INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERIA ELECTRÓNICA

Verificación funcional de un controlador de memoria para un dispositivo médico implantable.

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

Jean Carlo Rodríguez Mejía

Cartago, Junio del 2018

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

PROYECTO DE GRADUACIÓN

ACTA DE APROBACIÓN

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

El Tribunal Evaluador aprueba la defensa del proyecto de graduación denominado Verificación funcional de un controlador de memoria para un dispositivo médico implantable., realizado por el señor Jean Carlo Rodríguez Mejía y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

Ing. Aníbal Ruiz Barquero
Profesor lector

Ing. Esteban Baradán Mendez
Profesor lector

Ing. Roberto Molina Robles
Profesor asesor

Escuela de Ingeniería en Electrónica, 20 de junio de 2018

Declaración de autenticidad

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

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

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

Jean Carlo Rodríguez Mejía

Cédula: 1 1574 0188

Resumen

Cuando se realiza un diseño para un producto que necesita tener un funcionamiento estable, continuo y que no se espere que falle, se debe comprobar el funcionamiento de los dispositivos antes de entregarlos al usuario final, para este caso el microprocesador diseñado por la Escuela de Electrónica se debe asegurar que el funcionamiento casi perfecto ya que su aplicación como parte de dispositivos médicos implantables pueden asegurar la vida o causar la muerte del usuario. Para poder asegurar el correcto funcionamiento de estos dispositivos se va a usar la Metodología Universal de Verificación o UVM, esta metodología usada por las industrias de diseño de circuitos integrados asegura que desde etapas de diseño se pueda comprobar el funcionamiento de hasta el ultimo componente de este microprocesador; este documento se centra en el controlado de memoria y el diseño del entorno para poder realizar su verificación.
Palabras Clave: verificación funcional, UVM, verificación, controlador de memoria, code coverage, functional coverage.

Abstract

When a design is made for a product that needs to have a stable and continuous operation and also that is not expected to fail, the fucntional operation of the devices must be checked before delivering them to the end user, in this case the microprocessor designed by the Escuela de Electrónica must be ensured that the operation is almost perfect since its application as part of implantable medical devices so it can ensure life or cause the death of the user. In order to ensure the proper functioning of these devices, the Universal Verification Methodology or UVM will be used to verify the function of this device, this methodology is used by the integrated circuits design industries, it will ensure that from design stages it is possible to verify the operation of up to the last component of this microprocessor; This document focuses on memory control and the design of the environment to be able to verify it.
Palabras Clave: functional verification, UVM, verification, memory controller, code coverage, functional coverage.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Entorno del proyecto

Las aplicaciones de la electrónica en la actualidad se encuentran en todos los campos de la sociedad, esto se ha logrado por los avances que han tenido los sistemas digitales por las contantes mejoras en los procesos del silicio usado para crear estos mismos.

Estos sistemas han ido volviéndose más complejos por lo que tienden a ser más propensos a las fallas de diseño o de implementación causadas por los requisitos del diseño, pero hay aplicaciones que no pueden permitirse la posibilidad de tener un error en su funcionamiento ya que corre el riesgo de causar pérdidas al sistema, monetarias o de vidas humanas por lo que estas aplicaciones deben tener un riguroso proceso de debug que asegure el más completo y correcto funcionamiento del mismo, aquí es donde entran los procesos de verificación funcional en el controlador de memoria para un dispositivo medico integrado donde los errores van a afectar la vida de una persona por lo que no deben permitirse.

Conforme más se avanza en las de desarrollo el producto más difícil y costoso es reparar los errores encontrados por esto, verificar el correcto funcionamiento se debe realizar lo más pronto posible con un proceso de verificación funcional del mismo desde etapas de prototipado en lenguajes de descripción de hardware para asegurar que cumplen sus especificaciones y que no hay escenarios que pueden llegar a hacerlos equivocarse, por esto desde etapas tempranas se deben de realizar miles o millones de pruebas de tortura de diferentes escenarios para comprobar el correcto o incorrecto funcionamiento del sistema ante todas las circunstancias que puedan suceder en el sistema.

1.2. Definición del problema

1.2.1. Generalidades

En los diseños de hardware siempre ha sido en mismo diseñador el que debe asegurar que lo que esta realizando funciona correctamente por lo que este siempre ha sido encargado de verificar que cumple con todas las especificaciones establecidas, pero conforme los diseños se

volvieron más complejos se empezó a volver mas difícil realizar ambas tareas a la vez y además realizarlo con un tiempo limitado, también surge otra posibilidad en el proceso de verificación que es lo que sucede cuando se presentan escenarios no contemplados y que comportamiento va a tener el diseño en estos casos, generando así que se generen nuevos equipos encargados de comprobar el funcionamiento en lugar de los diseñadores lo que permite más libertad para las pruebas y diferentes escenarios, pero aun hay una variable que no se vuelve fácil de concretar que es la limitante del tiempo ya que los equipos de verificación deben tener los pruebas listas al mismo tiempo que los diseños entran en etapas de pruebas además de que conforme aumenta el tiempo se pase realizando las pruebas la profundidad de las pruebas y errores encontrados va a incrementar, aquí nace la primera metodología de verificación la Open Verification Methodology y de esta nace la Universal Verification Methodology estas establecen una manera de seccionar la verificación de manera que se pueda iniciar desde antes de tener las especificaciones del diseño que se desea verificar además de que poseen compatibilidad con otros diseños lo que hace que se mejore los tiempos de proyectos de verificación siguientes así como facilita hacer arreglos o mejoras a proyectos en curso y para mejorar la compatibilidad estas metodologías requieren la generación de un documento el cual permite que cualquier ingeniero pueda continuar cambiar o mejorar la verificación de un diseño sin tener que volver a reconstruir todo el sistema este se conoce como Plan de verificación; todo esto vuelve a la UVM la técnica escogida para la verificación del controlador de memoria.

1.2.2. Síntesis del problema

No hay manera de realizar la verificación funcional del controlador de memoria en el Instituto Tecnológico de Costa Rica.

1.3. Enfoque de la solución

Para completar esta tarea se debe construir un sistema de verificación para el controlador de memoria desde cero ya que no existe ningún sistema establecido que pueda completar esta tarea con facilidad, conociendo los beneficios de la metodología UVM la implementación de esta se vuelve la mejor manera de asegurar que la verificación del controlador de memoria, por lo que la implementación de una máquina de verificación completa que se ejecute en uno servidor de la Escuela de Electrónica es la base de la solución del problema al no tener certeza de las especificaciones o de los tiempos en los que se van a realizar las pruebas esto nos permite construir la máquina de verificación mientras se completan las especificaciones para luego después de diseñar las pruebas y modelo de referencia que van a ser usados para comprobar si el controlador diseñado cumple con todos los requerimientos establecidos y encontrar los errores no planteados en su funcionamiento, lo que lo lleva a generar estos errores y el comportamiento que muestra cuando suceden estos errores; y como parte final el generar el Plan de verificación correspondiente el cual permite que todo este trabajo realizado cumpla el objetivo de verificar el controlador de memoria aun después de terminado este proyecto y las siguientes etapas de desarrollo del microprocesador.

1.4. Meta

Establecer un sistema capaz de realizar la verificación funcional del controlador de memoria durante todo su proceso de diseño e implementación.

Objetivos

1.5. Objetivo General

Evaluar el funcionamiento del controlador de memoria por medio de la verificación funcional.

1.6. Objetivos específicos:

- Construir máquina de verificación en la cual se realizará la verificación funcional del controlador de memoria.
- Implementar interfaces y secuencias para la verificación del controlador de memoria en la máquina de verificación.
- Registrar los resultados en el plan de verificación.

Capítulo 2

Risk V y el CMB

2.1. El módulo por verificar

El módulo por verificar se llama Controlador de Memoria y Bus en su etapa final de diseño, inicialmente siendo llamado Controlado de Memoria o MMU por sus iniciales en inglés Memory Management Unit, ya que esta era la base para definir los comportamientos que iba a poseer esta unidad y su relación con él procesador, aun así en las etapas iniciales del desarrollo del diseño se fue cambiando su funcionamiento para darle un comportamiento mas cercano a interfaz de comunicación del procesador con cualquier dato, adquiriendo en su interior la RAM y manejador de bus, esto para lograr simplificar las instrucciones a nivel de software, llegando así a tener agregado los funcionamientos de controlar los datos que llegan desde el bus y entregarlos al procesador además de poseer internamente la RAM y su controlador, de aquí su nombre pasa a ser el Controlador de Memoria y Bus o CMB basado en sus iniciales de su nombre en español; las decisiones de porqué y cómo fueron tomadas las decisiones de diseño, acerca de cómo va a actuar respecto a los diferentes escenarios de diferentes entradas o solicitudes de datos y los bloques de operaciones que le fueron agregados, son ajenas a el trabajo del verificador y no son esenciales para realizar la verificación, además de que los diseñadores no poseen un repositorio de versiones anteriores o un control de cambios que se pueda acceder por un verificador; por esto si se desea saber acerca del proceso de diseño del CMB se recomienda acceder el documento respectivo acerca del diseño de ese ya que este documento se va a basar en los datos obtenidos e información presente en la ultima especificación entregada por parte del diseñador y ya que cualquier otro dato no puede ser respaldado sin la documentación acerca del proceso de diseño se va a mantener el mínimo las explicaciones acerca del funcionamiento del módulo.

La verificación se realiza en base a la lista de puertos y listado de funcionamientos que se pueden obtener de las especificaciones de funcionamiento del CMB, los puertos se pueden agrupar en secciones que permiten sus funcionamientos ser ejecutados en paralelo, aun así no se posee una clara definición acerca de si el procesador va a poder realizar todas estas operaciones en un mismo ciclo de reloj o va a poseer ciclos de espera entre estas operaciones esto a causa de que el controlador principal aún no se encuentra completamente diseñado o implementado al momento de escritura de este documento. En la figura 1 podemos ver las

entradas y salidas del CMB, por ser un trabajo en black box no se conoce ni es necesario tener información de los bloques internos.

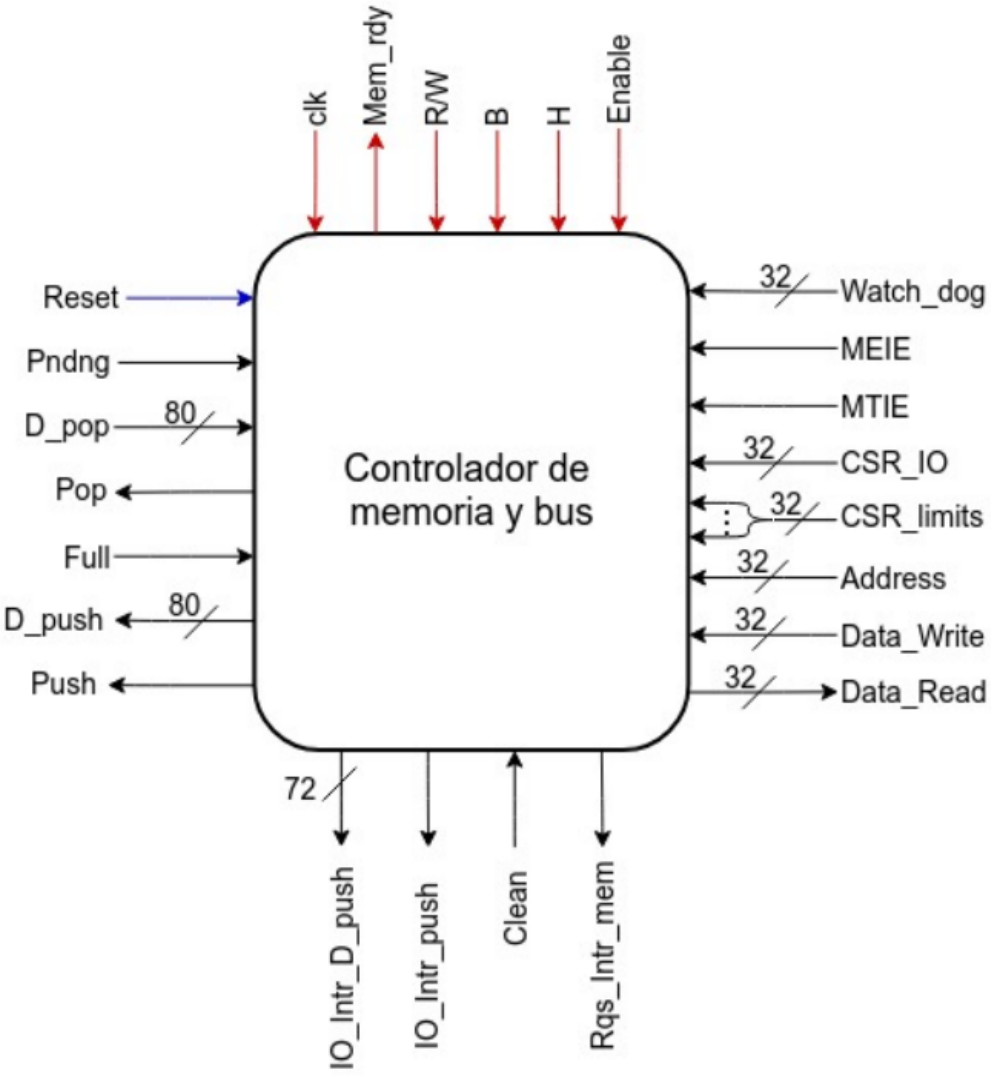


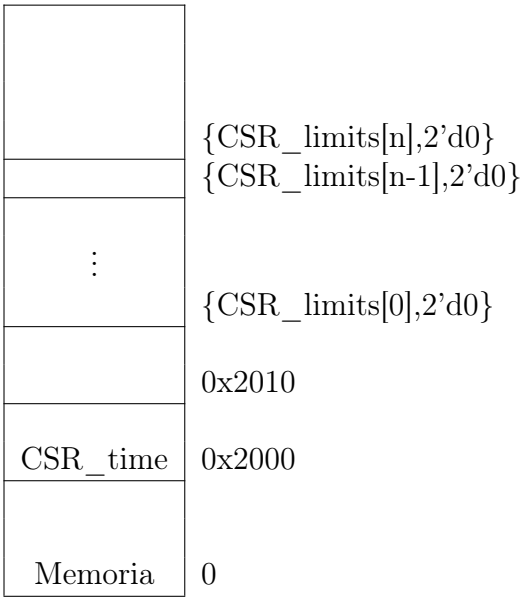
Figura 1: Diagrama de primer nivel del modulo a verificar.

2.1.1. Listado de Puertos por secciones funcionales:

Señales básicas de funcionamiento:

- Reset: input 1 bit, entrada asincrónica que permite el reinicio del funcionamiento del modulo.
- Clk: input 1 bit, reloj principal del sistema.

Cuadro 1: Mapeo de memoria



Interfaz con el bus:

- Pndng: input 1 bit, avisa de la presencia de dato en puerto de bus respectivo y solicita que sea procesado.
- Dpop: input 80 bits, dato de proveniente del bus se subdivide en las secciones mostradas en la figura(imgformatoDbus) estas se explicarán más adelante.
- Pop: output 1 bit, avisa al bus que el dato ha sido procesado y puede ser eliminado del puerto de datos.
- Full: input 1 bit, avisa que no se pueden recibir datos en el bus de datos de manera que no se puede procesar un Push de datos.
- D_push: output 80 bits, dato enviado al bus se subdivide en las secciones mostradas en la figura(imgformatoDbus) estas se explicarán más adelante.
- Push: output 1 bit, avisa al bus que debe ejecutar el dato colocado en puerto de bus respectivo.

Interfaz con el manejador de interrupciones

- IO_Intr_D_push: output 72 bits, dato enviado al manejador de interrupciones, se subdivide en las secciones mostradas en la figura(imgformatoDintr) estas se explicarán más adelante.
- IO_Intr_push: output 1 bit, avisa al manejador de interrupciones que el dato en el puerto de datos de interrupciones debe ser procesado.

- Clean: input 1 bit, avisa que se puede enviar el dato al manejador de interrupciones para ser procesado.
- Rqs_Intr_mem: output 1 bit, solicita al manejador de interrupciones permiso para enviar dato por medio del puerto de datos de interrupciones.

Interfaz con el procesador:

Interfaz de operación de datos del procesador:

- Mem_rdy: output 1 bit, señal que indica que el proceso de datos se terminó y que se puede continuar con las operaciones de lectura o escritura.
- R_W: input 1 bit, indica si la operación solicitada es una escritura al estar en alto y una lectura en bajo.
- B: input 1 bit, indicador de cantidad de bytes que va a ser procesados en la operación, opera según tabla 2
- H: input 1 bit, indicador de cantidad de bytes que va a ser procesados en la operación, opera según tabla 2
- Enable: input 1 bit, indicador de que hay una operación por parte del procesador.
- Address: input 32 bits, dirección donde se va a ejecutar la operación del procesador.
- Data_Write: input 32 bits, datos que se van a ejecutar en la operación del procesador.
- Data_Read: output 32 bits, datos resultantes de la operación del procesador.

Cuadro 2: Decodificación de alineado

	DIR Address[1:0]	Amount_bytes{B,H}
Alineado	00	00
Desalineado	01	00
Desalineado	10	00
Desalineado	11	00
Alineado	00	01
Desalineado	01	01
Alineado	10	01
Desalineado	11	01
Alineado	00	1X
Alineado	01	1X
Alineado	10	1X
Alineado	11	1X

Interfaz de operación de CSR's:

- Watchdog: input 32 bits, valor máximo de cuenta para operaciones de procesador que requieren espera.
- MEIE: input 1 bit, habilita la eliminación de datos provenientes del bus sin ser procesados mientras se encuentra en alto.
- MTIE: input 1 bit, inicia la cuenta de CSR de cuenta, cada vez que es pasado de cero a uno se reinicia el valor en CSR de cuenta y mientras esta en cero los CSR de cuenta son colocados en cero.
- CSR_IO: input 32 bits, cada uno de sus bits representa un IO de manera que si esta en alto o bajo el IO va a ser operado de manera diferente cuando sea accedido por el procesador.
- CSR_limits: input array de elementos de tamaño 32 bits, no se encuentra definido cuantos son, solo el ultimo es el primer byte después del ultimo byte de un IO mapeado y el primero debe estar por encima de 0x2010.

2.1.2. Listado de operaciones diferentes ejecutables por el CMB:

Operación de boot:

Para poder iniciar el funcionamiento este modulo necesita un proceso de boot iniciado por la bandera de Reset en 1, el reinicio del sistema es asíncrono, pero cabe destacar que el de este modulo no es negado, se ejecuta en los siguientes pasos.

- El CMB genera el dato de inicialización de boot 0x00014000000
- Después se reciben solo escrituras a RAM, cualquier otro dato desde él bus es eliminado por medio de un Pop.
- El bus finaliza el boot con un 0x00018000000

Operaciones de procesador:

Todas estas operaciones primero analizan alineado; para definir si los datos se encuentran alienados se usan las entradas B y H de manera que si B esta en alto la operación se realiza de 1 byte, mientras B este en bajo si H está en alto la operación es de 2 bytes o si esta en bajo es de 4 bytes, donde el byte de inicio de la Address debe corresponder al tipo de lectura solicitada sino se genera la interrupción de desalineado, esto se ejemplifica en la tabla(tabladesalineadopro); si se encuentran desalineados se genera la interrupción respectiva.

La siguiente comparación es el mapeado en memoria ya sea como RAM o IO; las operaciones de RAM y IO generan diferentes salidas en el sistema ya que los IO son accedidos por medio del bus, mientras la RAM se encuentra dentro del CMB; aparte de estos los CRS de cuenta y de comparación con accedidos como si fueran RAM aunque no son parte de esta y no poseen el mismo comportamiento ante los casos de lectura o escritura, estos se explicaran más adelante; si se encuentran fuera del mapeado se genera la interrupciones respectiva.

Lectura y escritura de memoria RAM se realizan sin análisis extra y colocando en Data-Read el valor leído o ceros respectivamente, la forma que los bytes son operados y colocados en sus destinos se pueden ver en tabla (tablaramformatos)

Las escrituras en IO generan un paquete solicitando datos en el puerto del bus y después de la generación de Push la operación termina.

Las lecturas de IO se dan de 2 maneras diferentes en base al valor de CSR_IO del io respectivo, si es cero entonces se envía el paquete y termina el procesamiento, si es 1 se inicializa el watchdog y se almacena en el fifo interno cualquier dato que no sea el dato de respuesta del io respectivo.

Estas operaciones siempre terminan con Mem_rdy sin importar cual haya sido el resultado.

Operaciones de CSR:

Estos pueden ser leídos y escritos pero solo toda la palabra ya que no son parte de la RAM y su interfaz es diferente, los CSR de comparación solo van a ser leídos cuando se encuentran en cuenta, si se encuentra en espera siempre se encuentran en cero aunque se escriba un dato en ellos; cuando los CSR de cuenta llegan al valor de los de comparación se genera la interrupción y se deja de contar hasta que la interrupción haya salido, donde van a ser desactivados por el procesador, aun así no se encuentra claro el funcionamiento de desactivado por parte del procesador ya que los controladores no han sido escritos aun así si que no se puede definir su comportamiento.

Operaciones de BUS:

Dependiendo de si MTIE esta activo o desactivado se va a pasar los datos a interrupción para que le procesador los maneje después o se va a realizar un pop para eliminar el dato sin procesarlo, estas operaciones siempre terminan en Pop después de realizar todo el procesamiento del dato respectivo.

Interrupciones:

Las interrupciones se envían al manejador de interrupciones si después de solicitar un Rqs_Int_mem la bandera de Clean se encuentra en alto de manera que se realiza un push con la información de la interrupción respectiva de acuerdo con la tabla (tablaintrlist), los

procesos que generan las interrupciones no son terminados hasta que se haya realizado el IO_Intrpush de los datos al manejador de interrupciones de manera que se quedan en espera hasta recibir el Clean.

Capítulo 3

Marco teórico

3.1. La verificación funcional, origen y funcionamiento

3.1.1. Origen

Con los avances de la tecnología de fabricación de silicio se logro integrar múltiples circuitos en solo uno en lo que se conoce como circuitos integrados, esto logra realizar más funcionalidades en un solo chip de una manera mas eficiente, volviendo los diseños mas complejos conforme los requiera la aplicación, esto genere lo que se conoce como un explosión en el espacio de estados ya que una pequeña maquina de estados puede manejar millones de combinaciones de posibles salidas o entradas; para ejemplificar esto podemos tomar un simple ejemplo de una operación de un registro que solo puede leer o escribir, si iniciamos con un tamaño de 8 bits son 256 posibles valores de lectura o escritura que se puede manejar con solo 1 bit, se tiene una cantidad de 256 x 2 = 512 diferentes operaciones para solo 1 registro de 8 bits, cuando se usa en un array para tener una memoria cache o RAM, usando solo 8 bits para direccionar se pasa a tener 256 x 256 x 2 = 131072 diferentes operaciones y cuando se pasa a 32 bits las posibles operaciones se encuentran fuera de la posibilidad comprobarlas todas.

Para poder asegurar el correcto funcionamiento de los circuitos los diseñadores deben asegurarse de definir todos los posibles estados de manera que el sistema no se salga de su control y ademas realizar pruebas para comprobar que su diseño se comporta de manera que se espera en todos los caso posibles, aun así estas tareas se vuelven muy difíciles de realizar cuando se toma en cuenta los plazos de tiempo para completar un diseño, mientras mas se tarde en completar un diseño mas costoso su desarrollo va a ser y los beneficios van a disminuir, lo mismo sucede para el tiempo de prueba, ya que mientras mas tarde se empiecen a realizar las pruebas mas se va a tardar para completar la comprobación de todos los posibles comportamientos pero estas mismas pruebas pueden tardar absurdas cantidades de tiempo por todos los millones de estados posibles y si se agregan análisis secuenciales los tiempos se vuelven infinitos; para disminuir la carga de trabajo se dividen las tareas de diseño y verificación en diferentes equipos de trabajo para así hacer que estos puedan trabajar de manera paralela volviendo las tareas más fáciles de completar.

3.1.2. La verificación funcional y metodología UVM

La verificación funcional nace como una manera de dividir tareas para aligerar las cargas y apresurar los tiempos para completar un diseño, de manera que se pueden generar los diseños a la vez que se construyen las pruebas y se hacen las pruebas para búsqueda de errores lo mas rápido posible; aun así la misma verificación presentaba múltiples problemas, con la cantidad posible de pruebas ademas de tener tiempos de trabajo y de pruebas limitados el diseño de un sistema para verificar cada diseño se vuelve un proceso muy complejo y por más pruebas que se realicen a cada diseño siempre van a existir casos que no se van a poder comprobar por la limitante del tiempo, todo esto se volvió un delicado equilibrio a manejar de manera que se busco la mayor eficiencia de manera que se puedan encontrar la mayor cantidad de errores lo más pronto posible en el proceso de manufactura.

La metodología UVM se deriva de la metodología OVM, las cuales fueron establecidas por accellera como para convertir el proceso de verificación en una tarea modular que podía ser reutilizada y que aligeraba la complejidad del diseño de entornos de pruebas para cada diseño, estas metodologías se basan en el uso de componentes genéricos que funcionasen de manera transparente unos a los otros, estos con la finalidad de aligerar el trabajo de diseño de pruebas y dar mas tiempo al trabajo de secciones especializadas que se construyen en base a cada diseño para conectar con los bloques genéricos que se pueden reutilizar de proyectos anteriores además de la generación de diferentes pruebas que permitan probar la mayor cantidad de escenarios diferentes usando la misma base; esto permite que se puedan tener diferentes pruebas ejecutándose al mismo tiempo pero realizando operaciones muy distintas sin necesidad de que haya ingeniero configurando las operaciones mientras se ejecutan.

3.2. Secciones que componen el entorno de verificación

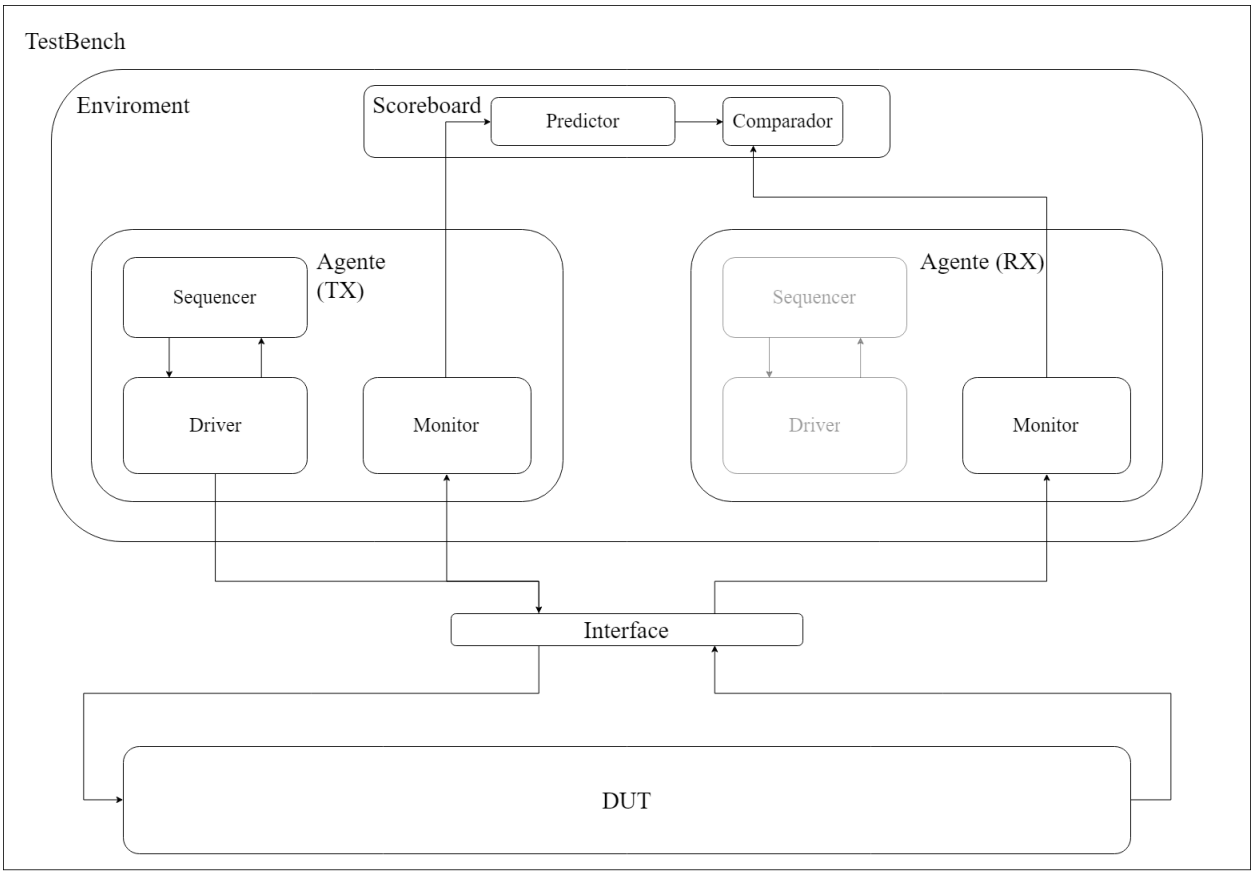


Figura 2: Diagrama de bloques de las principales secciones del entorno de verificación.

3.2.1. Interfaz

La comunicación de datos en la verificación es crítica ya que cualquier error puede generar que las pruebas generen falsos positivos a los test, por lo que hay que asegurar que los puerto se encuentren conectados de manera correcta y que los datos sean transmitidos y procesados sin ninguna alteración por comunicación, aun así cuando aumenta la complejidad de los diseños y la cantidad de puertos incrementa se vuelve más difícil manejar estas conexión y comprobar su funcionamiento; la interfaz solventa este problema.

- Su función es ser una conexión ordenada y a prueba de errores que simplifica el trabajo de conexión, mantenimiento y es reutilizable, todo esto en una sección de alta importancia en el manejo de datos.
- Se encuentra ubicado dentro de el testbench.
- Es conectado a el DUT y el Driver cada uno de estos va a manejar una parte de sus señales.

- Es un bloque personalizado ya que debe adaptarse a cada DUT y su forma depende totalmente de los puertos de este:

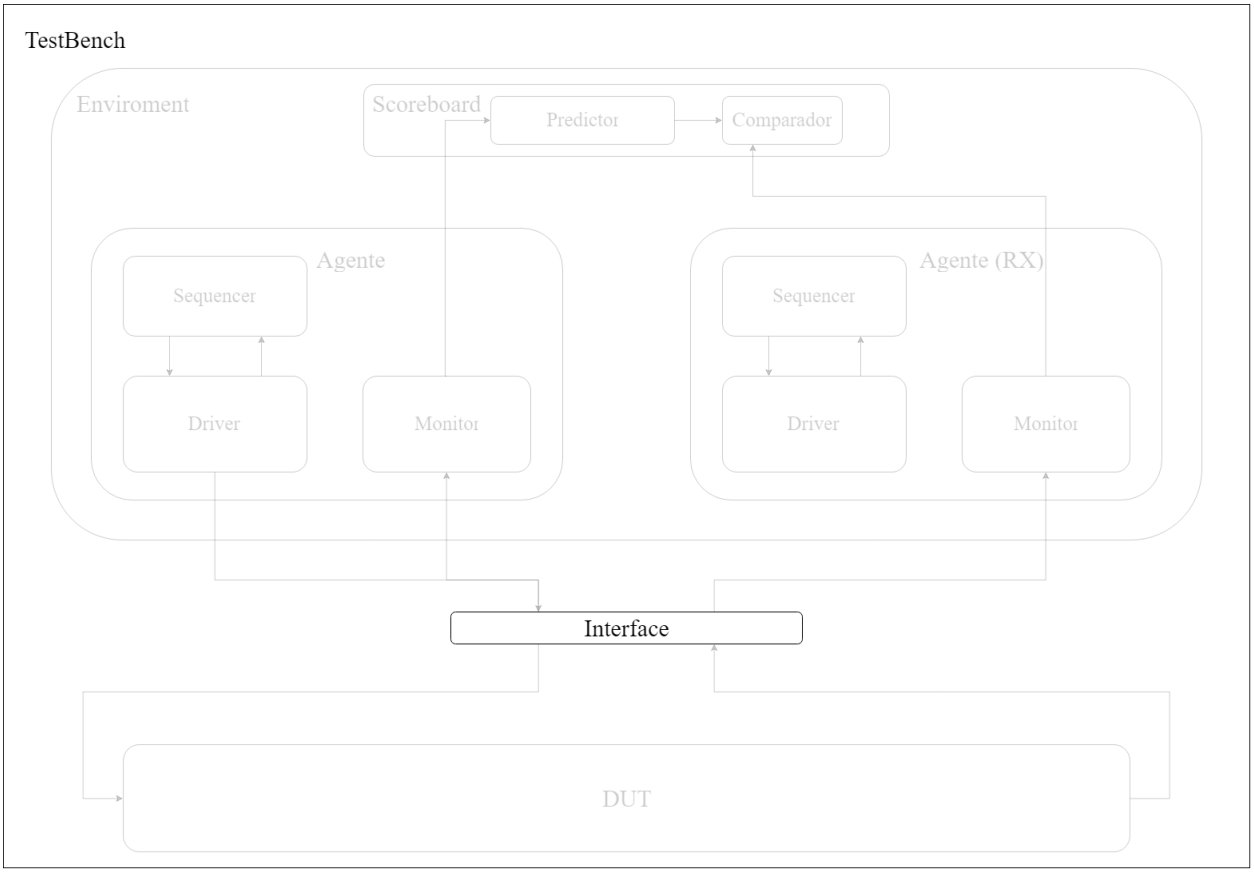


Figura 3: Diagrama de bloques de la interfaz

3.2.2. Sequencer

Bajo esta sección se agrupan los módulos necesarios para la creación y envió de secuencias hacia el driver, se agrupan bajo el nombre de secuencer ya que este es el modulo que se encuentra instanciado dentro del entorno y se comunica con el resto de bloques.

- Su función es generar las secuencias en el formato correspondiente que pueda ser entendido por el driver y entregar los elementos conforme son solicitados.
- Se encuentra ubicado dentro del agente y solo se encuentra activo en el agente de TX.
- Esta conformado por los ítem de secuencia, secuencias y secuenciador.
- No se realizan conexiones en estos módulos sino mas bien estos modulo son conectados en niveles superiores.

- Se divide en las siguientes secciones para poder mejorar su personalización y separar las partes reutilizables de las definiciones y funcionamientos que deben ser personalizados para cada DUT.

Ítem de secuencia

Es un listado de los puertos a usar en la generación de pruebas, en sus valores se definen si estos pueden ser aleatorizados o si van a ser manejados por el DUT o lógica del driver, también se definen los limites de la aleatorización para no generar datos imposibles, asi como cuales de estos valores deben ser reportados en el log y cuales son para manejo de datos interno, no es un modulo funcional sino un objeto de datos.

- Su función es la de ser el equivalente de los datos en la interfaz en un ciclo de reloj, se usa solo dentro del entorno ya que no se puede conectar directamente a la interfaz ya que sus formatos son diferentes.
- Se utiliza principalmente como parte de secuenciador, pero se crean objetos de este tipo en todo el entorno en el manejo de datos.
- No posee otras definiciones o bloques en su estructura.
- No se conectan otros bloques en este.
- Es un bloque completamente personalizado y dependiente del DUT y debe ser cambiado para poder manejar un DUT diferente así como cada objeto de este tipo utilizado, pero donde y como es usado no cambia cuando cambiar el DUT.

Secuencia

En las secuencias se producen los datos para realizar las pruebas, estos datos se producen y envían cada vez que son solicitados por el driver para mantener el orden y funcionamiento controlado.

- Su función es la generación de los datos para realizar las pruebas, se funcionamiento depende da le prueba a realizar ya que pueden ser desde completamente aleatorias hasta completamente controladas.
- Se llaman desde el test seleccionado cuando se inicia la verificación.
- No posee secciones o bloques dentro, pero se llaman secuencias dentro de secuencias para hacer comportamientos más complejos y estructurados.
- En este no se realizan conexiones, su comunicación se realiza por puertos de datos establecidos por el secuenciador.
- Es un bloque personalizado ya que las secuencias van a depender completamente del funcionamiento del DUT por lo que cada DUT usa secuencias diferentes.

Secuenciador

Este bloque conecta el driver que es un bloque de bajo nivel con las secuencias que son de alto nivel, esto lo hace con protocolos y rutinas definidas en la UVM; esto permite que se pueda realizar las comunicaciones entre estos bloques de una manera rápida y confiable.

- Su función es ser una interfaz de comunicación entre el driver y las secuencias.
- Se encuentra ubicado dentro del agente de tipo TX.
- No posee bloques o secciones internas.
- Posee las comunicaciones de los puertos del driver y los puertos de secuencias en las dos direcciones.
- Es un bloque genérico ya que solo posee puertos de comunicación estándar que no depender el tipo de objeto enviado por ellos.

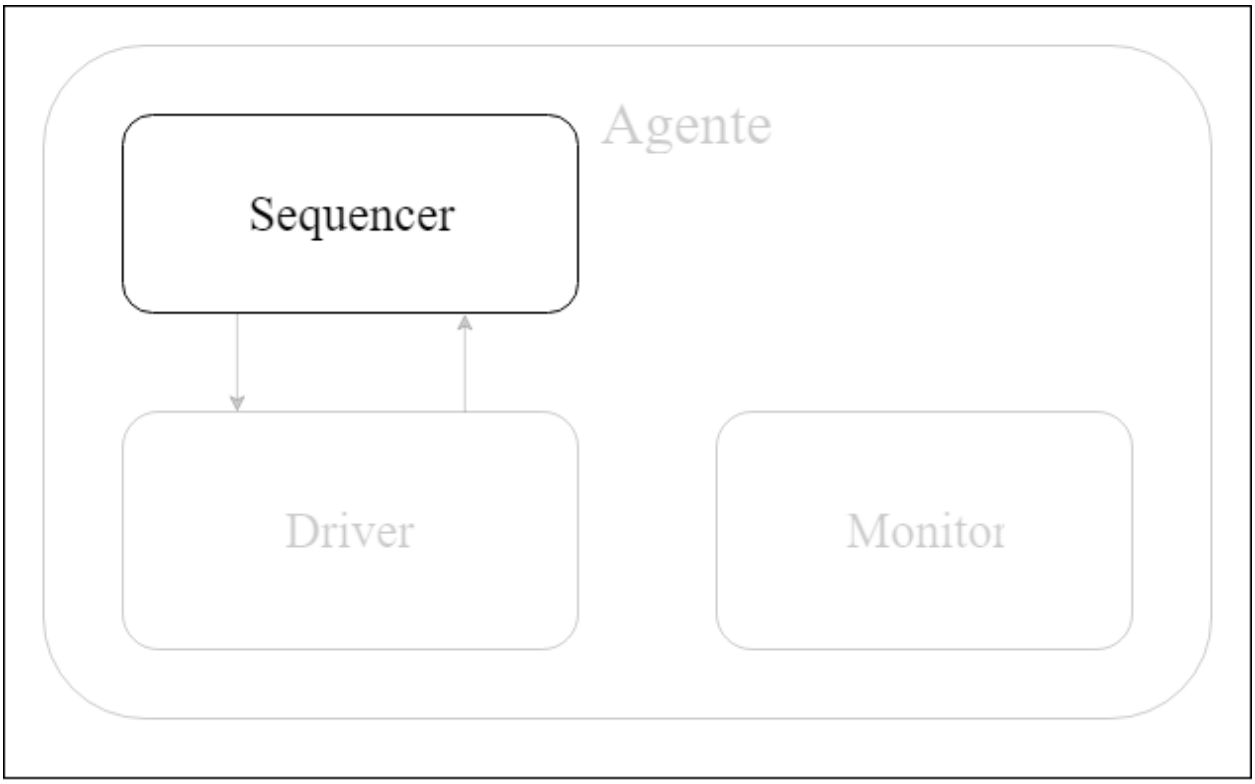


Figura 4: Diagrama de bloques del sequencer

3.2.3. Driver

Este modulo es la comunicación de bajo nivel que controla los datos que entran al DUT y es donde se programan los comportamientos de las unidades que se conectan por medio

de las interfaces a los DUT, para hacer esto se necesita una comunicación constante con las secuencias y verificar los datos en la interfaz para saber cuando debe colocar los datos de las secuencias en los puertos del DUT.

- Su función es manejar los puertos del DUT cada ciclo del reloj, envía los datos de las secuencias de manera correcta y emula el comportamiento de las interfaces del DUT para que este no pueda funcionar de manera normal.
- Se encuentra ubicado dentro de agente de TX
- No posee bloques instanciados, pero se basa en 2 secciones principales, la Run_phase que es la que solicita datos para decidir como se va a comportar y la sección de Drive donde se hace el manejo de puertos justo antes del posedge del reloj para tener los datos a tiempo en el DUT.
- No posee conexiones entre módulos pero si esta conectado a la secuencia por medio del secuenciador para enviar y recibir datos.
- Es un bloque completamente personalizado ya que debe comportarse como si fuera los módulos a los que se conecta el DUT e imitarlos sin ningún error ya que puede afectar el comportamiento real haciendo que se deje de comprobar el funcionamiento correcto del DUT.

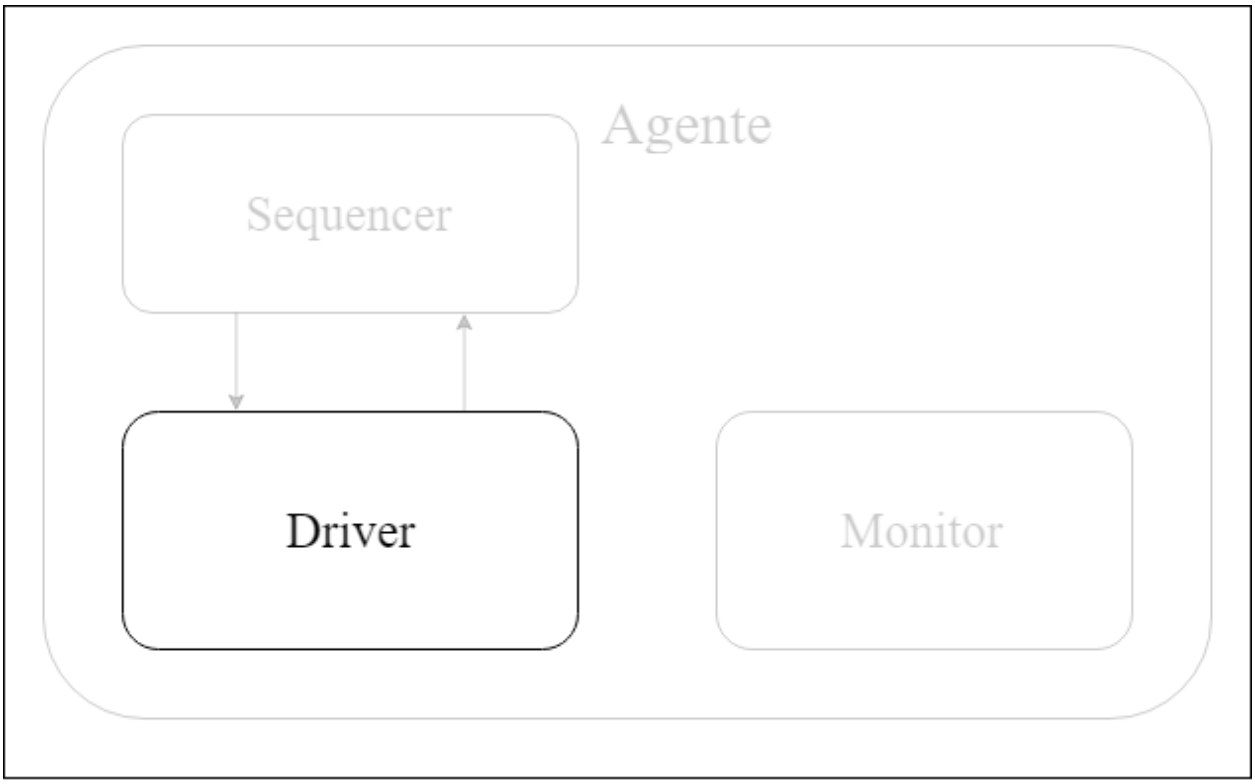


Figura 5: Diagrama de bloques del driver

3.2.4. Monitores

Estos son bloques pasivos que comprueban cada ciclo de reloj los datos en los puertos de la interfaz de conexión con el DUT y con base a la lógica de monitoreo traducen los datos de bajo nivel a formato de alto nivel, conocido como transacción, para ser procesado en el Scoreboard; para tener mayor observabilidad se recomienda configurarlos para realizar él envió de datos todos los ciclos de reloj, pero se puede dar lógica para monitorear solo ciertas operaciones y enviar datos a diferentes puertos.

- Su función es la de observar cada ciclo de reloj los datos que son enviados en la interfaz desde el driver y los que son enviados por el DUT para enviarlos por los respectivos puertos hacia el scoreboard.
- Se encuentra ubicado dentro de los agentes de TX y RX, observando diferentes puertos en cada uno de estos.
- No posee boques instanciados, sus secciones principales son las de monitoreo de puertos y envió de datos.
- No posee conexiones, solo las entradas de los puertos ya que su conexión se realiza por otro modulo.
- Este es un bloque personalizado ya que depende de los puertos que use el DUT y respecto al funcionamiento hace el muestreo de datos y los envía al scoreboard para ser procesados.

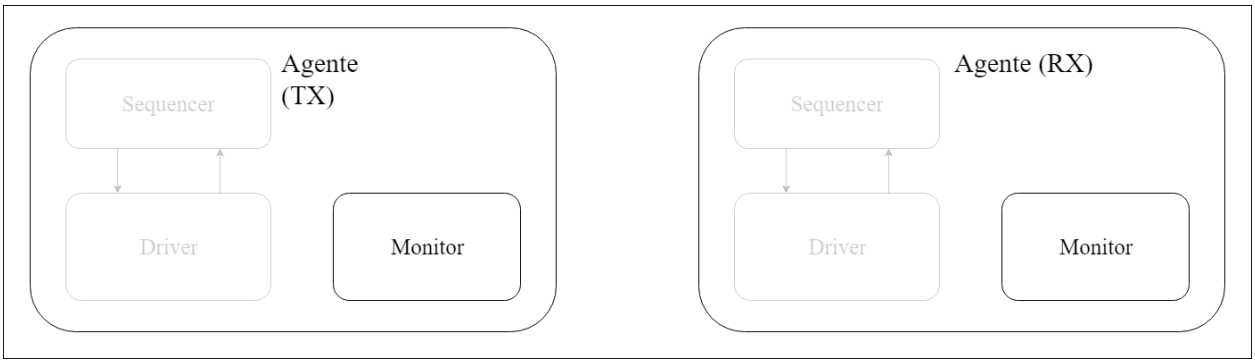


Figura 6: Diagrama de bloques del monitor

3.2.5. Agente

Los agentes son manejadores de datos del DUT son los bloques principales que se conectan a la interfaz con el DUT para manejo de datos de entrada y salida, son los bloques de los que depende hacer funcionar el DUT como si estuviera en operación normal y permiten realizar pruebas de funcionamiento del DUT y las secciones del entorno antes de haber completado todo el scoreboard.

Puede funcionar de manera activa o pasiva activando o desactivando driver y secuencer para solo mantener los monitores en caso de que el DUT no posea entradas o que sea controlado por otro bloque; una característica especial es que se pueden usar múltiples agentes para enviar datos o recibirlos de un DUT, pero lo recomendable es usar la cantidad mínima para poder mantener la observabilidad y controlabilidad de la mayor cantidad de variables.

- Su función es la de agrupar los bloques que manejar los datos del DUT para realizar hacerlo funcionar.
- Se encuentra ubicado dentro del entorno y puede ser de varios tipos por lo que se pueden crear de diferentes tipos y usarlos a la vez.
- Posee la instanciación de los monitores drivers y secuenciadores internamente, además de las configuraciones para activar solo los que sean requerido.
- Posee las conexiones del driver y secuenciador para que puedan comunicarse entre ellos pero estas comunicaciones no son monitoreadas por ningún modulo implementado.
- Es un bloque genérico que forma parte de la estrutura del UVM para el manejo de datos.

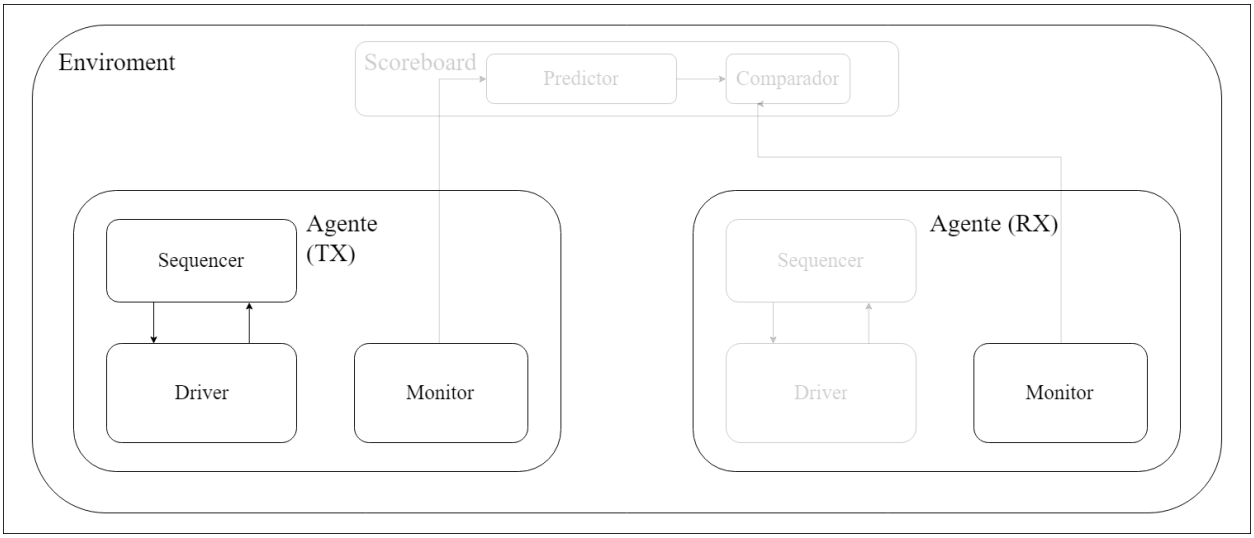


Figura 7: Diagrama de bloques de los componentes en el agente

3.2.6. Predictor

Este bloque se construye en base a la especificación del DUT, pretende emular a la perfección el DUT pero usando herramientas de alto nivel para poder producir los resultados de manera confiables siendo mucho mas fácil de diseñar, sus principal diferencia es el uso de lenguaje de alto nivel para su funcionamiento que hace que las operaciones se realicen completamente independiente de un reloj paro al mismo tiempo cualquier tiempo de espera se debe emular con las invocaciones al predictor desde el scoreboard.

- Su función es generar las mismas salidas que el DUT para cada ciclo de reloj sin ningún error.
- Se encuentra ubicado dentro de el scoreboard ya que en este bloque es que se procesan las salidas del DUT.
- No posee bloques internos, pero pueden ser creados y llamados si se desea.
- No posee conexiones, sus entradas y salidas son datos que el scoreboard maneja.
- Es un bloque personalizado debe comportarse de manera idéntica el DUT y cambiar cuando el DUT cambia.

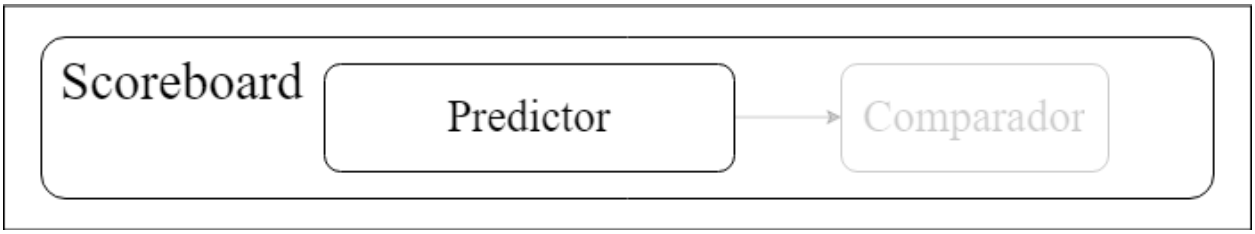


Figura 8: Diagrama de bloques del predictor

3.2.7. Comparador

El comparador es completamente genérico para poder ser reutilizado, pero se puede utilizar custom para tener un reporte configurado para las necesidades de la verificación o para realizar comprobaciones de un funcionamiento muy específico.

- Su función es compara las salidas del DUT y predictor para encontrar errores.
- Se encuentra ubicado dentro del scoreboard al igual que el predictor.
- No posee mas bloques ya que es de funcionamiento simple.
- NO posee conexiones, trabaja los datos que sean provistos por el scoreboard.
- Es un bloque genérico ya que compara datos sin importar la forma de estos ya que deben ser del mismo tipo para ser comparados y deben ser idénticos para que la comprobación resulte en correcta.

Figura 9: Diagrama de bloques del comparador

3.2.8. Scoreboard

Es un bloque que maneja secciones al igual que los agentes trabaja en alto nivel y conecta a predictor con comparador además de entregar los datos que requieren estos, los datos se obtiene de un puerto desde el enviroment, desde los puertos los datos son colocados en FIFOs para ser extraídos en orden para entregarlos al predictor y comparador para procesarlos y obtener los resultados de la verificación.

- Su función es la de manejar los datos que son generados entregarlos al predictor comparador y realizar los reportes.
- Se encuentra ubicado dentro del entorno como uno de los bloques principales, a diferencia del driver trabaja datos en alto nivel.
- Posee la instanciación de el predictor y el comparador además de los puertos y los ítems de datos para manejar los datos conforme se van generando.
- Posee las conexiones de los puertos de datos conectados en el enviroment a los ítems para el trabajo de datos en cada ciclo por el predictor y comparador.
- Es un bloque genérico en funcionalidad ya que todos los bloques que usa son siempre los mismos, pero debe ser configurado para que realice las conexiones y los diferentes modos que hayan sido implementadas o establecidas estas conexiones.

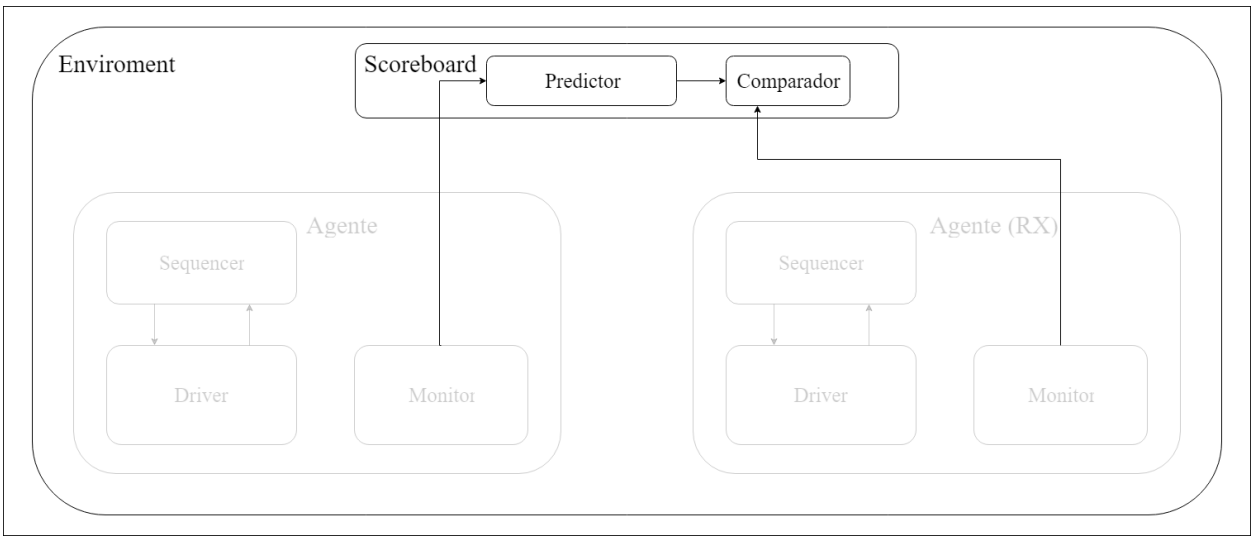


Figura 10: Diagrama de bloques de los componentes en el Scoreboard

Puertos de datos

Son parte del scoreboard pero se conectan usando al enviroment, son estructuras del mismo funcionamiento que el sequencer, pero son más simples ya que solo envían datos desde monitores a scoreboard por lo que no son un bloque separado sino un puerto de datos genérico que el fin es ser reutilizado.

3.2.9. Enviroment o entorno

Este es un bloque genérico usado para la instanciación de las secciones de manejo de bajo nivel que son los agentes y el manejo del los datos a alto nivel del scoreboard además del las conexiones de los puertos de datos a los monitores y a el scoreboard para que los datos del DUT puedan ser enviados a ser procesados.

- Su función es la contener y conectar las secciones con lógica o genéricas que permiten realizar la verificación del DUT.
- Se encuentra ubicado dentro del testbench como la sección del entorno de verificación donde esta contenida toda la lógica y estructura definida por la UVM.
- Posee la instanciación del socreboard, los agentes y los puertos por los que se comunican estas secciones.
- Posee las conexiones del los puertos de datos a monitores y de los puertos de datos a scoreboard, esto se hace de una sola conexión ya que no hay manejo de datos en el enviroment.
- Es un bloque genérico ya que solo hace llamadas a bloques definidos y los conecta, contiene a los bloques principales para manejo de datos y verificación permitiendo que todo este estructurado para ser reutilizado.

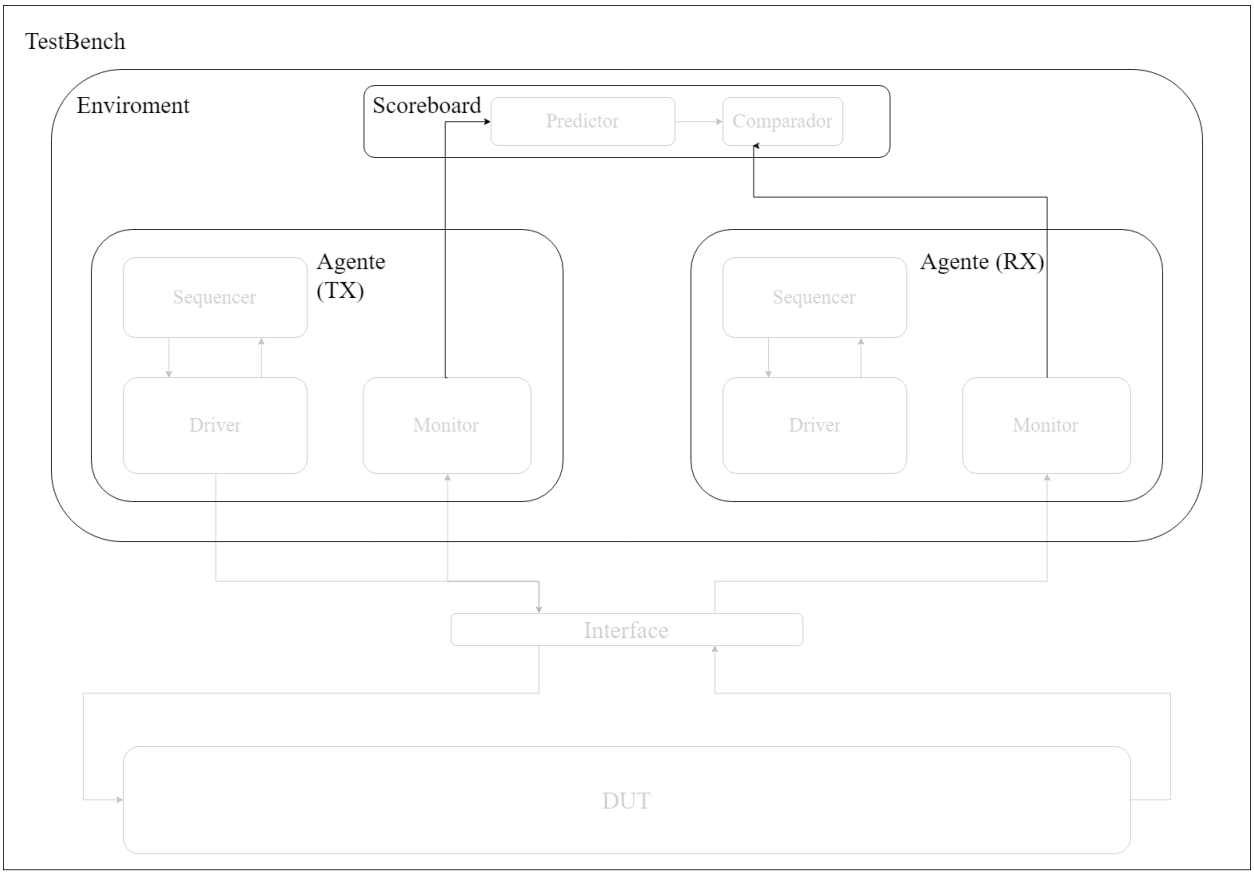


Figura 11: Diagrama de bloques de los componentes en el Enviroment

3.2.10. TestBench

El bloque principal para el funcionamiento de la verificación, contiene y conecta las secciones principales que son el DUT que es el bloque a verificar y el entorno de verificación, además de la interfaz para realizar las conexiones de estos; los reloj de bajo nivel se realizan aquí y se llaman las pruebas a realizar, es el bloque principal que es donde se inicia toda la simulación.

- Su función es la de llamar y conectar el bloque a verificar con el entorno de verificación al inicio de la simulación.
- Es el bloque principal llamado en la simulación.
- Posee la instanciación de enviroment interfaz y DUT, aparte de esto todas las conexiones y cualquier señal básica para funcionamiento como son los reloj.
- Posee las conexiones de la interfaz con drivers y la interfaz al DUT y los reloj a los mismos.

- Es un bloque genérico en su funcionamiento pero posee secciones que deben ser configuradas respecto al DUT y enviroment para que funcionen en base a las funcionalidades dadas por cada equipo de trabajo.

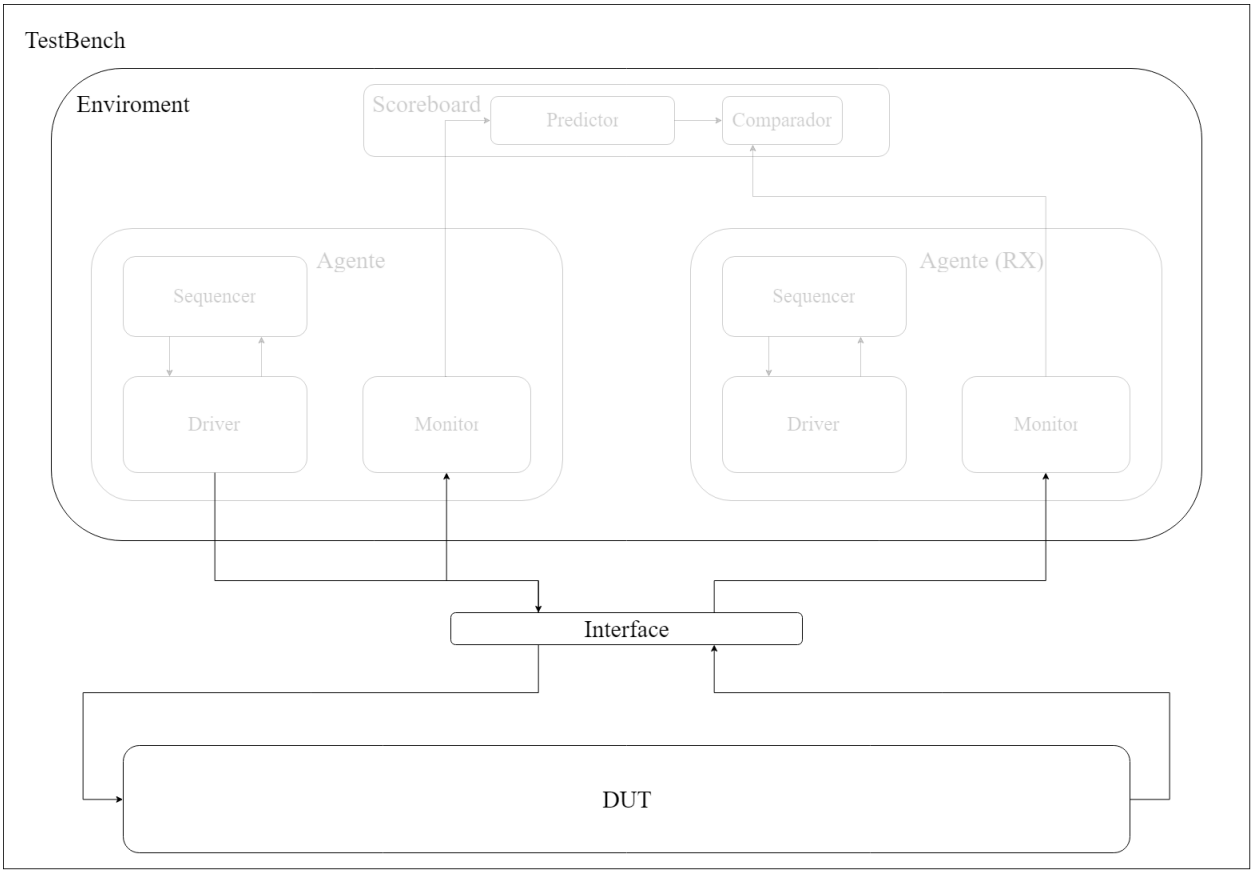


Figura 12: Diagrama de bloques de los componentes en la TestBench

3.3. Diseño del modelo de referencia

El modelo de referencia posee un comportamiento muy sencillo solo debe generar las salidas respecto a las entradas y colocarlas en las salidas en el momento justo, para hacer esto se divide en 3 secciones diferentes, la generación de datos, almacenamiento y los controladores de salidas; estos funcionan en conjunto para emular el comportamiento del DUT y realizar las funciones del predictor correctamente.

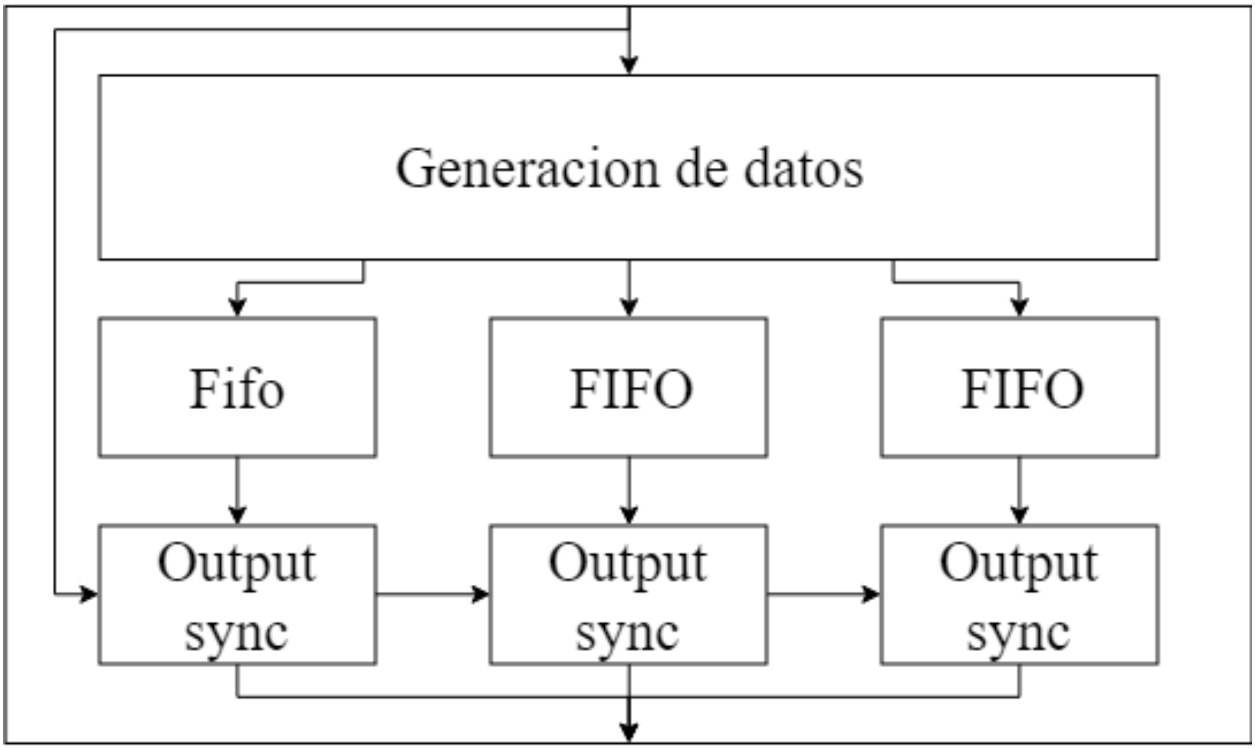


Figura 13: Diagrama de bloques de los componentes en el modelo de referencia

3.4. El coverage

El coverage es una métrica que se usa para definir los alcances de verificación logradas por una prueba en el diseño, sirve para obtener los caminos y datos utilizados en la prueba y mas importante los que no fueron utilizados lo que hace que se pueda saber si los test están logrando probas las secciones que se desean probar y las que no esta logrando probar.

Aunque los hay de diferentes tipos para este proyecto solo se van a implementar los siguientes

- Functional coverge: es el coverage implementado en los puertos escogidos por el verificador, se basa en comprobar cuales de todos los posibles valores que puede tener un dato han sido utilizados en las operaciones, aunque sirve para observar generalmente la respuesta que tiene el DUT dante determinados datos y la distribución de los resultados que se pueden lograr si se ejecutan los test con diferentes semillas o limitando los rangos de datos, no se puede confiar ciegamente en que un sistema al esta completamente verificado al lograr que el functional coverage llegue al 100 % ya que los sistemas combinacionales por definición presentan respuestas diferentes a diferentes secuencias de datos y eso es algo que el functional coverage por si solo no es capaz de cubrir.
- Code coverage: esta forma de coverage se realiza con la herramienta de simulación

se basa en el análisis del código durante la ejecución de los test para obtener los datos acerca de los efectos de las pruebas en las diferentes operaciones descritas por el lenguaje de descripción de hardware, estos fueron los utilizados en la ejecución de pruebas cabe destacar que no todos son validos para todos los bloques ya que algunos compruebas operaciones especificas que no se presentan en todos los bloques del DUT:

- Line: a como su nombre lo indica el realiza la verificación de cobertura de las lineas de código, lo que permite definir cuales secciones fueron probadas y cuales no, esto permite ver cuales son las secciones que se están ignorando y generar nuevas pruebas diseñadas para comprobar estos casos; esta versión de coverage por si sola posee la misma deficiencia del functional coverage ya que no puede diferenciar comparaciones que sean aceptadas o negadas y el camino por el cual fueron ejecutadas las lineas de código por lo que pierde la manera de observar las operaciones de tipo secuencial que hacen llamadas a mismo código por diferentes razones con diferentes datos, estas debilidades se mitigan con los siguientes tipos de code coverage.
- Cond: este coverage es de tipo condicional, trabaja exclusivamente en operaciones que posean múltiples condiciones para ser aceptadas y que generen un resultado cuando una o varias de estas condiciones se concreten, la manera más simple de ejemplificar es en resultados de operaciones .ºR"donde 3 de las 4 posibles combinaciones generan la misma salida, este tipo de coverage ayuda a comprobar que se ejecutados todas las combinaciones posibles o a notificar cuales son las combinaciones faltantes.
- Toggle: igual que el coverage condicional este también es especializado en una operación muy especifica, este a diferencia se basa en datos de puertos o variables dentro de los bloques y compruebas si estos varían, lo hace por medio de comprobaciones de bit al comprobar si pasa de cero a uno y de uno a cero; para análisis en general es de utilidad pero su utilidad baja si se desea saber variaciones de múltiples bits en conjunto, este análisis de hace de manera diferente.
- FSM: este coverage también se especializa en una operación definida que en este caso son las maquina de estados finitos, lo que reporta son los estados a los cuales entro a ejecutar durante las pruebas y también los cambios de estados posibles que sucedieron de manera que se puede saber en general cuales caminos de la maquina de estados se están recorriendo con determinadas pruebas.
- Branch: este coverage se ejecuta para casos en los que haya comprobaciones de datos dentro de comprobaciones de datos en los que ambas comprobaciones tengas múltiples soluciones, esto genera que para llegar a cada uno de los posibles resultados haya múltiples comprobaciones diferentes que solo se recorren una vez, cada uno de estos caminos es una ruta que requiere condiciones definidas para llegar a la salidas y este coverage verifica esas condiciones y cuales han sido los caminos de todos los posibles que ha sido ejecutados por los test.

Pruebas recomendada a implementar

Las pruebas recomendadas a implementar son las siguientes:

Pruebas básicas

- Prueba de inicio
- Prueba de Reinicio
- Prueba de escritura y lectura de Memoria
- Prueba de errores
- Prueba de escritura y lectura de IO
- Prueba de cuenta de CSR
- Prueba de watchdog

Pruebas complejas

- Prueba de errores en boot.
- Pruebas con diferentes longitudes de boot.
- Prueba de escritura Reinicio y lectura.
- Prueba de saturación de datos.
- Pruebas de generación de interrupciones.
- Pruebas de tiempos de espera
- Pruebas de escritura en cuentas de CSR.

Pruebas aleatorias

- Escritura y lectura aleatorios en memoria
- Escritura y lectura aleatorios en IO
- Reinicio aleatorio en el funcionamiento
- Generación de interrupciones

Pruebas para coverage

- Escritura y lectura de bloques de memoria secuencialmente.
- Escritura y lectura de bloques de memoria aleatoriamente
- Escritura y lectura de IO.
- Generación de interrupciones.

Capítulo 4

Marco metodológico

4.1. Implementación de entorno de verificación

4.1.1. Interfaz

La implementación de la interfaz no posee ninguna lógica, sino mas bien es un listado de los puertos a utilizar y depende completamente del DUT.

4.1.2. Sequencer

Para ejecutar el funcionamiento del sequencer no se utilizó a este como un bloque en sí, sino mas bien sus tareas son realizadas por las partes que lo conforman en módulos separados,en conjunto realizan su funcionamiento de la manera deseada.

Ítem de secuencia

Este al igual que la interfaz no posee ninguna lógica por si mismo, posee el listado de puertos a usar en la creación de las secuencias; este es el modulo donde se decide los valores que pueden tener un valor aleatorio para las pruebas y los que van a ser controlados por la secuencia; para facilitar las pruebas también posee los puertos de salida del DUT y puertos extra añadidos para comprobación.

Secuencia

La implementación de las secuencias varia en base el funcionamiento deseado, normalmente se usa una versión simple como la mostrada en la figura14a, pero a causa del funcionamiento del DUT se requirió preparar operaciones reactivas en las secuencias esto se hace al habilitar un camino de realimentación desde el driver hacia las secuencias como se muestra en la figura 14b, esto permite crear ciclos de espera o el envío de datos requeridos a secciones de puertos independientes para emular el comportamiento requerido por el DUT.

Secuenciador

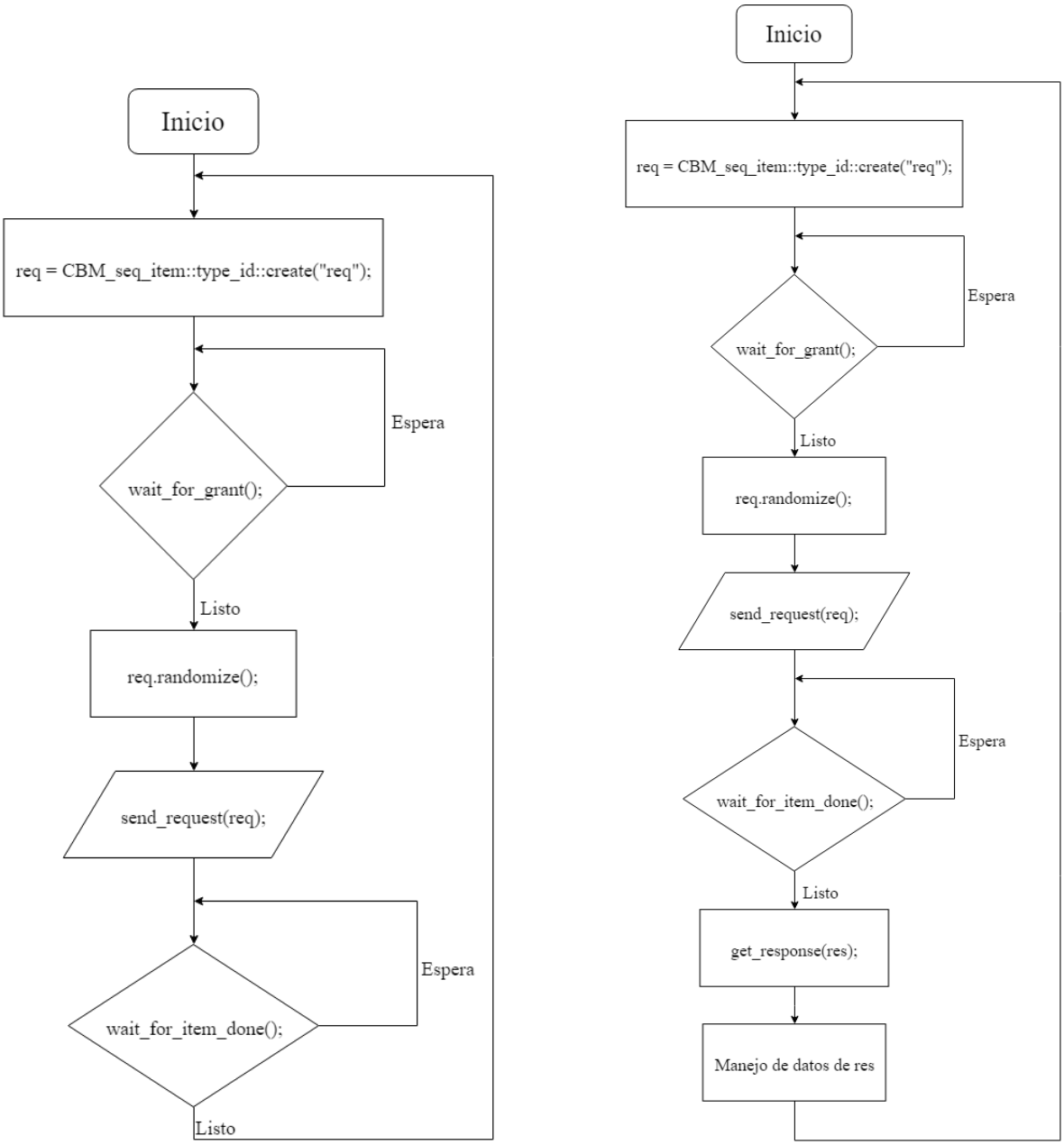
Este bloque es genérico y solo se debe de conectar en el agente con el driver para poder funcionar, la conexion a la secuencias a usar se hace en el test que es el que define las pruebas a ejecutar en el inicio del funcionameinto de la simulación.

Diagrama de funcionamiento

El funcionamiento del sequencer es conjunto con el del driver por lo que muchos de sus pasos se basan en comunicación con el driver o decisiones basadas en los datos recibidos desde el driver por medio del puerto de respuesta; la comunicación se hace por el secuenciador de manera automática como parte de la UVM y la lógica es dada por las Secuencias, las secuencias usadas en este proyecto son del tipo de la figura14b ya que el sistema es reactivo a ciertas solicitudes de datos o posee esperas con tiempos no especificado y esta es la secuencia que permite esos comportamientos de la mejor manera.

4.1.3. Driver

Al igual que secuanciador su comportamiento depende de este para continuar, cada salida de datos y ciclo en el driver y secuenciador corresponde a un ciclo y salida de datos en el otro modulo, el funcionamiento de drive esta dado por la especificación y se basa en reenviar los datos de vuelta para que el sequencer los copie en la secuencia para mandarlos de nuevo hasta que se complete la operación.



(a) Diagrama de flujo del funcionamiento del modelo de referencia

(b) Diagrama de flujo del funcionamiento del modelo de referencia

Figura 14: Diagramas de flujo de secuencias

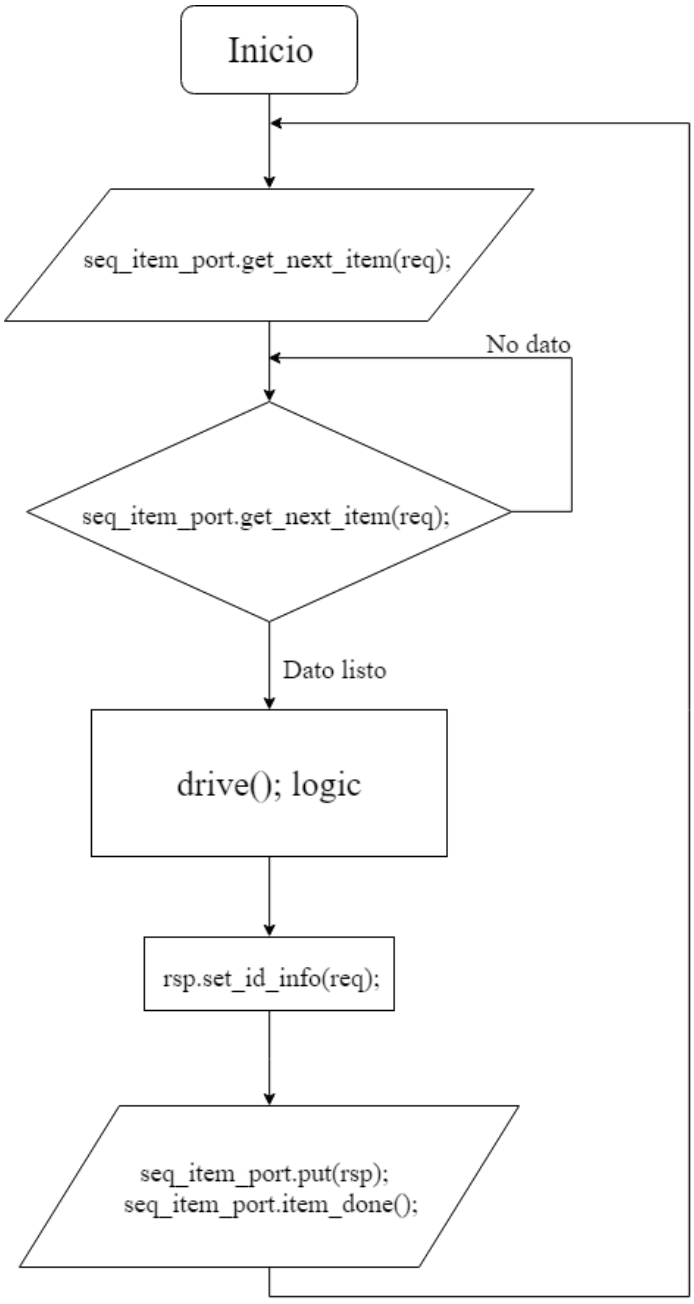


Figura 15: Diagrama de flujo del funcionamiento del driver

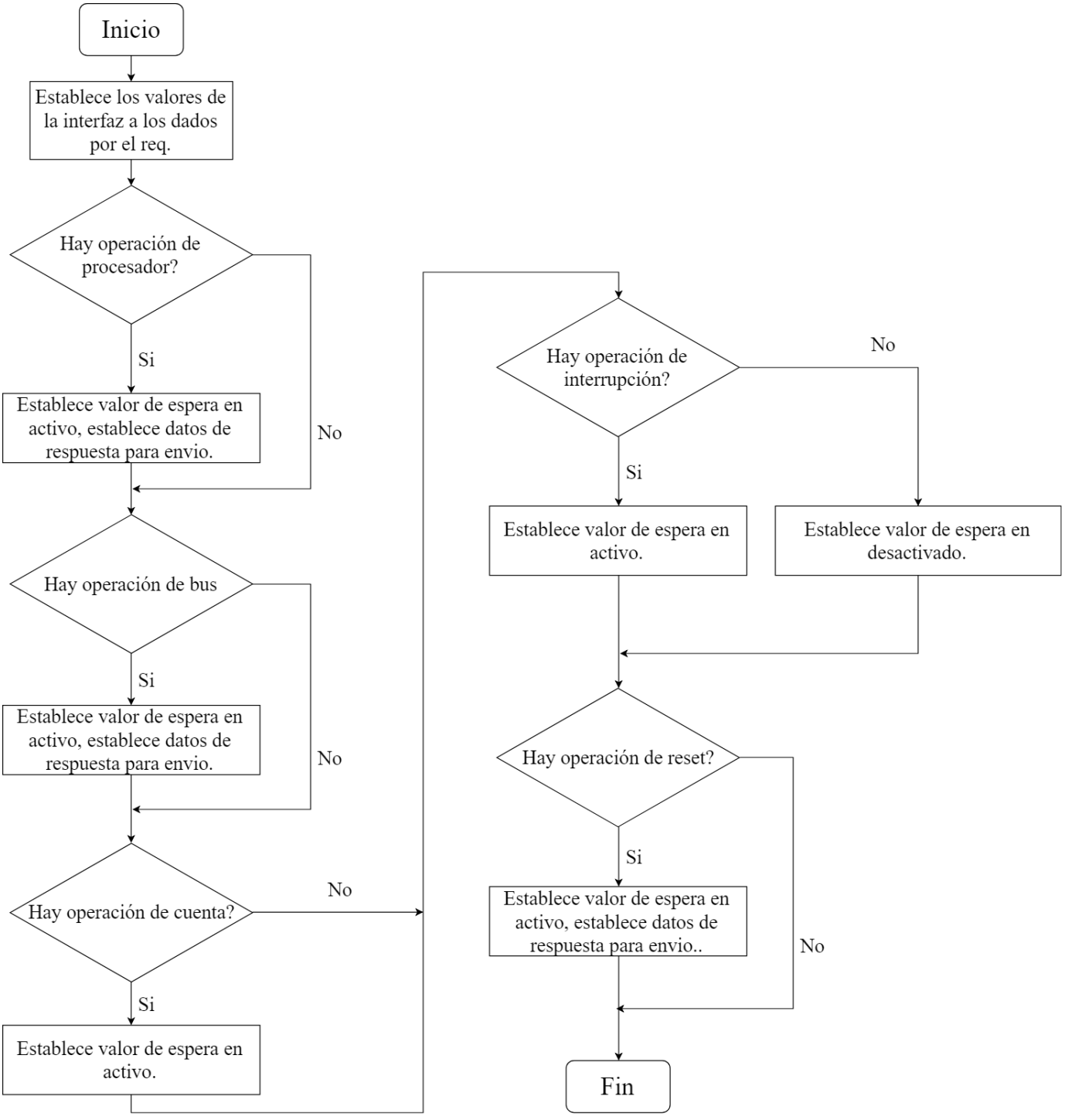


Figura 16: Diagrama de flujo de la lógica de drive

4.1.4. Monitores

Estos no posee lógica en su funcionamiento mas que revisar cual es su tipo y enviar los datos todos los ciclos de reloj.

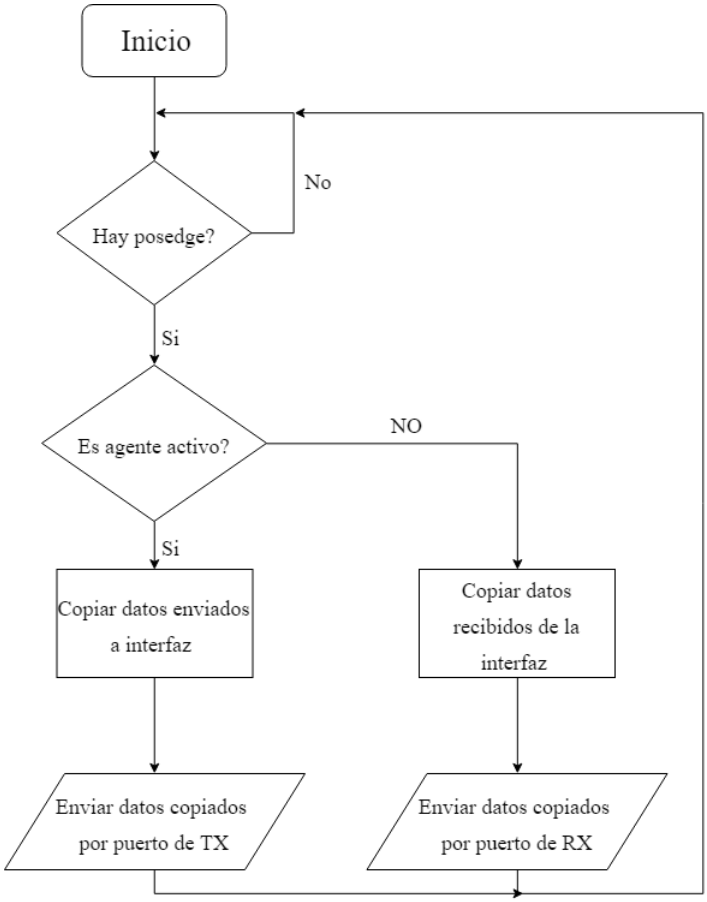


Figura 17: Diagrama de flujo del funcionamiento de los monitores

4.1.5. Agente

Los agentes funcionan como bloque genérico para conexión de bloques internos, no posee ninguna lógica en su funcionamiento, su implementación se basa en seguir el formato de cualquier otro agente ya que la mayoría deben ser iguales para poder ser reutilizables.

4.1.6. Predictor

El predictor depende del diseño del modelo de referencia, para este proyecto el modelo de referencia no necesito ninguna operación especial de datos por lo que son el mismo bloque y no se diferencian uno del otro.

4.1.7. Comparador

Aunque hay múltiples maneras de implementar comparadores para este proyecto por las dificultades para definir los tiempos de datos, se utilizo las señales del DUT para realizar la comparación solo cuando este genere datos y la comparación se hacer con una función genérica que forma parte de la UVM.

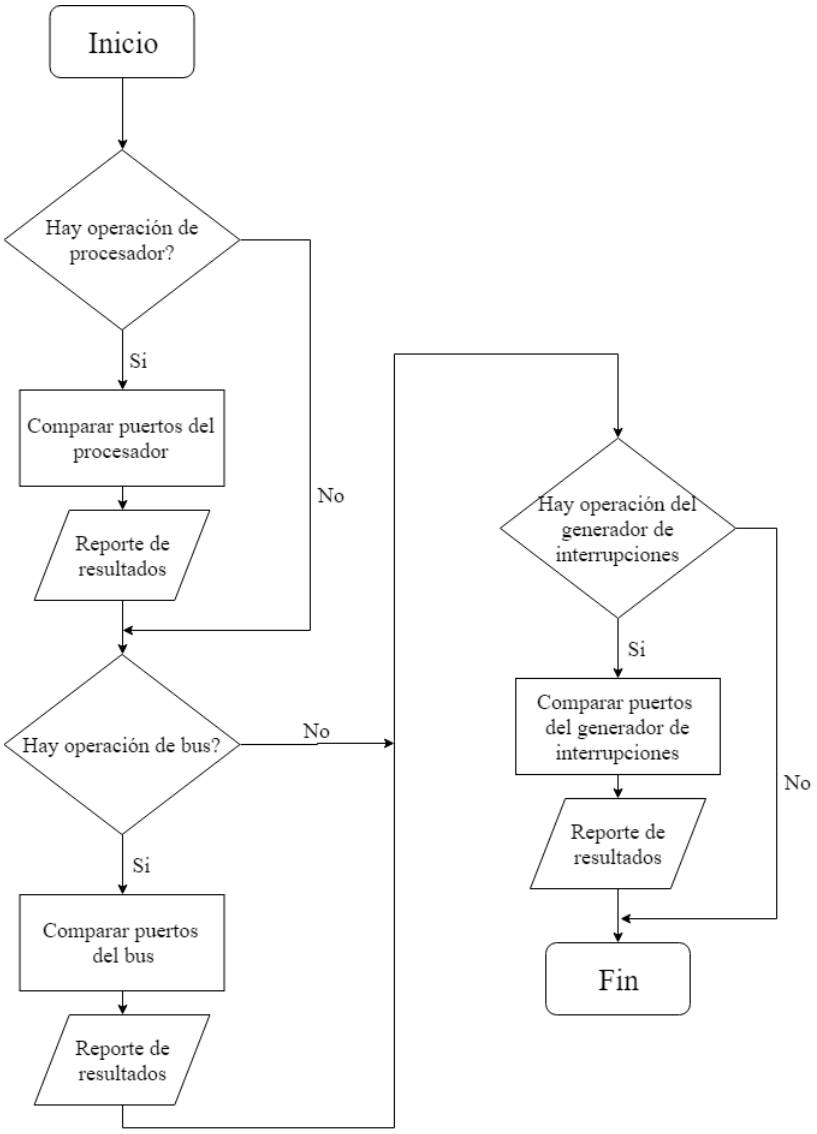


Figura 18: Diagrama de flujo del funcionamiento del comparador

4.1.8. Scoreboard

El scoreboard posee también una estructura muy genérica para poder ser reutilizado y las operaciones lógicas para su implementación son mínimas.

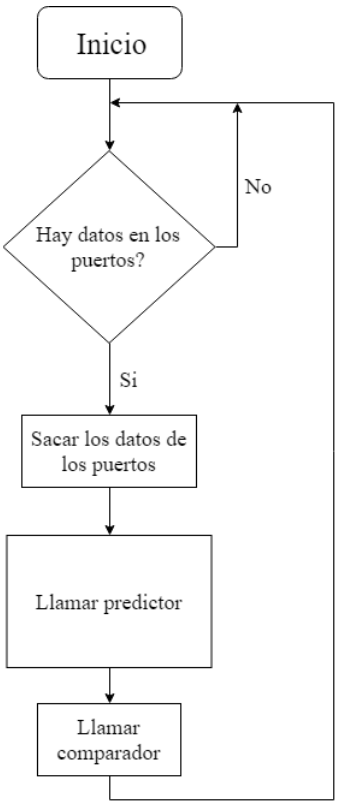


Figura 19: Diagrama de flujo del funcionamiento del scoreboard

4.1.9. Enviroment

Su implemtación es completamente genérica y no posee lógica asociada.

4.1.10. TestBench

Su implemtación es completamente genérica y no posee lógica asociada.

4.2. Implementación del modelo de referencia

El modelo de referencia es la sección mas compleja del diseño del entorno de verifiación este debe poseer el mismo comportamiento y generar las mimas salidas para cada entrada, pero se escribe en lenguaje de alto nivel para simplificar su diseño y funcionamiento, así como para asegurar que no posee errores; con mayores conocimientos y experiencia se puede definir la forma mas eficiente de realizar el modelo de referencia respecto a los funcionamientos del DUT.

Para este diseño se escogió basarse en el funcionamiento para generar las salidas de antemano, almacenarlas y luego entregarlas para comparación cuando el DUT genera sus salidas,

esto hace que el diseño se divida en secciones funcionales como se observa en la figura 13, esto genera problemas si se requiere realizar operaciones respecto a entradas anteriores, entradas que se mantienen o generar una salida repetidas veces sin tener desde el inicio definido cuantas veces va a suceder esto.
Cabe destacar que las secciones de generación de datos y de salidas sincronizadas son independientes una de la otra y siempre se van a ejecutar sin importar lo que realice la otra sección, de manera que cada diagrama de flujo que tenga un Ïnicio"se va a ejecutar cada vez que sea llamada una operación del modelo de referencia y para que inicie cualquier otro deberá ser llamado por uno de los que posee Ïnicio.º uno que sea llamado por uno de estos.

4.2.1. Diagramas de funcionamiento

Generación de datos

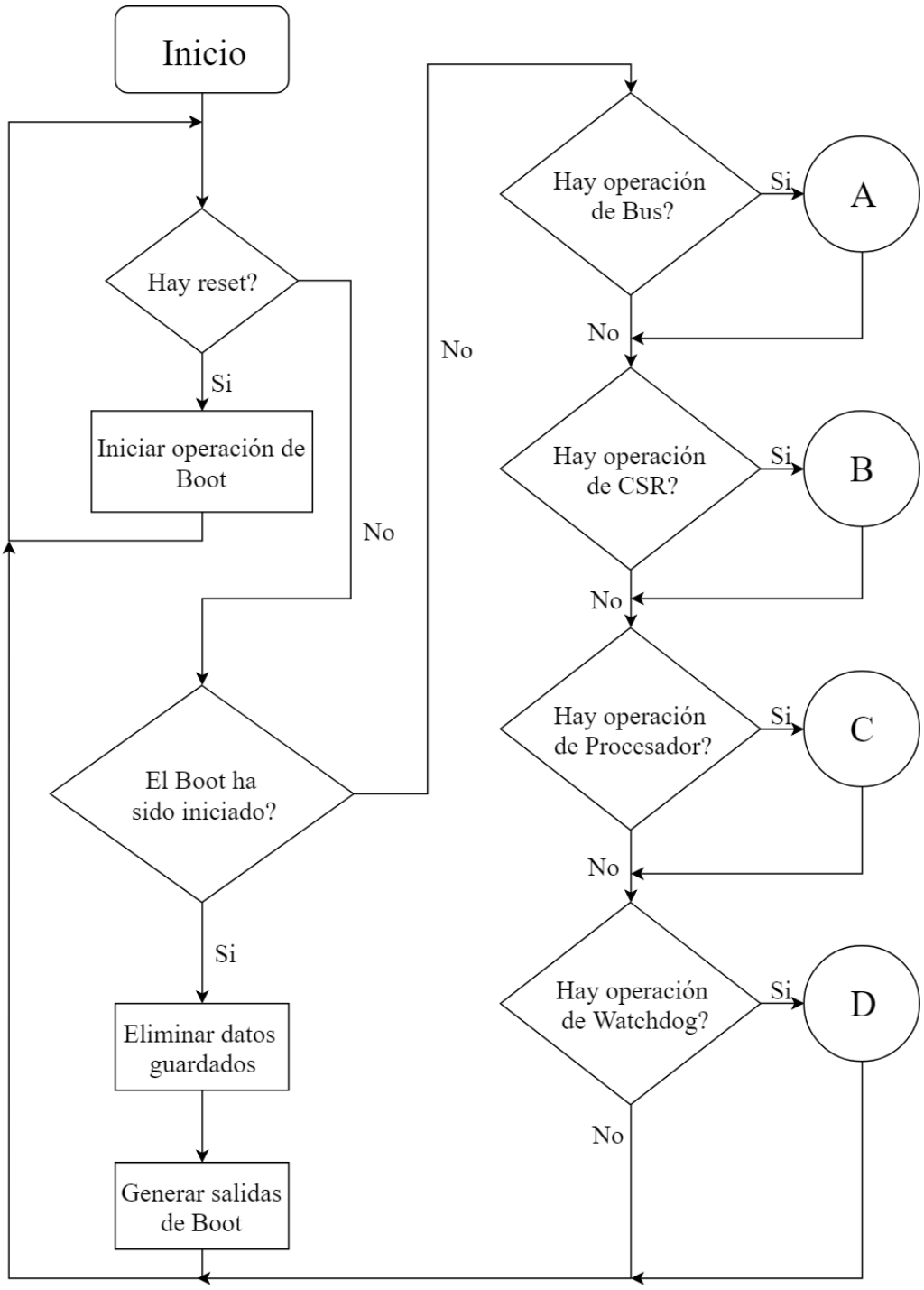


Figura 20: Diagrama de flujo del funcionamiento del la sección principal del generador de datos modelo de referencia.

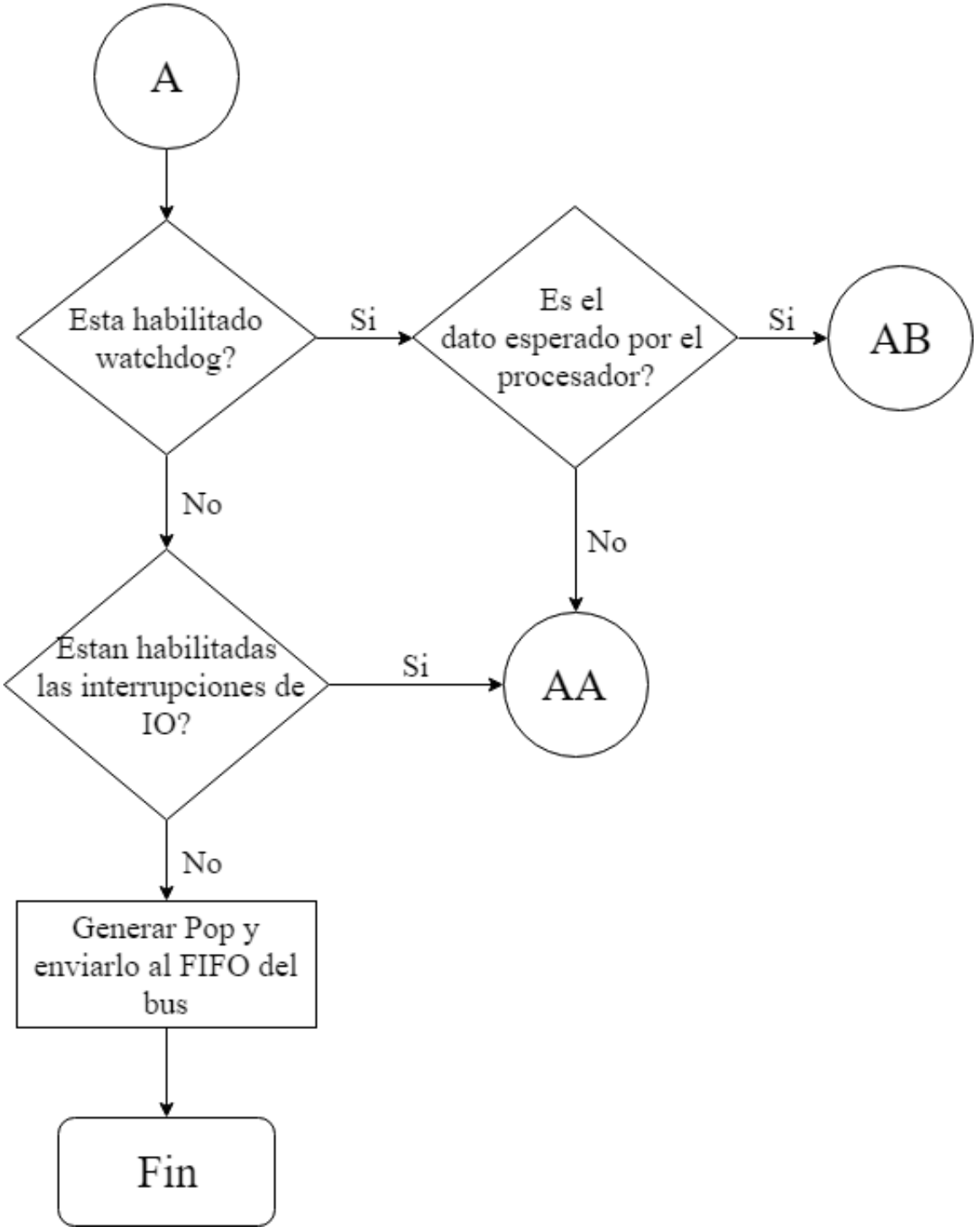


Figura 21: Diagrama de flujo del funcionamiento del las operaciones de bus.

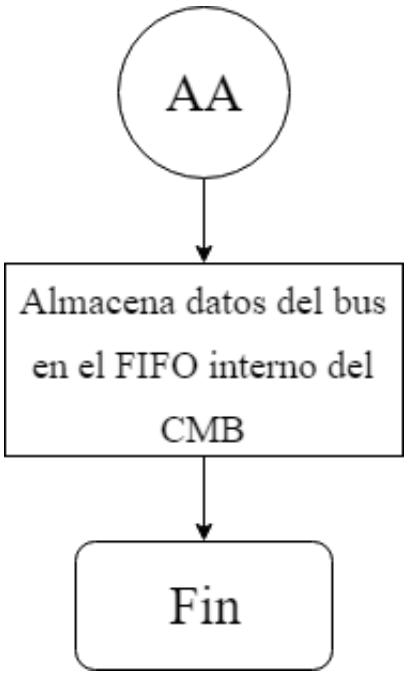


Figura 22: Diagrama de flujo del almacenamiento de datos de bus a FIFO interno.

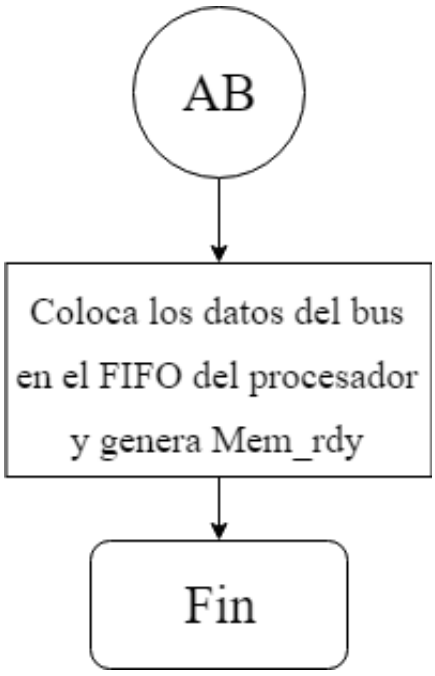


Figura 23: Diagrama de flujo del paso de datos de bus a procesador.

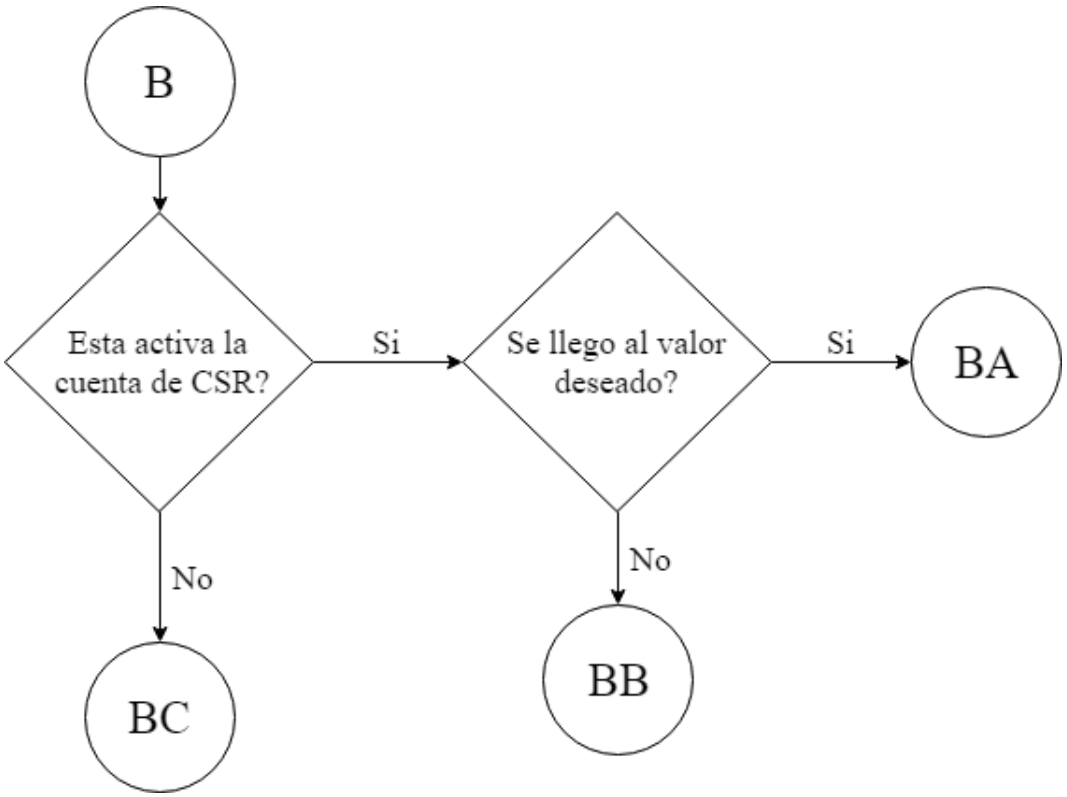


Figura 24: Diagrama de flujo del funcionamiento de los CSR de tiempo.

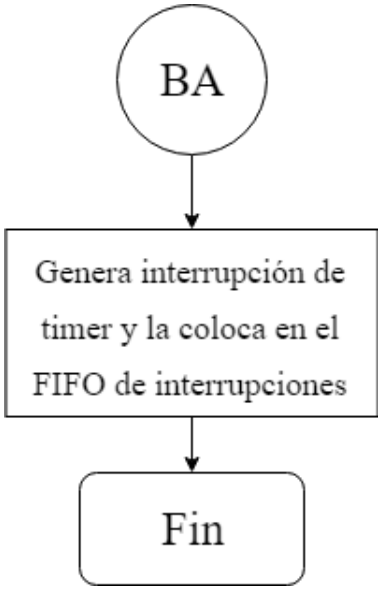


Figura 25: Diagrama de flujo la generación de interrupción por cuenta

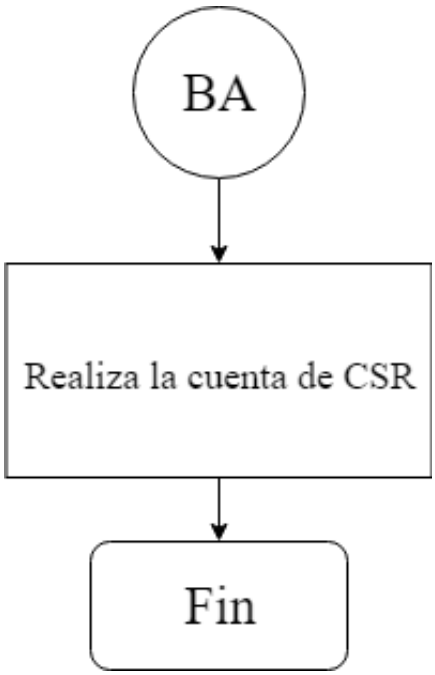


Figura 26: Diagrama de flujo de la cuenta de CSR

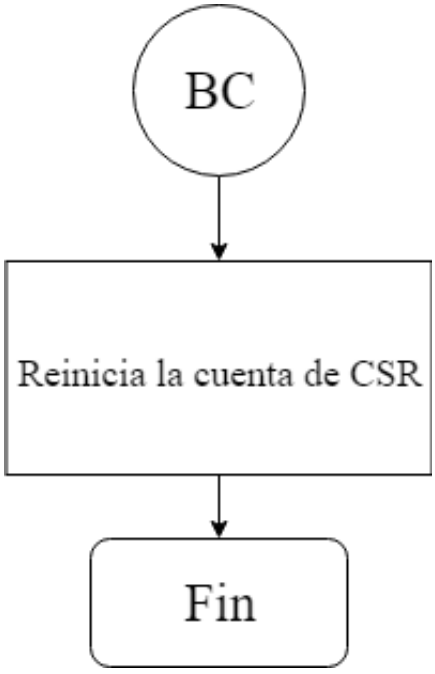


Figura 27: Diagrama de flujo el borrador de CSR

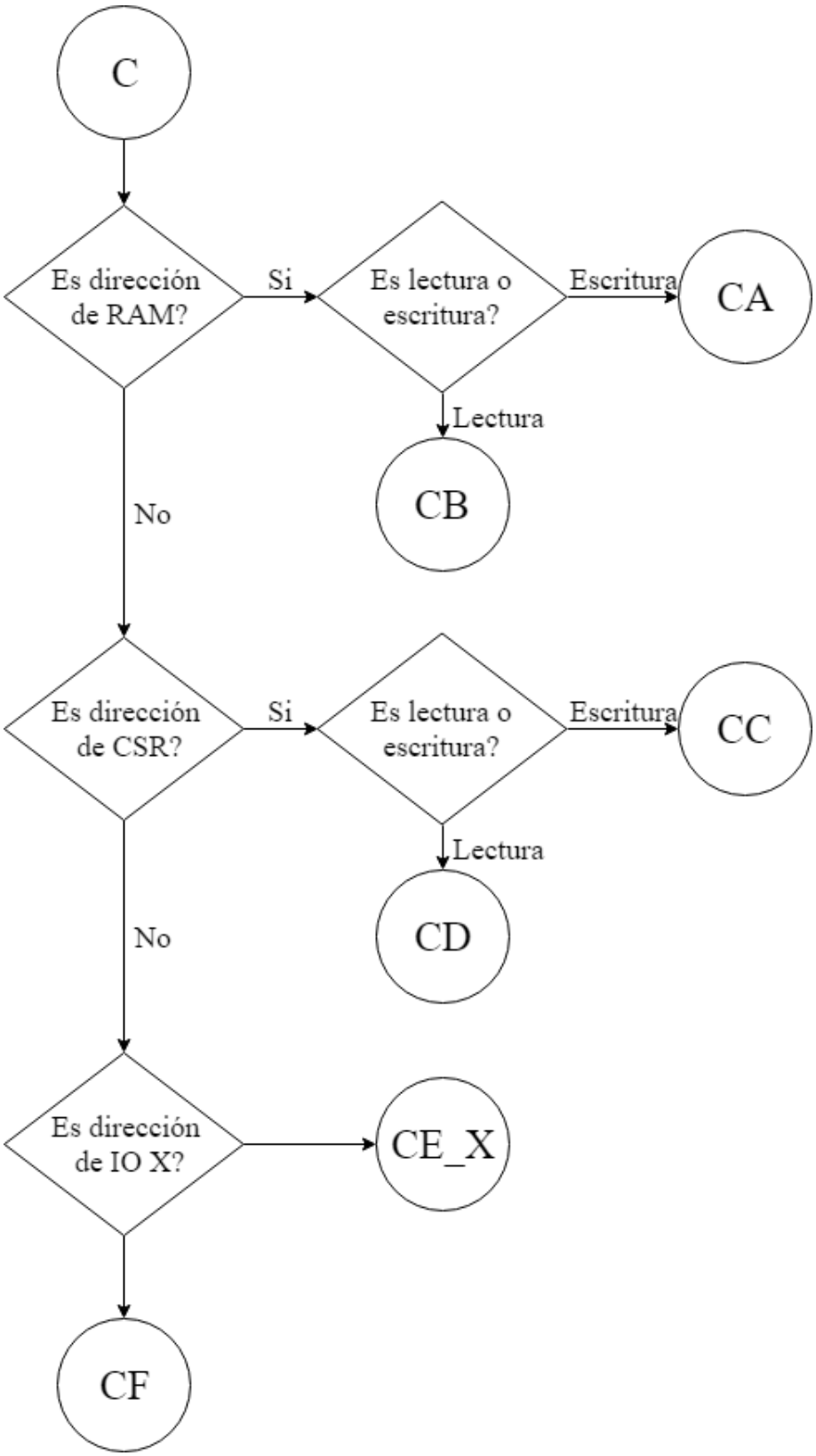


Figura 28: Diagrama de flujo de las operaciones del procesador

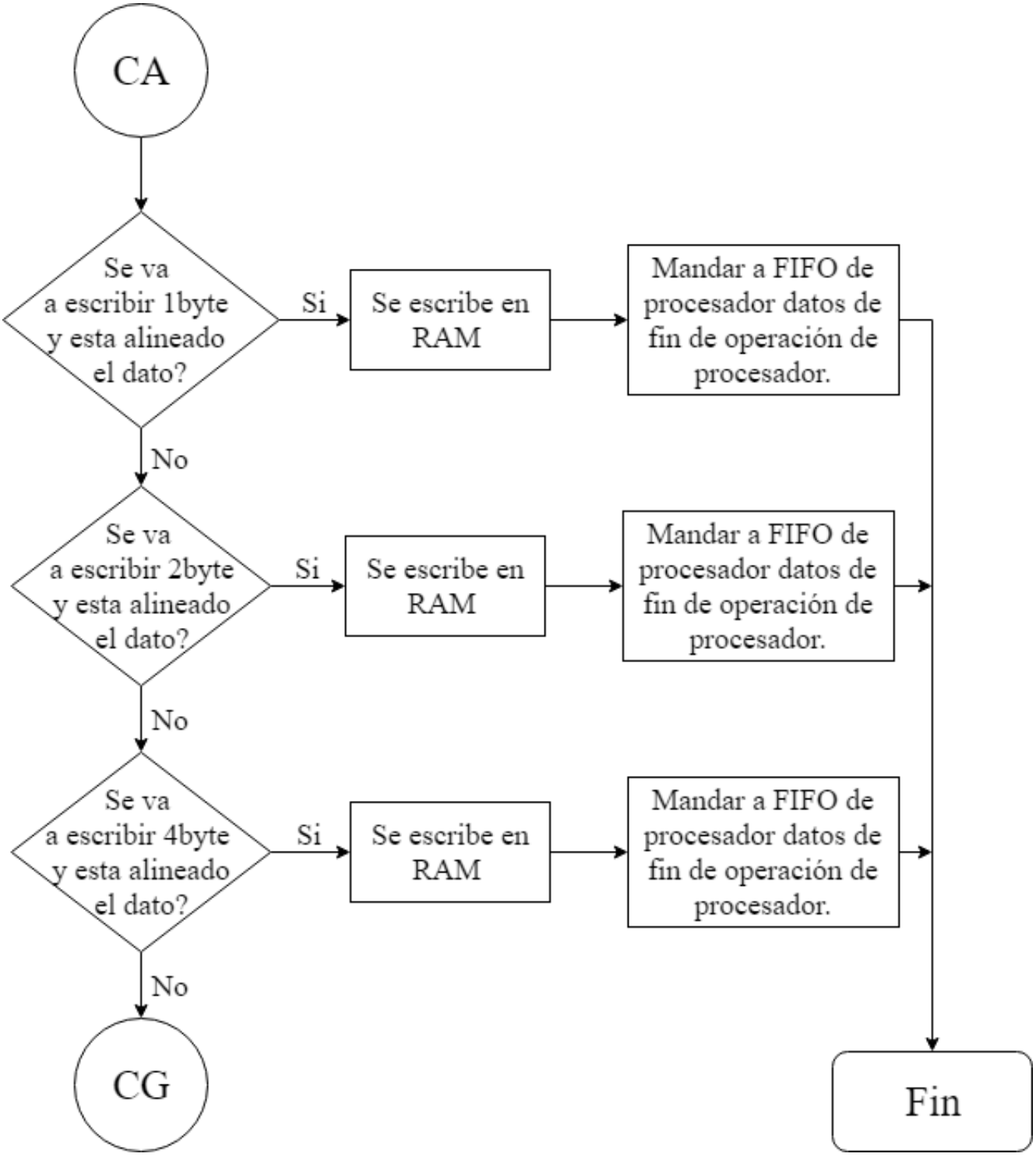


Figura 29: Diagrama de flujo de escritura de RAM desde el procesador

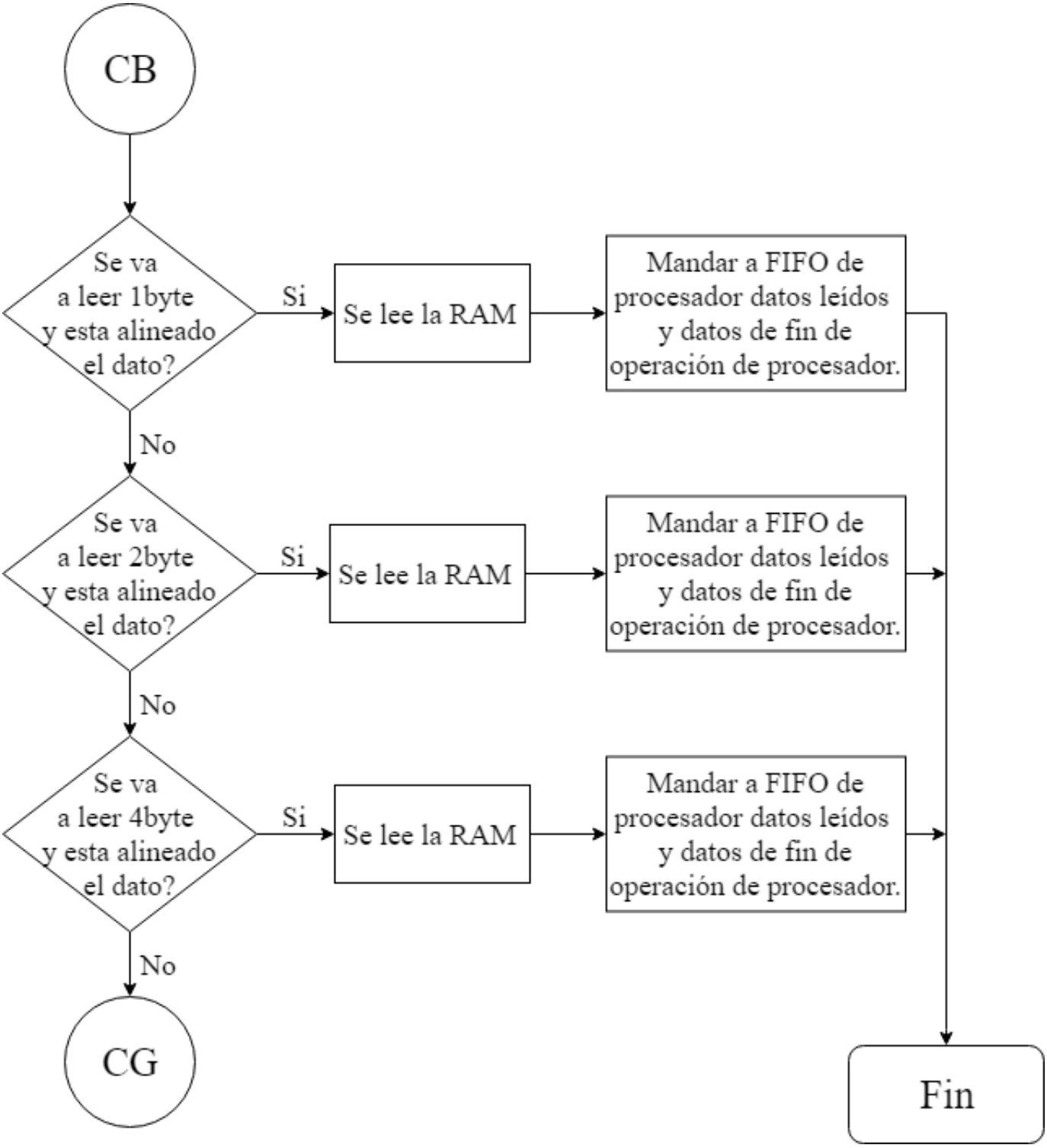


Figura 30: Diagrama de flujo de lectura de RAM desde el procesador

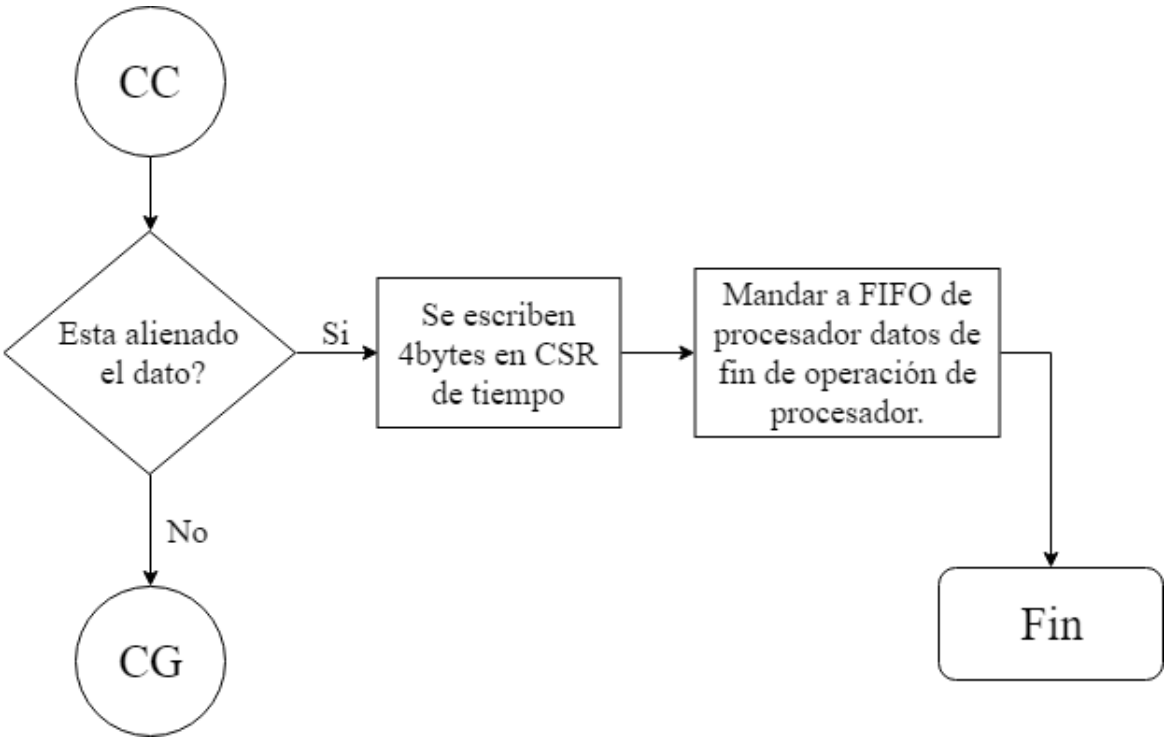


Figura 31: Diagrama de flujo de escritura de CSR de tiempo desde el procesador

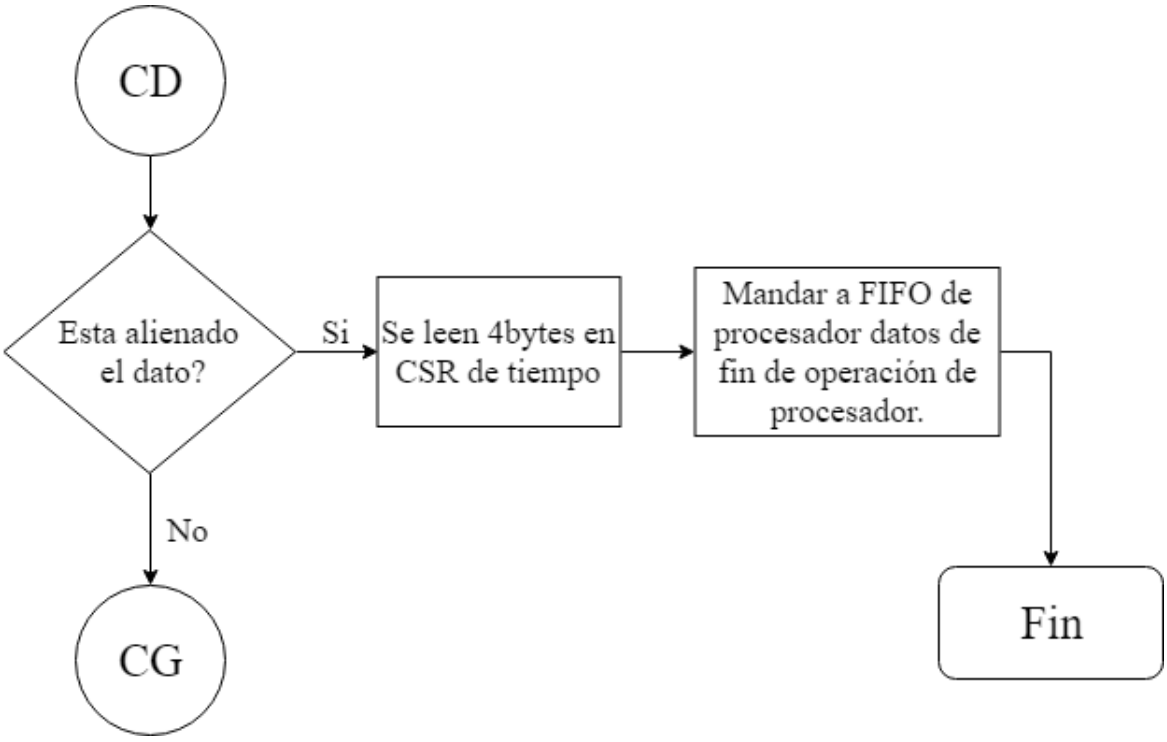


Figura 32: Diagrama de flujo de lectura de CSR de tiempo desde el procesador

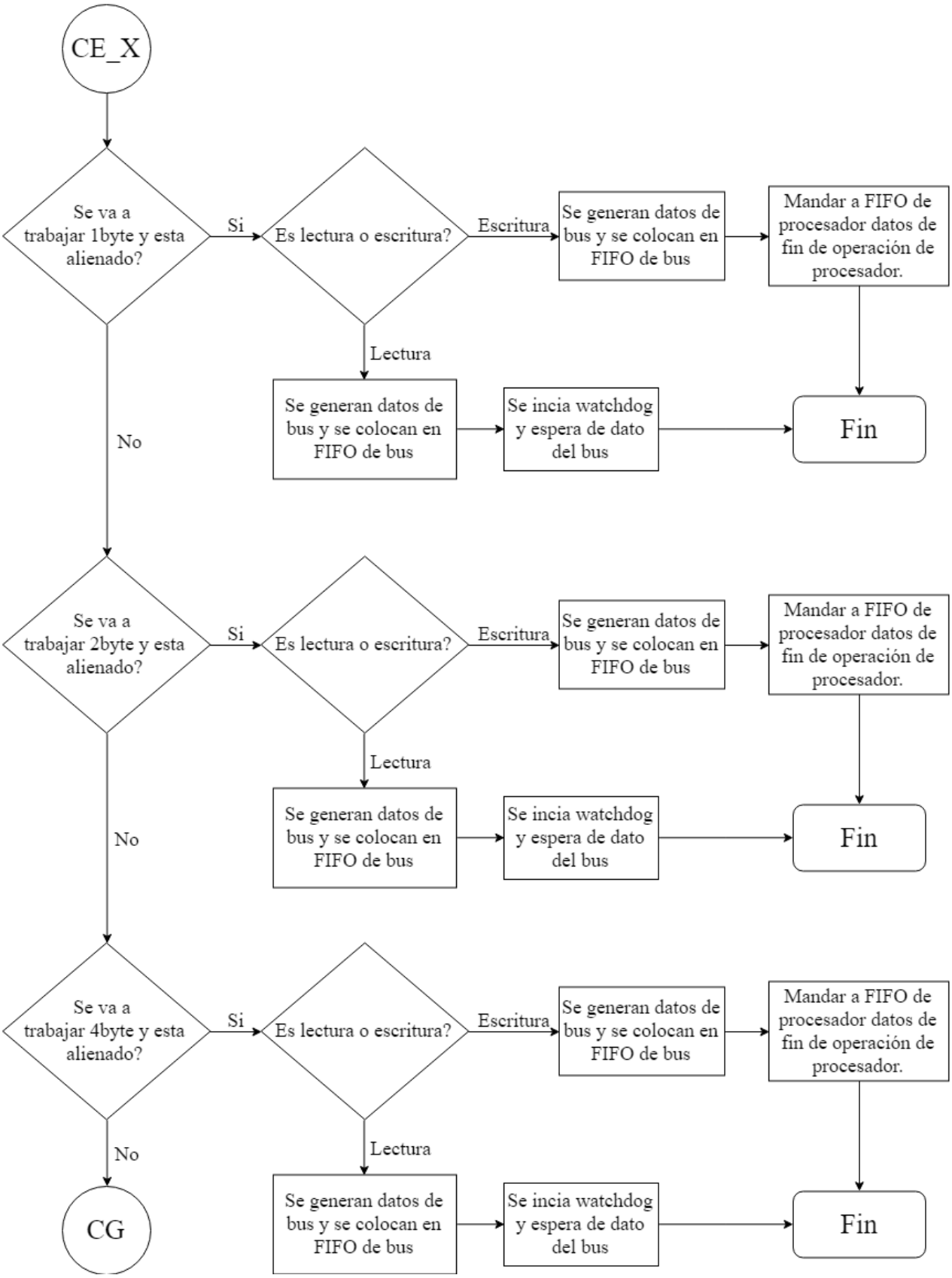


Figura 33: Diagrama de flujo de las operaciones de IO desde el procesador

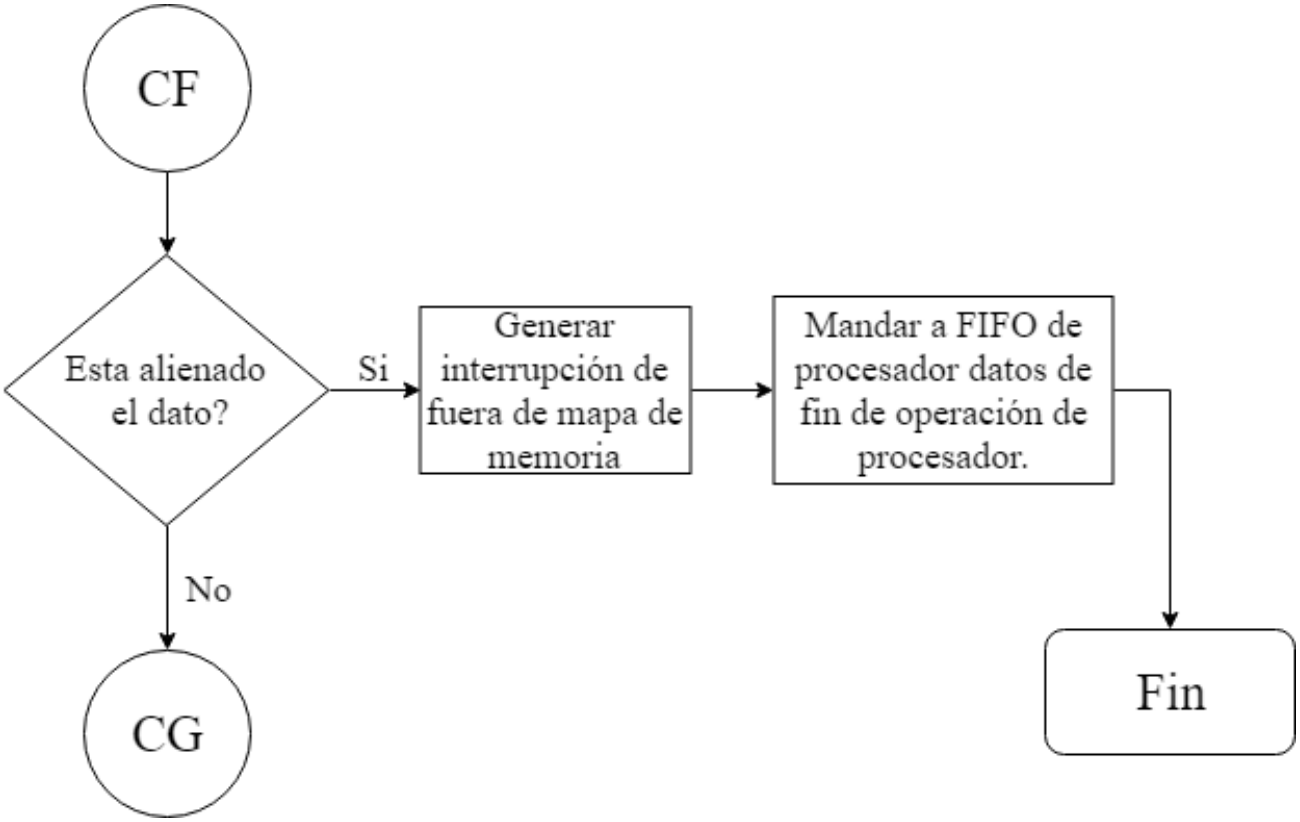


Figura 34: Diagrama de flujo de interrupciones de fuera del mapa de memoria, posee prioridad el desalineado

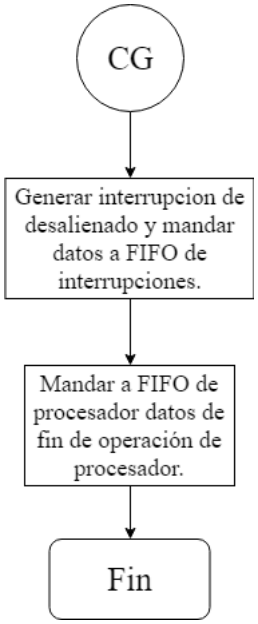


Figura 35: Diagrama de flujo la interrupción de desalineado

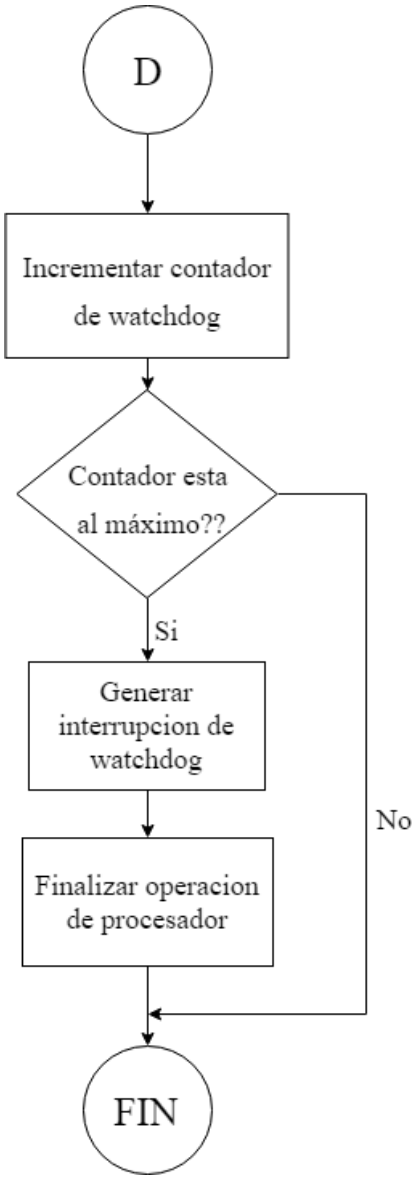


Figura 36: Diagrama de flujo del funcionamiento del watchdog

Sincronización de salidas

Cada FIFO usa la misma lógica para sincornización de salidas.

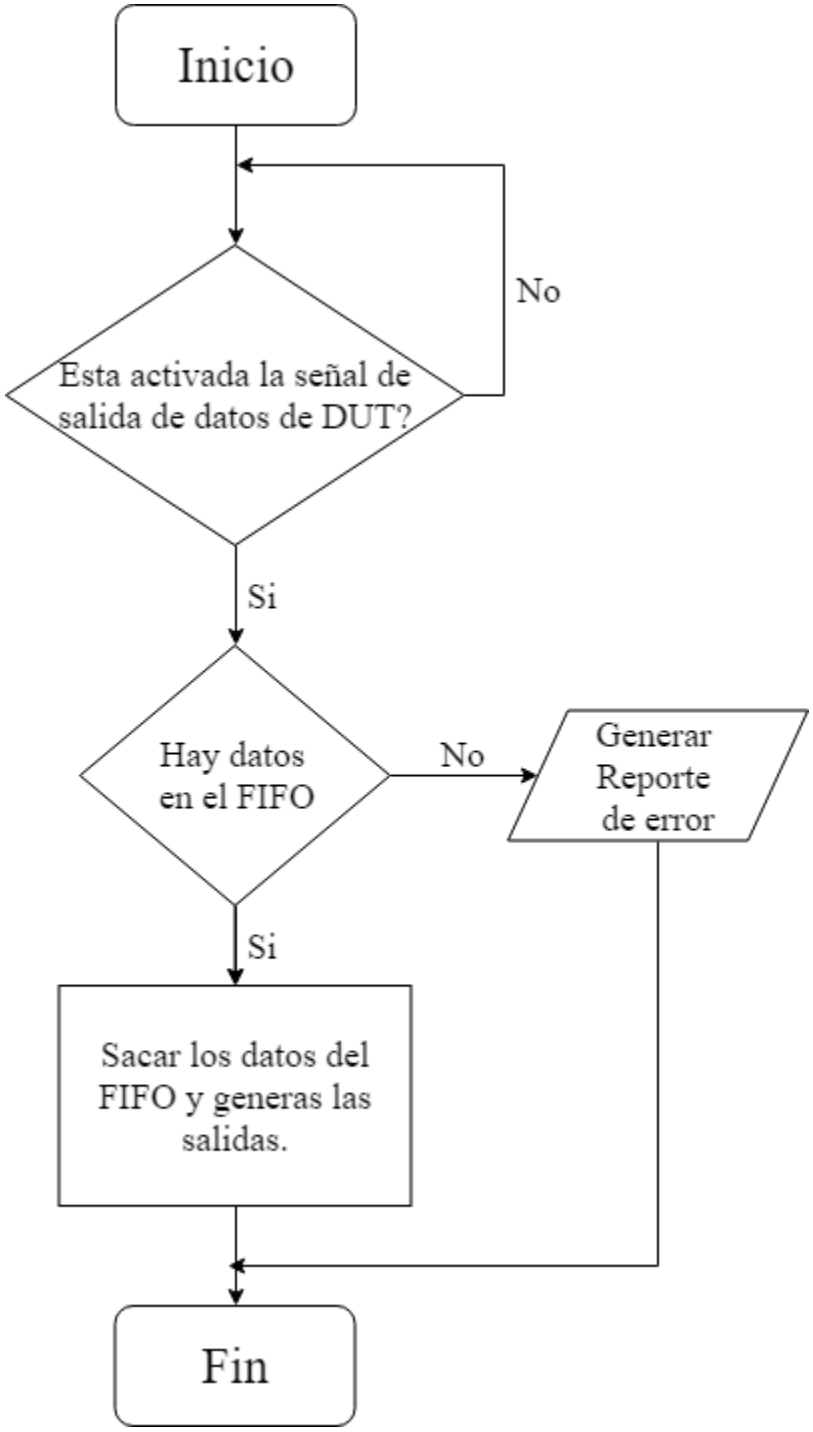


Figura 37: Diagrama de flujo del funcionamiento de las salidas del modelo de referencia.

4.3. Implementación del coverage

4.3.1. Functiona coverage

Se utilizaron 3 covergroups en los puertos de la interfaz de la siguiente forma:

4.3.2. Code coverage

Se utilizaron los siguientes comandos en la simulación para que la herramienta realizara el code coverage

4.4. Implementación de pruebas

Se utilizaron 2 pruebas distintas para comprobar el funcionamiento del entorno, una prueba aleatoria que puede entregar múltiples resultados distintos al enviar cada ciclo nuevos datos a cada sección de puertos; y una prueba en blanco para demostrar que los datos generados por los coverage si son causados por la prueba aleatoria.

Prueba aleatoria

Para generar esta prueba se usa se secuencia de la figura 14b usando la generación de datos aleatoria de la función de UVM req.randomize() esto permite que cada vez que se llama a la secuencia se genere un dato aleatorio.

Prueba en blanco

Esta prueba usa la secuencia de la figura 14b pero sin ninguna generación de datos y simplemente enviando las req vacías de manera que en el driver solo se hace colocan ceros en los puertos del DUT.

Capítulo 5

Resultados y Análisis

Para definir que se realizo de manera correcta la implementación del entorno se va a utilizar los resultados de la herramienta de simulación ya que esta posee un compilador que va a encontrar cualquier error de funcionamiento a ve a reportarlo en la compilación, pero el lograr la compilación no asegura que funciona a como se desea por lo que se van a realizar pruebas implementadas para obtener muestra de que si el entorno es capaz de verificar el CMB.

5.1. Resultados de la implementación del entorno

Como los trabajos de diseño y verificación iniciaron al mismo tiempo, en las primeras etapas no existia un diseño de CMB o un DUT para probar, por lo que se uso un dummy para ver los datos ser entregados.

5.1.1. Resultados pre-DUT

Starting vcs inline pass...
8 modules and 0 UDP read.
recompiling package vcs_paramclassrepository
recompiling package _vcs_DPI_package
recompiling package uvm_pkg
recompiling module MBC
recompiling package CBM_pkg
recompiling module CBM_if
recompiling module CBM_tbench_topblock
All of 9 modules done
..simv up to date
CPU time: 11.134 seconds to compile + .268 seconds to elab + .687 seconds to link

5.1.2. Resultados post-DUT

Starting vcs inline pass...
9 modules and 1 UDP read.
recompiling package vcs_paramclassrepository
recompiling package _vcs_DPI_package
recompiling package uvm_pkg
recompiling module CMB
recompiling module handler
recompiling package CBM_pkg
recompiling module CBM_if
recompiling module CBM_tbench_topblock
All of 10 modules done
..simv up to date
CPU time: 11.375 seconds to compile + .292 seconds to elab + .672 seconds to link

5.2. Resultados de las pruebas

5.2.1. Prueba aleatoria 500 datos

UVM_INFO appsvcsmxetcuvm-1.2srcbaseuvm_objection.svh(1270) @ 50365.00 ns: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO CBM_prueba_test.svh(53) [CBM_prueba_test] ----------
UVM_INFO CBM_prueba_test.svh(54) [CBM_prueba_test] —- TEST FAIL —-
UVM_INFO CBM_prueba_test.svh(55) [CBM_prueba_test] ----------
UVM_INFO appsvcsmxetcuvm-1.2srcbaseuvm_report_server.svh(847) @ 50365.00 ns: reporter [UVMREPORTSERVER]
— UVM Report Summary —

** Report counts by severity
UVM_INFO : 1312
UVM_WARNING : 0
UVM_ERROR : 1295
UVM_FATAL : 0
** Report counts by id
[CBM_prueba_test] 3
[CBM_scoreboard] 1295
[MISCMP] 1306
[RNTST] 1
[TEST_DONE] 1
[UVMRELNOTES] 1

$finish called from file ¨appsvcsmxetcuvm-1.2srcbaseuvm_root.svh; line 527.

$finish at simulation time 50365.00 ns
V C S S i m u l a t i o n R e p o r t
Time: 50365000 ps
CPU Time: 5.160 seconds; Data structure size: 0.4Mb

5.2.2. Prueba sin datos 500 repeticiones vacías

UVM_INFO appsvcsmxetcuvm-1.2srcbaseuvm_objection.svh(1270) @ 5365.00 ns: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
UVM_INFO CBM_nodata_test.svh(58) [CBM_nodata_test] ————————————————-
UVM_INFO CBM_nodata_test.svh(59) [CBM_nodata_test] —- TEST PASS —-
UVM_INFO CBM_nodata_test.svh(60) [CBM_nodata_test] ————————————————-
UVM_INFO appsvcsmxetcuvm-1.2srcbaseuvm_report_server.svh(847) @ 5365.00 ns: reporter [UVMREPORTSERVER]
— UVM Report Summary —

** Report counts by severity
UVM_INFO : 6
UVM_WARNING : 0
UVM_ERROR : 0
UVM_FATAL : 0
** Report counts by id
[CBM_nodata_test] 3
[RNTST] 1
[TEST_DONE] 1
[UVMRELNOTES] 1

$finish called from file ¨appsvcsmxetcuvm-1.2srcbaseuvm_root.svh¡ line 527.
$finish at simulation time 5365.00 ns
V C S S i m u l a t i o n R e p o r t
Time: 5365000 ps
CPU Time: 3.920 seconds; Data structure size: 0.4Mb

5.3. Resultados del coverage

5.3.1. Coverage general

Se realizaron pruebas usando 2 semillas diferentes y 4 cantidades iteraciones, para comprobar los efectos de aumentar o disminuir la cantidad de pruebas o los valores de diferentes semillas.

En la tabla 3 se puede observar los valores delos diferentes code coverages para las semillas y cantidad de pruebas y el valor del functional coverage agrupados los covergroups bajo la categoría GROUP, mientras que la tabla 4 nos muestra los aumentos porcentuales que tiene las pruebas por encima de la de 500 datos usando a las 500 pruebas como libea base para la comparación.

Cuadro 3: Valores de coverage para diferentes cantidades de pruebas y semillas

Semilla	Pruebas	SCORE	LINE	COND	TOGGLE	FSM	BRANCH	GROUP
43285432	500	30.54	48.32	36.84	8.98	22.86	35.71	32.39
	5000	30.55	48.32	36.84	9.00	22.86	35.71	35.35
	50000	30.72	48.32	37.72	9.00	22.86	35.71	37.90
	500000	30.90	48.32	38.60	9.00	22.86	35.71	39.28
83647256	500	30.37	48.32	35.96	8.98	22.86	35.71	32.32
	5000	30.55	48.32	36.84	9.00	22.86	35.71	35.32
	50000	30.55	48.32	36.84	9.00	22.86	35.71	37.89
	500000	30.90	48.32	38.60	9.00	22.86	35.71	39.27

Cuadro 4: Aumento porcentual en los valores de coverage respecto a la linea base de 500 pruebas

Semilla	Pruebas	SCORE	LINE	COND	TOGGLE	FSM	BRANCH	GROUP
43285432	Linea base	0 %	0 %	0 %	0 %	0 %	0 %	0 %
	10	0 %	0 %	0 %	0 %	0 %	0 %	9 %
	100	1 %	0 %	2 %	0 %	0 %	0 %	17 %
	1000	1 %	0 %	5 %	0 %	0 %	0 %	21 %
83647256	Linea base	0 %	0 %	0 %	0 %	0 %	0 %	0 %
	10	1 %	0 %	2 %	0 %	0 %	0 %	9 %
	100	1 %	0 %	2 %	0 %	0 %	0 %	17 %
	1000	2 %	0 %	7 %	0 %	0 %	0 %	22 %

Para mantener orden y la menor cantidad de datos repetidos solo se va a extender el converage de la prueba de 5000 datos de la semilla 83647256, ya que al ser la misma pruebas las variaciones van a ser mínimas en sus comportamientos como se puede observar de la tabla 4 donde apenas varian los valores de coverage, en la tabla 5 se puede observar los valores de coverage para cada bloque por separado de la prueba seleccionada para extender.

Para cada tipo de coverage se va a extender usando solamente los bloques que posean el valor más alto para ese tipo de coverage y el más bajo, excluyendo los valores de 100 % y los que no posee valor.

Cuadro 5: Valores de code coverage para 5000 pruebas de la semilla 83647256

NAME	SCORE	LINE	COND	TOGGLE	FSM	BRANCH
inst_controller	30.14	48.32	36.84	6.96	22.86	35.71
Watch_DG_Counter	32.09	60.00		2.94		33.33
error_manejador	64.17	84.62		19.00		88.89
fifo_int	30.39	56.67		1.16		33.33
gen_sgnl	42.41	68.29	33.33	15.62		52.38
maq_bus	20.68	42.75	14.29	2.45	25.00	18.92
maq_core	25.79	32.88	36.84	7.74	21.05	30.43
verf_algn	95.83	100.00	100.00	100.00		83.33

5.3.2. Line coverage

error_manejador

El line converage de este modulo se encuentra bastante completo a como se puede observar en la tabla 6, en la figura 38 se muestra el reporte de la herramienta de coverage mostrando la sección que no fue cubierta por la prueba.

Cuadro 6: Line coverage de el manejador de errores

	Line No.	Total	Covered	Percent
TOTAL		26	22	84.62
ALWAYS	34	6	6	100.00
ALWAYS	44	20	16	80.00

```
60          1/1                         case ({Rqs_bs,Rqs_cr})
61                                          2'b00: begin
62          1/1                                 {Go_cr,Go_bs} = 2'd0;
63          2/2                                 if(clean) Sgnl_int = 1'b1;
64          1/1                                 else Sgnl_int = 1'b0;
65                                          end
66                                          2'b01: begin
67          1/1                                 Go_cr = 1'b1 & clean;
68          1/1                                 Go_bs = 1'b0;
69          1/1                                 Data_int = Data_cr;
70          1/1                                 Sgnl_int = clean;
71                                          end
72                                          2'b10, 2'b11: begin
73          0/1     ==>                         Go_cr = 1'b0;

74          0/1     ==>                         Go_bs = 1'b1 & clean;

75          0/1     ==>                         Data_int = Data_bs;

76          0/1     ==>                         Sgnl_int = clean;

77                                          end
78                                          default: begin
79          1/1                                 {Go_cr,Go_bs} = 2'd0;
80          1/1                                 Data_int = {dat_size{1'b0}};
81          1/1                                 Sgnl_int = 1'b0;
```

Figura 38: Lineas no cubiertas por el line converage en el modulo manejador de errores

maq_core

A diferencia del manejador de errores este bloque posee un line coverage bastante deficiente además de ser un bloque mas complejo y poseer una mayor cantidad de lineas, se va a agregar una sección del reporte para extender los datos.

Cuadro 7: Line coverage de la maquina core

	Line No.	Total	Covered	Percent
TOTAL		222	73	32.88
ALWAYS	118	3	3	100.00
ALWAYS	127	23	0	0.00
ALWAYS	169	8	7	87.50
ALWAYS	193	188	63	33.51

```
126                         always @(posedge RDY) begin
127         0/1     ==>         if (RW_reg) begin

128         0/1     ==>             case (adrs_cod[1:0])

129                                     2'b00: begin
130         0/1     ==>                     casex ({B_reg,H_reg})

131         0/1     ==>                         2'b00: Data_out_reg = Data_Write_reg;

132         0/1     ==>                         2'b01: Data_out_reg = {Q[31:16],Data_Write_reg[15:0]};

133         0/1     ==>                         2'b1x: Data_out_reg = {Q[31:8],Data_Write_reg[7:0]};

                    ==>  MISSING_DEFAULT
134                                     endcase
135                                 end
136         0/1     ==>             2'b01: Data_out_reg = {Q[31:16],Data_Write_reg[7:0],Q[7:0]};

137                                 2'b10: begin
138         0/1     ==>                 casex ({B_reg,H_reg})

139         0/1     ==>                     2'b01: Data_out_reg = {Data_Write_reg[15:0],Q[15:0]};

140         0/1     ==>                     2'b1x: Data_out_reg = {Q[31:24],Data_Write_reg[7:0],Q[15:0]};

141         0/1     ==>                     default: Data_out_reg = {Q[31:24],Data_Write_reg[7:0],Q[15:0]};

142                                     endcase
143                                 end
144         0/1     ==>             2'b11: Data_out_reg = {Data_Write_reg[7:0],Q[23:0]};

                    ==>  MISSING_DEFAULT
145                                 endcase
```

Figura 39: Lineas no cubiertas por el line converage en el modulo maquina core

5.3.3. Conditional coverage

Al basarse en condiciones no todos los bloques poseen este tipo de coverage y las cantidades operaciones y variables depende del diseñador.

inst_controller

Cuadro 8: Cond coverage del inst controller

	Total	Covered	Percent
Conditions	3	1	33.33
Logical	3	1	33.33
Non-Logical	0	0	
Event	0	0	

```
LINE       176
EXPRESSION (pndng_int && (wt_for_read == 1'b0))
            ----1----    ----------2----------
```

-1-	-2-	Status
0	1	Covered
1	0	Not Covered
1	1	Not Covered

Figura 40: Reporte del conditional coverage del inst controller

maq_bus

Cuadro 9: Cond coverage de maquina bus

	Total	Covered	Percent
Conditions	7	1	14.29
Logical	7	1	14.29
Non-Logical	0	0	
Event	0	0	

```
LINE       110
EXPRESSION (Read_done && Wt_for_read)
            ----1----    -----2-----
```

-1-	-2-	Status
0	1	Not Covered
1	0	Not Covered
1	1	Not Covered

```
LINE       164
EXPRESSION ((Cod_op == 8'h10) || (Cod_op == 8'h11) || (Cod_op == 8'h13))
            --------1--------    --------2--------    --------3--------
```

-1-	-2-	-3-	Status
0	0	0	Covered
0	0	1	Not Covered
0	1	0	Not Covered
1	0	0	Not Covered

Figura 41: Reporte del conditional coverage de la maquina bus

5.3.4. Toggle coverage

error_manejador

Para este modulo solo se va a usar el resumen, la tabla 10 muestra los totales, ports y signals de este bloque.

Cuadro 10: Resumen del toggle coverage del error manejador

	Total	Covered	Percent
Totals	16	9	56.25
Total Bits	600	114	19.00
Total Bits 0->1	300	57	19.00
Total Bits 1->0	300	57	19.00
Ports	13	7	53.85
Port Bits	452	78	17.26
Port Bits 0->1	226	39	17.26
Port Bits 1->0	226	39	17.26
Signals	3	2	66.67
Signal Bits	148	36	24.32
Signal Bits 0->1	74	18	24.32
Signal Bits 1->0	74	18	24.32

fifo_int

Para este modulo solo se va a usar el resumen, la tabla 11 muestra los totales, ports y signals de este bloque.

Cuadro 11: Resumen del toggle coverage del fifo int

	Total	Covered	Percent
Totals	19	2	10.53
Total Bits	346	4	1.16
Total Bits 0->1	173	2	1.16
Total Bits 1->0	173	2	1.16
Ports	8	2	25.00
Port Bits	300	4	1.33
Port Bits 0->1	150	2	1.33
Port Bits 1->0	150	2	1.33
Signals	11	0	0.00
Signal Bits	46	0	0.00
Signal Bits 0->1	23	0	0.00
Signal Bits 1->0	23	0	0.00

5.3.5. FSM coverage

El FSM coverage se basa en los estados de las maquinas de estados y sus transiciones por lo que sus valores va a depender de la complejidad de las maquinas de estados necesitadas por el diseño.

maq_bus

La maquina bus es la que posee el mayor FSM coverage, los reportes se son los estados a los que ha llegado en a tabla 12 y las transiciones en la tabla 13.

Cuadro 12: Reporte de los estados del FSM coverage de la maquina bus

states	Line No.	Covered
ask_for_read	112	Covered
external_wait	241	Covered
internal_wait	110	Not Covered
send_error	147	Not Covered
push_wait	104	Covered
wait_memory	179	Not Covered

Cuadro 13: Reporte de las transiciones del FSM coverage de la maquina bus

transitions	Line No.	Covered
ask_for_read->external_wait	150	Covered
ask_for_read->internal_wait	139	Not Covered
ask_for_read->send_error	147	Not Covered
ask_for_read->push_wait	104	Not Covered
ask_for_read->wait_memory	179	Not Covered
external_wait->ask_for_read	124	Covered
external_wait->push_wait	104	Covered
internal_wait->ask_for_read	112	Not Covered
internal_wait->external_wait	119	Not Covered
internal_wait->push_wait	104	Not Covered
send_error->external_wait	217	Not Covered
send_error->internal_wait	219	Not Covered
send_error->push_wait	104	Not Covered
push_wait->external_wait	226	Covered
wait_memory->external_wait	209	Not Covered
wait_memory->push_wait	104	Not Covered

maq_core

Esta maquina posee más estados y transiciones que la maquina bus por lo que resulta más difícil, se tiene los estados cubiertos en la tabla 14 y las transiciones en la tabla 15.

Cuadro 14: Reporte de los estados del FSM coverage de la maquina core

states	Line No.	Covered
push_wait	265	Not Covered
read_rdy_wait	346	Not Covered
done_rdy	407	Not Covered
send_error	238	Covered
enable_wait	428	Covered
boot_wait	211	Covered
wait_memory	293	Not Covered

5.3.6. Branch coverage

EL branch coverage se basa en los caminos recorridos por las pruebas, se va a usar el bloque error manejador y la maquina bus para esta sección.

error_manejador

Se tiene el resumen de cooverage en la tabla 16 y un seccion en la figura 42.

```
34                  if (reset) begin
                    -1-
35                      Data <= {dat_size{1'b0}};
                        ==>
36                      Sgnl <= Sgnl_int;
37                  end else begin
38                      if (enable) Data <= Data_int;
                        -2-
                        ==>
                        MISSING_ELSE
                        ==>
```

Branches:

-1-	-2-	Status
1	-	Covered
0	1	Covered
0	0	Covered

Figura 42: Ejemplo del branch cov del error manejador

maq_bus

Se tiene el resumen de cooverage en la tabla 17.

Cuadro 15: Reporte de las transiciones del FSM coverage de la maquina core

transitions	Line No.	Covered
push_wait->read_rdy_wait	371	Not Covered
push_wait->send_error	366	Not Covered
push_wait->enable_wait	374	Not Covered
push_wait->boot_wait	211	Not Covered
read_rdy_wait->send_error	347	Not Covered
read_rdy_wait->enable_wait	346	Not Covered
read_rdy_wait->boot_wait	211	Not Covered
done_rdy->boot_wait	211	Not Covered
done_rdy->wait_memory	424	Not Covered
send_error->enable_wait	389	Covered
send_error->boot_wait	211	Not Covered
enable_wait->push_wait	265	Not Covered
enable_wait->send_error	238	Covered
enable_wait->boot_wait	211	Covered
enable_wait->wait_memory	293	Not Covered
boot_wait->enable_wait	227	Covered
wait_memory->done_rdy	407	Not Covered
wait_memory->enable_wait	411	Not Covered
wait_memory->boot_wait	211	Not Covered

Cuadro 16: Reporte de branch coverage del error manejador

	Line No.	Total	Covered	Percent
Branches		9	8	88.89
IF	34	3	3	100.00
IF	44	6	5	83.33

5.3.7. Group coverage

5.4. Análisis de las pruebas

- De la compilación del código en la herramienta se logra obtener que no hay errores pero, no muestra datos que den valor a que el entorno esta completo o que tan bien puede ejecutar sus tareas.
- Las pruebas no logran estresar todos los casos deseados, se deben realizar pruebas especializadas si de verdad se desea ejecutar una verificación funcional de que abarque la mayoría del DUT.
- Al realizar las pruebas se encontró un bug en el DUT que genero problemas para lograr continuar otras pruebas, este bug genera interrupciones de tipo fuera del mapa de memoria para direcciones de ram; este bug esta se encuentra en proceso de corrección,

Cuadro 17: Reporte de branch coverage de la maq bus

	Line No.	Total	Covered	Percent
Branches		9	8	88.89
IF	34	3	3	100.00
IF	44	6	5	83.33

su funcionamiento genera problemas con el avance del modelo de referencia ya que se desordenan los datos guardados en el FIFO de interrupciones cada vez que es generado al hacer una comprobación contra una interrupción que no debería suceder, el predictor no posee la capacidad de diferenciar los datos desde el DUT erróneos de datos para sincronizar; esto hace que pierda la confianza para pruebas muy largas.

- Al ejecutar las pruebas en blanco y aleatoria se lograr obtener que la prueba aleatoria es capaz de encontrar errores que la prueba en blanco no puede ver, esto es una claro resultado de que el entorno esta cumpliendo con su labor y es capaz de que bajo los estímulos correctos realizar comprobación del funcionamiento del DUT.

5.5. Análisis de los coverages

De los resultados de los coverages se puede observar que faltan realizar pruebas que sean más dirigidas para poder comprobar los procesos secuenciales del diseño, ya que los pruebas completamente aleatorias no son suficientes para poder comprobar todas las secciones deseadas sin cantidades absurdas de pruebas y aun las grandes cantidades de pruebas no pueden asegurar que se compruebe todo lo deseado a como se puede ver en la tabla 4, donde 1000 veces mas pruebas no lograr mejorar los code coverage aunque el group coverage siga creciendo.
Casi todos los code coverage muestran una falta de pruebas a casos específicos y los que poseen los mejores valores de coverage es porque posee un uso ligero de la operación que se esta reportando.

Cuadro 18: Variables para Group coverage cg_bined

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT
addrH	32768	32768	0	0.00
addrL	32768	32767	1	0.01
datawriteH	32768	32768	0	0.00
datawriteL	32768	32767	1	0.01
dpopdest	16	0	16	100.00
dpopfuen	16	0	16	100.00
dpopcod	16	0	16	100.00
dpopdir	4096	4096	0	0.00
dpopinfH	32768	32768	0	0.00
dpopinfL	32768	32768	0	0.00
dpushdest	16	15	1	6.25
dpushfuen	16	15	1	6.25
dpushcod	16	15	1	6.25
dpushdir	4096	4095	1	0.02
dpushinfH	32768	32768	0	0.00
dpushinfL	32768	32767	1	0.01
datareadH	32768	32768	0	0.00
datareadL	32768	32767	1	0.01
intf.R_W	2	0	2	100.00
intf.B	2	0	2	100.00
intf.H	2	0	2	100.00

Cuadro 19: Variables para Group coverage cg_unbined

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT
addr	64	63	1	1.56
datawrite	64	63	1	1.56
dataread	64	63	1	1.56
dpop	64	63	1	1.56
dpopdest	64	0	64	100.00
dpopfuen	64	0	64	100.00
dpopcod	64	0	64	100.00
dpopdir	64	0	64	100.00
dpopinf	64	0	64	100.00
dpush	64	63	1	1.56
dpushdest	64	63	1	1.56
dpushfuen	64	63	1	1.56
dpushcod	64	62	2	3.12
dpushdir	64	63	1	1.56
dpushinf	64	63	1	1.56
intrDpush	64	63	1	1.56
intf.Address	64	63	1	1.56
intf.R_W	2	0	2	100.00
intf.Data_Write	64	63	1	1.56
intf.Data_Read	64	63	1	1.56
intf.B	2	0	2	100.00
intf.H	2	0	2	100.00

Cuadro 20: Variables para Group coverage cg_strict

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT
addr	8200	880	7320	89.27
addr512	520	2	518	99.62
addr256	264	2	262	99.24
addr128	136	2	134	98.53
addr64	72	2	70	97.22
dpopdest	4	0	4	100.00
dpopfuen	4	0	4	100.00
dpopcod	8	0	8	100.00
dpopdir	8200	8176	24	0.29
dpushdest	4	2	2	50.00
dpushfuen	4	3	1	25.00
dpushcod	8	6	2	25.00
dpushdir	8200	8199	1	0.01
intf.R_W	2	0	2	100.00
intf.Data_Write	64	63	1	1.56
intf.Data_Read	64	63	1	1.56
intf.B	2	0	2	100.00
intf.H	2	0	2	100.00

Capítulo 6

Conclusiones y Recomendaciones

6.1. Conlusiones

Los porcentajes de coverage funcional permiten mostrar que se están realizando pruebas en ciertos rangos del las entradas del sistema, cuando se combina a los efectos mostrados por las diferentes pruebas se puede decir que se esta realizando una verificación del DUT que aunque no es completa ni esta cerca de ser aceptable, si confirma el funcionamiento del entorno de verificación y de las pruebas realizadas. EL uso de una secuencia aleatoria para probar el funcionamiento total de un sistema secuencial es casi imposible ya que las probabilidades de que se ejecuten los escenarios necesitados para comprobar funcionamientos específicos es casi imposible.
El entorno es capaz de verificar el funcionamiento del DUT de manera correcta y con la adición de nuevas pruebas preparado para que conforme vayan avanzando las etapas del proceso de manufactura del microprocesador compruebe que el funcionamiento y cualquier cambio realizado durante estas nuevas etapas pueda ser verificado que funciona de la manera que se desea y cuales son los casos es que no funciona como se desea.

6.2. Recomendaciones

Se deben usar las secuencias que ejecuten las funciones del CMB para lograr una mayor cobertura del código en vez de una secuencia aleatoria para lograr resultados en la verificación del dispositivo en una menor cantidad de pruebas.
Es recomendable aprovechar características de alto nivel de programación para simplificar el diseño del modelo de referencia, para futura iteraciones es mejor reemplazar las llamadas a nuevos task por uso de funciones que retornan valores respecto a las entradas, esto puede permitir simplificar el análisis del modelo de referencia y cualquier ajusto o arreglo ya que se pueden disminuir las secciones que realizan exactamente la misma función.

Bibliografía

[1] Accellera. (12 de Junio de 2014). Universal Verification Methodology (UVM) 1.2 Reference implementation. Recuperado Junio de 2018, de https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/

[2] Accellera. (2015). Universal Verification Methodology (UVM) 1.2 User's Guide. CA, USA: Accellera Systems Initiative.

[3] Aviral M. (s.f.). VCS and coverage . Recuperado Junio de 2018, de http://www.vlsiip.com/vcs/

[4] Battistutti, O. C. (2003). Metodología de la programación: algoritmos, diagramas de flujo y programas (2 ed.). (Alfaomega, Ed.)

[5] Liu, S. (2001). Sypnosis VCS Jumpstart training.

[6] LT Wang, C. W. (2006). VLSI Test Principles and Architectures: Design for Testability. California: Systems on Silicon.

[7] Patel, D. R. (07 de Mayo de 2015). VCS commands ease coverage efforts & speed.

[8] Smith, D. (2009). A Practical Look @ SystemVerilog Coverage. Recuperado Junio de 2018, de https://www.doulos.com/knowhow/sysverilog/DVClub_Austin_09/DVClub_Austin.pdf

[9] Tala, D. K. (2014). Asic-World. Recuperado Junio de 2018, de http://www.asic-world.com/systemverilog/index.html

[10] Verification Guide. (2016). Recuperado Junio de 2018, de http://www.verificationguide.com/p/uvm-tutorial.html

[11] Wile, B., Goss, J., & Roesner, W. (2005). Comprehensive Functional Verification. San Francisco, CA: Morgan Kaufmann.

[12] www.TestBench.in. (2017). Recuperado Junio de 2018, de http://www.testbench.in/UT_00_INDEX.html