DUV cuando el *driver* realiza una solicitud al *sequencer* para obtener la siguiente secuencia, esta solicitud y respuesta se hace a través de una transacción entre ambos bloques y se conectan por medio de un puerto TLM que tiene referencia en ambos bloques: en el *sequencer* como 'seq_item_export' y en el *driver* como 'seq_item_port'. Este proceso se puede ver en la Figura 10. [3]

- La ***interface virtual*** es lo que permite la conexión del ambiente de verificación con el DUV, según [5, pág. 711]

  > En su nivel más bajo, una interface es un paquete nombrado de redes y variables. La interface es instanciada en un diseño y permite ser accesada a través de un puerto como un solo elemento, y el componente de redes o variables referenciadas donde sea necesitada.

  Con lo anterior se puede decir que permiten el empaquetamiento de los puertos en el momento de la interconexión de módulos como el *driver* y DUV. Las interfaces poseen los llamados 'modports' los cuales son usados en SystemVerilog para restringir el acceso a la interface dentro de un módulo, en ellos se indican las direcciones de los puertos que poseen [5]. También existen los llamados 'Clocking Block' que son bloques que se encargan de sincronizar eventos, muestrear y sincronizar el envío de las señales que pasan por la interface.

- Las **Fases UVM** según [6], son un medio de sincronización que tiene el ambiente de verificación para la ejecución de la simulación. Las fases se representan como llamadas a métodos que pueden ser tanto funciones como tareas y que posee los bloques de un ambiente de verificación, por ejemplo existen las siguientes fases:

Fase	**Descripción**	**Forma de Ejecución**
build	Construye el componente	Top-Down
connect	Conecta los puertos TLM de los componentes	Botton-Up
end of elaboration	Configuraciones de testbench antes de simulación	Botton-Up
start of simulation	Imprimir información de testbench	Botton-Up
run	Generación, envio, monitoreo y chequeo	Parallel
extract	Sacar información del proceso	
check	Comprobar funcionamiento de DUV	
report	Resultado de Simulación	

Cuadro 1: Fases de Metodología UVM

- Un acceso por **Backdoor** [3], permite la verificación del contenido de los distintos registros y direcciones de memoria, esto se hace con el fin de encontrar errores de información que surgen debido a que las funciones de escritura y lectura se hacen por la misma ruta de acceso y provocando la corrupción en la información almacenada, de ahí la importancia del acceso back-door para encontrar esos errores generados. Por otra parte, la verificación utilizando back-doors permite eliminar el tiempo de simulación requerido para la configuración del DUV una vez que se haya comprobado que las interfaces físicas funcionen de manera correcta.

  Además de eso, el mayor desafío de la implementación de un back-door es la identificación de la jerarquía de las rutas de acceso y la implementación de los modelos de registros y de memoria.

  Por otra parte es importante mencionar que los valores almacenados en el modelo de registros pueden ser actualizados cuando se finalice una transacción a través de la interface de conexión con el DUV, esto permite utilizar los back-doors como una clase de monitor que revisan los contenidos de los registros internos.

- Las **Rutas de Acceso Jerárquico** [3], sirven para acceder a un registro o a una dirección de memoria especifica, pero para eso se debe de conocer la ruta que se debe de atravesar en la jerarquía del diseño RTL. Eso se logra especificando el componente que se desea de revisar utilizando una concatenación de los nombres identificadores de cada elemento por los que se deben de atravesar para acceder al registro deseado, se usa un punto como token para la concatenación: por ejemplo para revisar un registro llamado 'xn' que está dentro de un bloque de registros llamado 'R' se especifica la ruta como R.xn .

- El **Coverage** según [5] es una forma que sirve para dirigir los recursos de verificación mediante la identificación de porciones probadas o no probadas del diseño. Según [5]

  > Coverage es definido como el porcentaje de objetivos de verificación que se han cumplido. Es usado como una forma de medir el progreso de un proyecto de verificación con el fin de reducir el numero de ciclos de simulación gastados en verificar un diseño.

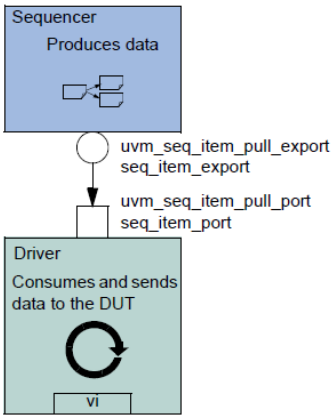

Figura 10: Transacción *driver-sequencer*

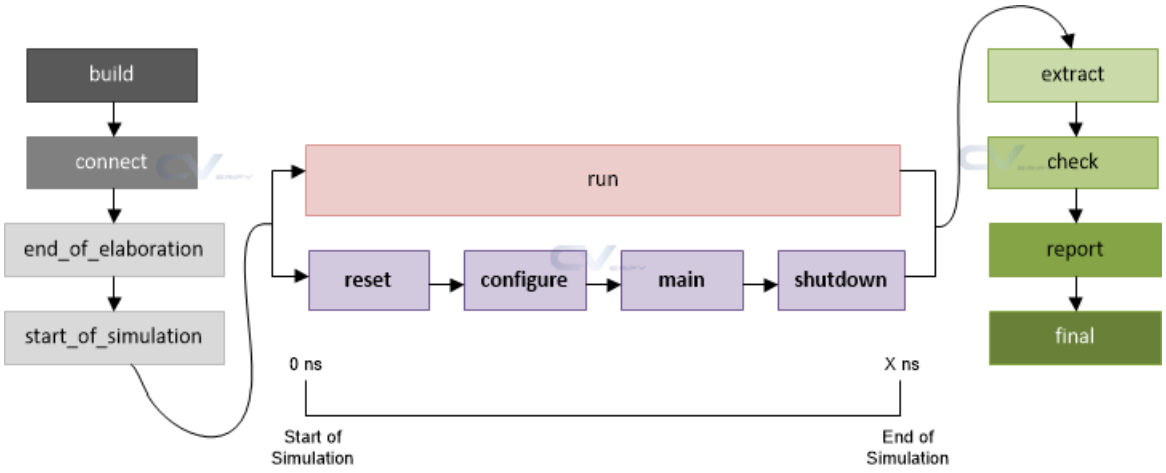

Figura 11: Sincronización de las Fases UVM [7]

3.4. Dispositivo bajo verificación

Otro elemento crucial para lo que es el diseño de un ambiente de verificación es el DUV ya que de este dependen algunas cosas como: las funciones que desempeñara el modelo de referencia, el tipo de secuencias que se deben de generar, el funcionamiento del driver para que respete el protocolo de comunicación necesitado, entre otras. Debido a esto es que es que debemos de conocer tanto el protocolo SPI utilizado por la memoria así como la arquitectura RISC-V 32I para diseñar el modelo de referencia.

3.4.1. Interface y protocolo SPI

Lo primero que debemos de conocer son las interfaces de comunicación con el dispositivo bajo verificación ya que a través de ellas se envían las señales de prueba generadas por el ambiente y se reciben las respuestas que genera el DUV. Cabe mencionar que en esta primera versión del ambiente solo se va a trabajar con la interface SPI debido a que es por este medio que se carga el programa de arranque o bootstrap, el cual esta almacenado en la memoria externa.

La interface SPI tiene la característica de que es un bus utilizado principalmente para la comunicación entre microcontroladores o microprocesadores y periféricos pequeños como: registros, memorias pequeñas o sensores. Además posee una línea de reloj separada de la línea de datos para lo que es la sincronización de envío y recepción entre el dispositivo maestro y los esclavos, tiene también líneas de selección que permiten al maestro escoger con cual periférico se desea comunicar [9].

El dispositivo que genera el Reloj es llamado 'Maestro' que corresponde al micrprocesador y los demás que están conectados a la interface son llamados 'Esclavos'. Es así que cuando los datos de la comunicación son enviados por el maestro a través de la interface, ese puerto se llama 'MOSI' (Master Out / Slave In); por el contrario si es el esclavo quien necesita enviar una respuesta al Maestro a través de la interface, lo hace por medio de un puerto llamada 'MISO' (Master In / Slave Out) y para ello espera la señal de reloj que lo habilita para el envío [9].

Por último, existe una línea extra que corresponde a la llamada 'SS' (Slave Select) la cual indica al periférico esclavo si debe de "descanzar.º si debe prepararse para la recepción y envío de datos.

La Figura 12 representa el método de conexión más común en las interfaces SPI el cual corresponde al de lineas de selección separada, existe también la forma de conexión daysi chain pero no es tan común utilizarlo [9].

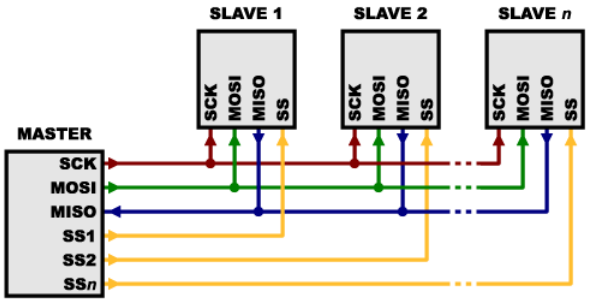

Figura 12: Interface SPI [9]

Por otra parte, algo importante de decir es que el protocolo SPI no es fijo ya que existen formas de configuración que dependen de las necesidades de los diseñadores. Esas formas de configuración puede ser respecto a la velocidad de envío, la omisión de las lineas de selección o también a los llamados modos de reloj.

Los modos de reloj son determinados por 2 bits de configuración: el primero es el CPOL

(Polaridad de Reloj) el cual determina si el estado IDLE está en alto o en bajo, el segundo es el CPHA (Fase de Reloj) el cual determina en que flancos de la señal de reloj se hace la trasmisión y recepción por las líneas MOSI y MISO. Debido a que esos bits poseen dos niveles entonces existen 4 modos de funcionamiento en donde las variaciones corresponde a en que flanco se hace la lectura y transmisión, y si estado IDLE es alto o bajo [10].

Sabiendo ya algunas características en general de la interface SPI es necesario conocer específicamente cual es la configuración escogida para la comunicación entre el microprocesador y la memoria externa, para eso se debe de ver la hoja de datos de la memoria 23A256 [11].

Esta memoria tiene las características de que cada posición almacena un byte, por lo que para tener una instrucción completa es necesario hacer recorridos de 4 direcciones consecutivas. Además de eso la frecuencia de transmisión máxima es de 20 MHz. Otra característica importante de esta memoria es que es necesario que el microprocesador envíe una instrucción a traves del puerto MOSI para indicarle a la memoria cual operación debe de realizar, esto se logra ver en la Figura 13:

Instruction Name	Instruction Format	Description
READ	0000 0011	Read data from memory array beginning at selected address
WRITE	0000 0010	Write data to memory array beginning at selected address
RDSR	0000 0101	Read STATUS register
WRSR	0000 0001	Write STATUS register

Figura 13: Instrucciones de operación de la memoria [11]

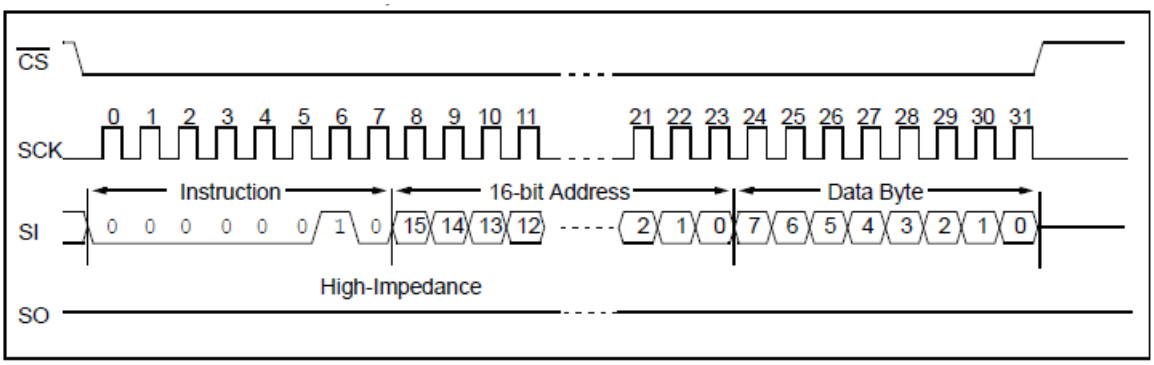

Figura 14: Modo de escritura de la memoria [11]

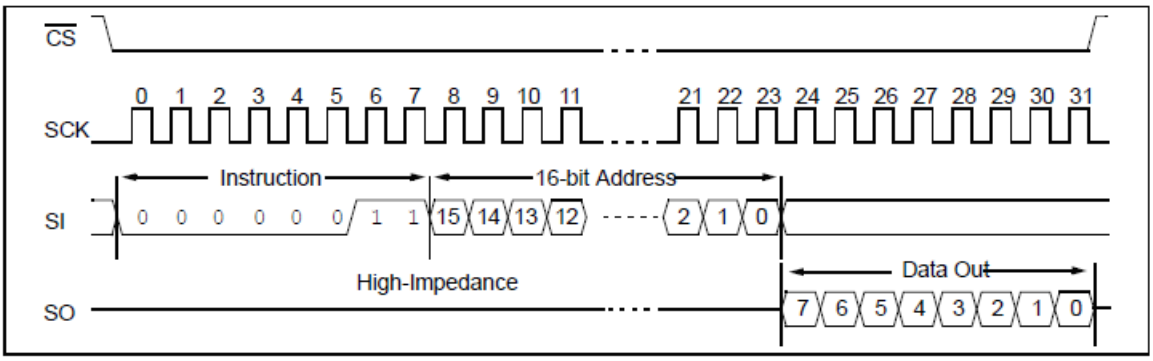

Figura 15: Modo de lectura de la memoria [11]

Por último se puede observar en la Figura 14 el diagrama de tiempo que indica como el protocolo escribe un byte en una posición de memoria determinada por la dirección que se envía. Primero es transmitida la cadena de la instrucción, luego sigue la dirección donde se va a escribir y por último se escribe el dato que se va a guardar en la memoria. Si se desea escribir mas de un byte de manera seguida lo que se hace es que se escribe la dirección de memoria del primer byte y se generan la cantidad de pulsos de reloj SCLK que corresponden a la cantidad de bits totales de la información [11]. De igual manera funciona la operación de lectura que se puede observar en la Figura 15.

3.4.2. Arquitectura RISC-V 32I

El otro elemento importante de conocer para el diseño de un ambiente de verificación es el funcionamiento del DUV ya que con este se debe de implementar el modelo de referencia que servirá para el proyecto de verificación.

La arquitectura RISC-V es un set de instrucciones abierto y gratuito que es relativamente nuevo porque comenzó en la Universidad de Berkley en el año 2010. Fue diseñado con la función principal de utilizarse en aplicaciones académicas y en investigación, sin embargo en la actualidad su uso se ha extendido en aplicaciones industriales que permitan la implementación de proyectos en arquitecturas libres [12]. Según [12]

> RV32I fue diseñado para ser suficiente para formar un objetivo de compilador y soportar ambientes de sistemas operativos modernos. El ISA fue diseñado también para reducir el requerimiento de hardware en una minima implementacion.

Con lo anterior se puede decir que el Instruction Set Arquitecture (ISA) trata de poseer la mínima cantidad de instrucciones con el fin de facilitar la implementación física del hardware y la creación del software. Este esta hecho como base de enteros, lo que quiere decir que la arquitectura tiene como principal tipo de dato números enteros, sin embargo si soporta la utilización de puntos flotantes con una extensión al set de instrucciones [12].

Por otra parte, existen 2 variantes principales de esta arquitectura las cuales son la 32I y 64I cuya diferencia es el ancho y la cantidad de registros disponibles y de ambas la más común en ser utilizada es la versión 32I. La versión 32I posee 32 registros, el ancho de cada

registro es de 32 bits y también este corresponde al tamaño de las instrucciones. Una variante de este set es la 32E la cual se utiliza en microcontroladores para aplicaciones embebidas y tiene la característica de tener solo 16 registros en total [12].

Además de eso otra característica que tiene la arquitectura RISC-V es que soporta cambios en el tamaño de las instrucciones, por ejemplo existe una versión comprimida de 16 bits. Estas variaciones en la longitud de la instrucción involucra que se tome como base la longitud de 16 bits por ser la mínima cantidad posible y por lo tanto en los casos de tamaño superior se toma que cada instrucción posee más de una base de 16 bits, estas extensiones siguen una convención de codificación las cuales se pueden diferenciar en los 2 bits menos significativos y esto se puede observar en la Figura 16 en donde aparecen las distintas bases de instrucción con la codificación respectiva:

Figura 16: Codificación de las instrucciones RISC-V

Según lo anterior, la versión ISA de 32 bits tiene como distintivo que los últimos dos tienen como valor 11 mientras que cuando el tamaño es de 16 bits la terminación puede ser 00,01 o 10 [12].

Caracteristicas del ISA 32I:

Como ya se mencionó la versión de 32 bits es la más utilizada por la facilidad de diseño del hardware y la implementación del software, es por eso que solo contiene 47 instrucciones únicas que se pueden observar en el Apéndice A. La primer característica que posee es la existencia de 32 registros de propósito general que almacenan valores enteros y con una longitud de 32 bits cada uno [12]. Los nombres físicos de esos registros son de la forma 'xn' en donde la 'n' representa el numero de registro que hace referencia.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Figura 17: Banco de Registros del ISA RISC-V 32I

La Figura 17 posee tanto los nombres físicos, los nombres de referencia en el lenguaje ensamblador y la información referente a los usos que tienen cada registro. Como se puede ver Como se puede ver x0 corresponde al registro 'cero' y x1 es el registro que almacena la dirección de regreso en una llamada, también hay registros utilizados como punteros, otros son utilizados para almacenar valores temporales que se pueden usar para saltos, otros son registros para almacenar datos, entre otros usos. También aparecen los registros de punto flotante, sin embargo estos solo están en el diseño del hardware cuando se requiere la extensión del set de instrucciones [12].

Después de conocer eso, es importante saber que el ISA RISC-V 32I tiene 4 formatos elementales de instrucciones las cuales son las R, I, S y U; entonces algo que tienen en común esos formatos es que mantienen los registros fuente (rs1 y rs2) y el registro destino (rd) en la misma posición para simplificar la decodificación. Según [12, pág. 11]:

> La decodificación de los especificadores de registros están usualmente en la ruta critica en la implementación, y entonces el formato de la instrucción fue escogido para mantener todos los especificadores de registros en la misma posición en todos los formatos con el costo de estar moviendo los bits inmediatos a través de los formatos.

Para completar los formatos de instrucción se debe de agregar 2 variantes más las cuales son la B y J que se basan en el manejo del valor inmediato [12], es así como se pueden observar en la Figura 35 todos los formatos disponibles:

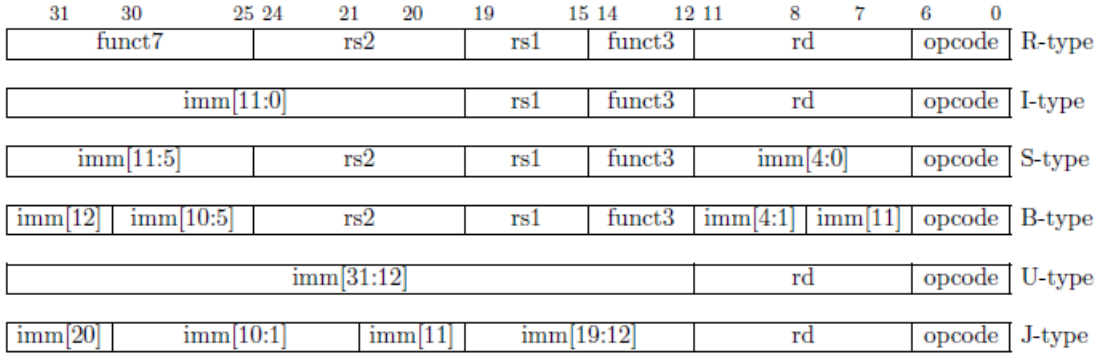

Figura 18: Formatos de instrucción ISA RISC-V 32I

Además de eso, se puede ver que existen coincidencias en las ubicaciones de los espacios para valores inmediatos entre los formatos B, S y J, U entonces [12] expresa que existen diferencias en la forma que se utilizan esos valores:

1. La diferencia entre el formato B y S es que el inmediato de 12 bits es utilizado para codificar los offsets de los branch en múltiplos de 2.

2. En el formato U el valor inmediato de 20 bits se desplaza en 12 bits para generar la codficación del inmediato, mientras que en el formato J solo se desplaza 1 bit.

Otra característica importante para la arquitectura, es la formación y calculo de los valores inmediatos para las instrucciones que requieran estos espacios. Como se puede ver en la Figura 19, los valores inmediatos vienen implícitos en la instrucción recibida y para formarlos se debe de decodificar la instrucción y asi determinar el tipo que corresponde, posteriormente se forma el valor dependiendo de las secciones destinadas para codificar el inmediato.

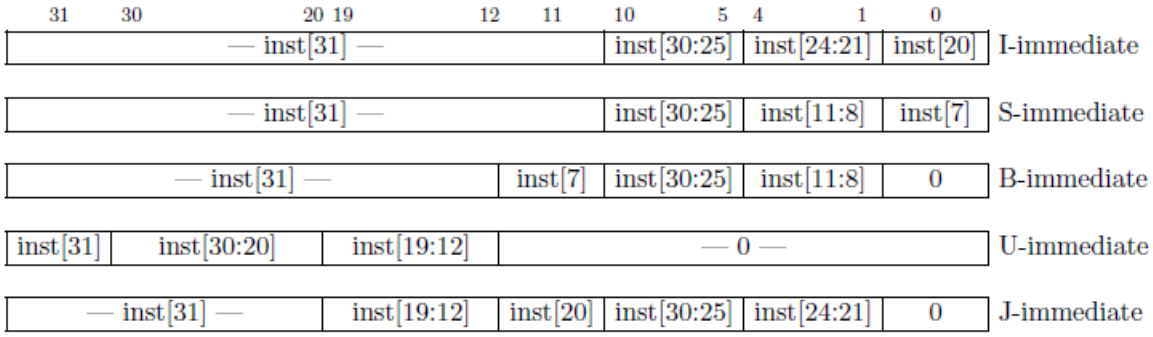

Figura 19: Tipos de inmediatos en la arquitectura RISC-V 32I

En los valores inmediatos la extensión de signo viene codificada con el bit número 31 de la instrucción, esta característica es muy importante porque según [12] permite la extensión en paralelo a la decodificación de la instrucción y los calculos con extensión de signo es un proceso de cuidado en los microprocesadores y es así que esta característica facilita el proceso que ejecuta el microprocesador.

Otra cosa importante en la arquitectura es que tiene la capacidad de hacer instrucciónes NOP, es decir instrucciones que no realizan ningún proceso pero que permiten al Program Counter avanzar. Esta instrucción se puede hacer al sumar un cero en valor inmediato al registro x0, el cual tiene almacenado un cero y posteriormente guardarlo en el mismo registro x0, o sea, la función NOP se puede implementar con un *ADDI x0, x0, 0*.

Por otra parte, uno de los elementos importantes para la arquitectura del microprocesador es el modelo de memoria. En primer lugar es necesario aclarar que este modelo esta desactualizado y que al momento de la publicación de [12] estaba en revisión. Para eso es necesario saber que la arquitectura RISC-V tiene la posibilidad de hacer múltiples hilos de procesos en hardware y cada uno tiene su propio registro de estado y contador de programa para ejecutar su propia secuencia de instrucciones. Además RISC-V tiene la posibilidad de interactuar con dispositivos por medio de los puertos I/O por medio de cargas y descargas de datos a las direcciones de memoria asignadas a estos puertos.

Además RISC-V trata a los demás hilos con sus propias operaciones de memoria de manera secuencial en el orden del programa, esto quiere decir que las operaciones que realizan otros hilos son sincronizadas y ordenadas por el microprocesador al utilizar las funciones FENCE. Por eso la función FENCE es usada para ordenar los dispositivos I/O y accesos de memoria hechos por dispositivos externos como coprocesadores.

En conjunto a esta función, existe otra llamada FENCE.I la cual es usada para el flujo de datos y sincronización de las instrucciones para dispositivos externos.

También, esta arquitectura cuenta con funciones dedicadas a el control del ambiente de operación, es decir controlan los estados que posee el microprocesador. En primer lugar están las funciones CSR o de Registros de Control de Estados, se encargan de hacer operaciones de lectura, escritura, leer y establecer bits o leer y limpiar bits. También se encuentran las funciones de llamada del ambiente, las cuales se encargan de ejecutar las interrupciones sincrónicas hechas por un disparador como un timer, o interrupciones asincronicas las cuales son hechas por dispositivos I/O.

Por último es necesario mencionar que la arquitectura tiene extensiones que se pueden utilizar para hacer otro tipo de operaciones, por ejemplo algunas extensiones son: La extensión M posee las funciones utilizadas para la multiplicación y división. La extensión A tiene la capacidad de modificar memoria por medio de lecturas o escrituras que realicen los dispositivos externos que accedan a una misma dirección de memoria. La extensión F se encarga de hacer operaciones de punto flotante y por lo que es necesario la expansión de la cantidad de registros que posee la arquitectura, ya que se requieren 32 registros que almacenen datos de tipo punto flotante.

Capítulo 4

Marco metodológico

Una de las principales decisiones que se debió tomar en cuenta para hacer la verificación funcional del microprocesador y de los bloques internos, fue con respecto a determinar el método que se iba a utilizar para la generación de las pruebas. Como primera opción se tenía la posibilidad de hacer un testbench deterministico, el cual según [1] sirven para probar funcionalidades básicas del DUV, sin embargo este método tiene la desventaja de que los ingenieros de verificación deben depender mucho sobre la información que los ingenieros de diseño les comparta, para de esta manera plantear los escenarios que se desean probar.

También otra desventaja que tiene esta forma de hacer las pruebas, es que el testbench es muy rígido debido a que se debe conocer con precisión las salidas del DUV para determinadas señales de entrada y así comparar con la respuesta que se espera obtener. Este proceso se repite para todos los casos que se desean de probar, haciendo el proceso de verificación muy tedioso.

La segunda opción era la creación de un testbench del tipo self-checking o de auto chequeo y consiste según [1], en que el ambiente de verificación es visto como el universo, es decir, todo el conocimiento necesario para hacer el chequeo de las pruebas esta en el ambiente y esto facilita en gran medida la verificación porque le da autonomía al sistema de pruebas, en este método los ingenieros de verificación se enfocan más en la creación del scoreboard y de los checkers para hacer al ambiente más autónomo.

En este tipo de testbench existen 3 variantes, de las cuales solo hay una que es importante para el caso de este proyecto, la cual es de mayor utilidad porque permite recrear el funcionamiento del DUV y despreocupándose sobre conocer con certeza las respuestas del DUV para ciertas entradas, esta forma de testbench de auto chequeo es el que posee modelo de referencia.

Según [1] .El modelo de referencia re-implementa el funcionamiento del DUV, usualmente en un lenguaje de programación de alto nivel o en un HVL"por lo que se puede decir que el modelo de referencia, una vez que este comprobado su funcionamiento de una manera correcta posee un alto grado de confiabilidad en el chequeo del DUV en cada ciclo de reloj y por eso se dice que es un *modelo preciso de ciclo*. Por lo tanto este fue el tipo de testbench elegido para diseñar el ambiente de verificación del microprocesador.

Como se mencionó anteriormente, existen metodologías que permiten a los desarrolladores de ambientes de verificación crear sistemas de pruebas que cumplan con un estándar, además facilitan el diseño y la implementación del ambiente de verificación. Es así como han

surgido algunas metodologías en los últimos años de las cuales las más conocidas son: la VMM (Verification Methodology Manual), OVM (Open Verification Methodology) y UVM (Universal Verification Methodology).

En primer lugar esta la VMM, la cual fue la primer metodología mayormente usada y que usaba SystemVerilog para crear ambientes de verificación altamente escalables. Esta metodología surgió en el año 2005 y tiene la característica que permite a los ingenieros desarrollar ambientes poderosos a nivel de transacciones, es decir permite generar secuencias complejas para las pruebas sobre los dispositivos. La desventaja que tiene esta metodología es que es de una complejidad mayor y además que no es libre, esto según [16], por esta razón en el año 2008 los usuarios pidieron que se hiciera una metodología abierta y simple, esto provocó que Acellera comenzara a unir esfuerzos para tener un comité que permitiera crear estos estándares. Posteriormente Synopsis apoyó la idea y dono la implementación completa de la VMM para acelerar la creación de esa metodología estándar que unificara la industria de la verificación. Por lo tanto esta metodología actualmente no se utiliza porque han surgido nuevas que mejoran y permiten a otras personas usar los estándares de una manera libre y simple.

Otra metodología es la OVM la cual fue desarrollada por las compañías Cadence y Mentor. Se usaron como base los estándares de lenguajes IEEE 1800^{TM} SystemVerilog, IEEE 1666^{TM} SystemC y IEEE 1647^{TM} e; según [17], esta metodología esta enfocada en la verificación basada en simulaciones y esta enfocado en facilitar la construcción del ambiente debido a que permite usar estructuras reutilizables. También esta metodología permite crear ambientes modulares, donde cada componente tiene una determinada funcionalidad, permitiendo una mayor flexibilidad porque los componentes se pueden adaptar a las necesidades que se requieran. Esta metodología sirvió de base para la UVM, por lo tanto actualmente es menos utilizada y es así que tampoco se eligió para el proyecto.

La última es la metodología UVM y anteriormente se hablo sobre las facilidades que posee, esta metodología surgió luego del esfuerzo en conjunto de Acellera para crear un estándar libre y fácil de utilizar. La metodología UVM junta las características de la VMM y la OVM, por esta razón es la más utilizada en la actualidad y la que mayor información tiene para el desarrollo de ambientes de verificación y por eso se tomó la decisión de usarla para el desarrollo del proyecto.

Capítulo 5

Diseño del ambiente

Para hacer el diseño del ambiente de verificación se debió trabajar en distintas etapas dedicadas a la implementación de los componentes UVM requeridos para las pruebas. La dependencia de tareas entre algunos bloques se pueden visualizar como componentes ficticios que identifican los pasos en el diseño del ambiente. Esos bloques se pueden observar en la Figura 20:

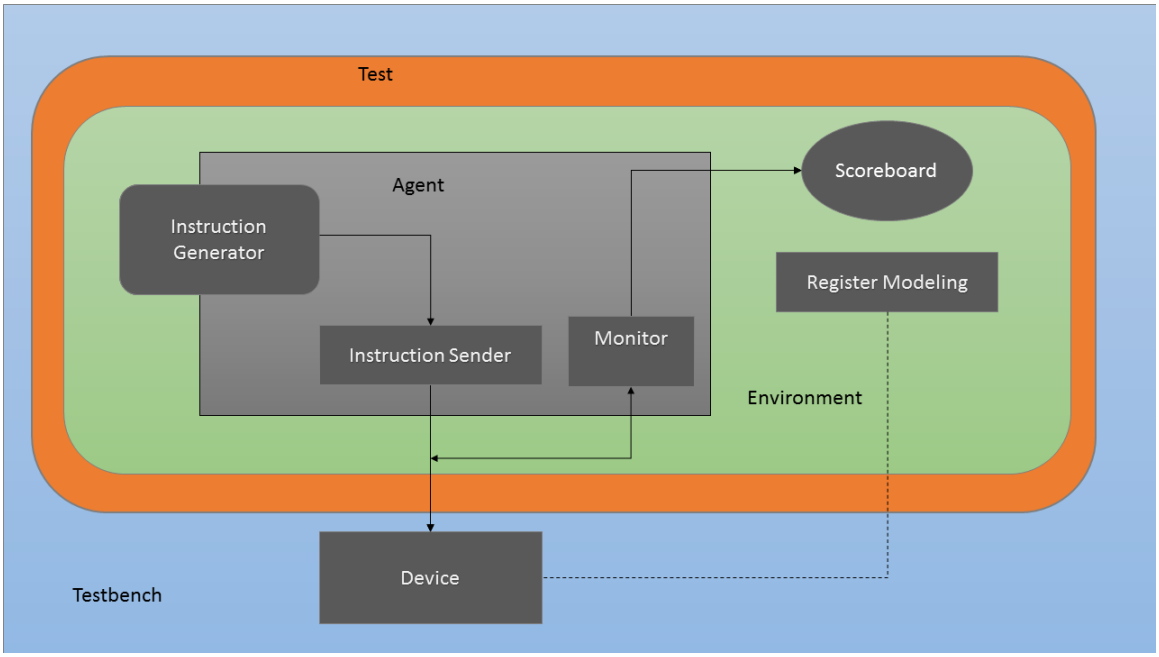

Figura 20: Diseño de ambiente de verificación

Además de eso el Cuadro 2 presenta cada una de esos bloques ficticios con los componentes internos que fueron construidos y la función que desempeñan en el ambiente:

Bloque	*Componentes internos*	*Función*
Instruction Sender	Sequencer, Driver, Interface	Solicitar y enviar la instrucción
Instruction Generator	Sequence Item, Sequence	Generar la instrucción
Monitor	Monitor	Observar la comunicación por la interface
Scoreboard	Scoreboard	Modelo de referencia y Checking
Register Modeling	Register Model, Adapter	Backdoor, Coverage

Cuadro 2: Etapas de diseño

Por último antes de explicar como se diseñaron todos los componentes del ambiente es necesario mencionar que se tomó como referencia el ambiente de ejemplo hecho por [13] y el modelo de registros hecho por [14].

5.1. Instruction Sender

Este fue el primer bloque funcional que se diseñó debido a que permite el envío de las instrucciones al DUV, por lo tanto tiene una de las tareas de mayor importancia durante el tiempo de simulación porque si la instrucción no se envía de manera correcta entonces el dispositivo no va a entender que es lo que tiene que hacer. Los bloques internos son los siguientes:

1. **Virtual Interface**: Para la implementación de la interface virtual fue necesario conocer los pines de comunicación que poseía la memoria externa [11] y ahí se observó que tenía los puertos comunes (SCLK, MOSI, MISO, SS) y un pin extra llamado HOLD el cual sirve para pausar al chip, sin embargo este no se tomó en cuenta. Lo siguiente fue entender la dirección que tenían esos pines con respecto al ambiente de verificación para así poder declarar los Clocking Blocks que van referenciados a los modports y así tener una sincronización en el envío de los datos a través de los puertos de la interface.

   Esas direcciones se pueden observar en la Figura 21 donde aparece que para el monitor todas son entradas de la interface ya que este solo se encarga de observar, mientras que para el driver la única salida es la señal 'MISO' que es por donde se envía la instrucción, el resto son entradas del bloque ya que son generadas por el Maestro.

2. **Driver:** Por otra parte, el driver fue uno de los elementos que tuvo más cuidado al momento del diseño ya que se debieron hacer cambios en la forma que se tenía pensada para el envío de los bits a través de la interface, por lo tanto esas correcciones se implementaron en 3 versiones distintas del driver.

   La primera versión consistió en solamente enviar un único bit de la secuencia, porque al principio se quería formar la instrucción bit por bit desde el instruction generator pero eso provocó que la instrucción no se enviara en el formato deseado y por lo tanto se descartó este método de envío.

   La segunda versión buscaba en recibir la instrucción completa que llegaba desde el instruction generator y después enviar bit a bit usando un ciclo 'for', el problema sucedió porque el primer bit enviado al DUV era un valor 'x' que generaba un corrimiento en

toda la secuencia y por lo tanto la instrucción no podía ser decodificada. Esto se debió a que en la etapa run phase del driver existió un pequeño retraso desde el momento que se tenia la instrucción lista para enviar hasta el momento que entraba en el ciclo de envío, lo que provocaba que el driver no obtuviera un valor conocido en ese primer bit.

La tercera versión es similar a la segunda sin embargo tiene una diferencia la cual es que el primer bit de la secuencia se envía por fuera del ciclo 'for' y de esta forma si se logró enviar la secuencia de manera correcta al DUV. Por otra parte, debido a que no se tenía el DUV a tiempo para hacer las pruebas se tuvo que generar la señal 'SS' desde el driver durante el momento que se enviaba cada bit, esto con el fin de que el ambiente tuviera la capacidad de leer las instrucciones validas con el monitor.

Por otro lado, la señal 'SS' generada tiene un valor en bajo durante la transacción en ejecución y así respetando el protocolo especificado en [11]. El diagrama de flujo del algoritmo implementado se puede ver en la Figura 22.

3. **Sequencer:** Este es uno de los bloques genéricos hechos bajo la metodología UVM porque solo hace la transacción cada vez que el driver solicita la información y por lo tanto en el ambiente lo único que se hizo fue cambiar el nombre a este componente.

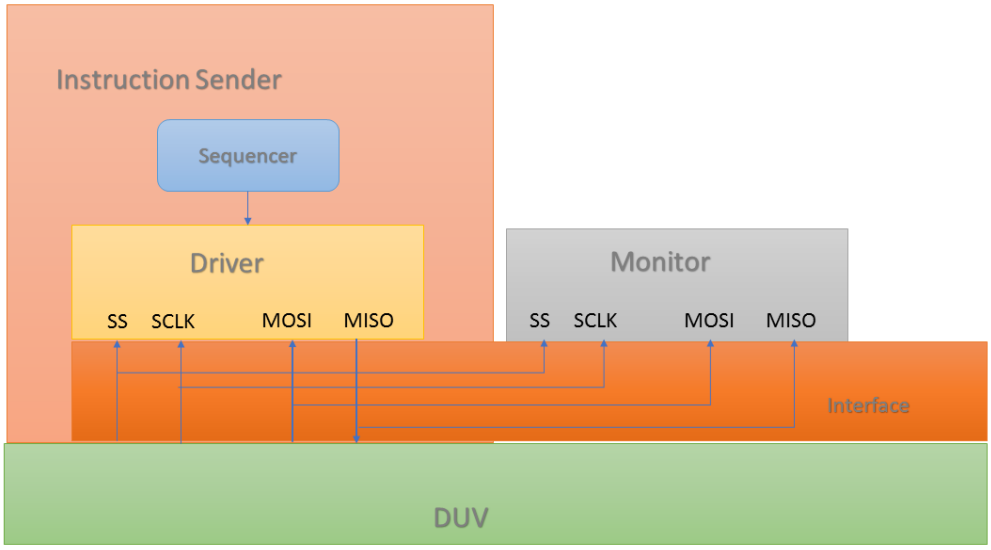

Figura 21: Diagrama de la interface en el Instruction Sender

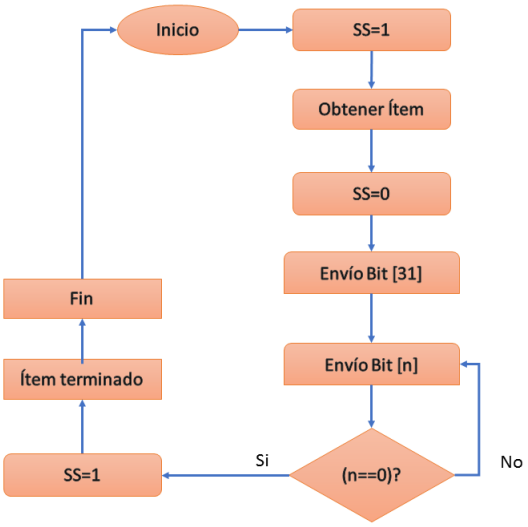

Figura 22: Diagrama de flujo del algoritmo de envío

5.2. Instruction Generator

La segunda etapa de diseño se basó en el llamado Instruction Generator, el cual tiene la función de generar una secuencia que corresponde a la instrucción enviada al DUV. Como ya se mencionó estas podían ser creadas de manera aleatoria o de manera estática por lo tanto se hicieron ambos tipos de secuencias para ciertas pruebas. Los componentes que están dentro de esta etapa son los siguientes:

- **Sequence Item:** Para la generación de las instrucciones se debió tomar en cuenta los dos tipos de secuencias que se deseaban realizar: el primer caso eran instrucciones con espacios aleatorios para la comprobación del funcionamiento de la etapa llamada Instruction Sender y comprobar el recibimiento de la secuencia en el scoreboard.

  Para lograr eso se crearon las diferentes secciones que conforman una instrucción R haciendo un sequence item para cada una de ellas, por lo tanto se hizo: un espacio llamado *opcode* con un tamaño de 7 bits el cual se generaba de manera estática para poder probar la decodificación en el scoreboard, también tres espacios correspondientes a los registros fuente y destino (*rs1,rs2* y *rd*) los cuales se generaban de manera aleatoria, además un espacio llamado *fun3* que tenia una longitud de 3 bits y con este se determinaba el tipo de instrucción R que se deseaba probar y por último se hizo un espacio llamado *transmited* con una longitud de 32 bits que tenía el objetivo de formar la instrucción completa dentro de sí y luego ser enviado a través del Instruction Sender.

- **Sequence:** Luego de tener las secciones correspondientes a la instrucción, el Sequence se encarga de unir cada parte y pegarlo en el espacio de secuencia llamado *transmited*, este proceso se observa en la Figura 23.

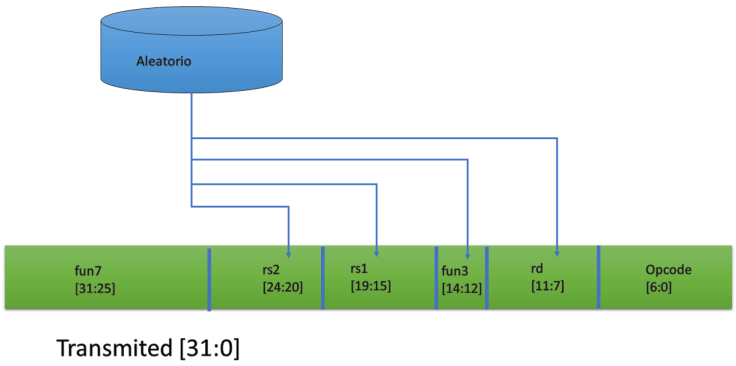

Figura 23: Generación de Instrucción con espacios aleatorios

5.3. Monitor

Por otra parte, durante la etapa de diseño del monitor se buscaba que la lectura y envío del dato al scoreboard se diera de manera directa en los flancos positivos de la señal de reloj cuando hubiera una transacción valida y esto se presentaba cuando la señal SS estuviera en bajo. Posteriormente se debió tomar en consideración aspectos importantes en la captura de los bits ya que existían fuentes de errores que originaban problemas en el dato observado por el scoreboard.

En primer lugar se notó que existía un retraso de dos ciclos de reloj desde el momento que el driver solicitaba al sequencer una transacción para generar la primera instrucción, hasta el momento que el monitor observaba el primer bit enviado por el driver. Este retraso se debe a que cada bit debe pasar a través de dos componentes del ambiente de verificación, los cuales son el sequencer y el driver, para llegar a la interface con el DUV y en donde el monitor realiza el muestreo de los bits. Para corregir este detalle fue necesario realizar en el run phase la espera de dos ciclos de reloj al inicio de la ejecución de las pruebas, una vez hecho esto se muestreaban los bits y se enviaban al scoreboard.

Otro problema que se encontró en la observación del envío de datos por la interface fue que durante la transmisión de la primera instrucción siempre se enviaba un cero al scoreboard que no estaba incluido en la secuencia, lo que provocaba un corrimiento en los bits recibidos. Esto se debía a que el monitor siempre comienza con un valor inicial en la ejecución de las pruebas, por lo tanto era necesario realizar la cuenta de un bit extra para poder empezar a leer de manera correcta las instrucciones, lo que quiere decir que se debían contar 33 bits en la primera instrucción y seguidamente se comenzaba a enviar al scoreboard desde el bit 32 y una vez terminado este proceso se proseguía a muestrear la siguiente instrucción de manera normal osea se lee a partir del bit 32.

Con lo anterior se puede extraer la Figura 24 que representa el diagrama de flujo con el funcionamiento del monitor.

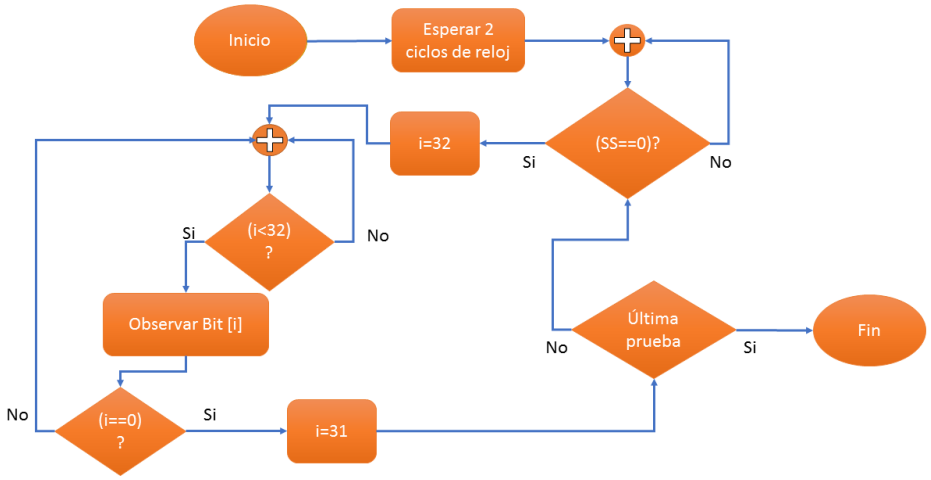

Figura 24: Diagrama de flujo del funcionamiento del monitor

5.4. Register Modeling

La etapa llamada register modeling se diseñó en paralelo al scoreboard debido a que era necesario para comprobar el funcionamiento del modelo de referencia al conocer los valores almacenados en los registros y chequear los resultados obtenidos por el modelo de referencia después de las operaciones efectuadas, para hacer este procedimiento fue necesario la implementación de un Backdoor y de esta manera acceder a los contenidos de los registros internos del microprocesador. Este bloque esta compuesto por 3 elementos importantes que interactuan entre si para poder hacer la lectura de los registros de manera correcta, esos tres elementos fueron:

- *Modelo de Registros:* Lo principal para conocer los valores guardados en los registros del DUV, es tener un sistema que simule el comportamiento de ellos. Para lograr eso fue necesario crear un modelo de registros utilizando la capa de clases UVM Register Layer.

  En primer lugar se hizo la clase llamada ' riscv_register' la cual tuvo el objetivo de crear un modelo que simulara un registro del banco en el microprocesador. En la etapa de construcción de la clase fue necesario indicar las características que poseían los espacios internos de los registros y debido a que solo se planeaba verificar el contenido total es que se creó un único uvm_reg_field, esas características se presentan en el Cuadro 3:

Característica	Valor	Descripción
parent	this	El registro padre de este espacio
size	32	Tamaño de Registro
lsb_pos	0	Posición del LSB
access	RW	Lectura y Escritura
volatile	0	No se toma en cuenta
reset	1	Valor del Reset
has_reset	1	Si hay reset
is_rand	1	El valor puede ser aleatorio
individually_accessible	1	Es el único espacio en el registro

Cuadro 3: Características de los registros

Las características del espacio creado tienen un significado especifico: el primero es un indicador que decide si el campo del registro va a tener como clase padre el registro que se está caracterizando, el segundo corresponde al tamaño del espacio creado por lo que debía ser de 32 bits, el tercero indica la posición del bit menos significativo la cual correspondía a la posición cero, la cuarta característica corresponde a los modos de acceso la cual se eligió del tipo 'RW' para así poder escribir y leer de los registros del DUV, el quinto espacio determina si el dato es consistente luego de las operaciones de escritura-lectura y se dejo como valor false para no tomar en consideración ese aspecto, la sexta característica era referente a la existencia de un reset para el valor almacenado en el espacio del modelo de registros por lo tanto al tener un valor true implica la existencia del reset para poder refrescar el dato periódicamente, el sétimo valor indica si el dato almacenado puede ser aleatorio y la última característica es una indicación si el register field es el único presente en el registro.

Por último, para completar el modelo de registros es necesario la creación de un register block el cual tiene la función de ser una estructura que almacena en su interior registros, memorias o register files; por lo tanto se utilizó este bloque con el fin de hacer el banco de registros del microprocesador al instanciar dentro de él 32 registros distintos con dirección de ruta de acceso según el registro que representa en el hardware, además de eso el bloque necesitó de un mapa de direcciones para poder tener una referencia de acceso a cada registro del bloque.

En la Figura 25 se presenta un diagrama del bloque de registros en donde se especifican los nombres de los registros y el mapa de direcciones.

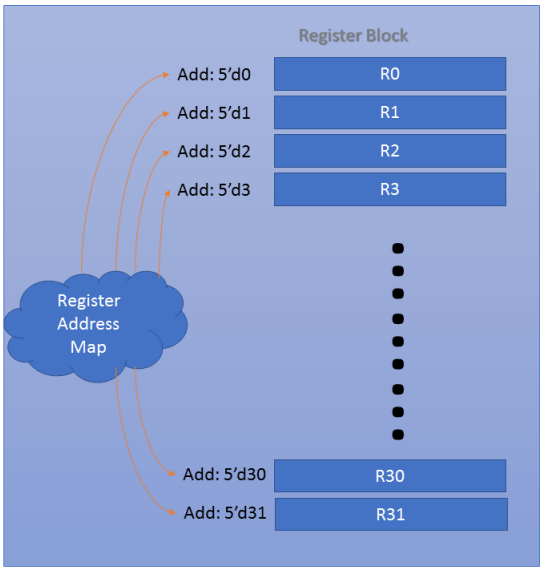

Figura 25: Diagrama del Bloque de Registros

- *Adaptador:* El Adaptador fue uno de los elementos más importantes para la implementación del backdoor ya que es el encargado de convertir o de adaptar las transacciones efectuadas entre el modelo de registros y los elementos que interactúan con el bus (interfaces, drivers, monitores) del ambiente.

  Debido a que el adaptador debe ser un elemento bidireccional, es que se realizan los métodos llamados 'reg2bus' para hacer escrituras a través del bus desde el modelo de registros hacia un target que esta presente en el DUV y 'bus2reg' para hacer lecturas en el DUV desde el target. El adaptador es una herramienta muy útil para la realización de frontdoor, sin embargo la utilidad que se aprovecha en un backdoor es la capacidad de extraer identificadores de los registros que están siendo modificados por un bus de datos.

  El método reg2bus no tiene tanta importancia para la realización de las pruebas sobre el DUV en esta versión del ambiente de verificación debido a que este método es utilizado en la mayoria de los casos para hacer escrituras en el DUV y lo que se busca es hacer lecturas de los datos almacenados en los registros del dispositivo. Además si no se creaba este método se generaba un error en la compilación del ambiente ya que por definición el adaptador es bidireccional y por lo tanto esta función se dejó como un cascarón hueco, es decir, no realiza ninguna función en especifico.

  Por otra parte el método bus2reg es el de mayor importancia porque es así que se obtiene los datos almacenados en los registros del DUV esto gracias a que identifica cuales registros son alterados por las operaciones del bus y con este identificación se puede hacer un read del backdoor. En primera instancia este método necesita una referencia a una transacción que se hace a través del bus, posteriormente se revisa los códigos de operación de dicha transacción para así poder mapear la dirección del

registro destino y hacer la lectura por medio del backdoor, por último se revisa que la transacción se completó de manera correcta al indicar que el estatus es UVM_IS_OK.

- *Secuencias de Registros:* Este tipo de secuencias son destinadas para la realización de la lectura o escritura hacia el dispositivo bajo verificación por medio de un acceso Backdoor. En el caso del ambiente diseñado solo se deseaba hacer lecturas a los registros, por lo tanto se utilizan los métodos peek o read presentes en la metodología UVM para poder sacar esa información y chequear los resultados de las operaciones.

  En este tipo de secuencias debe existir una referencia al bloque de registros y en el cual se guardan los datos leídos por las funciones peek o read. Además existe dos variables que permiten obtener la lectura: la primera se llama **status** que indica el estado del dato leído y la segunda variable corresponde **value** la cual almacena el valor capturado por la función peek o read.

5.5. Scoreboard

Por último para el diseño del ambiente se requirió hacer el modelo de referencia del DUV implementado en el scoreboard. Para hacer este modelo de referencia se tuvo que dividir en varias etapas con funciones determinadas: la primera se dedica a la recepción y empaquetamiento de la instrucción, esto porque las señales son recibidas en forma serial y es necesario reunir todos los bits correspondientes a cada instrucción, la segunda corresponde a la decodificación para determinar el tipo de instrucción e identificar los registros o direcciones de memoria en la cual se realizan las operaciones y la tercera etapa realiza la operación de la instrucción generando los resultados y almacenándolos en los registros o direcciones de memoria correspondientes.

En primer lugar se hizo la sección que se encargaba de recibir cada bit por el puerto de análisis proveniente del monitor y una vez capturados los 32 bits se guardan en una variable llamada instruction para así hacer la decodificación y continuar con la siguiente parte en el funcionamiento del modelo de referencia.

El algoritmo de recepción se muestra en la Figura 26, en primer lugar se empieza a recibir cada bit de la instrucción desde el 31 al 0 y se empaqueta en una variable llamada *instruction*, posteriormente cuando se comprueba que es el último bit se reinicia la cuenta para la siguiente instrucción, se activa una bandera para decodificar la instrucción en la siguiente etapa y se obtiene el código de operación de la instrucción.

Posteriormente la decodificación de la instrucción se implementó utilizando un case para cada código de operación lo que permitió encasillar cada instrucción recibida según los tipos que tiene la arquitectura RISC-V. Una vez determinado el tipo de instrucción se utiliza los identificadores *funct3* y *funct7* para determinar cual instrucción es la que se va a ejecutar. Y por último se obtienen los valores de los registros (rs1,rs2,rd), inmediatos (imm) según lo requiera la instrucción a ejecutar; este proceso se puede ver en la Figura 27.

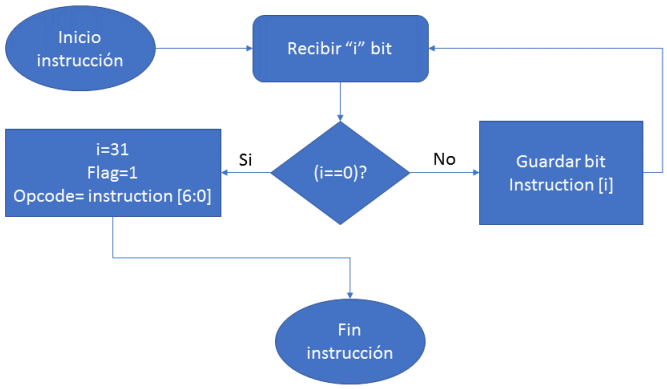

Figura 26: Diagrama de flujo de la recepción de la instrucción

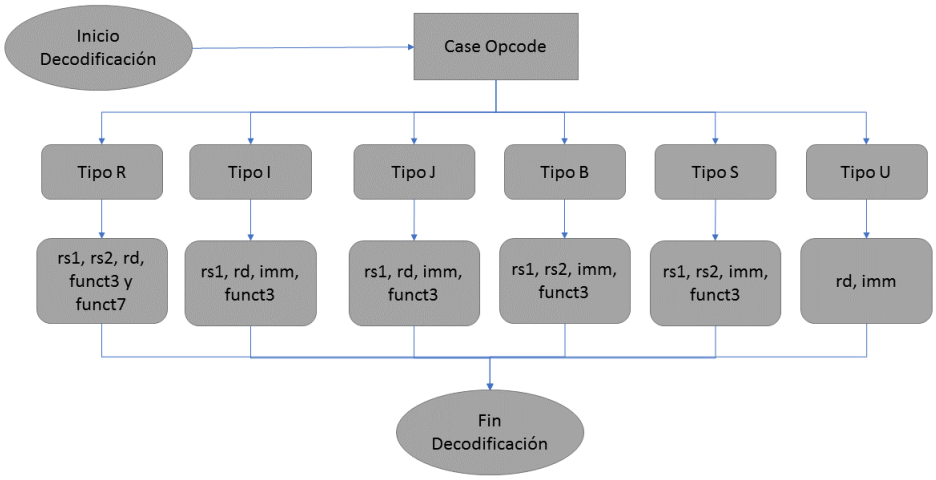

Figura 27: Diagrama de flujo de la decodificación de la instrucción

Por último en la etapa de ejecución de las operaciones se toman los valores contenidos en los registros o en las direcciones del mapeo de memoria que tiene el modelo de referencia, los cuales fueron obtenidos en la decodificación de la instrucción y esos valores se modifican luego de realizar la función que indica la decodificación.

Las funciones implementadas en el modelo de referencia se especifican en las siguientes tablas:

Opcode: *0110011*

funct3	funct7	Nombre	Función
000	0000000	Add	Suma
000	0100000	Sub	Resta
101	0000000	Srl	Corrimiento Lógico Der.
101	0100000	Sra	Corrimiento Aritmético Der.
111	xxxxxxx	And	Y
110	xxxxxxx	Or	O
100	xxxxxxx	Xor	O exclusivo
001	xxxxxxx	Sll	Corrimiento Lógico Izq.
010	xxxxxxx	Slt	Comparación sin signo
011	xxxxxxx	Sltu	Comparación con signo

Cuadro 4: Instrucciones R

Opcode: *0010011*

funct3	funct7	Nombre	Función
000	xxxxxxx	Addi	Suma inmediato
111	xxxxxxx	Andi	Y inmediato
110	xxxxxxx	Ori	O inmediato
100	xxxxxxx	Xori	O exclusivo inmediato
010	xxxxxxx	Slti	Comparación sin signo inmediato
011	xxxxxxx	Sltiu	Comparación con signo inmediato
101	0100000	Srai	Corrimiento Arit. Der. inmediato
101	0000000	Srli	Corrimiento Lóg. Der. inmediato
001	xxxxxxx	Slli	Corrimiento Log. Izq. inmediato

Cuadro 5: Instrucciones I

Opcode	Nombre	Descripción
0110111	Lui	rd=imm « 12
0010111	Aiupac	rd=pc+ (imm « 12)

Cuadro 6: Instrucciones U

Opcode	Nombre	Descripción
1101111	Jal	Salto y enlace
1100111	Jalr	Salto y enlace registro

Cuadro 7: Instrucciones Salto

Opcode: *1100011*

funct3	funct7	Nombre	Función
000	xxxxxxx	Beq	Branch Igual
001	xxxxxxx	Bne	Branch Distinto
100	xxxxxxx	Blt	Branch comparación <sin signo
110	xxxxxxx	Bltu	Branch comparación <con signo
101	xxxxxxx	Bge	Comparación >= sin signo
111	xxxxxxx	Bgeu	Comparación >= con signo inmediato

Cuadro 8: Instrucciones B

Opcode: *0000011*

funct3	funct7	Nombre	Función
010	xxxxxxx	Lw	Cargar palabra
000	xxxxxxx	Lb	Cargar byte signo extendido
001	xxxxxxx	Lh	Cargar media palabra signo extendido
100	xxxxxxx	Lbu	Cargar byte extención de cero
101	xxxxxxx	Lhu	Cargar 1/2 palabra extensión cero

Cuadro 9: Instrucciones S

Por último es importante mencionar que se dejaron las funciones de Store y CSR para la segunda iteración del ambiente de verificación, además algunas de las funciones implementadas no se van a utilizar en la versión final porque se busca disminuir las instrucciones de la arquitectura y así simplificar la construcción del microprocesador.

Capítulo 6

Pruebas del ambiente

Por último, se tuvo que tomar la decisión de realizar pruebas sobre los bloques funcionales del ambiente de verificación y no hacer test sobre el dispositivo bajo verificación, esto debido a que al momento de finalizar el proyecto no se contaba con la implementación del diseño fullchip del microprocesador y era necesario verificar la funcionalidad de los componentes más importantes del ambiente.

Simulación Instruction Generator:

El primer componente en ser probado fue el que se denominó durante la etapa de diseño como *instruction generator*, el cual estaba compuesto por el sequence item y el sequence. Este componente tenía la función de generar instrucciones de prueba para ver el comportamiento del modelo de referencia. En la siguiente tabla se presentan 5 instrucciones de tipo R que se generaron para hacer pruebas de comunicación con el scoreboard y funcionamiento de la interface:

funct7	**rs2**	**rs1**	**funct3**	**rd**	**opcode**	**transmited**
0	31	27	2	24	51	00000001111111011010110000110011
0	0	10	2	16	51	00000000000001010010100000110011
0	30	2	5	4	51	00000001111000010101001000110011
0	26	22	3	19	51	00000001101010110011100110110011
0	9	13	7	15	51	00000000100101101111011110110011

Cuadro 10: Instrucciones de prueba

Estas instrucciones primeramente se limitaron a ser de tipo R debido a que solo se buscaba comprobar la generación aleatoria de secciones en las instrucciones como lo es el caso de los registros operandos y el registro destino, así como el funct3 para poder aleatorizar el tipo de operación que se deseaba revisar en el modelo de referencia; estos cambios se pueden observar en el Cuadro 10.

Simulación Instruction Sender:

Una vez generadas las 5 instrucciones para las pruebas, se revisaron la transmisión de todos los bits que componen esas instrucciones a través del driver y de la interface con el

DUV para así determinar que la recepción de las instrucciones no tenga errores con respecto al dato enviado. Es así que se realizaron pruebas con el EpWave de EdaPlayground para ver la simulación de la comunicación a través de la interface y comprobar la sincronización del protocólo SPI.

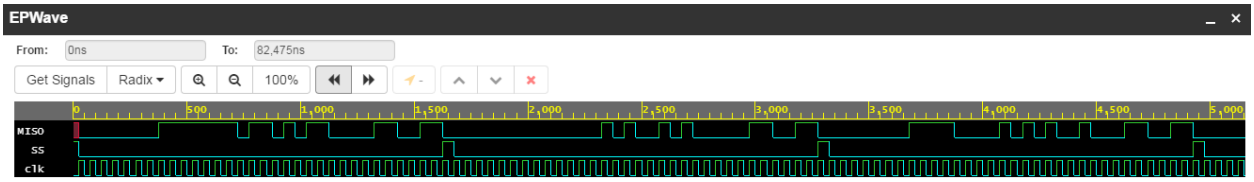

Figura 28: Diagrama de tiempos de envío de instrucciónes por interface SPI

Como se puede observar en la Figura 28 la señal de SS se generó de manera forzada ya que al no existir un dispositivo maestro con el cual comunicarse no existía la forma de habilitar el envío de las instrucciones por medio de la interface y por lo tanto SS se ponía en un nivel bajo cuando el driver hacia el envío de las instrucciones. Por otra parte en esa misma imagen se puede observar que los primeros bits de la primera instrucción en ser recibidos por el DUV son siete 0 los cuales corresponden al valor de *funct7*, posteriormente siguen siete veces 1, después sigue el código 01101011, después siguen cuatro veces un 0 y por último sigue el código 110011 el cual corresponde al opcode de la instrucción. Si se compara esa secuencia de bits con la primera instrucción generada que se presenta en el cuadro 10, se logra observar que son iguales y por lo tanto la instrucción se envía a través de la interface sin ninguna clase de error, si se comparan las demás secuencias de bits se concluye que en el el driver no existe ningún problema con la transmisión de las secuencias generadas por el ambiente de verificación.

Simulación Monitor:

Otro de los elementos importantes en el ambiente de verificación y que es necesario la comprobación de su funcionamiento es el monitor ya que permite que el modelo de referencia reciba u observe las secuencias que son enviadas a través de la interface para posteriormente responder de la misma manera que lo haría el DUV. La simulación del monitor se puede observar en la Figura 29, en la cual se puede ver que existen tres ciclos de reloj desde el momento que se genera la instrucción hasta el momento que se observa el primer bit de la secuencia y esto se debe a que uno de esos ciclos es lo que tarda la instrucción en generarse, el segundo ciclo es lo que tarda la instrucción en llegar al driver y el tercero es un ciclo de espera que se hace para la sincronización con el DUV.

Luego de eso se nota que todos los bits hasta el penúltimo se envían cada ciclo de reloj y en el último bit sucede la particularidad que existe un retraso de 1 ciclo, es decir, la lectura del último bit se da un ciclo después al esperado debido a que en el ciclo de reloj sobrante (el esta entre el penúltimo y el último bit de la instrucción) se hace la solicitud al sequence generator para la siguiente instrucción y el ambiente no permite realizar otra función extra a la misma vez, por lo tanto la lectura debe ocurrir un ciclo de reloj después a dicha solicitud.

Para corregir esto se intentó hacer una bandera que avisara cuando se observaba el último bit de la instrucción para así poder realizar la solicitud de la siguiente instrucción y de esa manera no hubiera un retraso en la observación del último bit por parte del monitor y eso implicaba que el driver debia esperar un ciclo para hacer la solicitud, sin embargo no funcionó esto y siempre se generaba el retraso.

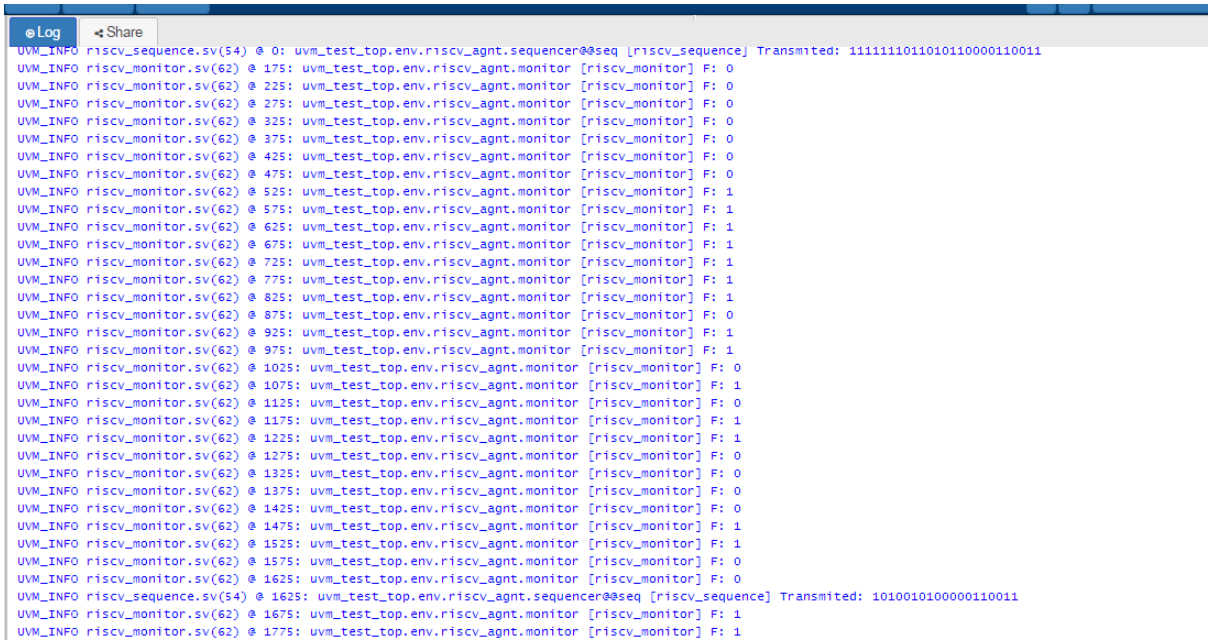

Figura 29: Diagrama de tiempos de envío de instrucciónes por interface SPI

Simulación Register Modeling:

Otro de los elementos a los que se le realizaron pruebas fueron el modelo de registros y los métodos para hacer el backdoor de los registros del DUV. Lo primero que se hizo fue hacer un RTL de prueba con un arreglo de 32 registros y el cual se conectó en lugar del microprocesador para así poder ligar el modelo de registros a un hardware bajo verificación, esto fue necesario hacer porque como ya se mencionó anteriormente para el momento de las pruebas no existía un diseño implementado del fullchip del microprocesador.

Entonces una vez listo el hardware de prueba, se enlazó el arreglo de registros en el DUV con en el modelo de registros del ambiente de verificación para posteriormente realizar una simulación. Cuando esto sucedió hubo un error de referencia al hardware, se puede observar en la Figura 30, y este error fue a causa de una redundancia en la referencia de la ruta de acceso al registro 0 del arreglo.

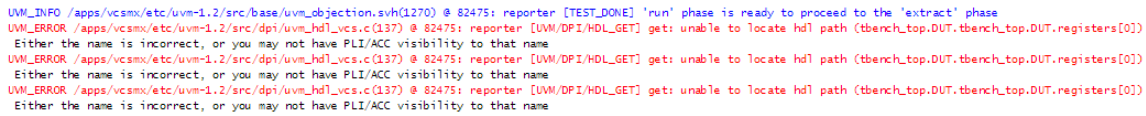

Figura 30: Error de referencia de hardware en el backdoor

Esta ruta del registro se describió como *"tbench_top.DUT.registers[0]"* y el bloque de registros tenía la ruta *"tbench_top.DUT"* por lo que se estaba haciendo referencia al DUV en dos secciones del modelo de registros y cosa que era incorrecto hacerlo así porque la metodología UVM permite hacerlo de una manera modular, es decir, primero se referencia al DUV en la parte del bloque de registros y posteriormente solo se referencia al registro de manera individual de la siguiente manera: *add_hdl_path(.path("tbench_top.DUT"))* es la ruta del DUV que se hace en el bloque de registros y R0.add_hdl_path_slice(registers[0]", 0, 32) es la ruta que se hace para referenciar al registro cero.

Además de esa prueba se comprobó el funcionamiento del método peek para extraer los valores almacenados en los registros del DUV por medio de un acceso Backdoor. Para hacer eso se debió crear un tipo de secuencia extra a las secuencias utilizadas anteriormente, estas son dedicadas a la comunicación entre el modelo de registros y los registros físicos cuando se hacen lecturas o escrituras por medio de un backdoor.

Para hacer las pruebas de este método se definió un valor estático para ser almacenado en un registro del arreglo presente en el DUV y posteriormente se debía imprimir el valor de la transacción efectuada guardado en el modelo de registros cuando se hacían las lecturas. Para hacer esta prueba se guardó un cero en R0 y un 5 en R6 y como se puede ver en la Figura 31 al hacer la lectura en los dos registros se comprueba que el ambiente de verificación si logró hacer las lecturas desde el hardware utilizando el acceso backdoor.

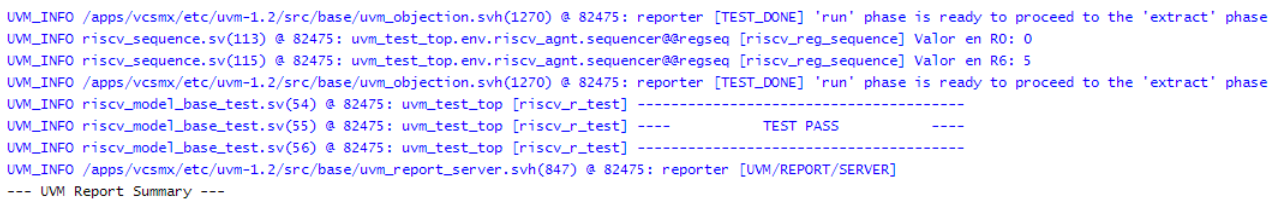

Figura 31: Comprobación de la extracción de datos en registros del DUV por medio de backdoor

Simulación Scoreboard:

Por último se realizaron las simulaciones que permitían comprobar el funcionamiento del Scoreboard. La primera prueba realizada para el scoreboard fué la etapa recepción y empaquetamiento de las instrucciones vistas desde la interface por medio del monitor, en la Figura 32 se puede ver que el scoreboard recibe la instrucción completa luego de que el ambiente

haya generado 2 instrucciones anteriormente. Esto debido a que el sistema se comporta como un pipeline en donde se genera una instrucción y después esta pasa a la sección que se encarga del envío, luego que se haya enviado cada bit se genera la siguiente instrucción mientras el scoreboard esta recibiendo los bits de la instrucción anterior observados por el monitor. En cuanto a los tiempos de recepción si se compara con la Figura 29 en donde el último bit de la instrucción es observado cuando el reloj es 1775, se puede ver que la instrucción es recibida por el scoreboard un ciclo después cuando el reloj es 1825 lo cual corresponde al retraso de pipeline que tiene el monitor y el scoreboard.

```
UVM_INFO riscv_sequence.sv(54) @ 0: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 11111110110101100000110011
UVM_INFO riscv_sequence.sv(54) @ 1625: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 1010010100000110011
UVM_INFO riscv_scoreboard.sv(99) @ 1825: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 11111110110101100000110011
UVM_INFO riscv_sequence.sv(54) @ 3275: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 11110000101010010000110011
UVM_INFO riscv_scoreboard.sv(99) @ 3475: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 1010010100000110011
UVM_INFO riscv_sequence.sv(54) @ 4925: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 11010101100111001101100011
UVM_INFO riscv_scoreboard.sv(99) @ 5125: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 11110000101010010000110011
UVM_INFO riscv_sequence.sv(54) @ 6575: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 10010110111101111110110011
UVM_INFO riscv_scoreboard.sv(99) @ 6775: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 11010101100111001101100011
UVM_INFO riscv_sequence.sv(54) @ 8225: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 10110000000100100011011
UVM_INFO riscv_scoreboard.sv(99) @ 8425: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 1001011011110111110110011
UVM_INFO riscv_sequence.sv(54) @ 9875: uvm_test_top.env.riscv_agnt.sequencer@@seq [riscv_sequence] Transmited: 10011001110010110000110011
UVM_INFO riscv_scoreboard.sv(99) @ 10075: uvm_test_top.env.riscv_scb [riscv_scoreboard] Received: 1011000000010010001100011
```

Figura 32: Recepción de instrucciones por el scoreboard

Finalmente se realizaron simulaciones para ver el comportamiento de la decodificación y funcionamiento de las instrucciones. Para eso se tuvo que hacer instrucciones de prueba por tipos de códigos de operación, es decir, se hacían simulaciones con instrucciones de un único opcode. En estas simulaciones se probaron algunas de las instrucciones implementadas en el modelo de referencia, las cuales fueron las instrucciones R,I y Load ya que estas involucran la interacción con los registros y memoria del microprocesador. También se intentaron de probar las instrucciones de control, sin embargo existió un error en la simulación en el calculo del *pc* que no se logró comprobar el funcionamiento de las instrucciones de branch y salto.

En primer lugar se simularon las instrucciones R y de estas se extrajeron algunos de los resultados obtenidos de la decodificación de cada instrucción con el fin de comprobar que esta etapa funcionaba correctamente y esos resultados fueron presentados en el Cuadro 11.

Por otra parte, en el Cuadro 12 se presentan los resultados de esas operaciones decodificadas. Además de eso es necesario aclarar que para poder realizar las operaciones se tuvo que asignar los valores almacenados en los registros operandos y hacer los calculos posteriormente para presentar los resultados generados por las instrucciones.

RS2	RS1	Funct3	RD	Función
31	27	2	24	SLT
26	18	5	4	SRL
26	22	1	4	SLL
26	22	3	4	SLTU
9	13	7	19	AND
25	15	4	30	XOR
9	24	0	14	ADD

Cuadro 11: Decodificación instrucciones R

Rs1	Rs2	Rd	Función
10101	-5	0	SLT
1001001	5	10	SRL
10101	5	1010100000	SLL
10101	4294967291	1	SLTU
10101	00101 (Binario)	101	AND
10101	101 (Binario)	10000	XOR
45	10	55 (Decimal)	ADD

Cuadro 12: Resultados de Operaciones R efectuadas

En el cuadro donde se presentan las decodificaciones se puede observar que el dato de mayor importancia es el *funct3* ya que es el que determina cual operación es la que se va a realizar luego de la decodificación de la instrucción, los demás datos son los indicadores de los registros que interactúan en cada instrucción realizada y se puede observar que existe repitencia de algunos registros lo cual es generado de manera .ªleatoria"por el ambiente de verificación.

Luego en el Cuadro 12 se puede observar que las instrucciones de comparación SLT y SLTU generaron resultados opuestos, es decir, que aunque los valores almacenados en los registros no variaron si existió un cambio numérico observado por el microprocesador debido que al tomar en cuenta el signo hace que el -5 sea visto como un número sin signo con magnitud de 4294967291 y provoca que ese dato sea mucho mayor al almacenado en Rs1 haciendo que la comparación de como resultado un 1. Por otra parte las operaciones de corrimiento si agregaron los 5 ceros a la derecha y a la izquierda respectivamente para hacer la modificación en los valores almacenados en Rs1 y guardarlos en Rd. Por último las instrucciones lógicas si cumplieron con la propiedad de funcionamiento bit a bit lo cual se puede ver de manera clara en la función XOR donde los primeros 4 bits de las palabras de Rs1 y Rd provocan un 0 en Rd y es el quinto bit el que genera un 1 en Rd. Además en la función de suma se nota que los valores se visualizaron como números decimales, donde se sumaba 10 y 45 lo que ocasionó un 55 en Rd.

De igual manera se probaron las instrucciones I en las cuales se obtuvo de manera aleatoria el valor del inmediato, algunos de los resultados obtenidos se presentan en el Cuadro 13 y en donde se puede ver que la comparación provoca un 1 en Rd al determinar que el inmediato es mucho mayor que el valor almacenado en Rs1. Las funciones lógicas también tienen un

funcionamiento bit a bit. La suma también presenta el resultado de forma decimal para comprobar que si funciona bien.

Rs1	**Imm**	**Rd**	**Función**
211	1031	1	SLTI
10101	11011000100	11011010101	ORI
10101	11011000100	11011010001	XORI
21	1065	1086	ADDI
10101	11000101001	1	ANDI
10101	6	10101000000	SLLI

Cuadro 13: Resultados de Operaciones I efectuadas

Por otra parte, las instrucciones de lectura desde la memoria fueron otras de las cuales se pudieron comprobar el funcionamiento. Para estas se calculaban la dirección de memoria base del primer byte y se guardaban cuatro números distintos en las siguientes 4 direcciones para asi tener un número de 32 bits en total. Luego de guardar esos valores se hacían lecturas a la memoria conociendo la dirección base del primer byte, esto se hizo con el fin de comprobar la carga del dato de 32 bits en el registro Rd.

Como se puede ver en el Cuadro 14 las cargas de cada palabra se hace con una concatenación de 4 bytes almacenados en direcciones de memoria seguidas, por lo que para poder apreciar esos valores cargados se separaron las cadenas de bits en grupos de 8. En el primer caso se guardaron y después cargaron los números 32, 43, 55 y 66; con eso se hizo la carga de una palabra completa. Después se hicieron la carga de un byte y de una media palabra lo que comprueba que las cargas de partes de una palabra también permiten ser utilizadas.

Add (Hexa)	**Dato (Binario)**	**Dato Separado**	**Función**
407	00100000 00101011 00110111 01000010	32 43 55 66	LW
e0	00100000	32	LB
4e0	00100000 00101011	32 43	LH
5e7	00100000	32	LBU
284	00100000 00101011	32 43	LHU

Cuadro 14: Resultados de Operaciones Load efectuadas

Finalmente se presenta en la Figura 33 lo ocurrido cuando se deseaba simular las instrucciones de Branch. Primero se deseaba imprimir el valor almacenado en el *pc* luego de cada instrucción, sin embargo, ese dato tomaba un valor desconocido y por eso era imposible revisar el funcionamiento de los branch ya que el valor en este registro es necesario para poder hacer los retornos luego de cada salto.

```
UVM_INFO riscv_scoreboard.sv(426) @ 3825: uvm_test_top.env.riscv_scb [riscv_scoreboard] Pc        x
UVM_INFO riscv_scoreboard.sv(357) @ 3875: uvm_test_top.env.riscv_scb [riscv_scoreboard] BGE

Warning-[STASKW_SFRTMATR] More arguments than required
riscv_scoreboard.sv, 426
  Number of arguments (1) passed to place in the format string are more than
  the number of format specifiers (0).

UVM_INFO riscv_scoreboard.sv(426) @ 3875: uvm_test_top.env.riscv_scb [riscv_scoreboard] Pc        x
UVM_INFO riscv_scoreboard.sv(357) @ 3925: uvm_test_top.env.riscv_scb [riscv_scoreboard] BGE

Warning-[STASKW_SFRTMATR] More arguments than required
riscv_scoreboard.sv, 426
  Number of arguments (1) passed to place in the format string are more than
```

Figura 33: Error en el calculo del PC

Conclusiones

Algunas de las cosas que se pudieron extraer luego del desarrollo de este proyecto y que se puede recalcar es que: La generación de instrucciones de prueba de manera aleatoria pueden ser utilizadas al momento de la verificación del microprocesador para encontrar corner cases, es decir, errores en eventos raramente vistos, por otra parte la generación de instrucciones por medio del toolchain de RISC-V permite verificar el comportamiento del desempeño del microprocesador.

También, la sincronización del protocolo SPI logró el envío de las instrucciones sin ninguna fuente de error, a pesar de la falta de un dispositivo maestro que generara la señal SS. También los retrasos en la propagación de las transacciones entre los componentes del bus (Sequencer, Driver y Monitor) provocaron que las instrucciones vistas por el modelo de referencia se puedan dar después de que el dispositivo bajo verificación las recibiera y provocando así un desfase en las ejecuciones del programa.

Por otro lado, la generación de una instrucción nueva no se pudo dar luego de que el monitor observara la instrucción actual, eso provocó un retraso de 2 ciclos de reloj en la observación de la instrucción por parte del modelo de referencia. En cuanto a las lecturas por medio del backdoor, no se logró poder determinar los registros que están siendo modificados por la instrucción al momento de realizar una lectura, por lo que estas se tuvieron que hacer de manera estática y quitando así autonomía a la verificación del DUV. Además es importante tener en cuenta que para poder realizar una lectura o escritura por acceso backdoor, es necesario conocer bien las características del elemento de memoria o registros a los cuales se va a hacer la verificación, así como la ubicación de estos en la jerarquía de memoria.

Por último y a pesar que no se logró implementar todo el ISA y que no se pudo comprobar todas las instrucciones implementadas, se logró comprobar el buen funcionamiento de las instrucciones que modifican los valores guardados en los registros y memoria, es decir las funciones que realizan calculos y movimientos de datos entre elementos de memoria.

También, se logró comprobar que las funciones implementadas en systemverilog lograron cumplir con las características de operación bit a bit que tiene el microprocesador. Además, las instrucciones de control de flujo de programa (Branch y Jump) no se pudieron comprobar debido a un fallo en el calculo del Program Counter.

Recomendaciones

Algunas de las recomendaciones que se pueden dar para las siguientes iteraciones del desarrollo del ambiente de verificación para el full chip es que se debe usar el toolchain de RISC-V para utilizar un programa hecho en C y generar instrucciones de prueba al momento de hacer la verificación del bootstrap del microprocesador, esto con el fin de probar el microprocesador con software real y ver la respuesta que el microprocesador genera. También una vez obtenido ese programa en lenguaje ensamblador RISC-V se recomienda implementar la funcionalidad en el ambiente de leer instrucciones de un archivo de texto para así tener un programa que no presente problemas en el flujo de ejecución.

También es necesario implementar las interfaces AXI y UART para posteriormente hacer pruebas de la verificación del mapeo de memoria del microprocesador. Además es muy importante terminar la implementación del ISA y consultar cuales son las instrucciones que se van a eliminar en la implementación final del microprocesador. Además se deben de verificar el funcionamiento de esas instrucciones faltantes.

Es necesario saber utilizar el adaptador y el predictor del modelo de registros para poder hacer de manera automática la identificación de los registros a los cuales hay que hacer la lectura por Backdoor, para de esta manera darle autonomía al ambiente al momento de realizar las pruebas en el servidor. Por último es muy importante realizar el coverage del modelo de registros del microprocesador y así determinar la cobertura en la utilización de la memoria y registros.

Bibliografía

[1] Goss, J., Roesner, W. (2005).Comprehensive Functional Verification: The Complete Industry Cycle. Systems on Silicon.

[2] Wang, L. T., Wu, C. W., Wen, X. (2006). VLSI Test Principles and Architectures. VLSI Test Principles and Architectures.

[3] (Accellera), A. S. I. (2015). Universal Verification Methodology (UVM) 1.2 User ' s Guide., 190.

[4] (Accellera), A. S. I. (2014). Universal Verification Methodology (UVM) 1.2 Class Reference, (June), 938.

[5] IEEE Computer Society, IEEE Standards Association Corporate Advisory Group. (2013). IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language, 2012(February), 1315.

[6] Verification Guide. (2016). UVM Phases [online]. Extraído de:
http://www.verificationguide.com/p/uvm-phases.html

[7] Chip Verify. (2015). UVM Phases [online]. Extraído de:
http://www.chipverify.com/uvm/phasing

[8] Akkem, S. (2015). UVM RAL : Registers on demand, 1–15. Extraído de:
https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/24_UVM_RAL_Registers_On_Demand_paper.pdf

[9] Grusin, M. (2013). Serial Peripheral Interface (SPI) [online]. Extraído de: https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

[10] Perez, E. L. (2008). INGENIERIA EN MICROCONTROLADORES Protocolo SPI(Serial Peripherical Interface). Technology, 1–10. Extraído de:
http://www.i-micro.com/pdf/articulos/spi.pdf

[11] Microchip Technology. (2011). 256K SPI Bus Low-Power Serial SRAM, (23), 28. Extrído de: http://ww1.microchip.com/downloads/en/DeviceDoc/22100F.pdf

[12] Krste Asanovic, Rimas Avizienis, Jacob Bachmeyer, Christopher F. Batten, A., J. Baum, Alex Bradbury, Scott Beamer, Preston Briggs, Christopher Celio, David Chisnall, P., Clayton, Palmer Dabbelt, Stefan Freudenberger, Jan Gray, Michael Hamburg, John Hauser, D., Horner, Olof Johansson, Ben Keller, Yunsup Lee, Joseph Myers, Rishiyur Nikhil, S. O., Albert Ou, John Ousterhout, David Patterson, Colin Schmidt, Michael Taylor, W. T., Matt Thomas, Tommy Thorn, Ray VanDeWalker, Megan Wachs, Andrew Waterman, R. W.-, Son, and R. Z. (2017). The RISC-V Instruction Set Manual v2.2. 2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings, I, 1–32.

[13] Verification Guide. (2016). UVM TestBench Example [online]. Extraído de: http://www.verificationguide.com/p/uvm-testbench-example.html

[14] Shimizu,K. (2015). UVM Tutorial for Candy Lovers – 24. Register Access through the Back Door [online]. Extraído de: http://cluelogic.com/2014/09/uvm-tutorial-for-candy-lovers-register-access-through-the-back-door/

[15] Verification Academy: Mentor. Registers/Adapter [online]. Extraído de: https://verificationacademy.com/cookbook/registers/adapter

[16] VMM Central. (2018). About VMM for SystemVerilog [online]. Extraído de: https://www.vmmcentral.org/aboutus.html

[17] Doulos. (2008). Getting Started with OVM: A Series of Tutorials based on a set of Simple, Complete Examples [online]. Extraído de: https://www.doulos.com/knowhow/sysverilog/ovm/tutorial_0/

Apéndice

Set de Instrucciones RISC-V 32I [12, pág. 04]

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 | pred	succ	00000	000	00000	0001111	FENCE
0000 | 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

Figura 34: Set de instrucciones ISA RISC-V 32I

Pseudoinstruction	Base Instruction(s)	Meaning
`la rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`call offset`	`auipc x6, offset[31:12]` `jalr x1, x6, offset[11:0]`	Call far-away subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Pseudoinstruction	Base Instruction	Meaning
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`fsrmi rd, imm`	`csrrwi rd, frm, imm`	Swap FP rounding mode, immediate
`fsrmi imm`	`csrrwi x0, frm, imm`	Write FP rounding mode, immediate
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags
`fsflagsi rd, imm`	`csrrwi rd, fflags, imm`	Swap FP exception flags, immediate
`fsflagsi imm`	`csrrwi x0, fflags, imm`	Write FP exception flags, immediate

Figura 35: Nemonicos de ensamblador RISC-V 32I