Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Diseño e implementación de un ambiente de verificación funcional con estándar UVM para segunda versión de microcontrolador Siwa.

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

Presenta:

Fabián Alberto Picado García

Cartago, Costa Rica

22 de junio de 2022

INSTITUTO TECNOLÓGICO DE COSTA RICA
ESCUELA DE INGENIERÍA ELECTRÓNICA
TRABAJO FINAL DE GRADUACIÓN
ACTA DE APROBACIÓN

Defensa del Trabajo Final 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 Trabajo Final de Graduación denominado Diseño e implementación de un ambiente de verificación funcional con estándar UVM para segunda versión de microcontrolador Siwa, realizado por el señor Fabián Alberto Picado García, 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

PABLO DANIEL MENDOZA PONCE (FIRMA)
Firmado digitalmente por PABLO DANIEL MENDOZA PONCE (FIRMA)
Fecha: 2022.06.08 09:59:14 -06'00'

Ing. Pablo Mendoza Ponce
Profesor lector

JORGE ALBERTO CASTRO GODINEZ (FIRMA)
PERSONA FISICA, CPF-01-1236-0930.
Fecha declarada: 09/06/2022 07:53:46 AM
Razón: Aprobación de acta

Ing. Jorge Castro Godínez
Profesor lector

ALFONSO CHACON RODRIGUEZ (FIRMA)
PERSONA FISICA, CPF-01-0702-0796.
Fecha declarada: 09/06/2022 02:49:01 PM
Razón: Doy fe
Lugar: Tres Rios Contacto: alchacon@tec.ac.cr

Ing. Alfonso Chacón Rodríguez
Profesor asesor

Cartago, 8 de junio de 2022

Yo, Fabián Alberto Picado García, declaro que el presente Trabajo Final de Graduación ha sido realizado en su totalidad por mi persona, aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos 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.

Fabián A. Picado.

Fabián Alberto Picado García
Cédula 1-1748-0068
Cartago, 20 de junio de 2022

Resumen

Actualmente la segunda versión del microcontrolador Siwa se encuentra en desarrollo, el cuál es un dispositivo que se ha elaborado completamente en la Escuela de Ingeniería Electrónica del Tecnológico de Costa Rica, como parte de las labores de profesores-investigadores asociados y los aportes de estudiantes. Como parte del flujo de desarrollo de circuitos integrados en general, es de suma importancia realizar revisiones periódicas al diseño, a fin de cuentas de evitar errores o posibles discrepancias entre la especificación funcional y el comportamiento observado del dispositivo bajo prueba (DUT). Para ello, en este proyecto se expone la metodología seguida para obtener una plataforma de pruebas o ambiente de verificación funcional que permita ejercitar el DUT. Tal camino inicia con la recolección de características claves del DUT, con el objetivo de formular un plan de pruebas. Una vez se obtuvo dicho plan de pruebas, se propuso una arquitectura de ambiente de verificación funcional acorde al estándar UVM. Finalmente, una vez concluido la conceptualización de la plataforma de pruebas, se muestran los resultados obtenidos, un breve análisis acorde a los errores obtenidos y las métricas de cobertura que generaron, a partir de la ejecución de cinco regresiones completas.

Palabras clave— Circuitos integrados digitales, microcontroladores, verificación funcional.

Abstract

Currently the second version of the Siwa microcontroller is under development, which is a device that has been completely developed in the Escuela de Ingeniería Electrónica of the Tecnológico de Costa Rica, as part of the work of associated professors-researchers and the contributions of students. As part of the integrated circuit development flow in general, it is very important to carry out periodic revisions to the design, in order to avoid errors or possible discrepancies between the functional specification and the observed behavior of the Device Under Test (DUT). This project exposes the methodology followed to obtain a testing platform or functional verification environment that allows the DUT to be exercised. Such a path begins with the collection of key characteristics of the DUT, with the objective of formulating a test plan. Once the test plan was obtained, a functional verification environment architecture according to the UVM standard was proposed. Finally, once the conceptualization of the test platform has been completed, the results obtained are shown, as well as a brief analysis according to the errors obtained and the coverage metrics they generated, from the execution of five complete regressions.

Keywords— Digital integrated circuits, functional verification, microcontrollers.

Agradecimientos

Gracias mamá, papá y hermano, lo logramos . . .

Una vez concluido estos 5 años de camino en el TEC, me percato de que a lo largo de este recorrido he estado rodeado de personas que me han ayudado a salir adelante.

Primeramente, agradezco de la manera más profunda y sincera a mi mamá y a mi papá, quienes me han apoyado y acompañado incansablemente durante toda mi vida, sin ustedes no lo hubiera logrado. Gracias por creer en mí y contagiarme de su carácter esforzado y espiritú alegre, positivo y honesto. Asimismo, agradezco a mi hermano por ser un gran ejemplo a seguir.

También le doy a las gracias a los profesores Alfonso Chacón, Ronny García y José Miguel Barboza por la pasión y entrega en los cursos que imparten, así como la paciencia y vocación por transmitir sus conocimientos y tener el interés real de formar profesionales que contribuyan al desarrollo del país.

A mi padrino y a mis tías, amigos y amigas, gracias por su apoyo.

Finalmente, para mi querida abuelita Carmen, sé que debes de estar muy feliz por este proyecto y todo lo que ha conllevado.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Actualmente, gracias al avance en ramas de la ingeniería como la microelectrónica y la confiabilidad y precisión en los procesos de manufactura, entre otros, es posible construir circuitos integrados con transistores (bloque fundamental) cuyas dimensiones rondan los 7 nm o 5 nm [4]. Es decir, tal como menciona Intel en una infografía sobre el proceso tecnológico de 22 nm, la humanidad es capaz de colocar, por ejemplo, "6 millones de transistores *tri-gate* en el punto final de esta oración" [5]. No obstante, para conseguir tal hito es imprescindible utilizar metodologías estandarizadas y revisiones periódicas del diseño, a fin de que la inversión en tiempo y recursos (tanto humano como económico) sea maximizado u optimizado lo más posible. Si se deja por un lado el proceso de la contextualización del diseño, lo anterior conlleva a la rama de ingeniería electrónica que se conoce Verificación Funcional de Circuitos Integrados. Dicha área de estudio determina las técnicas, pautas y métricas que permiten establecer cómo ejercitar el dispositivo bajo prueba y así, por consiguiente, concluir con cierto nivel de confianza que el diseño puede ser utilizado con seguridad por parte del cliente o sector de la sociedad meta. Esto último establece un gran abánico de escenarios y riesgos posibles en función de la intención de uso final, puesto que se puede tratar de un chip que operará en algo relativamente simple como una calculadora de escritorio o algo tan complejo como la computadora abordo de un avión comercial en medio del oceáno, incluso en dispositivos médicos vitales como un desfibrilador implantable, por mencionar algunos. En el presente informe, se presenta la metodología seguida para desarrollar un ambiente de verificación funcional para la segunda versión del microcontrolador Siwa, utilizando la metodología estándar llamada UVM. De este modo, se contará con una herramienta de software que permita validar el comportamiento de Siwa, a la luz de su especificación funcional y en apego de las directrices de la arquitectura RISC-V.

1.1. Entorno

Siwa es un microcontrolador que fue diseñado completamente en Costa Rica, por parte de un equipo de investigadores y estudiantes de la Escuela de Electrónica del Tecnológico de Costa Rica [4]. A diferencia de un microprocesador, Siwa integra, entre otros, unidades para la comunicación e interacción con dispositivos periféricos, lo cuál lo convierte en un microcontrolador, ya que se incorporan bloques adicionales al núcleo de procesamiento [4]. Aunque su diseño puede ser capaz de fungir como un microcontrolador de uso general, en su primer iteración de construcción fue ideado

como parte de un sistema estimulador cardiaco, orientado a un escenario de aplicaciones médicas [6].

Actualmente en la la Escuela de Electrónica del Tecnológico de Costa Rica, se encuentra en desarrollo una segunda versión del microcontrolador Siwa, el cual, a grandes rasgos, incorpora un segundo núcleo de procesamiento al sistema, con los retos, modificaciones y consideraciones de diseño que ello plantea. Además, se ha propuesto el objetivo que en esta nueva revisión, Siwa sea orientado a un escenario de uso más general. También hereda la característica de bajo consumo de potencia, aunque puede ser fácilmente ampliable a un mayor consumo. Otras características que se mantienen con respecto a la iteración anterior, es el empleo de una arquitectura RISC-V RV32I.

La fabricación y diseño de Circuitos Integrados (CI), de manera general, se apega a una sucesión de pasos ordenados, dada la complejidad de la tarea y la cantidad de personas que se requieren. Uno de dichos pasos, corresponde al diseño lógico de la microarquitectura que sustenta el funcionamiento meta del dispositivo. Para concretar el paso anterior, es necesario emplear un nivel de abstracción que permita al equipo diseñador traducir las ideas de diseño derivadas de especificaciones y requerimientos hacia un lenguaje de descripción de hardware (HDL, por sus siglas en inglés), con el fin de utilizar herramientas de diseño electrónico asistido por computadora (en inglés, EDA) [7].

De este modo, tanto al momento de construir los dispositivos lógicos, como a la hora de traducir a HDL, es posible que se susciten problemas o errores de diseño. Lo anterior da lugar a la rama de Verificación Funcional de Circuitos Integrados, cuyo objetivo es comprobar y validar un diseño (ya sea a nivel de sistema, a nivel de bloque, etc) de CI, a través de pruebas exhaustivas, antes de proseguir con las demás etapas del flujo de diseño de CI [8]. Más a fondo, se encarga de encontrar y reportar problemas funcionales que puedan surgir durante las diferentes pruebas de validación que se realicen, permitiendo que estos sean solucionados y reparados a la brevedad posible [8].

El resultado final de un proceso de verificación funcional es el de asegurar que un dispositivo cumple con los requisitos de diseño y se comporta de manera correcta en los casos de uso posible [8]. De manera antagónica, se evita a toda costa que un error no encontrado durante la fase de diseño, sea determinado por un cliente al utilizar el producto final; donde dicho cliente puede ser desde un usuario de oficina con su computadora personal o la computadora abordo de un avión de uso comercial en pleno vuelo.

Ahora bien, para llevar acabo la tarea de verificación funcional, usualmente se basa en el paradigma de Programación Orientada a Objetos (OOP, por las iniciales en inglés), ya que existe analogía entre la forma de construir hardware con interconexión de módulos que poseen funciones preestablecidadas y el uso de clases/objetos con métodos definidos que se comunican entre sí [9]. Esto da lugar a lenguajes de verificación de hardware (HVL, por sus siglas en inglés). Aunque existen diferentes alternativas, una de ellas es SystemVerilog, como un lenguaje HVL y HDL, ya que permite construir un entorno de verificación y generar código sintetizable en un mismo lenguaje.

Con el uso de SystemVerilog, es posible dar solución a la tarea de verificación funcional desde diferentes puntos de vista. Esto conlleva a diferentes metodologías de verificación, desde no estandarizadas hasta estandarizadas por gran parte de compañías proveedoras de herramientas EDA. Para el caso del presente proyecto, se establece como requisito el empleo de la Metodología de Verificación Universal (en inglés, UVM). Los principales beneficios de emplear UVM para la validación de un diseño son:

- Es una metodología con abundancia de documentación libre.
- Soportada por las principales compañías de software EDA.

- Permite una fácil portabilidad y escalabilidad del ambiente de verificación entre equipos de trabajo y diferentes revisiones de un mismo producto.
- Elimina la necesidad de codificar aspectos cotidianos dentro un ambiente verificación y traslada a la persona validadora la tarea exclusiva de definir, ejecutar y analizar pruebas al dispositivo bajo prueba (DUT, en inglés).

1.2. Síntesis del problema

¿Como podría validarse el diseño de la segunda versión del microcontrolador Siwa?

1.3. Enfoque de la solución

La importancia de la verificación de un diseño de circuito integrado, radica en la búsqueda de errores o problemas potenciales que se evitan manifestar en etapas más tardías del proceso de diseño, donde su repercusión a nivel económico y en tiempo invertido, sería considerable. Mediante el presente proyecto, se pretende desarrollar una plataforma de verificación funcional para el diseño de la nueva versión del microcontrolador Siwa.

Lo anterior implica, después de un estudio de alto nivel del dispositivo bajo prueba, un proceso de constitución de un plan de pruebas con todas las funcionalidades y características que se desean verificar, tanto para escenarios de uso común como aquellos casos especiales que podrían desencadenar problemas. Estos últimos se conocen como casos de esquina.

Seguidamente, una vez definido el plan de pruebas, es necesario el diseño de la estructura que dará soporte a las tareas de verificación. Para ello, existen dos importantes ramas o alternativas, donde se parte de una metodología estandarizada o una arquitectura personalizada. Dada la complejidad de la verificación en general y la necesidad de adoptar técnicas o conceptos de programación y de manejo de software, el uso de alternativas universales genera un nivel de abstracción donde las particularidades sobre software en general han sido ocultadas. De este modo, se busca que la persona verificadora se encargue únicamente de las tareas que le atañen, sin profundizar considerablemente en el manejo de las herramientas de simulación. Caso contrario de una plataforma personalizada, donde la persona encargada de validación diseña todo desde cero.

De la mano con la adopción de una metodología estandarizada, se obtienen una serie de bloques o módulos previamente definidos, donde la tarea se reduce únicamente a codificar las modificaciones pertinentes al dispositivo bajo prueba, por ejemplo, como la implementación del protocolo de comunicación en los pines de entrada y la necesidad de un interprete adecuado en los pines de salida. Esto manifiesta la relevancia de un estudio previo del DUT, pues la emulación o implementación de protocolos irá en función del diseño bajo análisis. Además, en conjunto con las modificaciones anteriores, también corresponde como tarea de la persona verificadora, establecer las comunicaciones necesarias entre los bloques que componen la plataforma para la ejecución de pruebas.

Finalmente, como posibles restricciones en cuanto al desarrollo de la solución, se visualiza el requisito de contar con una licencia de software para verificación funcional. Sin embargo, debido a que

el proyecto es parte de las actividades de investigación de profesores en la Escuela de Electrónica del Tecnológico de Costa Rica, se conoce de antemano que será posible contar con licencia para ejecutar un simulador comercial de verificación funcional.

Además, como se mencionó anteriormente, el diseño de la segunda versión del microcontrolador Siwa aún se encuentra en desarrollo, por lo que para evitar contratiempos en el diseño e implementación del ambiente de verificación que se plantea en este proyecto, se utilizará un modelo de alto nivel proporcionado por el estudiante Ángelo Hernández Cordero como parte de su trabajo de graduación, cuyo abordaje se realizará de manera paralela. Este módelo de alto nivel funcionará como el dispositivo bajo prueba del ambiente de verificación funcional del presente proyecto y contempla las estructuras que se muestran en la región delimitada de color rojo en la Figura 1.1. Nótese que el DUT no contempla las interfases I2C, SPI y UART, así como los pines de interrupciones externas y los arreglos de GPIO.

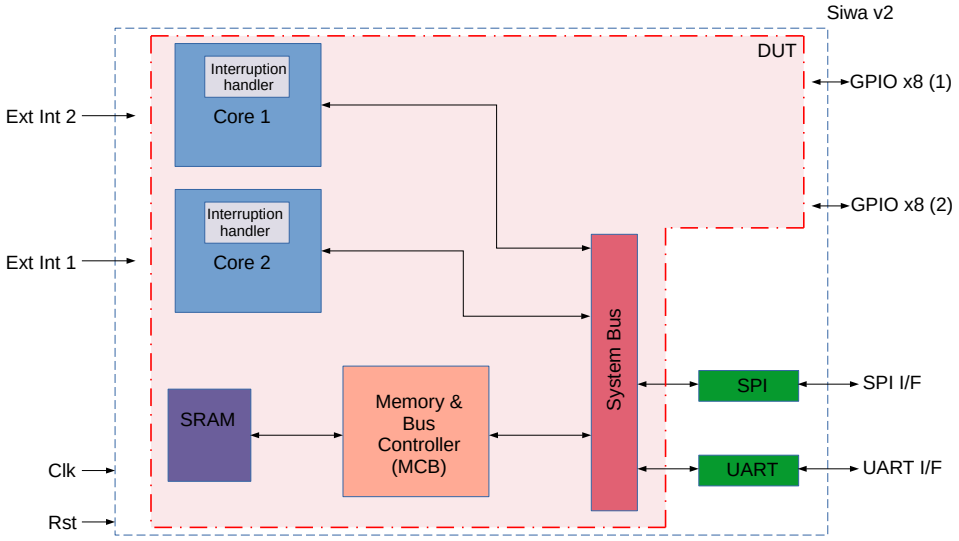

Figura 1.1. Boceto de la posible arquitectura de la segunda versión del microcontrolador Siwa y la delimitación del diseño que será considerado como DUT para el presente proyecto.

Con respecto a las posibles soliciones, se considerará como objeto de análisis la manera en la que se llevará a cabo la tarea de verificación, ya que sin importar la alternativa elegida, siempre será requisito cumplir con un plan de pruebas previo y la codificación de los manejadores e intérpretes de estándares de diseño o protocolos de comunicación que integran el DUT. En cuanto a paradigmas de programación en software, se parte de la necesidad de utilizar OOP. Así, tales soluciones factibles son:

1. Desarrollo de la plataforma de verificación mediante el lenguaje SystemVerilog, sin la adopción alguna de prácticas estandarizadas a nivel industrial.
2. Desarrollo de un ambiente de verificación, en conjunto con la codificación de pruebas, mediante UVM.
3. Desarrollo de una plataforma de verificación mediante síntesis del RTL en FPGA.

Como parte de las necesidades del proyecto al cual se ha adherido el presente trabajo, es requisito utilizar el estándar UVM para concretar la tareas de verificación funcional. Además, siempre será necesario partir de un estudio previo del DUT y de la elaboración de un plan de pruebas exhaustivo, así como el diseño de las métricas de cobertura funcional. De este modo, las posibles alternativas de solución se reducen a como ejecutar el proceso de verificación, tomando en cuenta los criterios mencionados anteriormente.

Aunque se presentan tres posibles soluciones, se supondrá por un momento que se desconoce el requisito de emplear algún estándar de verificación, por lo que el fin es discernir y justificar el porqué del estándar UVM solicitado.

Con el fin de establecer un proceso cuantitativo a la hora de discernir entre las potenciales soluciones encontradas, se utilizará la herramienta de Matriz de Pugh [10].

A continuación se presenta una lista de criterios que se han fijado como parte del análisis de las alternativas de diseño de la propuesta de la solución. Así mismo, tales criterios se utilizaron como insumo de la Tabla 1.1. Nótese que la numeración del criterio corresponde de manera resumida al criterio mostrado en dicha Tabla.

1. Módulos de verificación reutilizables y adaptables a futuras revisiones de Siwa.
2. Capacidad de ejecutar diferentes pruebas en tiempo de corrida de simulación, sin necesidad de volver a compilar.
3. Facilidad de entender el código desarrollado por parte de otras personas del equipo de verificación.
4. Tiempo necesario para lograr ejecutar la primera prueba.
5. Necesidad de plataforma de hardware como parte de la plataforma de verificación.

Tabla 1.1. Matriz de Pugh.

Criterios		**Solución 1**	**Solución 2**	**Solución 3**
1	Reutilización del código	-1	+1	+1
2	Diferentes pruebas sin necesidad de compilar de nuevo	-1	+1	+1
3	Facilidad de interpretación del código	-1	+1	+1
4	Inversión de tiempo	-1	+1	-1
5	Hardware requerido	+1	+1	-1
Suma positivos (+)		1	5	2
Suma negativos (-)		4	0	3
Suma general		-3	+5	-1

A partir de la información mostrada en el Tabla 1.1, se concluye que la solución óptima radica en utilizar el estándar UVM. Esto debido a que será posible desarrollar una plataforma que permita la reutilización de código y una comprensión sencilla por parte de terceras personas que podrían involucrarse en el diseño y validación de Siwa. Además, gracias a los niveles de abstracción y la codificación de funciones y aspectos comunes de manejo de software que emplea UVM, la inversión de tiempo será menor, pues únicamente será necesario dedicarse a las tareas de verificación funcional.

1.4. Objetivos

1.4.1. Objetivo general

Desarrollar un ambiente de verificación funcional para la validación del diseño de la nueva versión del microcontrolador Siwa mediante el empleo de la Metodología de Verificación Universal (UVM) y la herramienta de verificación funcional VCS.

1.4.2. Objetivos específicos

1. Identificar las características del diseño bajo prueba (DUT, por sus siglas en inglés).
2. Formular un plan de pruebas acorde a las características funcionales del DUT.
3. Establecer la arquitectura del ambiente de verificación mediante el estándar UVM, así como la composición de paquetes o transacciones y los canales de comunicación entre módulos.
4. Implementar la arquitectura del ambiente de verificación.

1.5. Estructura del documento

El capítulo 2 contiene una breve revisión de los conceptos técnicos que fueron necesarios para dar el sustento a la propuesta de solución y el análsis de resultados. Seguidamente, en el capítulo 3 se engloba todo el proceso que se siguió para dar lugar al ambiente de verificación planteado, comenzando con un estudio de las capacidades y características principales de la nueva versión de Siwa a la luz de lo que fué la primera iteración, luego se incluye un plan de pruebas que permitió verificar el funcionamiento básico esperado del DUT y por último se adjunta la arquitectura del ambiente de verificación y un compendio de los aspectos importantes en el diseño de cada uno de los componentes del banco de pruebas. De este modo, se llega al capítulo 4 para mostrar los resultados e información obtenida gracias a la formulación del plan de pruebas y la arquitectura propuesta. En dicho capítulo se muestra la metodología que se siguió para estandarizar la ejecución de los comandos y software necesarios que permitieron correr las simulaciones y el software externo de modelo de referencia, así como la realización de multiples iteraciones de regresiones completa y la visualización de base de datos de cobertura. Por último, en el capítulo 5 se reunen las conclusiones del proyecto y algunas recomendaciones que serán útiles en trabajos futuros.

En el apéndice A se enlistan las abreviaciones que se utilizaron a lo largo del documento y su breve significado.

Capítulo 2

Marco teórico

Como parte de la ejecución del ambiente de verificación y la posterior recolección de los resultados, es importante mostrar algunos conceptos técnicos de manera breve.

2.1. Fundamentos

Bajo la premisa de construir un ambiente de verificación reutilizable en el tiempo y de fácil mantenimiento entre las personas del equipo de verificación, entre otros, surge la necesidad de emplear metodologías estandarizadas de verificación funcional. Una de ellas corresponde a UVM, la cual es el fruto de distintas experiencias anteriores en metodologías de verificación y creada a partir de un consorcio las compañias EDA mas representativas del mercado [11, 12].

UVM es una biblioteca codificada en el lenguaje SystemVerilog y proporciona la base de componentes de verificación que permiten crear arquitecturas de banco de pruebas [12]. Está construida bajo un paradigma de programación OOP.

Por otro lado, el dispositivo bajo prueba será una versión de Siwa modelada en un lenguaje de alto nivel. Más adelante se hará mención de ello, especificamente en la sección 3.1 referente al estudio del DUT, donde se incluyen algunos aspectos relevantes a la tarea de verificación funcional.

Con el objetivo de guiar al DUT hacia escenarios de prueba diversos pero que tenga sentido, es necesario emplear la aleatorización controlada; es decir, por ejemplo, no sería correcto dejar que el simulador decida de manera aleatoria el valor lógico de una línea de bit, puesto que esta podría adquirir valores 1, 0, X (condición no importa) o, incluso Z (alta impedancia) [13]. Por ello es importante utilizar restricciones o *constraints* que permitan acotar el rango de escenarios posibles del espacio total de opciones [13].

Seguidamente, como parte del análisis posterior a la recolección de resultados arrojados por el ambiente de verificación funcional que se describirá en el capítulo 3, es importante definir el concepto de cobertura y sus alcances. La cobertura se define como una métrica para cuantificar la calidad y alcance del esfuerzo de verificación realizado [8]. Dicho concepto adquiere mayor relevancia cuando se utilizan estrategias de aleatorización controlada, puesto que el análisis de cobertura será la metodología a seguir para caracterizar si realmente tal aleatorización está teniendo el efecto planteado sobre el DUT, a partir de lo descrito en un plan de pruebas previo [13].

A la hora de formular las diferentes técnicas y subcategorias de cobertura que existen, para efectos

del presente proyecto, se tomaron en cuenta las alternativas provistas por las herramientas de software de Synopsys, como cobertura de:

- Línea o *line*: cuantifica cuantas líneas de código RTL han sido ejecutadas durante una simulación [13].
- *Toggle*: medición de conmutación o transición de valor lógico durante simulación [13]. Por ejemplo, más comunmente, transiciones de 1 lógico a 0 lógico y viceversa.
- *Branch*: permite caracterizar las estructuras condicionales sintetizables (sintáxis `if-else`, por ejemplo), desde el punto de vista de la dirección del flujo de código que se toma durante una simulación [13].
- Máquina de estados (FSM): hace posible la cuantificación de cuantos estados, del espacio total posible, han sido ejecutados [13].

Capítulo 3

Propuesta de solución

3.1. Estudio del DUT

Tomando como referencia la arquitectura desarrollada en la primera versión, en este caso se trata de un SoC de doble núcleo, considerando las modificaciones y retos que ello supone. Es importante rescatar que la segunda versión de Siwa aún se encuentra en desarrollo, por lo que las características, interconexiones y funcionalidades que aquí se consideran podrían cambiar en futuras iteraciones de este mismo modelo de Siwa. No obstante, el trabajo recabado ha sido presentado y comunicado con integrantes del equipo diseñador. En la misma línea, lo anterior justifica la razón del por qué se utilizará un modelo de alto nivel del nuevo Siwa como DUT que será ejercitado en el ambiente de verificación funcional.

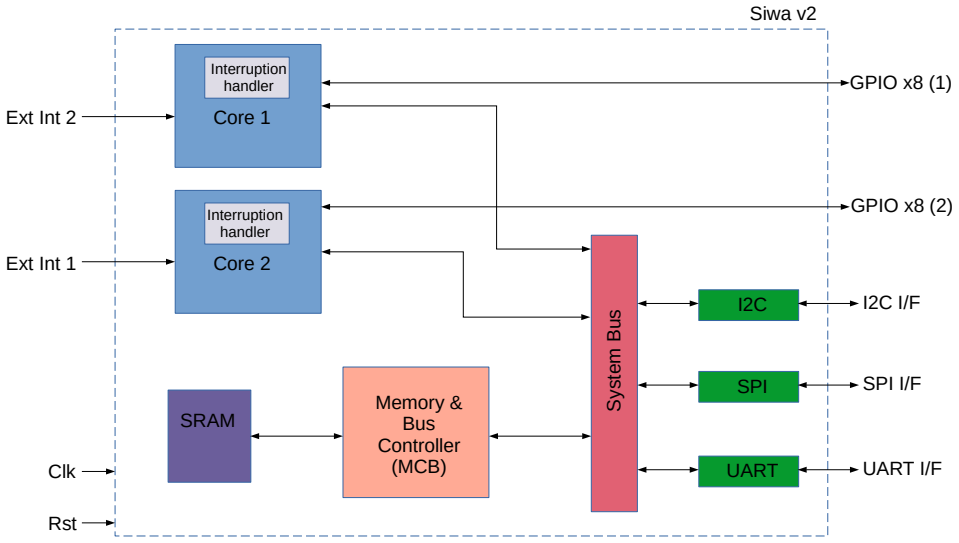

Figura 3.1. Diagrama de la arquitectura de la segunda versión del microcontrolador Siwa.

Para efectos del presente proyecto se ha delimitado el dispositivo bajo prueba en la región de color

rojo que se muestra en la Figura 3.2. Nótese que aquí han quedado por fuera las interfases SPI, I2C y UART, por lo que no será necesario caracterizar su comportamiento ni contemplar su protocolo en el plan de pruebas que se presentará más adelante. Con respecto a la conexión equivalente de I2C con el bus del sistema, esta se ha descartado por completo del presente proyecto, ya que aún no se tiene certeza de si será o no contemplada en el diseño final de la segunda versión del microcontrolador. La misma decisión se tomó para el caso de la conexión de arreglos de pines GPIO para cada núcleo, así como la entrada de interrupciones externas. Esto corresponde a una recomendación por parte del equipo asesor, con el fin de acotar la tarea de investigación y campo de acción.

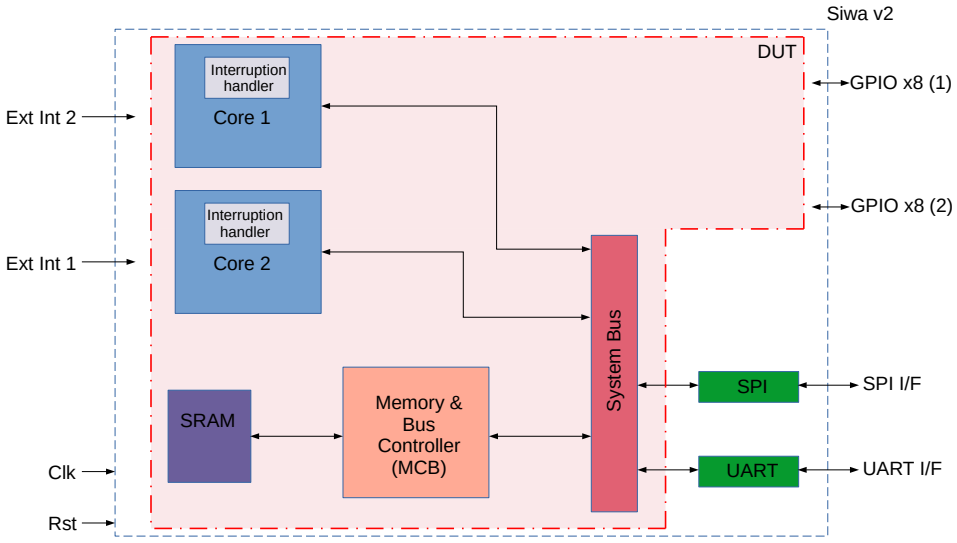

Figura 3.2. Definición del DUT a partir del diagrama de la arquitectura de la segunda versión de Siwa.

3.1.1. Núcleo Siwa

Es similar en cuanto a la microarquitectura multiciclo que implementó en la primera versión de Siwa [14]. Se apega al estándar de RISC-V en gran medida (utiliza el conjunto de instrucciones RV32I), pues presenta algunas diferencias a la hora de implementación de los registros de control y estado (CSR, por sus siglas en inglés), ya que el estándar propone direcciones de 12 bits para identificar cada CSR, lo que conlleva a un espacio de 4096 registros por implementar. Sin embargo, por razones de ahorro de área en el chip final, las direcciones se han reducido a 4 bits de longitud y, por consiguiente, 16 CSRs en total aunque solo 8 CSRs finalmente codificados en el RTL [1]. En el Tabla 3.1 se referencian los CSRs que incluye Siwa.

Tabla 3.1. Registros de control y estado (CSRs) que implementa Siwa [1] [3].

Nombre del CSR	**Dirección dada por estándar RISC-V**	**Dirección en el espacio de Siwa**	**Comentarios**
mie	0x304	0x00	-
mip	0x344	0x01	
mepc	0x341	0x02	
mvtec	0x305	0x06	
mcauseA	0x342	0x03	El equivalente según las especificaciones RISC-V sería *mcause*.
mcauseB (`trapB`)	N/A	0x04	No se apegan al estándar RISC-V. Fueron diseñados específicamente para Siwa.
mcauseC (`trapC`)	N/A	0x05	
CSR_IO	N/A	0x07	

No obstante, durante el desarrollo del presente proyecto, se notificó que aún existe un nivel de incertidumbre por cuanto como la segunda versión de Siwa implementará y manejará la información en cada CSR que se presentó en la Tabla 3.1. Como resultado de ello, se optó por descartar el tratamiento de los CSRs en el presente proyecto y la siguiente propuesta de verificación. Ello también repercute en el análisis o estudio de las interrupciones de Siwa, pues se sabe de antemano que a través de los CSRs se informan las causas y fuentes de interrupción, por citar algunas funcionalidades [1].

Con respecto a las instrucciones que Siwa es capaz de ejecutar, estas fueron tomadas de [1] y recopiladas en el Tabla 3.2. Cabe recalcar que la versión de las especificación RISC-V utilizada en la primera versión de Siwa fue lanzada en el 2017, sin embargo la versión más reciente es del año 2019. La importancia de lo anterior radica que, entre otras cosas, las instrucciones de manejo de CSRs fueron retiradas del set RV32I y consideradas como un subset Zicsr. Siendo estrictos, la nueva versión de Siwa implementa los estándares RV32I y Zicsr aunque por simplicidad y compatibilidad con documentación anterior, se utilizará como referencia RV32I para decir que implementa tanto el set principal como el subset de manejo de CSRs.

Tabla 3.2. Instrucciones RV32I que soporta Siwa [1], así como su respectivo *opcode* [2]

Instrucción [mnemónico]	***opcode***	**Instrucción [mnemónico]**	***opcode***
addi	0010011	sra	0110011
slti	0010011	add	0110011
sltiu	0010011	slt	0110011
andi	0010011	sltu	0110011
ori	0010011	and	0110011
xori	0010011	or	0110011
slli	0010011	xor	0110011
srli	0010011	sll	0110011
srai	0010011	srl	0110011
		sub	0110011
lui	0110111	jal	1101111
auipc	0010111	beq	1100011
bne	1100011	blt	1100011
bge	1100011	bltu	1100011
sb	0100011	sh	0100011
sw	0100011	jalr	1100111
lb	0000011	lh	0000011
lw	0000011	lbu	0000011
lhu	0000011	ecall	1110011
ebreak	1110011	bge	1100011

Nótese que en la Tabla 3.2 no se hace referencia a instrucciones de manejo de CSRs, ya que, en concordancia con lo mencionado anteriormente, estas no fueron considerados como parte de la segunda versión de Siwa al momento del desarrollo actual del diseño, por lo que tampoco se toman parte del estudio del DUT ni del posterior esfuerzo de verificación.

Con respecto a la instrucción `fence`, de manera similar, en la versión 2017 era parte del set RV32I aunque ahora para la versión del 2019, se ha incorporado como una extensión llamada Zifencei, quedando por fuera del set RV32I [2]. No obstante, debido a la arquitectura planteada para la nueva versión de Siwa, podría ser de relevancia, dada su funcionalidad para configuraciones multi-núcleo.

Por otro lado, se hereda la funcionalidad de ejecutar programas en modo máquina privilegiado como el único nivel implementado en hardware, lo que presupone que Siwa no será capaz de ejecutar sistema operativo alguno [1]. En notación de RISC-V, es posible concluir que Siwa es una arquitectura *bare-metal*. Esto repercute en la elección del modelo de referencia, cuya presentación en este documento se realizará más adelante.

3.1.2. Controlador del bus del sistema y memoria

La interacción con las interfases de I/O sucede a través de accesos de memoria [1] (MMIO, por sus siglas en inglés), cuyas regiones se definen en el mapa de memoria de la Figura 3.3. Dicho mapa

corresponde a una actualización del presentado en [1], ya que no se considera una región exclusiva de los actuadores de dispositivos médicos que se incluían en el espacio de 4 MB hasta 8 MB inclusive, Además, se resalta que ahora la región de memoria general es direccionable para ambos *cores* y para efectos del presente proyecto se consideró como parte de la región de datos de RISC-V; aún se considera su implementación como SRAM del sistema. Por otro lado, se tienen regiones exclusivas para cada núcleo, cuyo uso se asume como región de texto de cada *core*. Sin embargo, como se ha reiterado anteriormente, la segunda versión del microcontrolador Siwa se encuentra en desarrollo, por lo que esto conlleva a la necesidad de implementar un mapa de memoria parametrizable en el lado del ambiente de verificación y, por consiguiente, adaptarse ágilmente a futuras modificaciones que puedan surgir por parte del equipo diseñador.

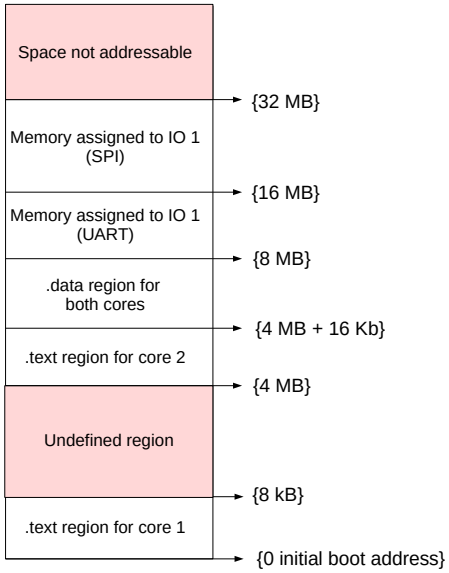

Figura 3.3. Estimación del mapa de memoria de la segunda versión del microcontrolador Siwa, a partir del presentado en la primera revisión de Siwa [1].

De este modo, al módulo MCB se le atribuye una tarea fundamental, pues será el responsable de suministrar y/o almacenar los datos que ambos núcleos pueden manipular a través de instrucciones `load` y/o `store` [1]. Así mismo, dado que las interfases SPI y UART son visibles en cada CPU de Siwa por medio de espacios de memoria delimitados, el MCB será encargado de mapear cada transacción de información que se recibe/transmite a través de ellos, mediante: a) el manejo del bus principal del sistema, b) la codificación/decodicación de la información que se transmite/recibe (considerando el estándar de conformación de paquetes por parte del bus, el cuál se abordará más adelante) y, por último, c) el almacenamiento o suminstro de datos válidos desde/hacia la memoria SRAM principal del sistema.

Por otro lado, luego de accionar el `reset` del sistema, el bloque MCB ejecuta por defecto la rutina de *boot* del sistema [1], la cuál básicamente consiste en el almacenamiento del software ubicado en una memoria externa y conectada a Siwa mediante la interfaz SPI. Dicho proceso involucra un protocolo detallado en la sección "Boot load process"de [1].

3.1.3. Bus del sistema

La interconexión entre el núcleo y las interfases para dispositivos periféricos se realizó mediante un bus con árbitro distribuido, según la topología documentada en [1]. Esto no será la excepción en el contexto de la segunda versión de Siwa, pues al contrario su funcionalidad adquiere mayor relevancia, dada la capacidad y necesidad de comunicación entre ámbos núcleos del SoC, tal como se constató en la Figura 3.1.

Como se mencionó inicialmente, no será objetivo del presente proyecto comprobar el funcionamiento de las interfases SPI y UART propiamente. Sin embargo, aún así es necesario comprender el protocolo de comunicación que sigue el bus del sistema. Para ello, a partir de la construcción del modelo de alto nivel del DUT, se conoce de antemano que tampoco incluyen las pilas FIFO entre la interfaz del bus y el dispositivo. Por consiguiente, las señales involucradas en el protocolo de comunicación con el bus del sistema se resaltan en la Figura 3.4.a).

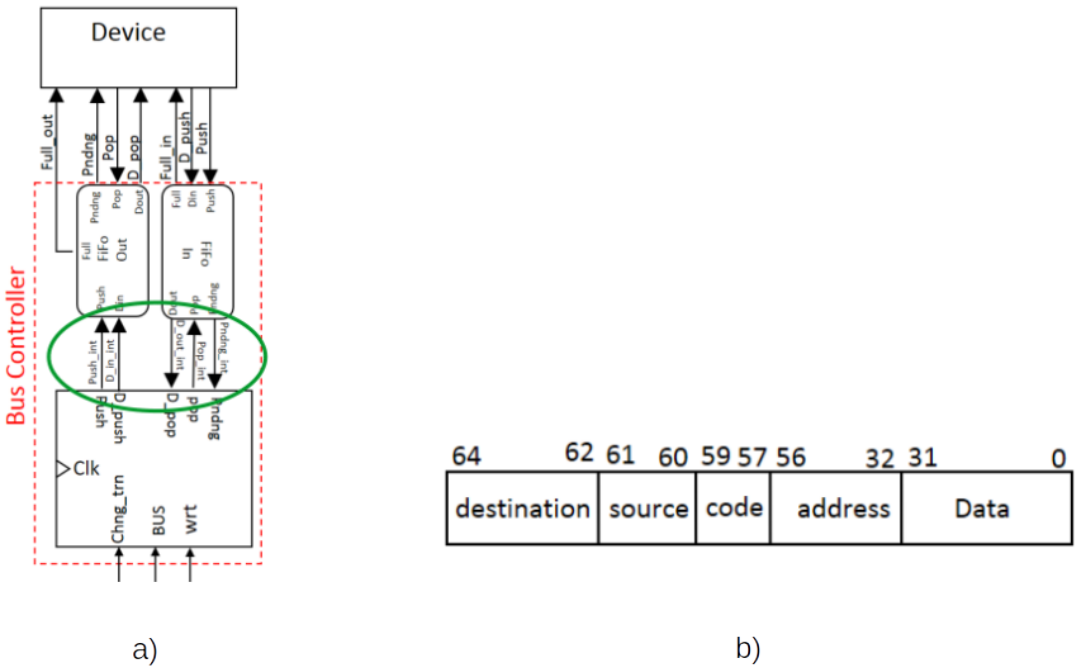

Figura 3.4. a) Resaltado de las señales que se considerarán como parte del protocolo de comunicación con el bus del sistema. b) Campos y bits respectivos que componen un paquete de información hacia/desde el bus del sistema. Tomado de [1].

Así, considerando el párrafo anterior y la composición de un paquete del bus de acuerdo a la Figura 3.4.b), se enlistan las señales involucradas y breves aspectos relevantes para la caracterización de los *drivers* y *monitors* que integrará el ambiente de verificación funcional. Nótese que esto aplica para una sola interfaz con el bus y que Siwa posee 2 para dispositivos I/O.

- Grupo de señales para el comportamiento desde el controlador del bus hacia el ambiente de verificación:

  1. `push_int`: señal de control activa en alto. Luego de un flanco positivo, se interpreta como una transacción proveniente del bus (por medio de `d_in_int`) que debe ser almacenada y

posteriormente decodificada para su manejo en el ambiente. Posee una duración de ciclo de reloj por cada dato que debe ser almacenado [1].

2. `d_in_int`: señal de información. Posee un ancho de 65 bits. Transporta la transacción proveniente del bus [1]. Su funcionamiento es en conjunto con la señal anterior `push_int`.

- Grupo de señales para el comportamiento desde el ambiente de verificación hacia el controlador del bus:

1. `pndng_int`: señal de control activa en alto. Se mantendrá asertada siempre y cuando exista un paquete pendiente por enviar a través del bus, de lo contrario permanecerá en bajo [1].
2. `pop_int`: señal de control activa en alto. Similar en comportamiento a la señal `push_int`, en este caso permite avisar al controlador del bus que debe almacenar el paquete que se colocará en la señal `d_out_int` [1].
3. `d_out_int`: señal de información. Posee un ancho de 65 bits y transporta la transacción que se dirige hacia el controlador del bus [1].

Por otro lado, los metadatos de un paquete de información válido para el bus del sistema (*destination, source* y *code*) siguen la codificación mostrada en el Tabla 3.3. Con respecto a la dirección o *address*, será considerada como una dirección de memoria válida.

Tabla 3.3. Codificación de los diferentes metadatos de un paquete válido del bus del sistema. Tomado de [1] y comentarios acerca de las modificaciones en la nueva versión de Siwa.

destination/source		***code***	
Dispositivo	**Valor**	**Acción**	**Valor**
MCB	0x4	Lectura 1 byte	0x1
SPI	0x2	Lectura 2 byte	0x2
UART	0x1	Lectura 4 byte	0x0
Núcleo 1	0x0	Escritura 1 byte	0x5
Núcleo 2	0x3	Escritura 2 byte	0x6
		Escritura 4 byte	0x4
		Comienzo *boot*	0x7
		Fin *boot*	0x3

Finalmente, cabe recalcar que para el presente proyecto, es de importancia conocer la codificación y manejo de paquetes que ingresan/egresan del bus del sistema a través de las interfases de I/O. Por consiguiente, el ambiente de verificación será transparante al manejo interno que se le dé al paquete en el DUT.

3.2. Plan de pruebas

A continuación, en la siguiente lista se describen las pruebas que componen el *testplan*. Es importante destacar que el objetivo es desarrollar un ambiente de verificación funcional capaz de llevar a cabo la tarea a un nivel de *full-chip*, por lo que no es propósito del proyecto realizar pruebas específicas u orientadas a módulos de hardware en específico de la microarquitectura, pues se supone que estos ya fueron revisados exhaustivamente antes de su integración. Se persigue la conformación de pruebas capaces de verificar comportamientos en el funcionamiento a alto nivel o de integración entre dos o más módulos de hardware de Siwa.

1. **Prueba: Transferencia de instrucciones válidas para CPU 1**

   Objetivo: Comprobar la funcionalidad del bus del sistema para transferir información y verificar la operación básica del núcleo 1.

   Descripción: En esta prueba se pretende ejercitar la funcionalidad de transferencia y carga de un programa como si este proveniera de la memoria flash externa. Para ello se inyectarán instrucciones aleatorizadas controladamente, tal como se describe en la sección 3.2.1, en el espacio de memoria del núcleo 1 del sistema, con lo que se busca que el *core* 1 sea el responsable de la ejecución del programa.

   Finalmente, para el caso de la interfaz cuya futura conexión será UART, inicialmente permanecerá en estado de espera, pues al finalizar la prueba sí será utilizada para obtener los resultados finales que fueron alojados en la memoria SRAM del sistema.

   Criterio de aprobación: Si el contenido de memoria del DUT y el simulado por el modelo de referencia coinciden completamente en cada espacio en memoria al finalizar la fase de corrida del *test*.

   Nombre clave de la secuencia UVM: `cpu_one_val_inst_seq.sv`

   Nombre clave de la prueba: `cpu_one_val_inst_test`

2. **Prueba: Transferencia de instrucciones válidas para CPU 2**

   Objetivo: Comprobar la funcionalidad del bus del sistema para transferir información y verificar la operación básica del núcleo 2.

   Descripción: Corresponde a una extensión de la prueba 1, dado que en este caso las instrucciones aleatorizadas controladamente serán colocadas en el espacio de memoria asignado para el núcleo 2. De este modo, la ejecución de las instrucciones y la prueba como tál serán realizadas por el *core* 2 del sistema.

   Criterio de aprobación: Si el contenido de memoria del DUT y el simulado por el modelo de referencia coinciden completamente en cada espacio en memoria al finalizar la fase de corrida del *test*.

Nombre clave de la secuencia UVM: `cpu_two_val_inst_seq.sv`

Nombre clave de la prueba: `cpu_two_val_inst_test`

3. **Prueba: Transferencia de instrucciones válidas para ambos CPUs**

   Objetivo: Comprobar la funcionalidad del bus del sistema para transferir información y verificar la operación básica de ambos núcleos.

   Descripción: Corresponde a una extensión de la prueba 1 y 2, ya que ahora las instrucciones serán alojadas en el espacio de memoria para cada núcleo de manera simúltanea. De este modo, el fin de la prueba será que ambos CPUs se encuentren activos con la ejecución de programas independientes. La salvedad que se realiza en la independencia de los programas radica en que un *core* no necesitará de los resultados generados por el otro *core*, de manera general.

   Criterio de aprobación: Si el contenido de memoria del DUT y el simulado por el modelo de referencia coinciden completamente en cada espacio en memoria al finalizar la fase de corrida del *test*.

   Nombre clave de la secuencia UVM: `cpu_both_val_inst_seq.sv`

   Nombre clave de la prueba: `cpu_both_val_inst_test`

4. **Prueba: Ejecución de bucles con diferentes condiciones de salida**

   Objetivo: Corroborar la capacidad de realizar saltos condicionales e incondicionales.

   Descripción: En esta prueba se pretende conformar 6 bucles o ciclos distintos, donde cada una de las condiciones de salida serán determinadas por las siguientes instrucciones RV32I: `beq`, `bne`, `blt`, `bge`, `bltu` y `bgeu`. Para ello, uno de los operandos (`rs1`) de tales instrucciones tipo B será inicializado de tal manera que, al menos en una ocasión, se tome el salto. Por otro lado, el operando restante (`rs2`) será controlado por un contador (instrucción `add`) cuyo valor de inicio será 0 antes de ejecutar cada ciclo. De este modo, se cubre la utilización de instrucciones para saltos condicionales.

Tabla 3.4. Valor en el registro equivalente al operando `rs1` durante la ejecución de cada bucle de prueba.

Instrucción	Valor del operando 1	Comentarios
`beq`	0	El ciclo se ejecutará una vez al menos, pues el operando en `rs2` siempre se inicializa en 0.
`bne`	2	Se realizarán 2 iteraciones, ya que el operando en `rs2` incrementa en una unidad por cada ciclo.
`blt`	10	Ocurrirán 10 iteraciones, ya que `rs2` inicia en 0 y se incrementa en una unidad por repeteción.
`bge`	0	Dado que la especificación indica mayor o igual, al menos se ejecutará una repetición.
`bltu`	10	Similar al caso con `blt`
`bgeu`	0	Similar al caso con `bge`

En caso de no cumplirse la condición de cada instrucción tipo B, se ejecutará un salto incondicional (tipo J) para redirigir el contador de programa a la dirección base de memoria donde comienza el software del bucle siguiente. Además, se guardará en memoria de datos el último valor del contador en direcciones distintas. Así, hasta ejecutarse 6 bucles diferentes en total.

Al igual que los casos de prueba con instrucciones lógicas y aritméticas aleatorizadas controladamente, se utilizarán instrucciones `store` para mover el contenido de la región de datos hacia la memoria direccionable por UART.

Criterio de aprobación: Si el contenido de memoria del DUT y el simulado por el modelo de referencia coinciden completamente en cada espacio en memoria al finalizar la fase de corrida del *test*.

Nombre clave de la secuencia UVM: `cond_incond_jumps_seq.sv`

Nombre clave de la prueba: `cond_incond_jumps_test`

3.2.1. Acerca de la aleatorización controlada de instrucciones

3.2.1.1. Funcionalidades aritméticas y lógicas básicas.

De acuerdo a las especificaciones y documentación consultada hasta el momento, así como conversaciones previas con el equipo diseñador, se sabe que la segunda versión del micontrolador Siwa integra el set RV32I.

Con el objetivo de crear secuencias de programa orientados a explorar la funcionalidad aritmética y lógica de cada núcleo, se pretendía inyectar instrucciones válidas enfocadas en generar resultados a partir de uno o dos operandos cuyos valores han sido inicialmente colocados en el banco de registros por medio de instrucciones `lui`. En este punto se evitaba modificar el flujo de ejecución del programa; es decir, las instrucciones del tipo `branch` y `jump` no serán consideradas como parte de la presente estrategia.

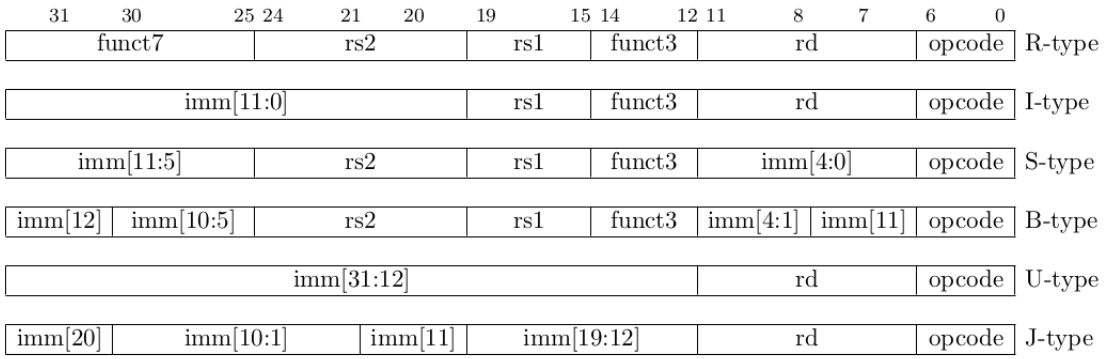

Figura 3.5. Codificación de instruccions RISC-V pertenecientes al set RV32I [2]

De este modo, la estrategia de aleatorización consistía en tomar aleatoriamente 32 instrucciones de las mostradas en el Tabla 3.5. Indistintamente del formato de codificación que sigue cada instrucción, los registros operandos `rs1` y `rs2`, así como el registro destino `rd` siguen una aleatorización tal que se busca que al menos cada registro sea utilizado en una ocasión.

Finalmente, la composición de un programa con aleatorización controlada de instrucciones RISC-V es tal como se observa en la Figura 3.6. Las instrucciones `lui` iniciales permiten escribir constantes de 32 bits en cada uno de los registros y el último grupo de instrucciones `store` permiten transportar los resultados finales en cada registro hacia ubicaciones de memoria de dispositivos de entrada o salida (por ejemplo, la interfaz asignada para UART).

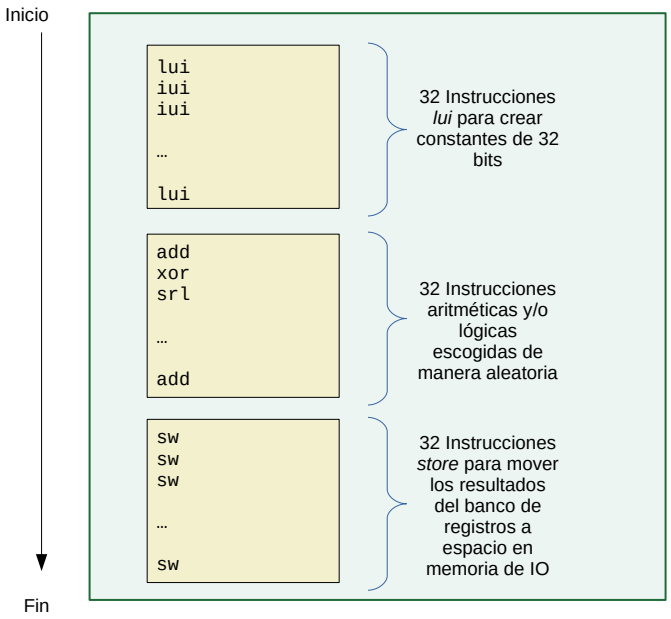

Figura 3.6. Ejemplificación de la composición de un programa con aleatorización controlada de instrucciones del set RV32I.

Nótese que en la Figura 3.6 no se hace referencia a instrucciones iniciales para configurar la dirección base en memoria donde se ubica la rutina de software encarga de atender las interrupciones (ISR). Aunque no es fin de esta estrategia generar condiciones o escenarios de interrupciones, inicialmente se pensó que, como medida precautoria en caso de que exista algún error en la microarquitectura de Siwa que se traduzca en una interrupción no esperada (por ejemplo, una instrucción válida que reconoce como inválida), se podría configurar la dirección base del ISR y así poder detectar y reportar la interrupción, por ejemplo. Sin embargo, dada la problemática alrededor de modelar el comportamiento de instrucciones CSRs, no fue posible concretar dicha idea, pues idealmente es necesario ejecutar `csrrw` para configurar el registro `mtvec` con la etiqueta de inicio del ISR en la región de texto de la memoria direccionable por ambos *cores*.

Tabla 3.5. Instrucciones aritméticas y lógicas a considerar en la aleatorización controlada de instrucciones. Tomado del set RV32I [2]

Instrucción [mnemónico]	**Formato codificación**	**Instrucción [mnemónico]**	**Formato codificación**
addi	I	sra	R
slti	I	add	R
sltiu	I	slt	R
andi	I	sltu	R
ori	I	and	R
xori	I	or	R
slli	I	xor	R
srli	I	sll	R
srai	I	srl	R
		sub	R
lui	U		
auipc	U		

3.2.2. Con respecto al modelo de referencia

Con el fin de automatizar la revisión de cada prueba *test* una vez finalizada y que el ambiente de verificación se encargue de dictaminar si dicho resultado fue satisfactorio o incorrecto, es necesario incluir un modelo de referencia [8]. Tal modelo de referencia tiene la tarea exclusiva de implementar ciertos criterios de aprobación basados en el estándar RISC-V para determinar si la información obtenida al finalizar la prueba se apega o aleja del comportamiento esperado.

Ahora bien, cabe recalcar que el modelo de referencia que se menciona en esta sección y el modelo de alto nivel que se ha nombrado anteriormente son completamente diferentes. Aunque en otro momento, el modelo elaborado por el estudiante Ángelo Hernández Cordero pueda utilizarse como modelo de referencia, para la presente ejecución del proyecto se optó por utilizar un software externo para el modelo de referencia y establecer una clara diferencia entre ellos. Así, a) no se incurrió en un trabajo doble al volver a elaborar un modelo de alto nivel de Siwa y b) no fue necesario ejecutar tareas aisladas propias de la validación del modelo de referencia, pues el software externo es respaldado por la organización de RISC-V como parte del software disponible para el ecosistema de la arquitectura [15], por lo que se supone que se adhiere completamente al estándar RISC-V, al igual que Siwa.

Dado la gran cantidad de opciones dentro del portafolio de software que se enlista en la página de RISC-V [15], fue necesario establecer una serie de criterios que permitieran discerninr cual opción convenía más para el presente proyecto, los cuales se presentan a continuación. Por su facilidad de uso e implementación y cantidad abundante de documentación, inicialmente se elige RARS y Spike como opciones de software. En la Tabla 3.6 se enlistan ambas opciones y su adherencia o no a cada criterio planteado.

1. Sea capaz de recibir un programa en código ensablador para poder realizar la simulación a partir de el.

2. Al finalizar la simulación, sea capaz de imprimir o hacer *dump* de la región de datos del mapa de memoria, con el fin de comparar la región simulada con la región provista desde el DUT.
3. Tenga la posibilidad de reconfigurar facilmente el mapa de memoria, ya que Siwa se encuentra en desarrollo y, por consiguiente, este puede cambiar con facilidad.
4. Posea la funcionalidad de emular desde uno hasta dos o más núcleos RISC-V.
5. Es necesario que trabaje directamente en *bare-metal*, pues Siwa opera con código privilegiado; es decir, sin la necesidad de un sistema operativo o *kernel*

Tabla 3.6. Elección del software externo para modelo de referencia.

Software	**Criterio 1**	**Criterio 2**	**Criterio 3**	**Criterio 4**	**Criterio 5**
RARS	✓	✓	✓	✕	✓
Spike	✓	✓	-	✓	✕

De este modo, a partir de la Tabla 3.6, el modelo de referencia elegido fue el software RARS. Una de las decisiones de peso para elegir RARS sobre Spike fue relacionado con que Spike solo funciona utilizando un *kernel* de por medio, lo cual dista en gran medida de Siwa. Por otro lado, una de las fuertes desventajas de RARS es que solo trabaja con un núcleo. Sin embargo, si se parte de la premisa de que se buscaba un simulador a nivel de instrucciones RISC-V y no uno propiamente de la microarquitectura de Siwa, esto se soluciona en gran medida, ya que lo que se busca es comparar el resultado de ejecutar instrucciones y no necesariamente replicar la forma en como ello sucedió. De este modo, el DUT y el modelo de referencia serían transparentes.

Así, una vez que se definió la elección del modelo de referencia, fue posible establecer los criterios de aprobación de cada prueba presentada en la sección 3.2, basándose en las capacidades y funcionalidades que ofrece RARS. En la siguiente sección se hablará sobre como se logró la implementación con el ambiente de verificación funcional, pues RARS está escrito en Java y debe funcionar en sincronía con pruebas y un ambiente de verificación codificado en lenguaje SystemVerilog. Finalmente, para ser estrictos, se utilizaron dos instancias de RARS, una exclusiva para cada núcleo, tal como se mostrará en la siguiente sección.

3.3. Ambiente de verificación funcional

3.3.1. Arquitectura del ambiente

En la Figura 3.7 se muestra la arquitectura propuesta del ambiente de verificación funcional a desarrollar. Dicho diagrama corresponde a las necesidades detectadas luego de la formulación del plan de pruebas (ver sección 3.2). Como aspectos importantes, cabe recalcar que, en cuanto al *scoreboard*, intencionalmente este fue dibujando parcialmente por fuera de los bloques *environment*, por ejemplo, lo cual no es usual en UVM. Sin embargo, lo anterior obedece a una forma de representar explicitamente que el modelo de referencia fue un software externo escrito en lenguaje Java, el cual interactua mediante funciones DPI y accesos `System` de lenguaje C, con el resto de bibliotecas UVM y la codificación completa del ambiente en lenguaje SystemVerilog. A continuación, en las siguientes subsecciones se brindarán aspectos importantes de cada uno de los componentes principales que incluye el ambiente de verificación.

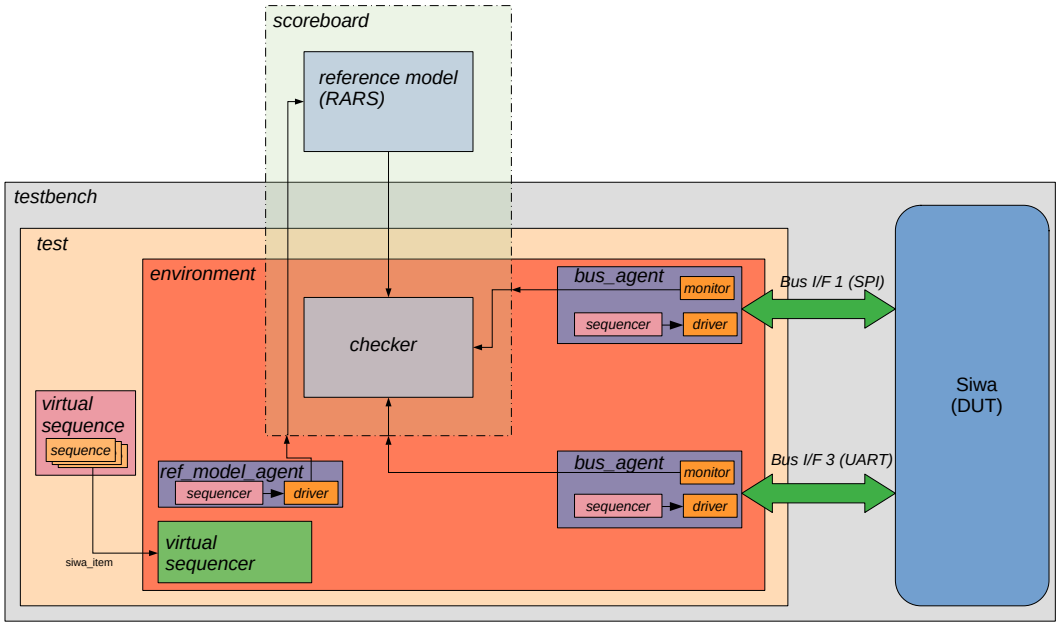

Figura 3.7. Arquitectura del ambiente de verificación funcional a desarrollar.

3.3.1.1. Interfaces

Con el objetivo de encapsular las señales de información o pines que existen entre la interfaces del controlador del bus (ubicadas en el DUT) y los agentes para el envío de los patrones de prueba, así como el observación de la respuesta parte del diseño, se creó la siguiente interfaz mostrada en el Código 3.1. Nótese que los nombres de cada señal corresponde directamente con las que se obtuvieron durante el estudio del DUT, como se detalló en la Figura 3.4.a. Así mismo, el ancho del paquete del bus que se envía o recibe a través de `d_out_int` o `d_in_int`, según corresponda, se ha dejado parametrizado mediante el macro `PKT_SIZE`. Para efectos del presente proyecto, dicho macro se definió con un valor de 64 bits, en concordancia con el ancho máximo mostrado en la Figura 3.4.b.

```
interface bus_if;
    logic push_int;
    logic [`PKT_SIZE-1:0] d_in_int;
    logic [`PKT_SIZE-1:0] d_out_int;
    logic pop_int;
    logic pndng_int;

endinterface
```

Código 3.1. Interfaz de señales entre un controlador del bus y las pilas *FIFO in* y *FIFO out*.

3.3.1.2. *Virtual sequencer*

Tal como se observa en la Figura 3.7, el ambiente de verificación funcional posee en total dos agentes que interactuan con diferentes interfases y, además, cada agente posee un secuenciador propio encargado de adiminstrar los patrones de prueba al *driver* asociado. Tales agentes deberán ser accionados, cada uno, desde una secuencia (o *sequence*, según se muestra en la Figura 3.7). De este modo, se creó un secuenciador virtual con el objetivo de que exista un espacio en común donde se administren los punteros de cada secuenciador y, por consiguiente, esto fuera transparente a la codificación de cada prueba.

```
    // Handle of sequencer for SPI equivalent interface
    uvm_sequencer #(siwa_item) b_sqcr_spi;

    // Handle of sequencer for I2C equivalent interface
    uvm_sequencer #(siwa_item) b_sqcr_i2c;

    // Handle of sequencer for UART equivalent interface
    uvm_sequencer #(siwa_item) b_sqcr_uart;

    // Handle of sequencer for clk & rst operations
    uvm_sequencer #(clk_rst_item) cr_sqcr;

    // Handle of sequencer for send the assembly program
    // to reference model
    uvm_sequencer #(siwa_item) b_sqcr_ref;
```

Código 3.2. Extracto de la estructura del *virtual sequencer* implementado.

3.3.1.3. *Drivers*

3.3.1.3.1. Interfaz del bus

Para la elaboración de cada *driver* o manejador de la interfaz del bus, se optó por construir una clase base `bus_driver` y luego se crearon *drivers* específicos que extendía tal clase base, con el fin exlcusivo de adquirir el puntero de la interfaz correspondiente a través de la base de datos de configuración de UVM. Por ejemplo, en el Código 3.3 se muestra la implementación del manejador para la interfaz SPI; nótese que extiende a la clase base llamada `bus_driver`.

```
class spi_driver extends bus_driver;
  `uvm_component_utils(spi_driver)
    function new (string name = "spi_driver", uvm_component parent = null);
        super.new(name, parent);
    endfunction : new
    // Build phase
    //  Acquired the pointer of interface from config data base
    virtual function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual bus_if)::get(this, "", "spi_if", vif))
   begin
            `uvm_fatal(get_name(), "Failed to get the pointer of interface with
    System Bus on DUT")
        end
    endfunction : build_phase
endclass
```

Código 3.3. Implementación del *driver* para el proceso de *bootstrap* a través de la interfaz equivalente de SPI.

Los métodos que implementa la clase base son:

- Un *task* `hndl_pop` para implementar el protocolo de `pop` y `pndng` como resultado de la emulación de la pila *FIFO in* mostrada en la Figura 3.4.a.
- Un *task* `build_bus_pkt` para la conformación de paquetes, siguiendo el estándar mostrado en la Figura 3.4.b. En este caso, el punto clave corresponde al traspaso de la dirección de memoria correcta y válida. Los campos *destination*, *source* y *code* se obtienen directamente del `sequence_item`, por lo que fueron aleatorizados o definidos previamente, según el contexto de la prueba. Se hace la salvedad de que *source* concuerda con el *sequencer* utilizado para comunicar el patrón de prueba.
- Una función de soporte `mem_addr_cntr` para la elaboración de una dirección de memoria válida, tomando en cuenta el núcleo al cual se pretende escribir y la dirección de memoria utilizada anteriormente.

Por último, el *driver* adquiere mayor relevancia para la comunicación por la interfaz equivalente de SPI, pues será el responsable de suminstrar los paquetes codificados durante el proceso de *bootstrap*. Por otro lado, el uso del *driver* para la interfaz equivalente de UART no será tan extensivo, ya que no se plantearon escenarios de prueba donde se inicie la comunicación por este medio, solo para la recepción de información.

3.3.1.3.2. Compilación del programa en lenguaje ensamblador

Por otro lado, con respecto al *driver* encargado de la escritura del código ensamblador, se trató de una estructura muy diferente a la expuesta anteriormente. En este caso, el objetivo era distinto, pues se recibían las transacciones con los campos de: nmemónico, operandos e inmediatos en variables distintas, por lo que era tarea del *driver* reunirlos o acomodarlos según correspondía al tipo de

instrucción RISC-V y, finalmente, escribirlos en un archivo con extensión `.asm` mediante la función `$fdisplay`, propia del *task* `build_asm` mostrado en el Código 3.4.

Aunque no se ha mencionado anteriormente, el fin con esta tarea adjudicada al *driver* del model de referencia, corresponde a una simplificación del trabajo y remoción del factor humano como posible fuente de error. Es decir, a partir del hecho de que se utilizaba RARS como simulador dentro del *scoreboard*, también se determinó en la documentación de dicho software [16], que era posible obtener de RARS el archivo binario a partir de un programa en lenguaje ensamblador.

De este modo, el funcionamiento se resume en las siguientes viñetas. Como referencia, en el Código 3.4 se muestra un extracto del *driver* para el archivo `.asm` que requiere RARS. Con respecto a las funciones DPI, estas se abordarán con más detalle en la siguiente sección sobre el *scoreboard* y en el Código 3.8.

1. El ambiente de verificación crea el programa en un archivo con extensión .asm, mediante el método `build_asm`.

2. Luego, el *driver* del modelo de referencia ejecuta RARS mediante la función DPI `rars_assembly_core_1/2` (según el núcleo que corresponda), donde el software externo recibe el archivo .asm y genera un archivo .s con cada instrucción codificada en su equivalente binario, considerando las etiquetas del compilador, así como las direcciones de los registros operandos y las direcciones de memoria válidas y.

3. Finalmente, el *driver* acciona el evento `assembly_op_ev` como bandera para que el ambiente de verificación pueda proseguir con la lectura del archivo .s y transmitirlo línea por línea a través del *sequencer* de SPI.

```
...
import "DPI-C" function void rars_assembly_core_1();
import "DPI-C" function void rars_assembly_core_2();
    ...
        // 1. Get next item from the sequencer
        seq_item_port.get_next_item(sent_item);
        `uvm_info(get_name(),"Waiting for write instruction on asm file",
   UVM_HIGH)
        // 2. Write the correspondent assembly instruction
        core_nmbr = sent_item.get_core_nmbr();
        if (core_nmbr == 1) begin
             build_asm(sent_item, fd_c1);
             cntr_c1++;
        end
        else if (core_nmbr == 2) begin
             build_asm(sent_item, fd_c2);
             cntr_c2++;
        end
        else `uvm_fatal(get_name(), "Unsoported core number")
        `uvm_info(get_name(), "Package already wrote to the asm file", UVM_NONE
   )
        // 3. Tell the sequence that driver has finished current item
        seq_item_port.item_done();
```

```
22      end : feed_loop
23      $fclose(fd_c1);
24      $fclose(fd_c2);
25      if (cntr_c1 != 0) begin
26          rars_assembly_core_1();
27          `uvm_info(get_name(), "Finish with assembly operation for core 1",
    UVM_NONE)
28      end
29      if (cntr_c2 != 0) begin
30          rars_assembly_core_2();
31          `uvm_info(get_name(), "Finish with assembly operation for core 2",
    UVM_NONE)
32      end
33      -> assembly_op_ev;
34      `uvm_info(get_name(), "The assembly operation is now over", UVM_NONE)
35      endtask : run_phase
36
37      task build_asm(siwa_item t, int pointer);
38          case (t.inst_type)
39                  type_I: begin
40                      if (t.nmemonic == lb ||
41                          t.nmemonic == lh ||
42                          t.nmemonic == lw ||
43                          t.nmemonic == lbu ||
44                          t.nmemonic == lhu) begin
45                              $fdisplay(pointer,
46                              $sformatf("%s %s, 0x%h(%s)", t.nmemonic.name,
47                                                           t.rd.name,
48                                                           t.imm,
49                                                           t.rs1.name
50                                                           )
51                              );
52      ...
53
```

Código 3.4. Extracto de la clase `ref_model_driver` donde se observa gran parte de la codificación de la fase de corrida y se finaliza con la sección inicial del método `build_asm` correspondiente a la escritura de la línea de código ensamblador en función del tipo de instrucción y nmemónico RISC-V. Posteriormente, dicho código ensamblador será compilado por RARS.

3.3.1.4. *Monitors*

Para la implementación de los monitores de cada agente, se opotó por una estrategia similar, al construir una clase base llamada `bus_monitor` con los métodos inherentes al control de los pines de la interfaz y luego extender dicha clase para conformar los *monitors* en específico que se necesitan. A manera de ejemplo, en el Código 3.5 se muestra la implementación del monitor para UART como clase hijo de `bus_monitor`.

```
1 class uart_monitor extends bus_monitor;
```

```
  `uvm_component_utils(uart_monitor)
    function new (string name = "uart_monitor", uvm_component parent = null);
        super.new(name, parent);
    endfunction : new
    // Build phase
    virtual function void build_phase (uvm_phase phase);
        super.build_phase(phase);
        // Create the instance of the analysis port
        mon_analysis_port = new("uart_analysis_port", this);
        if (!uvm_config_db#(virtual bus_if)::get(this, "", "uart_if", vif))
   begin
            `uvm_fatal(get_name(), "Failed to get the pointer of interface with
    System Bus on DUT")
        end
    endfunction : build_phase
endclass
```

Código 3.5. Implementación del *monitor* para la interfaz equivalente de UART.

En este caso, debido a la sencillez del protocolo de comunicación en los pines de salida entre la pila *FIFO out* y la interfaz del bus como se detalló en la Figura 3.4.a, no fue necesario codificar métodos por aparte. Al contrario, se obtiene una `run_phase` simple, donde luego de cada flanco positivo en `push_int`: 1) se capturan el paquete proveniente del bus y, luego, 2) se encapsula dentro de una instancia del `sequence_item` creado para el trasiego de información y, por último, 3) se envía a través de un puerto de análisis cuyo destino es el *scoreboard*. Lo anterior se observa en el Código 3.6.

```
    // Run phase
    virtual task run_phase (uvm_phase phase);
        siwa_item receive_item = siwa_item::type_id::create("receive_item",
   this);
        forever begin : receive_loop
            @(posedge vif.push_int);
            receive_item.set_pkg(vif.d_in_int);
            `uvm_info(get_name(), $sformatf("Package received on bus interface
   with ID = %d", bus_if_id), UVM_NONE)
            mon_analysis_port.write(receive_item);
            @(negedge vif.push_int);
        end : receive_loop
    endtask : run_phase
```

Código 3.6. Implementación del protocolo de `push` proveniente de la interfaz del bus hacia el ambiente de verificación, como parte de la fase de corrida de `bus_monitor`.

Finalmente, con respecto al puerto de análisis, su puntero fue declarado como parte de la clase base, por lo que tanto el monitor de SPI como de UART, ambos poseen un puerto de análisis. Sin embargo, a partir de lo expuesto en [1], se sabe de antemano que la interfaz de SPI fue creada exclusivamente para dar soporte al proceso de *bootstrap*, por lo que no sería correcto recibir información (desde el DUT/Siwa hacia el ambiente de verificación/mundo externo) a través de tal interfaz. Por consiguiente, en la fase de conexión de la clase ambiente, solo se contempla la conexión del puerto de análisis del monitor de UART con el *scoreboard*, tal como se detalla en el Código 3.7.

```
    // Connect phase
    virtual function void connect_phase (uvm_phase phase);
        super.connect_phase(phase);
        ...
        uart_eq_agnt.uart_mnt.mon_analysis_port.connect(m_scbd.ap_imp);
        // There is no connection between SPI monitor and SB
    endfunction : connect_phase
```

Código 3.7. Extracto del `connect_phase` de la clase ambiente.

3.3.1.5. *Scoreboard*

El *scoreboard* corresponde a uno de los bloques de mayor trascendencia, pues fue el responsable de automatizar la determinación sobre el resultado de cada prueba, ya que finalmente concluía si los datos obtenidos eran esperados y, por lo tanto, una prueba existosa. Al contrario, si la información generada por el DUT distaba en alguno o varios puntos con lo obtenido del modelo de referencia, el resultado era una prueba incorrecta o no exitosa. Gran parte de esta sección será dedicada a la implementación y uso que se le dió al software externo que se eligió.

Anteriormente, en la sección 3.2.2, se detalló sobre la elección del software RARS como modelo de referencia del ambiente de verificación. En esta ocasión, se abordará el cómo se llevó a cabo la implementación e integración con el ambiente de verificación funcional. Esto último presentó uno de los mayores retos, pues desde un inicio se trataba de conjugar tres lenguajes de programación en un mismo escenario:

- Por un lado, de manera inherente, la codificación de todo el proyecto con SystemVerilog y luego VCS (herramienta de verificación funcional utilizada), al compilar, traduce todo a su equivalente en lenguaje C.
- En otro lado, se tenía RARS, el cual estaba escrito en Java.

De este modo, a partir de la compilación en C por parte de VCS y que dá como resultado, entre otras cosas, un archivo de *headers* `vc_hdrs.h`, fue posible aprovechar la funcionalidad de DPI para implementar funciones sencillas en C [17]. De este modo, gracias al uso de tales funciones en C y la función de libreria propia del lenguaje llamada `system()`, se dotó al ambiente de verificación la capacidad de controlar la ejecución de RARS en total sincronía, mediante la sintáxis de comandos para uso en línea de comando [16]. El Código 3.8 corresponde a la librería de funciones creadas para el presente proyecto, considerando lo recien expuesto.

```
#include<stdio.h>
#include<stdlib.h>
#include "../../vc_hdrs.h"

void rars_assembly_core_1() {
    printf("\nC program. Calling RARS for obtain .s assembly file core 1 \n\n")
    ;
    system("java -jar rars/rars_two_cores_custom_v3.jar dump .text BinaryText
    inst_1.s mc CoreOne test_1.asm a");
}
```

```
9
10 void rars_assembly_core_2() {
11     printf("\nC program. Calling RARS for obtain .s assembly file core 2\n\n");
12     system("java -jar rars/rars_two_cores_custom_v3.jar dump .text BinaryText
    inst_2.s mc CoreTwo test_2.asm a");
13 }
14
15 void rars_sim_core_1() {
16     printf("\nC program. Calling RARS for simulate core 1 seq \n\n");
17     system("java -jar rars/rars_two_cores_custom_v3.jar  mc CoreOne test_1.asm
    nc 0x800000-0x80007c > data_1.dat");
18     system("sed -i -e 1,2d data_1.dat ");
19     printf("\nC program. Done with simulate & process core 1 seq \n\n");
20 }
21
22 void rars_sim_core_2() {
23     printf("\nC program. Calling RARS for simulate core 2 seq \n\n");
24     system("java -jar rars/rars_two_cores_custom_v3.jar  mc CoreOne test_2.asm
    nc 0x800080-0x8000fc > data_2.dat");
25    system("sed -i -e 1,2d data_2.dat ");
26    printf("\nC program. Done with simulate & process core 2 seq \n\n");
27 }
```

Código 3.8. Biblioteca de funciones en C que permitieron la integración del software RARS (escrito en Java) con el ambiente de verificación (codificado en SystemVerilog), ya que con ello se habilitó la ejecución de RARS de manera sincronizada con las fases de UVM y el flujo en general de compilación y *bootstrap* en Siwa.

Ahora bien, con respecto al mapa de memoria en cada simulación de RARS, fue necesario establecer un método que permitiera concordancia entre el mapa de memoria del a) ambiente de verificación, b) ensamblador/simulador RARS y c) el dispositivo bajo prueba. Con respecto al punto a), este se abordará en la siguiente sección. En el caso del software RARS, inicialmente se evaluaron las opciones por defecto que se muestran en la Figura 3.8, sin embargo ninguna de las tres opciones se apegaba al mapa de memoria adoptado, tal como se detalló en la Figura 3.3. Por tal motivo, se optó por crear una versión personalizada de RARS, cuya una diferencia radica en la ubicación de la región de texto para ambos *cores*. Los archivos fuentes se pueden ubicar en la carpeta `rars` del repositorio del presente proyecto. Cabe recalcar que las modificaciones en el código fuente fueron posibles debido a la licencia MIT que le otorgaron a RARS [16].

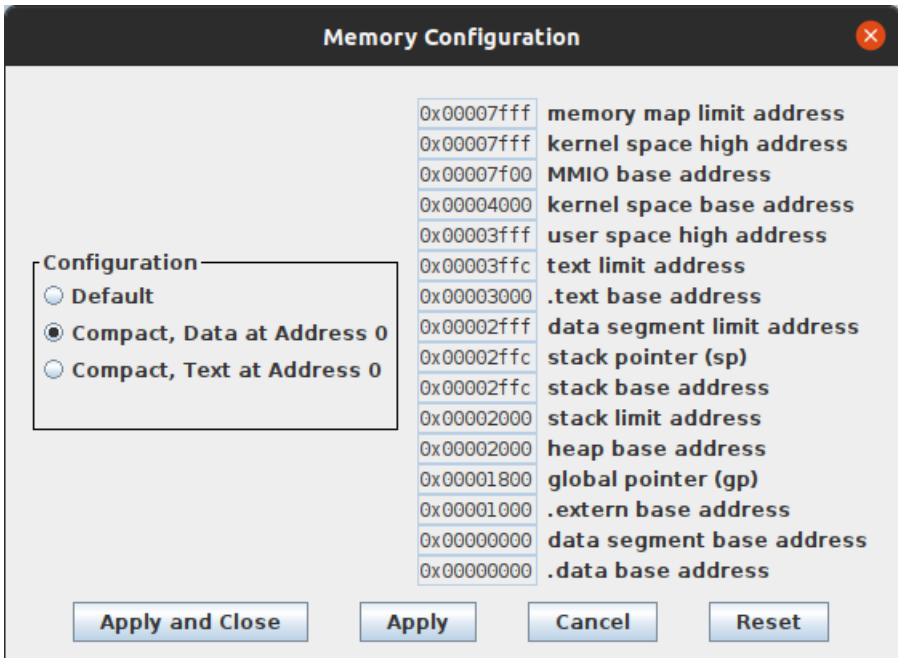

Figura 3.8. Captura de pantalla con las opciones de configuración del mapa de memoria que presenta el software RARS por defecto; es decir, antes de haber compilado RARS con el mapa de memoria personalizado, contemplando los espacios o rangos del mapa de memoria de Siwa mostrados en la Figura 3.3.

Para la construcción de mapas de memoria personalizables, se siguió la sugerencia mostrada en el *issue* número 22 del repositorio del proyecto RARS [16]: al modificar el archivo `rars/rars/riscv/hardware/MemoryConfigurations.java`, es posible definir las direcciones de inicio, entre otros parámetros, de cada una de las regiones de un mapa de memoria bajo el estándar RISC-V. En el Código 3.9 se muestra la sección relevante a las modificaciones realizadas sobre el código fuente del software RARS. Nótese que las direcciones `.text Base Address` y `text limit address` concuerdan con la región de texto de cada núcleo mostrado en la Figura 3.3. Finalmente, como prueba de concepto, una vez compilado el ejecutable .jar de RARS, al abrir la interfaz gráfica y dirigirse a la opción `Memory configuration..`, se visualiza tal como se muestra en la Figura 3.9.

```
1    // Memory map for siwa core 1 (exclusive text region)
2    private static int[] siwaCoreOne= {
3            0x00000000, // .text Base Address
4            0x00404004, // Data Segment base address
5            0x10000000, // .extern Base Address
6            0x10008000, // Global Pointer $gp)
7            0x10010000, // .data base Address
8            0x10040000, // heap base address
9            0x7fffeffc, // stack pointer $sp (from SPIM not MIPS)
10           0x7ffffffc, // stack base address
11           0x7fffffff, // highest address in user space
12           0x80000000, // lowest address in kernel space
13           0xffff0000, // MMIO base address
14           0xffffffff, // highest address in kernel (and memory)
```

```
15              0x02000000, // data segment limit address
16              0x00002000, // text limit address
17              0x10040000, // stack limit address
18              0xffffffff  // memory map limit address
19      };
20
21      // Memory map for siwa core 2 (exclusive text region)
22      private static int[] siwaCoreTwo= {
23              0x00400000, // .text Base Address
24              0x00404004, // Data Segment base address
25              0x10000000, // .extern Base Address
26              0x10008000, // Global Pointer $gp)
27              0x10010000, // .data base Address
28              0x10040000, // heap base address
29              0x7fffeffc, // stack pointer $sp (from SPIM not MIPS)
30              0x7ffffffc, // stack base address
31              0x7fffffff, // highest address in user space
32              0x80000000, // lowest address in kernel space
33              0xffff0000, // MMIO base address
34              0xffffffff, // highest address in kernel (and memory)
35              0x02000000, // data segment limit address
36              0x00404000, // text limit address
37              0x10040000, // stack limit address
38              0xffffffff  // memory map limit address
39      };
```

Código 3.9. Extracto del archivo `MemoryConfigurations.java` modificado para incluir las regiones de texto exclusivas para cada núcleo de la segunda versión de Siwa.

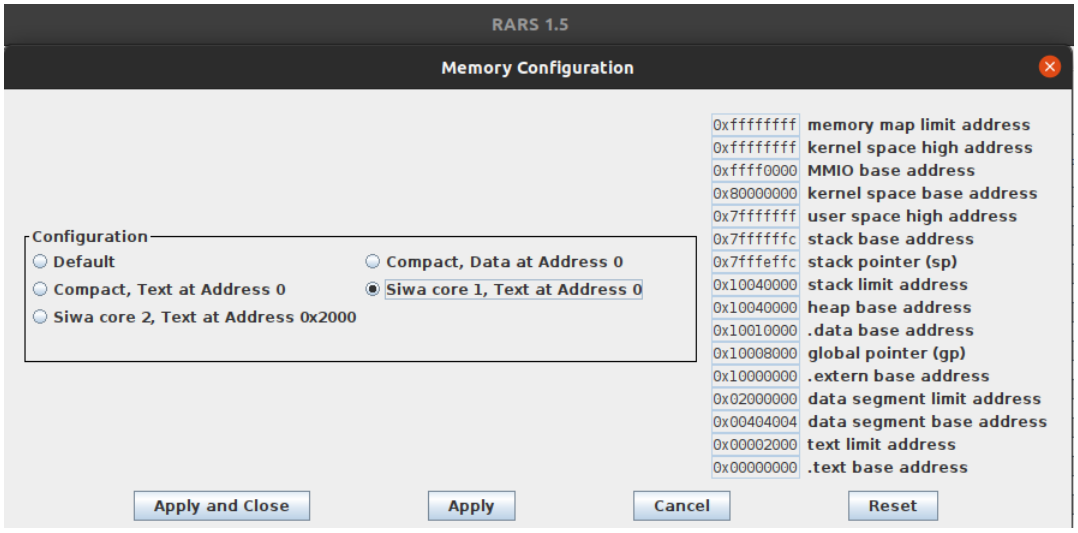

Figura 3.9. Captura de pantalla con las opciones personalizadas del mapa de memoria en RARS para adaptarse a la arquitectura de Siwa.

3.3.1.6. Manejo del mapa de memoria

Para la implementación del mapa de memoria mostrado en la Figura 3.3, se optó por utilizar macros en formato Verilog, de modo tal que cada uno de ellos defina la dirección en memoria del inicio y final de cada región del mapa. De este modo, se favorece la necesidad de reconfiguración del ambiente en futuros escenarios, ya que únicamente será necesario cambiar el valor por el cuál sustituye el macro y el ambiente de verificación se podrá adaptar sin problema alguno. En el Código 3.10 se pueden observar dichos macros en lenguaje SystemVerilog.

```
1  // Start address of text memory region (core 1)
2  `define TXT_MEM_START_C_ONE 25'd0
3
4  // End address of text memory region (core 1)
5  `define TXT_MEM_END_C_ONE 25'h2000
6
7  // Start address of text memory region (core 2)
8  `define TXT_MEM_START_C_TWO 25'h400000
9
10 // End address of text memory region (core 2)
11 `define TXT_MEM_END_C_TWO 25'h404000
12
13 // Start address of data memory region (SRAM)
14 `define DT_MEM_START 25'h404004
15
16 // End address of data memory region (SRAM)
17 `define DT_MEM_END 25'h800000
18
19 // Start address of IO memory region (UART)
20 `define IO_UART_START 25'h800004
21
22 // End address of IO memory region (UART)
23 `define IO_UART_END 25'h1000000
24
25 // Start address of IO memory region (SPI)
26 `define IO_SPI_START 25'h1000004
27
28 // End address of IO memory region (SPI)
29 `define IO_SPI_END 25'h2000000
```

Código 3.10. Extracto del archivo `global_defines.sv`, donde se muestran los macros utilizados para definir el mapa de memoria a lo largo de todo el ambiente de verificación desarrollado.

3.4. Integración del modelo de alto nivel (DUT)

Tal como se mencionó anteriormente, dado que la segunda versión del microcontrolador Siwa aún se encuentra en desarrollo, no fue posible utilizar el RTL como DUT. Por consiguiente, se utilizó un modelo de alto nivel desarrollado por el estudiante Ángelo Hernández como DUT del presente proyecto. Es decir, el bloque correspondiente a Siwa o DUT en la Figura 3.7 se trata de un modelo escrito en SystemVerilog y utiliza el paradigma de programación OOP.

La estrategia que se siguió para integrar el modelo consistió en elaborar una clase *wrapper* que hiciera transparente el funcionamiento y requerimientos propios del modelo con el comportamiento propio del ambiente de verificación desarrollado en este proyecto. Para ello se hizo uso de la base de datos de configuración de UVM, con el objetivo de obtener los punteros de las interfaz de SPI y UART declaradas en el *testbench*. Luego, mediante el uso de métodos o *tasks* fue posible emular los protocolos de los elementos *FIFO in* y *FIFO out* mostrados en la Figura 3.4.a, ya que la forma en que el modelo captura y envía información es mediante el empleo de puertos de análisis de UVM. En el Código 3.11 se muestra el método encargado de traducir las señales provenientes de la interfaz SPI para transmitir las transacciones de bus hacia el modelo del DUT. En dicho Código se incluye la implementación del envío de paquetes hacia el bus del sistema mediante el protocolo de `pndng_int` y `pop_int` mencionado en la sección 3.1.3. Seguidamente, como una característica propia del modelo utilizado como DUT, se envían paquetes vacíos con el objetivo de mantener el modelo activo durante la fase de corrida y ejecutando la rutina de software programada.

```
1    ...
2    // Task for handle the fifo_in side of SPI bus interface
3    task fifo_in_side_spi(uvm_phase phase);
4        forever begin
5            // 1. Wait for external pending
6            `uvm_info(get_name(), "1. Wait for external pending", UVM_NONE)
7            @(posedge spi_vif.pndng_int);
8            // 2. DUT answers with posedge pop_int
9            `uvm_info(get_name(), "2. DUT answers with posedge pop_int and
   acquire data", UVM_NONE)
10           spi_vif.pop_int = 1;
11           if (spi_vif.d_out_int[59:57] == 3'h3) begin
12               // This subroutine is for match behavior of Angelo's
13               // high level model
14               dut_model_seq_item.push_spi   = 1;
15               dut_model_seq_item.D_push_spi = spi_vif.d_out_int ;
16               spi_ap.write(dut_model_seq_item);
17               `uvm_info(get_name(), "4. End boot condition. Exiting loop for
   SPI side", UVM_NONE)
18               break;
19           end
20           else begin
21               dut_model_seq_item.push_spi   = 1;
22               dut_model_seq_item.D_push_spi = spi_vif.d_out_int ;
23               spi_ap.write(dut_model_seq_item);
24               dut_model.m_boot_ended_main_core = 1;
25               dut_model.m_boot_ended_sec_core  = 1;
```

```
26                 `uvm_info(get_name(), "4. Finished ", UVM_NONE)
27             end
28         end
29         `uvm_info(get_name(), "5. Start sending ghost transations and keep the
       model alive ", UVM_NONE)
30         phase.raise_objection(this);
31         forever begin
32             dut_model_seq_item.push_spi   = 0;
33             dut_model_seq_item.reset      = 0;
34             dut_model_seq_item.D_push_spi = 'd0;
35             spi_ap.write(dut_model_seq_item);
36             `uvm_info(get_name(), "Ghost sequence ", UVM_HIGH)
37             if (dut_model.finish == 1'b1 || dut_model.s_finish == 1'b1) begin
38                 `uvm_info(get_name(), "Exit condition was hit.. ", UVM_HIGH)
39                 phase.drop_objection(this);
40                 break;
41             end
42             #30; // arbitrary wait time
43         end
44     endtask : fifo_in_side_spi
45     ...
```

Código 3.11. Implementación del método `fifo_in_side_spi` para la conexión entre la interfaz SPI del ambiente de verificación y el puerto de análisis del modelo del DUT. Aquí se contempló el envío de paquetes hacia el bus del sistema mediante el protocolo de `pndng_int` y `pop_int`. Luego se envían paquetes vacíos con el fin de colocar el modelo o DUT en un estado activo durante la fase de corrida.

Inicialmente se pudo haber considerado la estrategia de construir *drivers* y *monitores* que interactuaran directamente con el puerto de análisis del modelo en lugar de utilizar interfaces, pues hubiera simplificado la necesidad de codificar una clase intérprete o *wrapper* entre ambos proyectos. Sin embargo, bajo la premisa de elaborar un proyecto reutilizable y con miras a adaptaciones futuras al RTL final de Siwa, se optó por el uso de interfaces, ya que, de este modo, la conexión de dicho RTL y funcionamiento sería transparente con el ambiente de verificación y minimizaría al máximo la cantidad de ajustes necesarios.

Finalmente, el código correspondiente al modelo de alto nivel fue utilizado tal y como se encuentra en su repositorio respectivo, sin cambios o modificaciones alguna para efectos de este proyecto. Por consiguiente, se utilizó una variable de ambiente que apunta a un clon local del repositorio remoto creado por el estudiante Ángelo Hernández Cordero.

```
1     -y $DUT_MODEL
2     +incdir+$DUT_MODEL
3     $DUT_MODEL/interface.sv
```

Código 3.12. *Switches* que se utilizaron para incluir el modelo de alto nivel a la compilación del ambiente de verificación funcional con el software VCS.

Capítulo 4

Resultados

A continuación se mostrarán los resultados de ejecutar las diferentes pruebas propuestas en la sección 3.2. Para ello se hará uso de las extensiones de estadística incorporadas en el software RARS, así como las herramientas de análisis de cobertura presentes en las herramientas de VCS y Verdi. Con respecto a la versión del módelo de alto nivel que se utilizó como DUT, se tomó una copia del repositorio de dicho proyecto cuyo último *commit* posee el número de referencia `d303ec1e785b0cf2fce758d8170ee0d5aea69242`.

4.1. Metodología

Con el objetivo de simplificar la ejecución de comandos y promover la creación de un flujo estandarizado de limpieza, compilación, corrida de pruebas y visualización de resultados, se utilizó un archivo Makefile. Dicho archivo se encuentra en la siguiente ruta dentro del repositorio del presente proyecto: `SystemVerilog/Makefile`. En total, se tienen los siguientes *targets* o métodos:

- `compile`: ejecuta el comando de VCS para compilar el ambiente de verificación funcional e incluir los archivos del modelo de alto nivel como DUT del proyecto.
- `clean`: borra la carpeta `csrc` de código fuente generado por VCS posterior a la compilación, además de borrar la carpeta `target` con el ejecutable `simv`, los archivos de registro de simulación, bases de datos de cobertura, entre otros.
- `sim`: permite ejecutar una simulación y obtener un archivo de ondas en formato FSDB, así como guardar el registro de simulación. No utiliza semilla aleatoria. Recibe dos variables: `testname` para indicar el nombre de la prueba que desea correr y `verbosity` para configurar el nivel de verbosidad de UVM deseado.
- `seed_sim`: similar al método `sim`, solo que ahora opera con semillas configuradas automáticamente, para favocerer la aleatorización controlada de las variables en *run-time*.
- `view`: abre el software Verdi y carga un archivo con extensión `.fsdb`. Trabaja con la variable `testname` para indicar el nombre de la prueba cuyo archivo de ondas desea visualizar en Verdi.

- `full_regression`: ejecuta una regresión completa con las pruebas descritas en la sección 3.2. Luego de ejecutar cada una de las pruebas, automáticamente se almacenan los registros de simulación, los archivos en lenguaje ensamblador, el binario de compilación generado por el modelo de referencia y el volcado de la región de datos provisto por RARS. Necesita de una variable llamada `iter`, cuyo objetivo consiste en agregar un sufijo al nombre de cada archivo generado. Tales archivos son preservados y no se ven afectados luego de ejecutar una limpieza con `clean`.

- `cov_results`: permite abrir el software Verdi para visualizar la base de datos de cobertura y luego almacena dicha base de datos en un directorio aparte y que se preserva aún después de ejecutar `clean`.

Finalmente, en el Código 4.1 se muestran la sucesión de métodos que se ejecutarón en una sesión de terminal en Linux para obtener los resultados que se mostrarán en las secciones posteriores del presente capítulo. Lo anterior con el fin de que se quiera replicar la metodología que se siguió en futuras revisiones del presente proyecto.

```
git clone https://github.com/fabianpg3/Proyecto---SIWA-2-Core
cd Proyecto---SIWA-2-Core/SystemVerilog/
source setup_script.sh
make clean
make compile
make full_regression iter=0
make full_regression iter=1
make full_regression iter=2
make full_regression iter=3
make full_regression iter=4
make cov_results
```

Código 4.1. Comandos utilizados para clonar el repositorio del proyecto, cargar los punteros de las herramientas EDA, compilar el ambiente de verificación, ejecutar 5 iteraciones de regresión completa y concluir mostrando la base de datos de cobertura. Esto a través de una sesión de terminal en Linux.

4.2. Resultados de ejecución de pruebas y breve análisis

Luego de ejecutar cuatro iteraciones de una regresión completa, la cual se compone por una instancia de cada una de las pruebas del *testplan*, se obtuvieron los resultados que se muestran en la Tabla 4.1. Allí se enlistan la cantidad de errores, advertencias y escenarios fatales que se generó en cada corrida. Además, nótese que también se incluyen la semilla utilizada, con el fin de poder replicar facilmente el escenario y, por consiguiente, volver a generar los escenarios de error para un escenario más profundo. Tal como se menciona en [8], los sistemas computacionales no son capaces de generar secuencias 100 % aleatorias, por lo que dependen de una llave para poder generar secuencias pseudo-aleatorias; dicha llave se conoce como semilla o *seed*. Se definen los siguientes criterios para dictaminar el resultado de cada prueba:

- Aprobó: no se obtuvo error alguno, por lo que existe concordancia al 100 % entre los resultados generados por el DUT y los datos provistos por el modelo de referencia.
- Falló: se obtuvo al menos un o varios escenarios de error, lo que se traduce en discrepancias entre el valor final esperado de un registro en específico (generado por el modelo de referencia) y el valor final de tal registro en el DUT. Además, otro escenario de error puede ser debido a una diferencia en la cantidad de valores almacenados en la memoria, como parte de la región de datos del modelo de referencia y la obtenida del DUT.

Tabla 4.1. Datos y resultados obtenidos al ejecutar 4 iteraciones de una regresión completa.

	Prueba	**Cantidad errores UVM**	**Cantidad *warnings* UVM**	**Cantidad *fatal* UVM**	**Semilla utilizada**	**Resultado**
1	`cond_incond_jumps_test`	2	0	0	1026923779	Falló
2					1919011662	
3					1829692438	
4					2791463595	
5					819937328	
1	`cpu_both_val_inst_test`	1	0	0	2909364250	Falló
2		4			2133933892	
3		1			4275890890	
4		2			1430089377	
5		0			1575775270	Aprobó
1	`cpu_one_val_inst_test`	0	0	0	3269341973	Aprobó
2		1			294976326	Falló
3		1			2344751335	
4		1			3900296391	
5		0			3046237624	Aprobó
1	`cpu_two_val_inst_test`	5	0	0	585114821	Falló
2		2			2752329855	
3		0			2744379814	Aprobó
4		0			984260968	
5		1			1281796337	Falló

Al analizar los resultados obtenidos de la prueba para saltos condicionales e incondicionales (`cond_incond_jumps_test`), se pudo constatar que las direcciones de memoria 0x800008 y 0x80000c eran recurrentes en mostrar diferencias con respecto al valor provisto por el modelo de referencia. Sin embargo, es importante recalcar que dicha prueba es mas un escenario de análisis dirigido, pues el principal objetivo fue ejercitar la sección de la arquitectura para saltos o *branches*. Por ello, la ejecución de 5 semillas distintas siempre arrojó los mismos resultados. Así, de acuerdo con la sección del programa ensamblador en el Código 4.2, los registros con error persistente corresponden al puntero del *stack* (`sp`) y el puntero global (`gp`) [2].

En el caso del registro gp, se trata de la dirección base en memoria donde se almacena el contenido de las clases de almacenamiento static de una variable en lenguaje C [18]. Por otro lado, el registro sp es un puntero a una región de memoria donde se almacenan y/o remueven valores de registros; sigue un funcionamiento de pila y, como resultado, trabaja con métodos *pop* y *push*, entre otros [18]. Tales registros de punteros se encuentran fueran del alcance del proyecto asociado al modelo de alto nivel que se utilizó como DUT en el presente informe. Sin embargo, a partir de la Figura 4.1, se evidenció que el valor reportado como esperado corresponde a las direcciones base de gp y sp mostradas en la Figura 3.9. En términos generales, las restantes pruebas de instrucciones válidas para el núcleo 1, 2 o ambos, lo anterior no representó un escenario de error, puesto que se aseguró que todos los registros fueran escritos y/o utilizados como operandos. De este modo, el valor de gp y sp pasó a ser un valor conocido, en lugar de ser un dato controlado enteramente por el software RARS.

```
...
lui t1, 0x00000800
addi t1, t1, 0x00000000
...
sw sp, 0x00000008(t1)
sw gp, 0x0000000c(t1)
...
```

Código 4.2. Extracto del programa en lenguaje ensamblador para las pruebas cond_incond_jumps_test.

```
UVM_ERROR verif/scoreboards/main_scoreboard.sv(115) @ 2530000:
uvm_test_top.siwa_env.m_scbd [m_scbd] The RARS output differs
with the output from DUT at address 0x800008
 Memory line from RARS  : 7fffeffc
 Memory line from DUT   : 0

UVM_ERROR verif/scoreboards/main_scoreboard.sv(115) @ 2530000:
uvm_test_top.siwa_env.m_scbd [m_scbd] The RARS output differs
with the output from DUT at address 0x80000c
 Memory line from RARS  : 10008000
 Memory line from DUT   : 0
```

Figura 4.1. Captura de pantalla de los errores obtenidos luego de ejecutar las repeticiones de la prueba cond_incond_jumps_test mostrada en la Tabla 4.1.

Con respecto a las pruebas cpu_both_val_inst_test, cpu_one_val_inst_test y cpu_two_val_inst_test, uno de los errores más recurrentes fue relacionado con la escritura de valores en el registro zero. Tal como se constató en los resultados obtenidos, se verificó que el DUT no es capaz de retener los 32 bits en cero en el registro x0, lo cual dista en gran medida de las especificaciones de RISC-V [2]. Por ejemplo, en el Código 4.3 se detalla una de las secuencias que generó el error que se observa en la Figura 4.2. La instrucción lui permitió inicializar el registro t6 con un valor 0x0004705c. Luego, se realizó un desplazamiento a la izquierda de 0x0000001c (28 base 10) posiciones sobre el valor de t6, lo que terminó en un valor de 32 bits en bajo almacenado en s9. De este modo, una vez que se ejecutó la instrucción ori, el valor que se almacenó en zero fue igual a

0x000004fc, ya que se realizó un una operación OR lógica con uno de los operandos igual a cero. Sin embargo, se esperaba que al final de la simulación el registro `x0` se mantuviera igual a cero.

```
...
lui t6, 0x0004705c
...
slli s9, t6, 0x0000001c
...
ori zero, s9, 0x000004fc
...
lui t1, 0x00000800
addi t1, t1, 0x00000000
sw zero, 0x00000000(t1)
...
```

Código 4.3. Extracto del programa en lenguaje ensamblador para la repetición 2 de la prueba `cpu_one_val_inst_test` mostrada en la Tabla 4.1.

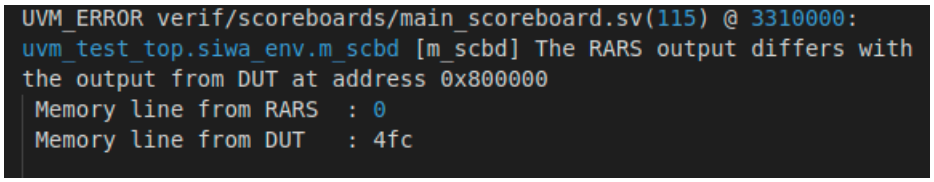

Figura 4.2. Captura de pantalla del error asociado con la escritura en el registro `zero`, como resultado de la repetición 2 de la prueba `cpu_one_val_inst_test` mostrada en la Tabla 4.1.

Ahora, en el caso de la prueba `cpu_both_val_inst_test`, se puede observar que la iteración número 2 fue el escenario donde la mayor cantidad de errores se generaron. En el Código 4.4 se detalla un fragmento de la rutina de software cargada en el DUT. Nótese que intencionalmente se han resaltado los registros `a0`, `s1` y `s3` y como estos fueron evolucionando en el tiempo. A partir de dicha rutina y de los errores que se detallan en la Figura 4.3, se determinó que los problemas se suscitaron a partir de los valores almacenados en los registros `s1` y `a0`; además del caso de escritura al registro `zero`, del cual ya se mencionó anteriormente.

Con respecto al registro `a0`, se trató de una suma entre el puntero del *stack* y el registro `zero`. Para dicho instante, ya se sufrían las consecuencias del error de escritura en `x0`, producto del desplazamiento aritmético entre `s9` y `a3` que se almacenó allí. Caso contrario, el valor esperado en el registro `s1` sería una copia del registro `sp`. Lo mismo sucedió con la operación OR lógica al utilizar `zero` como uno de los operandos, luego la suma de dicho resultado con el puntero global `gp` y, por último, la discrepancia en el resultado almacenado en `a0`. Cabe recalcar que los punteros `sp` y `gp` no presentaron error alguno en esta prueba, ya que los valores por defecto del simulador RARS ya se habían sobreescrito, debido a que se utilizó la estrategia de aleatorización mencionada en la sección 3.2.1 y la semilla que se enlista en la fila correspondiente de la Tabla 4.1.

```
...
lui a0, 0x000123dd
...
sra zero, s9, a3
```

```
...
or s3, a0, zero
...
add a0, sp, zero
add s1, s3, gp
auipc zero, 0x000010e8
...
lui t1, 0x00000800
addi t1, t1, 0x00000000
...
sw zero, 0x00000000(t1)
...
sw sp, 0x00000008(t1)
sw gp, 0x0000000c(t1)
...
sw s1, 0x00000024(t1)
sw a0, 0x00000028(t1)
...
```

Código 4.4. Extracto del programa en lenguaje ensamblador para la repetición 2 de la prueba `cpu_both_val_inst_test` mostrada en la Tabla 4.1.

```
UVM_INFO verif/scoreboards/main_scoreboard.sv(117) @ 5330000: uvm_test_top.siwa_env.
m_scbd [m_scbd] Succesfull comparison at address 0x800008
UVM_INFO verif/scoreboards/main_scoreboard.sv(117) @ 5330000: uvm_test_top.siwa_env.
m_scbd [m_scbd] Succesfull comparison at address 0x80000c
UVM_ERROR verif/scoreboards/main_scoreboard.sv(115) @ 5330000: uvm_test_top.siwa_env.
m_scbd [m_scbd] The RARS output differs with the output from DUT at address 0x800000
 Memory line from RARS   : 0
 Memory line from DUT    : 10e80f4

UVM_ERROR verif/scoreboards/main_scoreboard.sv(115) @ 5330000: uvm_test_top.siwa_env.
m_scbd [m_scbd] The RARS output differs with the output from DUT at address 0x800024
 Memory line from RARS   : 180b000
 Memory line from DUT    : ef42d657

UVM_ERROR verif/scoreboards/main_scoreboard.sv(115) @ 5330000: uvm_test_top.siwa_env.
m_scbd [m_scbd] The RARS output differs with the output from DUT at address 0x800028
 Memory line from RARS   : 0
 Memory line from DUT    : efdef657
```

Figura 4.3. Captura de pantalla con resultados de comparación exitosos (parte superior) y errores en direcciones de memoria (parte inferior) debido a valores incorrectos en los registros `s1` y `a0`, así como el error de escritura en el registro `zero`. Esto como producto de la ejecución de la repetición 2 de la prueba `cpu_both_val_inst_test` mostrada en la Tabla 4.1.

4.3. Estadística y conteo de instrucciones

Una vez ejecutada la regresión completa, se almacenaron cada uno de los colaterales generados por el ambiente de verificación en cuanto las rutinas de software se refiere; es decir, los archivos con

extensión: `.asm` (programa en lenguaje ensamblador), `.s` (archivo binario luego de compilar el programa) y `.dat` (archivo resultante del volcado de la región de datos, tanto de la región entre 0x800000–0x80007c o 0x800080–0x8000fc). De este modo, se procedió a cargar cada uno de ellos en el software RARS pero ahora utilizando la interfaz gráfica, en lugar de acceder e interactuar mediante la línea de comandos. Así, fue posible aprovechar la herramienta de *instruction counter*. En la Tabla 4.2 se muestran los resultados obtenidos luego de cargar cada uno de los archivos `.asm` en RARS. Para el caso de la prueba de instrucciones lógicas y aritméticas en ambos núcleos, se enlistó el resultado de la suma del aporte en el núcleo 1 y el aporte del núcleo 2, ya que, como se mencionó en secciones anteriores, cada rutina de software se ejecutaría en instancias aparte de RARS

Tabla 4.2. Conteo de instrucciones ejecutadas por cada prueba, así como el núcleo ejercitado y considerando el aporte de cada tipo de instrucción utilizada. Datos obtenidos con la herramienta *instruction counter* del software RARS.

	Prueba	**Instrucciones ejecutadas**	**Núcleo ejercitado**	**Tipos de instrucciones ejectuadas**					
				R	**I**	**S**	**B**	**U**	**J**
1	`cond_incond_jumps_test`	94	1	0	29	32	25	1	6
2									
3									
4									
5									
1	`cpu_both_val_inst_test`	196	1 y 2	38	19	64	0	73	0
2				45	16	64	0	69	0
3				41	14	64	0	75	0
4				46	17	64	0	67	0
5				40	22	64	0	68	0
1	`cpu_one_val_inst_test`	98	1	19	10	32	0	36	0
2				19	9	32	0	37	0
3				16	12	32	0	37	0
4				19	11	32	0	35	0
5				19	11	32	0	35	0
1	`cpu_two_val_inst_test`	98	2	22	8	32	0	35	0
2				16	12	32	0	37	0
3				23	9	32	0	33	0
4				16	13	32	0	36	0
5				18	12	32	0	35	0

A partir de la información recopilada en la Tabla 4.2, se extiende el gráfico de la Figura 4.4, donde se observa la contribución de cada archivo en lenguaje ensamblador al conteo final de número de instrucciones ejecutadas, considerando la contribución de cada tipo de instrucción RISC-V. De este modo, es posible determinar que las instrucciones tipo S fueron utilizadas ampliamente, lo cual era de esperar dada a la dependencia de instrucciones `sw` para mover el contenido del banco de registros a

memoria direccionable por UART. En la misma línea, nótese el alto uso de instrucciones tipo U en las pruebas de instrucciones lógicas y aritméticas, lo cual concuerda con la metodología para inicializar variables en los registros por medio de `lui`. Por otro lado, la importancia de la prueba para saltos condicionales e incondicionales radica en quer permitió subsanar la ejecución de instrucciones tipo B y J, ya que son la unica prueba que hicieron posible la verificación de dicha área en el DUT.

Figura 4.4. Representación gráfica de la información en la Tabla 4.2. Los datos son distribuidos por el aporte en cada uno de los archivos *.asm* generados por el ambiente de verificación durante la fase de corrida o *run*.

4.4. Análisis de cobertura

A continuación se presentan los resultados de cobertura obtenidos. Para ello, se conformó una base de datos de cobertura (`simv.vdb/`)con el resultado de 5 iteraciones de una regresión completa con semillas aleatorias en cada prueba. En la Figura 4.5 se observan los datos de cobertura en cada uno de los módulos del banco de pruebas. Debido a que el DUT, para efectos del presente informe, consistía de una clase en lenguaje SystemVerilog, las funcionalidades de cobertura de VCS y Verdi no funcionaron en su totalidad, ya que lo usual y lo esperado por parte de tales softwares de verificación funcional es analizar un módulo en RTL, con sintaxis propia de lenguaje de descripción de hardware. Por consiguiente, la metodología que se siguió fue recolectar la cobertura la cobertura de `toggle` o conmutación en todas las lineas de bits de cada puerto de entrada y salida definida en las interfases entre el ambiente de verificación y la conexión con el bus del sistema en el DUT.

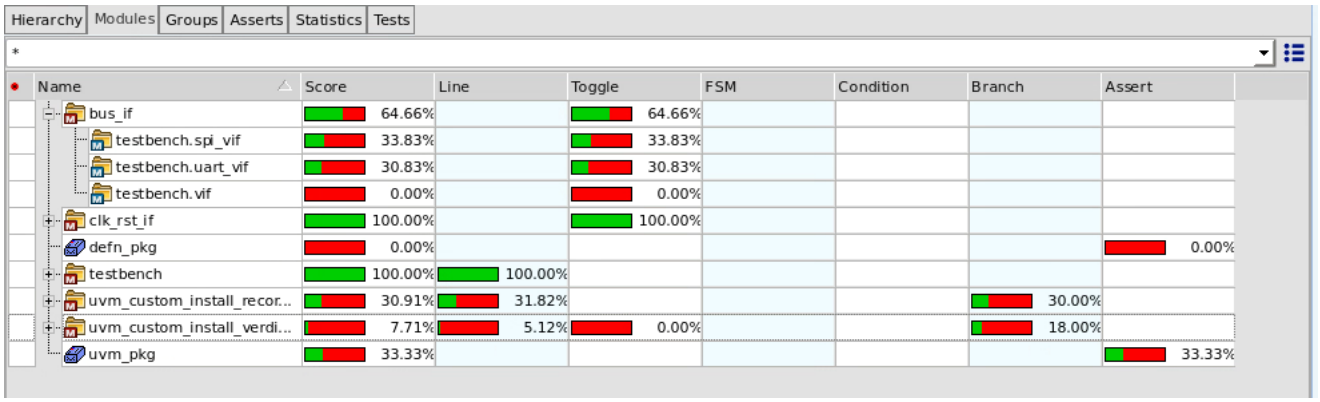

Figura 4.5. Resultados de cobertura de *toggle* sin considerar el uso de exclusiones debido a bits que de antemano se sabe que no presenta conmutación alguna.

De la Figura 4.5 se extrae que las instancias de `bus_if` son las de mayor interés para efectos del presente proyecto, ya que la interfaz para el reloj y *reset* (`clk_rst_if`) no fue implementada en el modelo de alto nivel pero si se dejó prevista (por ello no fue mostrada en la arquitectura de la Figura 3.7). Bajo la jerarquía de `bus_if` se encuentra la instancia de SPI y UART. Para mayor detalle, en la Figura 4.6 y en la Figura 4.7 se incluye en el desglose de la cobertura de *toggle* en las interfases equivales de SPI y UART, respectivamente.

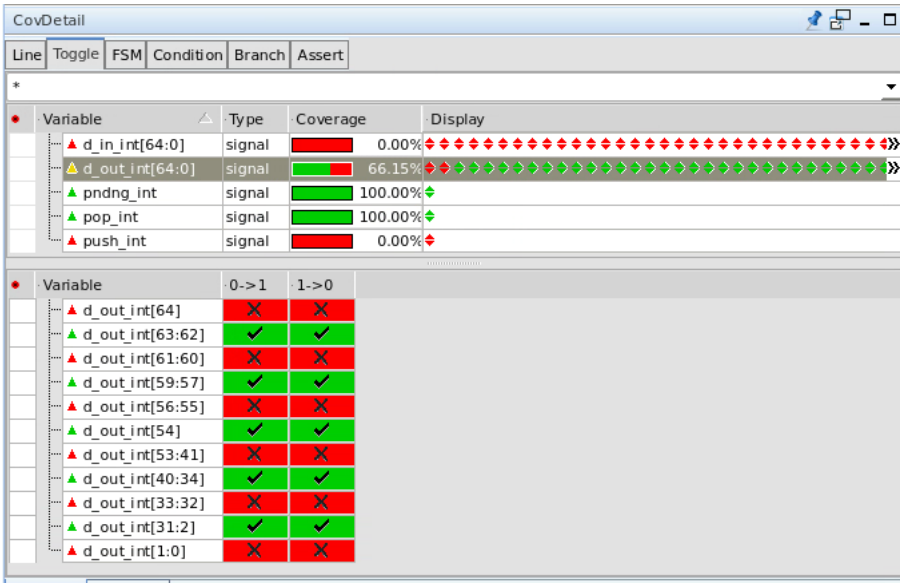

Figura 4.6. Distribución de los resultados de cobertura en cada uno de los puertos y líneas de bits de la interfaz para SPI. El color rojo representa aquellos bits en específico que no presentaron transición alguna. Caso contrario, se representa con el color verde. Nótese que aún no se empleó el uso de exclusiones.

Con respecto al resultado de cobertura en la interfaz equivalente de SPI, se puede observar que existió un 0 % de conmutación en los bits de los puertos `d_in_int` y `push_int`. Esto era esperable, puesto que la intención del puerto SPI en general es para dar soporte a la transferencia de la rutina

de software a ejecutar, por lo que el tráfico de información es desde el exterior hacia el interior y no en dirección contraria, tal como se detalló en la sección 3.1. Sin embargo, para ello se implementan el uso de las exclusiones, las cuales permiten dejar por fuera de manera intencional ciertas líneas de bits o áreas específicas del diseño en general para el análisis de cobertura.

Por otro lado, ocurre algo similar en el caso de la cobertura de la interfaz equivalente de UART, ya que el objetivo de las pruebas era ejercitar el trasiego de información desde el DUT hacia el ambiente de verificación funcional y así poder acceder a los valores finales en el banco de registros. Por consiguiente, no se estimularon los puertos `d_out_int`, `pndng_int` y `pop_int`. Lo anterior fue más una imposición a raiz del modelo de referencia utilizado, ya que no era posible simular el envío de datos de manera asincrónica a través de un bus con arquitectura personalizada. No obstante, esto corresponde a un escenario ideal a incluir dentro del archivo de exclusiones, pues se sabe de antemano que tales puertos y las líneas de bits que engloban no van a ser conmutadas con el plan de pruebas propuesto en la sección 3.2.

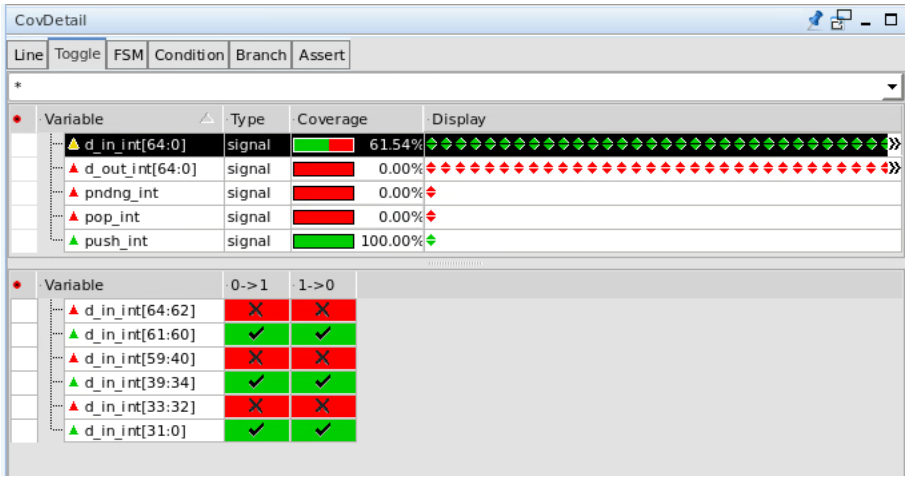

Figura 4.7. Distribución de los resultados de cobertura en cada uno de los puertos y líneas de bits de la interfaz para UART. El color rojo representa aquellos bits en específico que no presentaron transición alguna. Caso contrario, se representa con el color verde. Nótese que aún no se empleó el uso de exclusiones.

Luego de aplicar las exclusiones en los puertos `d_in_int` y `push_int` de SPI y en los puertos `d_out_int`, `pndng_int` y `pop_int` de UART se obtuvo un resultado preliminar de cobertura en las interfases equivalentes de 67.16 % y 62.12 %, respectivamente. Nótese que, aunque el porcentaje final se incrementó en alrededor de un 30 % con respecto a los valores mostrados en la Figura 4.5, aún existe área de mejora para formular de pruebas que permitan conmutar los bits marcados con rojo en la parte inferior de la Figura 4.6 y la Figura 4.7. En la Tabla 4.3 se muestra un breve análisis entre los bits que mostraron nula cobertura de *toggle* y los escenarios de prueba planteados. De este modo, es posible aplicar una segunda iteración de exclusiones, cuyo resultado corresponde a los valores finales cobertura alcanzados: 95.74 % para SPI y 100 % para UART. Dichos valores se muestran en la Figura 4.8.

Tabla 4.3. Análisis de bits de cada puerto en específco que no presentaron aporte a la cobertura de *toggle*. Datos obtenidos a partir de las Figura 4.6 y 4.7.

Interfaz	**Puerto de entrada/salida**	**Bits sin conmutar**	**Significado en el paquete de bus (Figura 3.4.b)**	**Comentarios**
SPI	`d_out_int`	[64]	Corresponde al bit más significativo del campo de destino.	Dado que el destino siempre fue el núcleo 1 y/o 2 (0x0 o 0x3), no era posible una transición en dicho bit. La única forma sería comunicandose directamente con la SRAM en MCB, el cual sería el código 0x4.
		[61:60]	Codifican el origen del paquete o quién lo envía.	El destino siempre fue 0x2 dado que se trata de la interfaz con SPI, por lo que no era posible una transición en estos bits.
		[56:55], [53:41], [33:32]	Dirección de memoria asociada al paquete	Dado que la región de texto en el mapa de memoria de Siwa está entre las regiones [0x0:0x200] para el núcleo 1 y [0x400000:0x404000] para el núcleo 2, gran cantidad de bits se mantienen sin transición alguna, más aún que tales regiones no fueron escritas en su totalidad.
		[1:0]	Instrucción RISC-V	-
UART	`d_in_int`	[64:62]	Corresponde al bit más significativo del campo de destino.	En este caso, no fue posible establecer transición alguna, debido a que el código de destino siempre fue 0x1.
		[59:40], [33:32]	Dirección de memoria asociada al paquete	La region direccionable por el puerto UART comienza en 0x800000 y termina en 0x1000000, aunque en este proyecto se trabajó únicamente en el rango de [0x800000:0x8000fc], por lo que gran cantidad de bits no se pudieron conmutar.

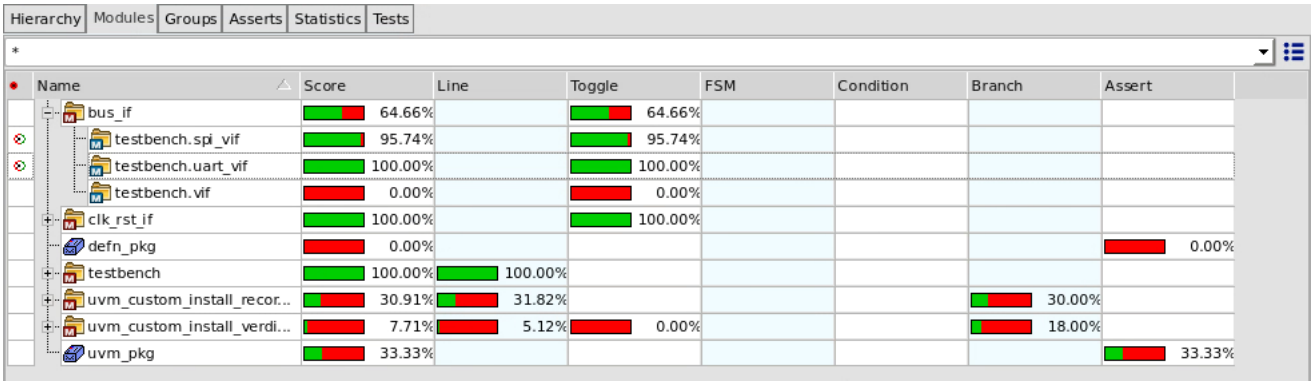

Figura 4.8. Resultados finales de cobertura de *toggle*, una vez aplicado el uso de exclusiones justificado en la Tabla 4.3.

Capítulo 5

Conclusiones

- Aunque la segunda versión del microcontrolador Siwa se encuentra en desarrollo, fue posible generar una lista con las principales características que se esperan del nuevo diseño, así como una revisión de las funcionalidades que se heredan de la primera versión.
- El plan de pruebas que se formuló permitió verificar de manera inicial el apego al estándar RISC-V. Sin embargo, dado a que la implementación del hardware de interrupciones no es muy claro en la presente fase del desarrollo de Siwa, los escenarios de interrupciones enmascarables y no mascarables no fueron cubiertos.
- La arquitectura propuesta del ambiente fue capaz de adaptarse a las necesidades de verificación, además de que su funcionamiento fue transparente al modelo de alto nivel que se utilizó como DUT e invita a una facil conexión, en un futuro, con el diseño en RTL.
- Mediante la ejecución de pruebas, resultados recopilados y su respectivo análisis, se demuestra el estado actual operativo del ambiente de verificación, así como la capacidad de detectar errores iniciales, tal como lo fue la escritura al registro `zero`.

5.1. Trabajo futuro

- Una vez la fase de diseño de RTL de la segunda versión de Siwa se encuentre más avanzada, el modelo de alto nivel utilizado como DUT podría pasar a ser parte del *scoreboard* y funcionar en paralelo con el software externo de modelo de referencia.
- Considerar los escenarios de interrupción y manejos de instrucciones CSRs en futuras revisiones del plan de pruebas propuesto.

Bibliografía

[1] HPC-LAB, “Micro - architectural specification: Tec-riscv,” 2022.

[2] A. Waterman and K. Asanović, “The risc-v instruction set manual, volume i: Unprivileged isa, version 20191213,” CS Division, EECS Department, University of California, Berkeley, Tech. Rep., December 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[3] A. Waterman, K. Asanović, and J. Hauser, “The risc-v instruction set manual, volume i: Unprivileged isa, version 20191213,” CS Division, EECS Department, University of California, Berkeley, Tech. Rep., December 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[4] M. Guzmán. Con desarrollo de primer microcontrolador, se abren importantes oportunidades para el tec. [Online]. Available: https://revistas.tec.ac.cr/index.php/investiga_tec/article/view/4670/4253

[5] Intel. Fun facts. exactly how small (and cool) is 22 nanometers? [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/corporate-information/history-moores-law-fun-facts-factsheet.pdf

[6] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Castro-Gonzalez, A. Arnaud, M. Miguez, J. Gak, R. Molina-Robles, G. Madrigal-Boza, M. Oviedo-Hernandez, E. Solera-Bolanos, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, and R. Rimolo-Donadio, “Siwa: a RISC-V RV32I based Micro-Controller for Implantable Medical Applications,” in *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 2020, pp. 1–4.

[7] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed. USA: Addison-Wesley Publishing Company, 2010.

[8] B. Wile, J. Goss, and W. Roesner, *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[9] R. Salemi, *The Uvm Primer: A Step-By-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.

[10] O’Reilly. Technique 40: Pugh Matrix. [Online]. Available: https://www.oreilly.com/library/view/the-innovators-toolkit/9781118331873/9781118331873c40.xhtml

[11] Accellera Systems Initiative. About us. [Online]. Available: https://www.accellera.org/about

[12] ——. Universal verification methodology (uvm) working group. [Online]. Available: https://www.accellera.org/activities/working-groups/uvm/

[13] C. B. Spear, *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[14] R. Garcia-Ramirez, A. Chacon-Rodriguez, R. Molina-Robles, R. Castro-Gonzalez, E. Solera-Bolanos, G. Madrigal-Boza, M. Oviedo-Hernandez, D. Salazar-Sibaja, D. Sanchez-Jimenez, M. Fonseca-Rodriguez, J. Arrieta-Solorzano, R. Rimolo-Donadio, A. Arnaud, M. Miguez, and J. Gak, "Siwa: A custom risc-v based system on chip (soc) for low power medical applications," *Microelectronics Journal*, vol. 98, p. 104753, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026269219303787

[15] RISC-V. Risc-v exchange. [Online]. Available: https://riscv.org/exchange/?_sft_exchange_category=software&_sfm_exchange_software_type=Simulators

[16] B. Landers, "RARS – RISC-V Assembler and Runtime Simulator," https://github.com/TheThirdOne/rars, 2022.

[17] A. I. Munteanu. How to call c-functions from systemverilog using dpi-c. [Online]. Available: https://www.amiq.com/consulting/2019/01/30/how-to-call-c-functions-from-systemverilog-using-dpi-c/

[18] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

Apéndice A

Abreviaciones comúnes

A continuación se presenta una lista del significado de abreviaciones que fueron utilizadas ampliamente a lo largo del presente documento.

CI: Circuitos Integrados.
HDL: *Hardware Description Language* o lenguaje de descripción de hardware.
OOP : Programación Orientada a Objetos, por sus siglas en inglés
UVM : Metodología de Verificación Universal, por sus siglas en inglés.
RTL : Codificación a nivel de transferencia de registros, por sus siglas en inglés.
FPGA : *Field Programmable Gate Array*.
ISR: *Interrupt Service Routine*.
FSM: *Finite State Machine*
DPI: *Direct Programming Interface*