UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Informática de Sistemas y Computadores

Inyección de fallos basada en simuladores de código libre para sistemas modelados mediante lenguajes de descripción de hardware

Trabajo Fin de Máster

Máster Universitario en Ingeniería de Computadores y Redes

AUTOR/A: Navarro López, Emilio

Tutor/a: Andrés Martínez, David de

Cotutor/a: Baraza Calvo, Juan Carlos

CURSO ACADÉMICO: 2023/2024

Resumen

La inyección de fallos es una metodología esencial para evaluar el comportamiento de un sistema en presencia de fallos, permitiendo estimar su robustez, detectar cuellos de botella relacionados con su confiabilidad y verificar los mecanismos de tolerancia a fallos. La inyección de fallos en modelos del sistema durante las fases iniciales del diseño facilita la detección y corrección de problemas de manera más económica y sencilla antes de la disponibilidad de un prototipo físico, aunque la precisión de los resultados dependerá de la exactitud del modelo empleado.

Este Trabajo Fin de Máster tiene como objetivo diseñar, implementar y evaluar una herramienta de inyección de fallos basada en simulación para modelos realizados en lenguajes de descripción hardware (concretamente VHDL y Verilog) para su uso en simuladores de código libre como Icarus Verilog o GHDL.

Gracias al uso de biblioteca Cocotb (*COroutine based COsimulation Testbench*), podremos introducir distintos tipos de fallos en componentes hardware simulados. Basándose en esta biblioteca que posee una interfaz común y un nivel superior de abstracción para simuladores, llegaremos generar bancos de prueba (*testbench*). Con ello, podremos dotar de funcionalidad de inyección a simuladores que originalmente no lo tienen, de una manera automática y común para los distintos simuladores y lenguajes.

Palabras clave: VHDL, Verilog, inyección de fallos, hardware, open-source, simuladores, Cocotb

Abstract

Fault injection is an essential methodology to evaluate the behaviour of a system in the presence of faults, allowing to estimate its robustness, detect reliability-related bottlenecks and verify fault tolerance mechanisms. The injection of faults into system models during the initial phases of the design facilitates the detection and correction of problems in a cheaper and simpler way before the availability of a physical prototype, although the accuracy of the results will depend on the accuracy of the model used.

This Master Thesis aims to design, implement and evaluate a simulation-based fault injection tool for models made in hardware description languages (namely VHDL and Verilog) for use in open source simulators such as Icarus Verilog or GHDL.

Thanks to the use of the Cocotb library (COroutine based COsimulation Testbench), we can introduce different types of faults in simulated hardware components. Based on this library, which has a common interface and a higher level of abstraction for simulators, we can generate testbenches. With this, we will be able to provide injection functionality to simulators that originally do not have it, in an automatic and common way for the different simulators and languages.

Keywords: VHDL, Verilog, fault injection, hardware, open-source, simulators, Cocotb

Índice de contenidos

Índice de ilustraciones

Índice de Tablas

Glosario

API: Una interfaz de programación de aplicaciones que permite la comunicación entre diferente software.

DUT: Dispositivo Bajo Prueba, el componente o sistema que se está probando en el entorno de banco de pruebas.

FLI: Interfaz de Lenguaje Extranjero, una interfaz para integrar lenguajes de programación extranjeros con simuladores VHDL.

FRAMEWORK: Una plataforma para desarrollar aplicaciones de software, proporcionando una base sobre la cual los desarrolladores de software pueden construir programas para una plataforma específica.

FST: Rastro de Señal Rápida, un formato de archivo similar a VCD, pero optimizado para un acceso más rápido y un tamaño de archivo más pequeño.

GCC: Colección de Compiladores de GNU, un sistema de compiladores producido por el Proyecto GNU que admite varios lenguajes de programación.

GHW: GHDL Waveform, un formato de forma de onda utilizado por el simulador VHDL GHDL.

HDL: Lenguaje de Descripción de Hardware, un lenguaje informático especializado utilizado para describir la estructura y el comportamiento de circuitos electrónicos.

HWIFI: Marco de Inyección de Fallos de Hardware, utilizado para probar la robustez de los diseños de hardware.

IP: Propiedad Intelectual

LLVM: Máquina Virtual de Bajo Nivel, una colección de tecnologías de compilador y cadena de herramientas modulares y reutilizables.

MAKEFILE: Un archivo que contiene un conjunto de directivas utilizadas con la herramienta de automatización de compilación 'make' para compilar y enlazar un programa.

OPEN-SOURCE: Software cuyo código fuente original está disponible de forma gratuita y puede ser redistribuido y modificado.

RTL: Nivel de Transferencia de Registros, un nivel de abstracción utilizado para describir el funcionamiento de circuitos digitales.

SWIFI: Marco de Inyección de Fallos de Software, utilizado para probar la robustez de las aplicaciones de software.

TESTBENCH: Banco de pruebas para verificar la corrección de un diseño o modelo.

VCD: Valor de Cambio de Volcado, un formato de archivo utilizado para volcar los cambios en los valores de señal de un circuito digital.

VHPIDIRECT: Interfaz de Procedimientos VHDL Directa, una interfaz que permite llamadas directas entre VHDL y lenguajes extranjeros.

VPI: Interfaz de Procedimientos Verilog, una interfaz estándar en el lenguaje de descripción de hardware Verilog.

XML: Lenguaje de Marcado Extensible, es un metalenguaje que define información de manera estructurada.

1. Introducción

En el proceso de desarrollo de hardware, la detección e inyección de fallos juegan un papel crucial para garantizar la confiabilidad y robustez de los sistemas. La capacidad de predecir y mitigar fallos potenciales durante las fases tempranas del diseño no solo permite identificar y corregir problemas de manera más eficiente, sino que también reduce significativamente los costos asociados a correcciones tardías en prototipos avanzados o productos finales.

A continuación, se describen la motivación, los objetivos y la estructura del presente Trabajo de Fin de Máster.

1.1 Motivación

La inyección de fallos permite simular condiciones de funcionamiento adversas, y evaluar el comportamiento del hardware bajo diferentes escenarios de fallo, proporcionando valiosa información para mejorar los mecanismos de tolerancia a fallos y asegurar un funcionamiento continuo y fiable.

Este enfoque proactivo en la detección y gestión de fallos es esencial para el desarrollo de sistemas electrónicos avanzados, donde la confiabilidad es un requisito primordial, especialmente en aplicaciones críticas como la automoción, la aeronáutica y la medicina. Por lo tanto, contar con herramientas efectivas y accesibles para la inyección de fallos es fundamental para avanzar en la creación de hardware robusto y confiable.

En nuestro caso, el uso de herramientas de código abierto para la inyección de fallos mediante simulación ofrece una alternativa accesible y flexible a las soluciones comerciales. Estas herramientas permiten a los desarrolladores simular condiciones de fallo en las primeras etapas del diseño, facilitando la detección y corrección de errores de manera temprana y sobre todo económica.

Utilizar librerías y simuladores de hardware *open-source* no solo reduce los costos, sino que también proporciona un entorno que cualquier persona puede utilizar, teniendo mayor control y adaptabilidad, con una comunidad que da un respaldo fiable.

1.2 Objetivo

El objetivo principal de este Trabajo de Fin de Master es utilizar mecanismos y técnicas de inyección clásicas en la inyección por simulador con herramientas *open-source*, utilizando una interfaz común que sea compatible con los simuladores de los distintos lenguajes de descripción de hardware que existen en el mercado.

Se empleará la biblioteca Cocotb para añadir a estos simuladores la funcionalidad de la inyección por simulador (comandos de simulador). Probaremos el alcance y fiabilidad de estas funcionalidades añadidas inyectando distintos tipos de fallos y mediante distintas métricas como, por ejemplo: su cobertura, su tasa de fallos, prestaciones…

1.3 Estructura

El siguiente TFM está dividido en 5 secciones diferentes (excluyendo la introducción) en las que evaluaremos distintos temas.

En la sección 2 “Estado del arte” haremos una breve descripción del panorama actual en cuanto a los fallos hardware, sus tipos y los métodos de inyección, proporcionando una visión global de la situación actual.

A continuación, en la sección 3 “Tecnología utilizada”, repasaremos las herramientas disponibles, tanto comerciales como de acceso libre, que utilizaremos en el trabajo, dando una pequeña descripción de su uso y proponiendo qué utilidades esperamos del software utilizado.

Posteriormente, en la sección 4 “Análisis de las posibilidades de inyección de fallos mediante Cocotb” aprovecharemos para comprobar el verdadero alcance de la biblioteca Cocotb en la inyección de fallos por órdenes de simulador, aprovechando su capacidad de generar bancos de prueba (*testbenches*) y midiendo su compatibilidad contra los distintos simuladores mencionados en la sección anterior.

En la sección 5 “Metodología para la realización de campañas de inyección de fallos mediante Cocotb” probaremos las capacidades de la herramienta mediante mecanismos de inyección clásicos (funciones como `force`, `deposit` y `release`) en escenarios de generación de experimentos. De esta manera, se probarán distintos tipos de fallos de forma más automatizada, lo cual será útil para escenarios reales de inyección de fallos, como lo son las campañas de inyección.

Por último, con nuestros hallazgos en las campañas de inyección creadas en la sección anterior, generaremos campañas para diversos procesadores en la sección 6 titulada “Resultados de campañas de inyección reales”.

Esto nos ayudará a sacar unas mejores conclusiones y nos ayudará a evaluar mejor el comportamiento de un sistema inyectado gracias a la biblioteca que utilizaremos a lo largo del trabajo. Todo el código utilizado y creado se puede encontrar en el repositorio online: https://github.com/Emnalo/cocotb-tfm .

2. Estado del arte

Este capítulo analiza el estado del arte en inyección de fallos hardware, abarcando tres áreas principales: fallos hardware y su clasificación, inyección de fallos y sus tipos, y simulación hardware y lenguajes. Se exploran los tipos de fallos, sus causas, y se detallan metodologías de inyección, centrándonos en la parte de simulación. Esto nos ayudará a entender el estado actual en el campo de la inyección de fallos hardware.

2.1 Fallos hardware

En el contexto de los sistemas informáticos y electrónicos, un fallo se refiere a cualquier condición que pueda impedir que un sistema funcione correctamente. Un fallo puede manifestarse como un error en el software, un mal funcionamiento del hardware o un error humano. Los fallos (*faults*) pueden provocar que un sistema no cumpla con sus especificaciones o expectativas de rendimiento, lo que puede llevar a averías del sistema (*failures*) o degradación del servicio. [1] [2]

Para este TFM nos centraremos en los fallos hardware en particular. Estos son problemas que ocurren en los componentes físicos de un sistema informático y que son el origen de un error. Estos fallos pueden deberse a defectos de fabricación, desgaste por uso, condiciones ambientales adversas o daños físicos.

Como podemos observar en la cadena de amenazas mostrada en la Ilustración 1, estos fallos provocan que el sistema se desvíe de su estado esperado, pudiendo generar errores. Estos errores al propagarse pueden provocar un servicio incorrecto, manifestándose en averías que trastornan el normal funcionamiento del sistema y los servicios que ofrece de cara al usuario [3] .

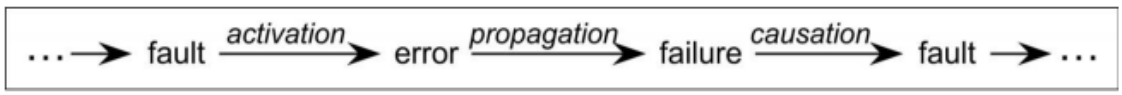

Ilustración 1 Cadena de amenazas

Los fallos son generalmente clasificados por diversos criterios como por ejemplo su duración, causa, extensión y variabilidad [3] [4],creando así posibles modelos generalistas para su identificación. En nuestro proyecto tendremos el mayor interés por los fallos según su origen (causa) y por su duración (persistencia) por lo que destacaremos los siguientes tipos:

- Agrupados por causa:
  - **Fallos aleatorios:**
    - **Descripción:** Estos fallos ocurren de forma no determinista y son impredecibles. Son causados por factores físicos internos y externos y no se repiten bajo las mismas condiciones. Común en hardware.
    - **Causas:** Internas como desgaste de los componentes por uso o envejecimiento o externas por corrosión, estrés, presión, radiación…

- **Fallos sistemáticos:**
    - **Descripción:** Estos fallos se producen de forma determinista debido a errores humanos en el diseño, desarrollo, o mantenimiento del sistema. A menudo se repiten en condiciones específicas y son más fáciles de reproducir y solucionar que los fallos aleatorios. Común en software.
    - **Causas:** Errores humanos, tanto en la creación como en el ciclo de vida del sistema por errores de diseño o configuración.

- Agrupados por persistencia:
    - **Fallos permanentes:**
        - **Descripción:** Estos fallos son duraderos y no desaparecen hasta que el componente defectuoso es reparado o reemplazado.
        - **Causas:** Defectos de fabricación, daños físicos, desgaste por envejecimiento.
    - **Fallos transitorios:**
        - **Descripción:** Son fallos temporales que desaparecen después de un corto período de tiempo.
        - **Causas:** Interferencias electromagnéticas, radiación, fluctuaciones de energía.

- **Fallos intermitentes:**
    - **Descripción:** Estos fallos aparecen y desaparecen de manera irregular, a menudo dificultando su detección y diagnóstico.
    - **Causas:** Problemas de conexión, variaciones ambientales, estrés mecánico o térmico.

Esto nos lleva a construir modelos de fallo [5]. Estos nos ayudan a entender, simular y analizar el comportamiento de sistemas hardware en situaciones de fallo. Los modelos nos permiten desarrollar estrategias para la prevención de fallos o su posible mitigación. En este TFM nos centraremos en los fallos hardware aleatorios, permanentes y transitorios, por lo que ofrecemos algunos modelos de fallo comunes, incluyendo:

- **Modelos de fallos permanentes:** *Stuck-at* (0,1), cortocircuitos, circuitos abiertos.
- **Modelos de fallos transitorios:** *Bit-flip*, pulso, indeterminación.
- **Modelos de fallos intermitentes:** Combinaciones de fallos permanentes y transitorios.

El desarrollo conjunto de estos modelos da lugar a mecanismos implementables en la creación de hardware para poder tolerar dichos fallos como por ejemplo la redundancia, la corrección de errores o reconfiguración.

2.2 Inyección de fallos hardware

Para verificar el correcto funcionamiento de los mecanismos de detección y/o tolerancia a fallos en un sistema hardware necesitamos hacer pruebas previas mediante la introducción de fallos en el componente hardware de forma deliberada y controlada. Esto nos permite observar y estudiar su comportamiento para poder evaluarlo, lo que nos ayuda a comprender como reaccionará el componente frente a fallos reales.

Este proceso se describe como inyección de fallos en hardware y existen diversos tipos/técnicas. Con ello conseguimos como hemos explicado anteriormente beneficios clave, los cuales se pueden resumir en:

- Una comprensión en profundidad del sistema incluyendo su comportamiento en presencia y ausencia de fallos lo que permite diseñar mecanismos de gestión y recuperación de errores.
- Una evaluación de la robustez del sistema.
- Identificación de vulnerabilidades o punto débiles del sistema hardware lo que nos revela puntos que podrían ser punto de explotación tanto de agentes internos como externos.
- Validación de mecanismos de tolerancia a fallos integrados en el diseño del software/hardware implementado.

Una metodología óptima que seguir para lograr estos objetivos a la hora de evaluar un sistema mediante la inyección de fallos es utilizar el modelo FARM (*Faults*, *Activation*, *Readouts*, *Measures*) [6] [7]:

- ***Faults* (Fallos):** Conjunto de fallos que se inyectan en el sistema objetivo durante los experimentos. Estos fallos pueden ser específicos, como fallos en el hardware o software, que se introducen deliberadamente para estudiar la respuesta del sistema.
- ***Activation* (Activación):** Datos o entradas utilizados para ejercitar el sistema durante la experimentación. Este componente asegura que el sistema se someta a condiciones operativas normales o extremas para observar cómo maneja los fallos inyectados.
- ***Readouts* (Lecturas):** Lecturas obtenidas del comportamiento del sistema durante y después de la inyección de fallos. Estas lecturas pueden incluir datos relevantes que ayudan a entender el impacto de los fallos inyectados.
- ***Measures* (Medidas):** Medidas extraídas del análisis de las lecturas. Estas medidas son utilizadas para evaluar la resiliencia, fiabilidad y otros atributos de seguridad del sistema bajo prueba.

2.2.1 Tipos de inyección

En este apartado mencionamos diversas técnicas de inyección que listamos a continuación:

Técnicas Físicas: (HWIFI: *Hardware Implemented Fault Injection*): Se utiliza hardware específico y especializado para introducir fallos en la memoria o registros de hardware.

- **Inyección de fallos a nivel de pines:**
  - **Descripción:** Se inyectan fallos directamente en los pines de los componentes electrónicos utilizando dispositivos externos [8].
  - **Objetivo:** Probar la robustez del hardware ante alteraciones eléctricas directas.
- **Radiación de iones pesados:**
  - **Descripción:** Se utiliza radiación ionizante para inducir fallos transitorios en los semiconductores [9].
  - **Objetivo:** Simular el efecto de partículas cósmicas en componentes críticos, común en aplicaciones aeroespaciales.
- **Alteración de suministro de energía (EMI: *Electromagnetic interference*, modificación de fuente de energía):**
  - **Descripción:** Se manipulan las condiciones de energía para inducir fallos por ejemplo generando pulsos de alto voltaje, provocando interferencias electromagnéticas [10].
  - **Objetivo:** Evaluar la resistencia del sistema ante variaciones en el suministro eléctrico.
- **Láser:**
  - **Descripción:** La técnica de inyección de fallos utilizando láser *(Laser Fault Injection*, LFI) implica el uso de un láser para inducir fallos en componentes específicos del hardware, alterando su comportamiento normal mediante la generación de pares electrón-hueco en las uniones PN de los semiconductores [11].
  - **Objetivo:** Evaluar la robustez del hardware y la efectividad de los mecanismos de tolerancia a fallos en presencia de fallos inducidos por láser.
- **Depuradores y puertos de depuración especializados:**
  - **Descripción:** La inyección de fallos mediante depuradores y puertos de depuración especializados (como los puertos Nexus) utiliza herramientas de depuración para manipular la memoria y registros del procesador y así introducir fallos sin alterar el software del sistema objetivo [12].
  - **Objetivo:** Evaluar la robustez del hardware y la efectividad de los mecanismos de tolerancia a fallos en presencia de fallos inducidos en puertos

Técnicas software (SWIFI: *Software Implemented Fault Injection*): Se utilizan programas para introducir fallos en la memoria o registros de hardware a través de comandos de software, simulando fallos físicos sin necesidad de dispositivos de inyección físicos.

- **En tiempo de compilación:**
  - **Descripción:** Se modifica el código fuente o binario para emular la ocurrencia del fallo [13].
  - **Objetivo:** Evaluar la robustez del hardware ante posibles errores introducidos durante la fase de desarrollo.

- **En tiempo de ejecución:**
  - **Descripción:** Se utilizan *traps* o excepciones hardware para ejecutar el código de inyección en el manejador definido [14].
  - **Objetivo:** Probar la respuesta del sistema en tiempo real ante la ocurrencia de fallos.

Técnicas basadas en emulación: Implementación del modelo RTL (*Register Transfer Level*) en un dispositivo lógico programable en el que se inyecta el fallo.

- **En tiempo de compilación:**
  - **Descripción:** Se modifica el modelo antes de su implementación para hacerlo "inyectable" [15].
  - **Objetivo:** Facilitar la inyección de fallos durante el diseño del hardware para verificar su comportamiento.
- **En tiempo de ejecución:**
  - **Descripción:** Se modifica la memoria de configuración del dispositivo para afectar al circuito implementado [16].
  - **Objetivo:** Introducir fallos en el circuito implementado para evaluar el sistema en condiciones operativas reales.

Técnicas basadas en simulación: Requiere de un modelo del sistema bajo estudio en el que se inyectan los fallos durante su simulación.

- **Simulación a nivel de transistor, puertas:**
  - **Descripción:** Requiere de un modelo del sistema a nivel de transistor o puertas lógicas en el que se inyectan fallos durante la simulación para analizar el comportamiento de los componentes más básicos [17].
  - **Objetivo:** Evaluar la robustez de los circuitos a nivel más bajo, asegurando unos resultados más representativos ya que esta representación es más cercana a la realidad.
- **Simulación a nivel de Lógica y RTL (*Register Transfer Level*):**
  - **Descripción:** Se utilizan simuladores de circuitos para introducir fallos en modelos en VHDL (*VHSIC Hardware Description Language*) o Verilog de los componentes del sistema [18].
  - **Objetivo:** Evaluar la propagación de fallos tanto en la lógica combinacional como en la secuencial y su impacto en los registros del sistema.
- **Simulación a nivel de sistema:**
  - **Descripción:** Emplea modelos del sistema completo para inyectar fallos durante la simulación, abarcando desde el hardware hasta el software y su interacción [19].
  - **Objetivo:** Evaluar la integridad y robustez del sistema en su conjunto, identificando puntos críticos y asegurando que todos los componentes funcionan correctamente bajo condiciones de fallo.

2.2.2 Inyección por simulación

En nuestro caso de estudio analizaremos estas técnicas basadas en simulación con mayor interés debido a que nos centraremos en este campo. En general, estas técnicas, son métodos que utilizan modelos de simulación de los componentes y circuitos del sistema para introducir fallos y analizar sus efectos, sin necesidad de realizar pruebas destructivas en el hardware físico real [20].

En este tipo de inyecciones podemos distinguir dos claros tipos, la modificación directa del código de descripción del hardware a simular y la inyección mediante órdenes del simulador. Dentro de la modificación del propio código hardware nos encontramos con 2 variantes principales:

- **Saboteadores**: Son componentes especiales que se añaden al modelo hardware originalmente descrito. Estos tienen como objetivo alterar el valor y/o las características temporales de una o varias señales cuando un fallo es inyectado. Poseen una señal de control que se activa en la inyección y pueden ser controlados, determinando su instante y duración. Gracias a esto, podemos modificar el valor que existe entre la entrada y la salida del componente.

- **Mutantes**: Estos son componentes que reemplazan a componentes existentes en nuestro modelo hardware, esto se logra añadiendo sus descripciones a estructuras del lenguaje o modificándolas. Mientras está inactivo trabaja como un componente normal. Al activarse se comporta como si el mismo estuviera en presencia de fallos.

Por otro lado, tenemos la inyección mediante órdenes. Estos dejan el componente descrito sin modificar y actúan en tiempo de simulación. Esta técnica es la más sencilla de implementar. Sin embargo, es extremadamente dependiente de la funcionalidad ofrecida por el simulador:

- Inyección mediante órdenes: Las señales pueden ser inyectadas con una secuencia de órdenes en particular para cada tipo de fallo en el propio simulador. La siguiente secuencia en pseudo código, muestra el curso de acción para la inyección de fallos transitorios en señales. Se basa en un forzado por línea de órdenes de una señal objetivo, para un valor concreto durante un tiempo determinado. Después, se libera esta señal y procedemos a observar los efectos que tiene en el sistema.

    1. *Simulate_Until [injection instant]:* Simulamos hasta el periodo donde queremos inyectar el fallo.
    2. *Modify_Signal [signal name] [fault value]:* Modificamos el valor de la señal a inyectar, en muchos simuladores se le llama a este comando `force` o `deposit`.
    3. *Simulate_For [fault duration]:* Continuamos la simulación el tiempo que queremos que dure el fallo.
    4. *Restore_Signal [signal name]:* Devolvemos el valor original de la señal que habíamos inyectado previamente, en muchos simuladores se le llama `noforce` o `release`.
    5. *Simulate_For [observation_time]:* Simulamos de nuevo para ver el comportamiento del sistema después del fallo.

Para la inyección de fallos permanentes deberíamos omitir los pasos 3 y 4, con lo que la duración sería permanente y observaríamos el sistema con este fallo inyectado. Por último, para inyectar fallos intermitentes simplemente habría que repetir el ciclo de órdenes.

En nuestra investigación nos centraremos en este último tipo de inyección por órdenes del simulador, ya que es la más accesible por su bajo coste y su nula intrusividad. Además, permite la inyección de muchos tipos de fallos en diversos puntos de manera simultánea, siendo rápidos y repetibles en caso de la realización de múltiples pruebas. Por último, recalcar los principales puntos fuertes y débiles que tiene el acercamiento a la inyección por la vía de la simulación.

Puntos fuertes:

- **No destructivas:** Permiten realizar pruebas exhaustivas sin dañar el hardware físico, reduciendo costos y riesgos asociados con pruebas físicas.

- **Flexibilidad y control:** Ofrecen un alto grado de control sobre los tipos y momentos de inyección de fallos, permitiendo escenarios de prueba detallados y específicos.

- **Repetibilidad:** Facilitan la repetición de experimentos bajo condiciones idénticas, lo cual es crucial para validar resultados y comparar la efectividad de diferentes estrategias de tolerancia a fallos.

- **Escalabilidad:** Pueden aplicarse a sistemas de diferente complejidad, desde pequeños circuitos integrados hasta sistemas completos.

Puntos débiles:

- **Modelado preciso:** La precisión de los resultados depende de la fidelidad del modelo del sistema. Un modelado inexacto puede llevar a conclusiones erróneas.

- **Recursos computacionales:** La simulación de sistemas complejos puede requerir una cantidad significativa de recursos computacionales y tiempo.

- **Cobertura de fallos:** Puede ser difícil garantizar que todos los posibles fallos se hayan cubierto en las simulaciones, especialmente en sistemas muy grandes y complejos.

2.2.3 Simulación hardware y lenguajes

Para hacer posible la verificación de nuestro sistema hardware utilizamos herramientas de simulación, estas herramientas logran reproducir el comportamiento del hardware a través de un programa software, lo que permite al usuario poder evaluar de una manera más o menos fiel el comportamiento de un sistema real a pesar de que este no exista realmente [21].

Esto evita el costo económico y temporal que conlleva un prototipo teniendo como ventajas una gran genericidad, accesibilidad y controlabilidad, intrusividad mínima y una gran capacidad de automatización y reutilización, a cambio de una pérdida de precisión comparado con el modelo real.

Pero para ello necesitamos herramientas que puedan describir nuestro hardware y su comportamiento, ahí es donde entran los HDL (*Hardware Description Language*). Estos son lenguajes especializados en la especificación de la estructura, diseño y operación de los circuitos digitales permitiendo descripciones a nivel de comportamiento, estructural o de puerta lógica [22].

En la industria actual los lenguajes HDL más utilizados son VHDL y Verilog, ambos lenguajes proporcionan un gran nivel de abstracción:

• **VHDL:** Desarrollado en los años 80 deriva de ADA, es un código estructurado, altamente tipado y menos flexible que Verilog. Este lenguaje soporta una gran variedad de tipo de datos, bibliotecas y paquetes. Su estándar es el IEEE 1076 y fue patrocinado por el departamento de defensa de EE.UU. [23].

• **Verilog:** Desarrollado en el 1984 por *Gateway Design Automation* con el estándar IEEE 1364. Se asimila al lenguaje C, con una estructura menos estricta que VHDL. Este lenguaje es muy flexible, permitiendo un gran soporte para la concurrencia y modularidad [24].

3. Tecnología utilizada

Tal y como hemos mencionado en capítulos anteriores, este trabajo se centrará en el método de inyección de fallos por software que hemos descrito como órdenes del simulador. En este apartado, describimos distintos simuladores que ocupan los lenguajes más populares de descripción de hardware que hemos mencionado anteriormente (VHDL y Verilog), los cuales dividiremos en 2 categorías.

En primer lugar, la solución comercial, es decir software propietario y de pago, teniendo una producción, distribución y soporte profesional y privado. Por otro lado, tenemos la categoría *open-source*, este código es de acceso público, por lo que se puede ver y modificar de una manera libre, esto hace que su desarrollo se vea descentralizado y que se mantiene gracias a una comunidad.

Recalcamos el punto anterior debido a la importancia que tienen las capacidades y funcionalidades del propio simulador a la hora de utilizar el método de inyección por órdenes del simulador. Por lo que cada simulador es un escenario distinto donde estamos ligados a este, ya que la capacidad de inyección dependerá en gran medida de las características y funcionalidades de cada herramienta.

3.1 Simulador comercial

Para evaluar la inyección de fallos en entornos comerciales, utilizamos Questa-Sim [25], uno de los simuladores más completos y con mayor funcionalidad en el mercado de la mano de Siemens. Questa-Sim proporciona un amplio soporte para los dos lenguajes HDL que hemos mencionado con anterioridad, VHDL y Verilog además de muchos otros.

También ofrece capacidades avanzadas para la depuración y análisis de diseño para la verificación de hardware. Además, permite la inyección de fallos a través de órdenes específicas en su consola y con interfaz gráfica, facilitando la simulación de fallos de todo tipo.

Por ende, al ser el simulador más completo lo tomaremos como base de la funcionalidad deseada para la tupla de simuladores libres y la biblioteca Cocotb. Para demostrarlo, mostraremos un sencillo ejemplo de cómo funciona una simple inyección por órdenes de simulador en un pequeño componente descrito en VHDL en este programa.

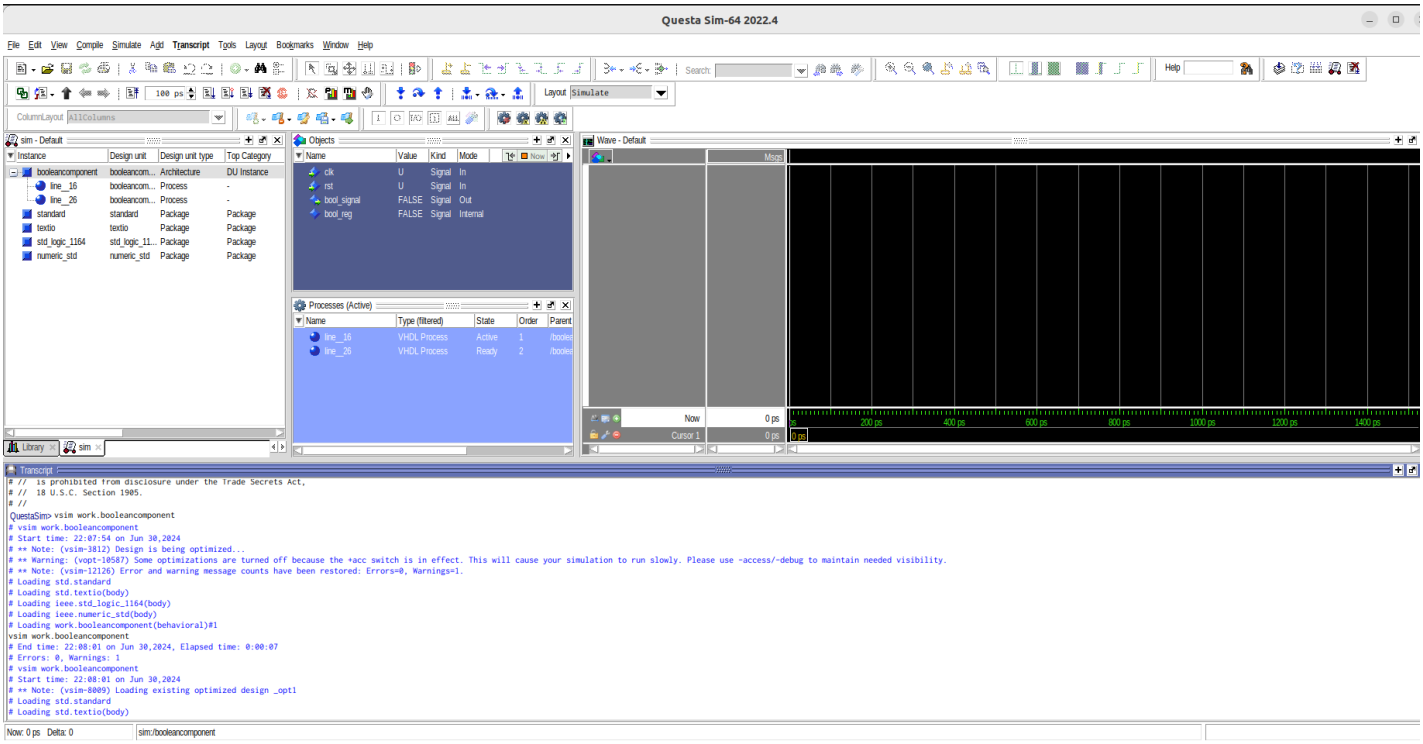

Ilustración 2 Interfaz Visual Questa-Sim SIM

En la Ilustración 2 podemos ver la interfaz gráfica del simulador Questa-Sim, con gran variedad de opciones disponibles. En la izquierda de la imagen identificamos un pequeño componente VHDL llamado *booleancomponent* mostrado en la Ilustración 4 . Este componente tiene un registro interno booleano que cambia de valor alternativamente en cada ciclo de reloj. Como podemos ver en la imagen en el apartado *Objects,* vemos todas las señales que lo forman, su valor y atributos. Al mismo tiempo, podemos arrastrarlas al apartado *Wave* para poder observar su comportamiento en forma de onda. Adicionalmente, podemos controlar sus valores en un menú desplegable que se puede abrir haciendo clic derecho sobre cada una de ellas.

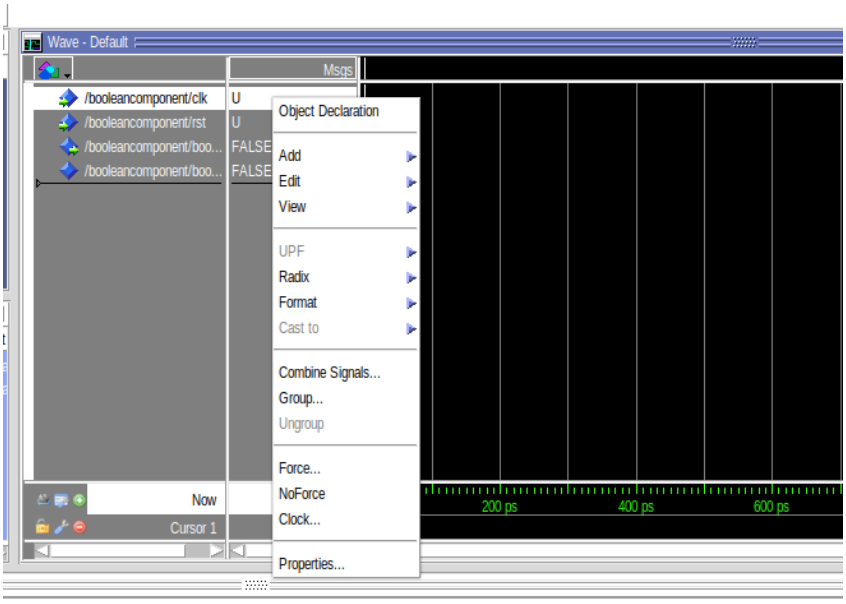

Ilustración 3 Menú de control de las señales

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity BooleanComponent is
    Port (
        clk     : in  std_logic;  -- Señal de reloj
        rst     : in  std_logic;  -- Señal de reset
        bool_signal : out boolean -- Señal de tipo boolean
    );
end BooleanComponent;

architecture Behavioral of BooleanComponent is
    signal bool_reg : boolean := false; -- Registro interno de tipo boolean
begin
    process(clk, rst)
    begin
        if rst = '1' then
            bool_reg <= false; -- Reiniciar el registro a false
        elsif rising_edge(clk) then
            bool_reg <= not bool_reg; -- Alternar el valor del registro
        end if;
    end process;

    -- Asignar el valor del registro a la salida
    bool_signal <= bool_reg;
end Behavioral;
```

Ilustración 4 Descripción del componente BooleanComponent.VHDL

Para poder empezar a observar el comportamiento del componente debemos inicializar su reloj, ya que depende de él, y asignamos como reloj a la señal con nombre *clk* en nuestro componente. Para esto el simulador nos ofrece una herramienta llamada *Clock*, como hemos visto anteriormente en la Ilustración 3. En este nuevo menú mostrado en la Ilustración 5, se despliega al abrir la opción y podemos establecer los valores del reloj, en nuestro caso, asignamos un periodo de 10ns (expresado en ps).

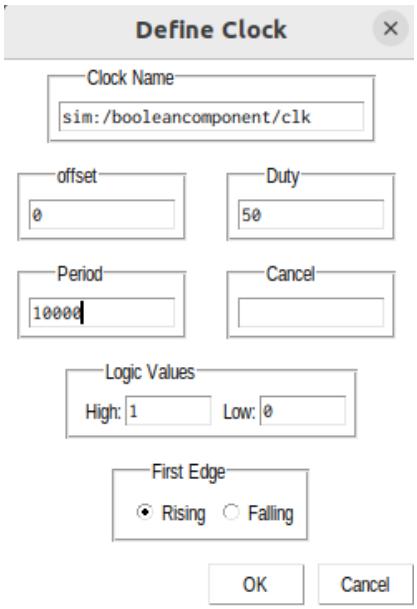

Ilustración 5 Menú de configuración para el reloj

A continuación, accedemos al menú mostrado en la Ilustración 6 con la opción *Force* que podemos ver en la Ilustración 3. Forzamos el valor de entrada de la señal *rst* a 0, esto nos permitirá tener las señales de entrada listas. A partir de este momento podremos simular el componente para ver su comportamiento antes de la inyección. En nuestro caso simularemos unos 50ns. Esto se puede conseguir con el comando "run 50ns" en la parte inferior del simulador.

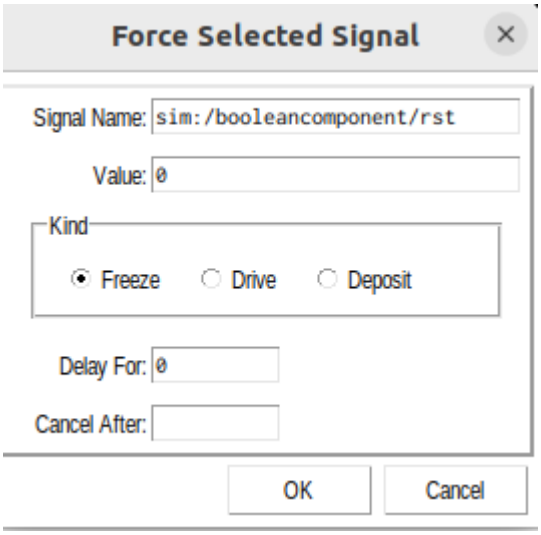

Ilustración 6 Menú de forzado de señales

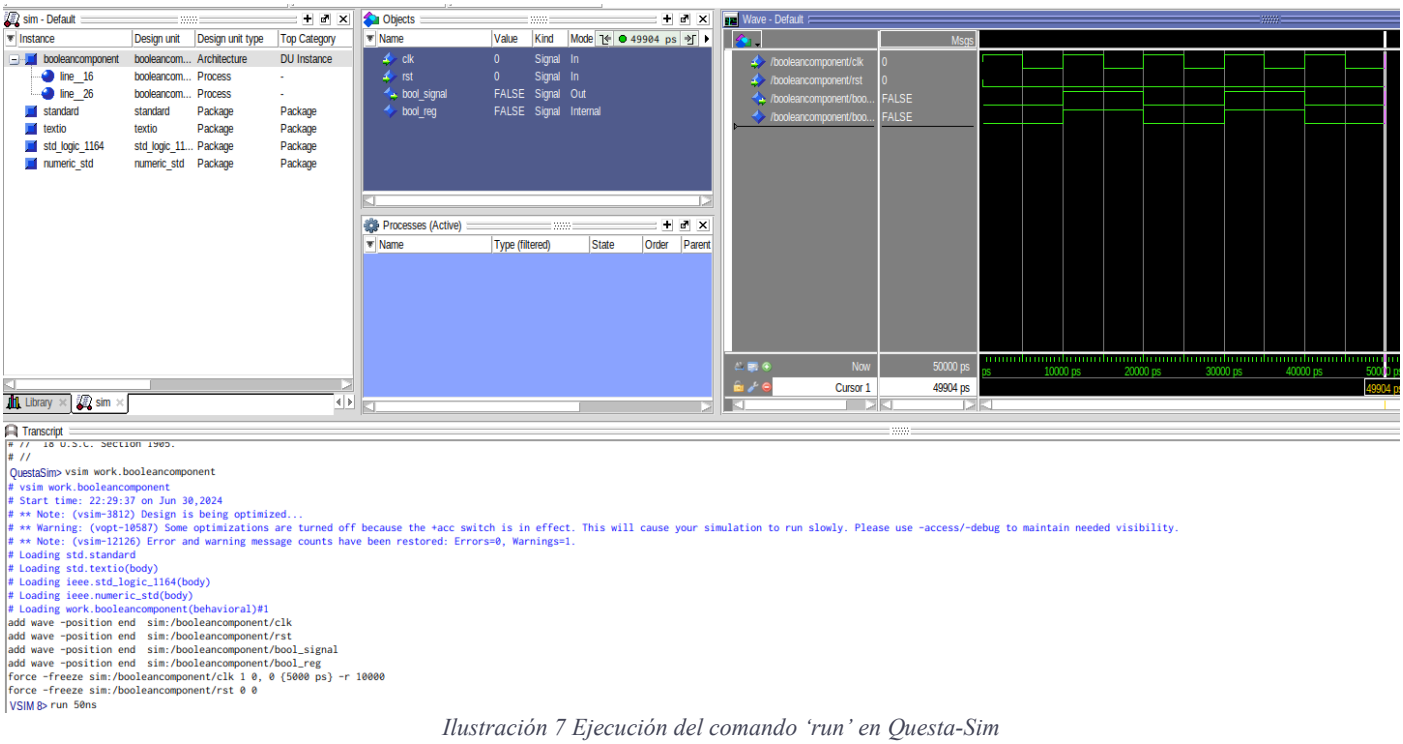

Ilustración 7 Ejecución del comando 'run' en Questa-Sim

Podemos observar en la Ilustración 7, el comportamiento del componente en el apartado de señales como formas de onda y viendo que su comportamiento es adecuado. Como sugerencia adicional hay que indicar que todos los comandos previos ejecutados gráficamente pueden ser ejecutados de la misma manera en la consola como indica la parte inferior.

Ahora que hemos asegurado el correcto funcionamiento del componente, probamos a inyectar en la señal de salida *bool_signal* un fallo transitorio. Tal y como hemos hecho para el componente de entrada, podemos forzar su valor y seguir avanzando la simulación con el comando *run* durante un tiempo y observar su comportamiento. En nuestro caso lo forzaremos a un valor TRUE y avanzaremos la simulación 20ns para ver el comportamiento.

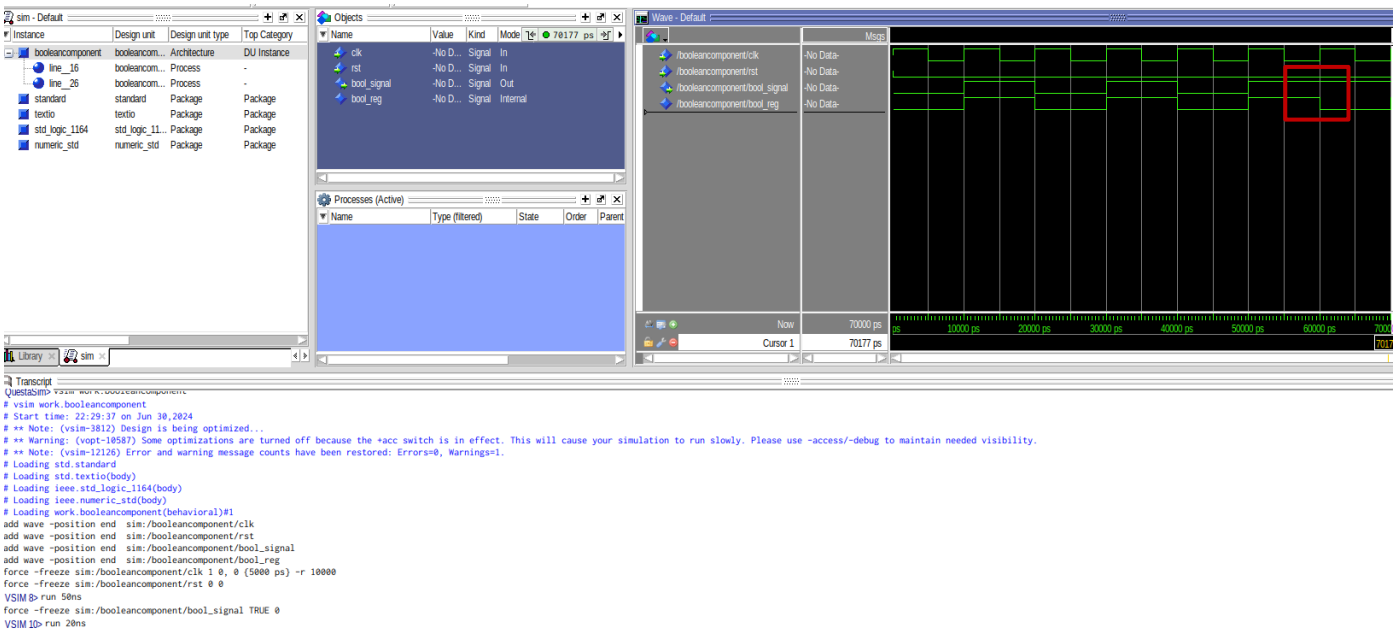

Ilustración 8 Forma de onda después de la inyección en bool_signal

En la zona donde se muestran las formas de onda a partir del nanosegundo 60 de la Ilustración 8 observamos la clara inyección del valor *TRUE* en la señal y como por ende su resultado da lugar a avería. Para hacer que esta inyección sea transitoria, debemos dejar de forzar su valor y para ello tenemos la opción *NOFORCE* que vimos en la Ilustración 3. Una vez liberada, continuamos simulando el componente durante unos nanosegundos (20) más para observar como el componente funciona con normalidad de nuevo como se muestra en la Ilustración 9.

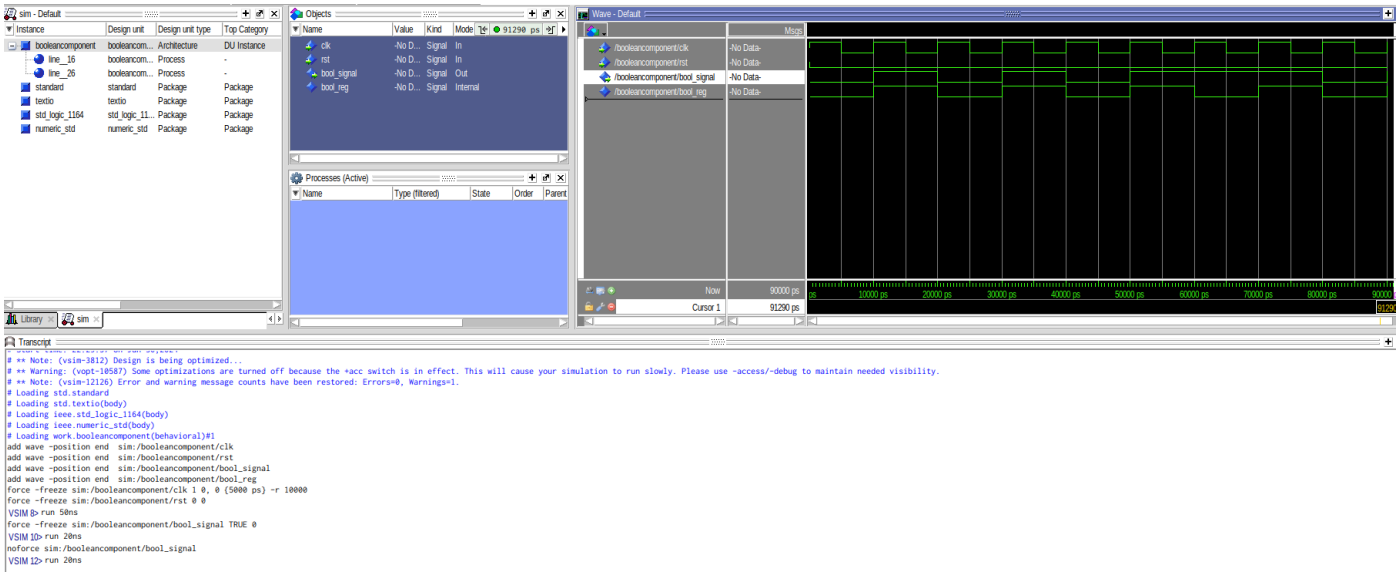

Ilustración 9 Simulación completa en Questa-Sim

Con este pequeño ejemplo dejamos clara la capacidad que tiene el simulador QUESTA-SIM para simular e inyectar fallos con suma facilidad, y al mismo tiempo nuestro estándar deseado por el resto de los simuladores libres en cuanto a funcionalidad se refiere.

3.2 Simuladores *open-source*

En el siguiente apartado examinamos algunos de los simuladores que usaremos a lo largo del trabajo, estos son simuladores de código abierto, los cuales ofrecen distintas funcionalidades y están implementados por la comunidad. Esto significa que cada una de estas herramientas tiene sus propias capacidades a las cuales nos tendremos que adaptar.

1. **GHDL:** Esta herramienta *open-source* funciona como analizador, compilador, simulador y en parte (experimental) como sintetizador para componentes hardware descritos únicamente con lenguaje VHDL. En concreto, para las versiones de 1987, 1993 y 2002 del estándar del lenguaje y parcialmente también, para las versiones de 2008 y 2019. Esta herramienta funciona utilizando compiladores como LLVM Y GCC presumiendo de ser mucho más rápido que un simulador interpretado.
   A su vez, puede escribir formas de onda en un archivo de formato vcd, fst o ghw necesitando un visor gráfico de terceros para poder visualizarlas.
   Como cualidades a destacar su funcionalidad se basa completamente por órdenes en consola (no tiene GUI propia) y soporta bibliotecas de terceros para añadir funcionalidades y corrutinas, en concreto VPI (*Verilog Procedural Interface*) / VHPIDIRECT. No soporta inyección de fallos de manera nativa. Utilizamos la versión 5.0.0-dev [26].

2. **NVC:** La herramienta NVC también *open-source* es un compilador y simulador que soporta casi todo el estándar del lenguaje VHDL del año 1993 y 2002, además soporta de manera experimental funciones del estándar para el año 2008 y 2019. Al igual que GHDL, soporta funcionalidades y corrutinas para añadir funcionalidades adicionales por terceros a través de interfaces como VPI.
   También es un simulador guiado de manera exclusiva por órdenes en consola y que también puede producir archivos donde escribir formas de onda para su posterior visualización en una herramienta visual. Cabe mencionar que su soporte para inyección de fallos por comandos de simulador es muy limitado y es experimental. Utilizamos la versión 1.12 de la herramienta [27].

3. **ICARUS VERILOG:** La herramienta icarus Verilog o iverilator, se trata de un compilador y simulador interpretado para componentes hardware descrito en el lenguaje HDL Verilog, en particular soporta las versiones de 1995, 2001 y 2005 del estándar. Como las herramientas mencionadas con anterioridad, iVerilog necesita de extensiones y bibliotecas (como VPI) para poder llevar a cabo funcionalidades adicionales como la inyección de fallos.
   Soportado para la mayoría de los sistemas operativos de sobremesa, utiliza el compilador de C++. Este simulador funciona por órdenes y es capaz de generar ficheros vcd para la visualización de ondas. Utilizamos la versión 12.0 [28].

4. **VERILATOR:** Compilado en C++ la herramienta verilator es un simulador y compilador para el lenguaje Verilog y SystemVerilog, al igual que las otras herramientas mencionadas es un simulador de código abierto, dirigido al ámbito profesional. Soporta extensiones y bibliotecas de terceros para funcionalidades extra y funciona mediante órdenes (argumentos) en consola.
   Puede hacer uso de herramientas de visualización de ondas gracias a su generación de archivos vcd, fst…Utilizaremos la versión 5.020 de la herramienta [29].

3.3 Cocotb

Debido a las limitaciones de los simuladores en cuanto a la inyección de fallos, introducimos la herramienta Cocotb **(***Coroutine based Cosimulation Testbench***)** [30]. Cocotb es un *framework* en Python que permite la creación de *testbenches* para simuladores HDL que soporten diseños de componentes en *Register Level Transfer* (nivel de abstracción donde se describe el hardware como el flujo de los datos entre registros y sus operaciones en esos datos), aprovechando la flexibilidad y potencia de Python para extender las capacidades de los simuladores.

Con su amplio soporte para simuladores más populares como: VCS, NVC, ModelSim, Questa, Xcelium, Riviera-PRO, Active-HDL, GHDL, CVC, Verilator y Icarus Verilog lo hace una herramienta perfecta para nosotros, ya que buscamos un *framework* común que incluya las funcionalidades que buscamos en los simuladores libres aportados anteriormente, en concreto para la inyección de fallos por órdenes del simulador.

Esto es posible gracias a la biblioteca GPI (*Generic Procedural Interface*) que ofrece Cocotb, una combinación de abstracción de las librerías VPI (*Verilog Procedural Interfaces*), VHPI (*VHDL Procedural Interface*) y FLI (*Foreign Language Interface. Mentor Graphics' equivalent to VHPI*). Estas API permiten la interacción directa entre los simuladores y herramientas externas como por ejemplo scripts en Python, permitiendo la manipulación de señales en la ejecución concurrente del simulador y los test de prueba creados por Cocotb.

Un test en Cocotb es simplemente una función en Python, que instancia un DUT (*Design Under Test*), el cual es el componente de mayor nivel de un diseño hardware en el simulador sobre el que estemos trabajando en ese momento. El test en Python podrá mandar estímulos en las señales, entradas y salidas del componente y monitorizarlos. Esto ocurre ya que en la ejecución tenemos tanto el avance del tiempo marcado por el simulador como las posibles funciones del código Python que estemos ejecutando, formando las llamadas "corrutinas". Para un mejor entendimiento mostramos un esquema de su funcionalidad en la Ilustración 10.

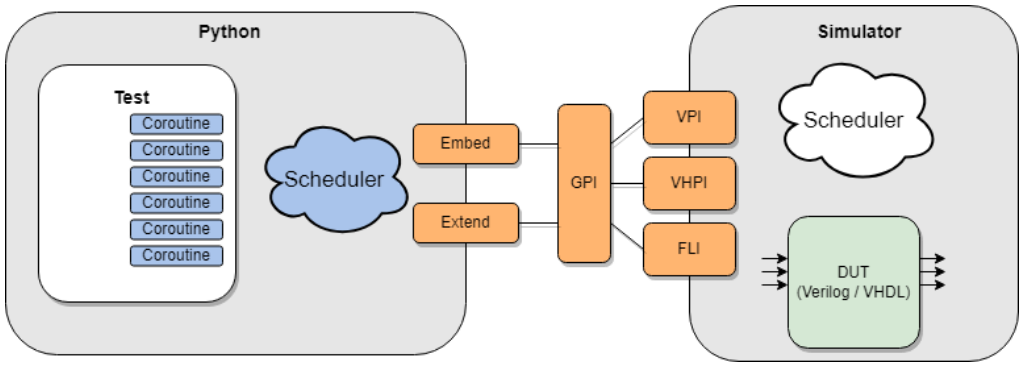

Ilustración 10 Esquema de la funcionalidad de Cocotb

A continuación, en la Ilustración 11 mostramos un simple ejemplo de un test Cocotb en Python para el componente mencionado en el apartado 3.1 (ilustración 4), donde habíamos descrito un componente VHDL en forma RTL.

```python
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer, RisingEdge
from cocotb.handle import Force, Release

async def inject_fault(signal, clk, dut, log_name):
    dut._log.info(f"Inyección de fallos en {log_name}")
    original_value = signal.value
    dut._log.info(f"Valor original en {log_name}: {original_value}")
    await Timer(10, units='ns')
    signal.value = Force(True)
    await Timer(10, units='ns')
    injected_value = signal.value
    dut._log.info(f"Valor inyectado en {log_name}: {injected_value}")
    signal.value = Release()
    await Timer(60, units='ns')
    final_value = signal.value
    dut._log.info(f"Valor final en {log_name}: {final_value}")

@cocotb.test()
async def FailTest(dut):
    # Genera señal de reloj
    cocotb.start_soon(Clock(dut.clk, 10, units='ns').start())

    # Aplica valores iniciales
    dut.rst.value = 1
    await Timer(10, units='ns')
    dut.rst.value = 0
    await Timer(10, units='ns')

    # Inyección de fallos en bool_signal
    await inject_fault(dut.bool_signal, dut.clk, dut, "bool_signal")
```

Ilustración 11 Testbench en Python llamado FailTest.py

Tal y como habíamos tratado en Questa-Sim, el script para Cocotb describe un comportamiento similar a la inyección del fallo transitorio para el componente. En este caso de la misma manera inicializamos el reloj del componente y la señal de reset. Luego, generamos la

función *inject_fault* que nos permite inyectar un fallo con valor TRUE en la señal deseada con el comando `force` y cómo podemos liberar el valor después de un tiempo con el comando `release`.

Después, continuamos simulando y observamos su comportamiento. Esta secuencia de órdenes nos debe resultar familiar ya que es la misma que se indicó como procedimiento en el apartado 2.2.2.

Para poder ejecutar el script y ver nuestros resultados necesitamos crear un pequeño *Makefile* con una serie de variables importantes para la prueba mostrado en la Ilustración 12.

```
1 SIM_ARGS += --wave=nvc.vcd --format=vcd
2 TOPLEVEL_LANG=vhdl
3 VHDL_SOURCES +=$(shell pwd)/BooleanComponent.vhdl
4 TOPLEVEL =booleancomponent
5 MODULE =ForceTest
6 include $(shell cocotb-config --makefiles)/Makefile.sim
7 #make SIM=nvc
```

Ilustración 12 Descripción del Makefile para Cocotb

En este *Makefile* podemos observar distintas variables, entre las que tenemos:

- SIM_ARGS: Donde indicamos argumentos específicos para el simulador que estamos tratando
- TOPLEVEL_LANG: Lenguaje HDL a utilizar para el simulador en cuestión.
- VHDL_SOURCES/VERILOG_SOURCES: indicamos todos los componentes descritos en lenguaje HDL que forman nuestro hardware, en caso de que haya más de uno se pueden añadir.
- TOPLEVEL: Nombramos al componente más alto en la jerarquía de nuestro hardware, en nuestro caso solo tenemos uno. Este es el DUT del test de Python.
- MODULE: El nombre de nuestro test escrito en Python.

Para poner en marcha este *makefile* en una consola procedemos a escribir 'make SIM= (nombre del simulador a utilizar)'. Podemos ver su ejecución completa en la siguiente Ilustración 13.

```
emnalo@emnalo-VirtualBox:~/Test/BooleanTest/NVC$ make SIM=nvc
rm -f results.xml
"make" -f Makefile results.xml
make[1]: Entering directory '/home/emnalo/Test/BooleanTest/NVC'
# Make sure all libs in SOURCES_VHDL_* are mentioned in VHDL_LIB_ORDER and vice versa
\
/home/linuxbrew/.linuxbrew/bin/nvc  --work=work:sim_build/work -L sim_build -a /home/emnalo/Test/BooleanTest/NVC/BooleanComponent.vhdl
rm -f results.xml
TESTCASE= COCOTB_TEST_FILTER= MODULE=ForceTest TOPLEVEL=booleancomponent TOPLEVEL_LANG=vhdl \
   /home/linuxbrew/.linuxbrew/bin/nvc  --work=work:sim_build/work -L sim_build \
  -e booleancomponent --no-save  \
  -r --load /home/emnalo/.local/lib/python3.10/site-packages/cocotb/libs/libcocotbvhpi_nvc.so  --wave=nvc.vcd --format=vcd
** Note: loading VHPI plugin
        /home/emnalo/.local/lib/python3.10/site-packages/cocotb/libs/libcocotbvhpi_nvc.so
     -.--ns INFO     gpi                                ../mbed/gpi_embed.cpp:79   in set_program_name_in_venv       Did not detect Python virtual environment. Using system-wide Python interpreter
     -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101   in gpi_print_registered_impl      VHPI registered
     0.00ns INFO     cocotb                             Running on nvc version 1.12.0
     0.00ns INFO     cocotb                             Running tests with cocotb v2.0.0.dev0+36832c15 from /home/emnalo/.local/lib/python3.10/site-packages/cocotb
     0.00ns WARNING  gpi                                VHPI: Other iterator types not implemented yet
     0.00ns INFO     cocotb                             Seeding Python random module with 1719791716
     0.00ns INFO     cocotb.regression                  Found test ForceTest.FailTest (1/1)
    20.00ns INFO     cocotb.BOOLEANCOMPONENT            Inyección de fallos en bool_signal
    20.00ns INFO     cocotb.BOOLEANCOMPONENT            Valor original en bool_signal: 0
    40.00ns INFO     cocotb.BOOLEANCOMPONENT            Valor inyectado en bool_signal: 1
   100.00ns INFO     cocotb.BOOLEANCOMPONENT            Valor final en bool_signal: 0
   100.00ns INFO     cocotb.regression                  ForceTest.FailTest passed
   100.00ns INFO     cocotb.regression                  ****************************************************************************************
                                                        ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        ****************************************************************************************
                                                        ** ForceTest.FailTest            PASS         100.00           0.00      62091.84  **
                                                        ****************************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0                100.00           0.01       7296.98  **
                                                        ****************************************************************************************
** Note: 100000001fs+0: VHPI plugin requested end of simulation
make[1]: Leaving directory '/home/emnalo/Test/BooleanTest/NVC'
```

Ilustración 13 Resultado de la ejecución del comando 'make'

También hemos indicado a nuestro simulador que genere un fichero vcd (esto será diferente para cada simulador que utilicemos y dependerá de si está implementada esta función o no, y de cómo lo hace) donde podremos observar la forma de onda generada en el proceso para un estudio más detallado. Utilizaremos el software libre GTKWave el cual ofrece una interfaz visual para poder abrir el fichero e interpretar los resultados como se muestra en la Ilustración 14.

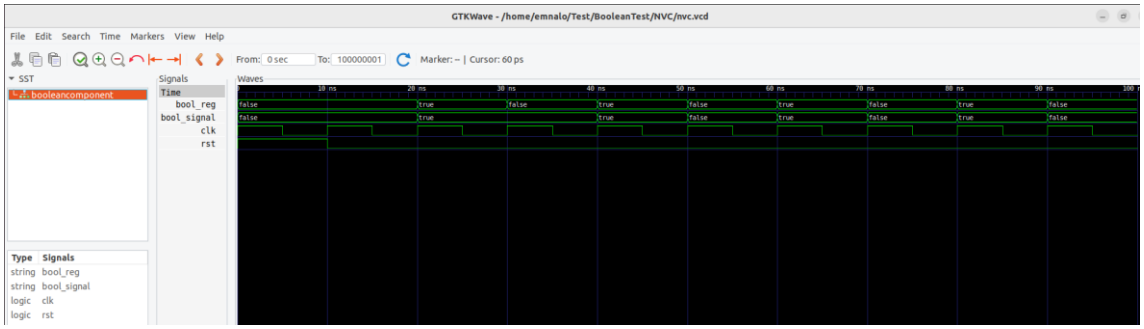

Ilustración 14 Visualización del archivo VCD

Con este pequeño ejemplo podemos concluir que Cocotb ofrece las herramientas necesarias para que simuladores que no disponen de inyección de fallos por órdenes de simulador (y aquellos que tienen también pueden utilizarlo) puedan hacer uso de sus bondades de una manera rápida, común para todos, en un lenguaje interpretado como lo es Python.

Sin embargo, nuestro objetivo principal es comprobar el alcance y compatibilidad que verdaderamente tienen los distintos simuladores con esta herramienta, ya que como hemos comentado al principio al estar implementados por la comunidad su estructura es realmente heterogénea e incluso para las mismas funcionalidades su estructura puede ser totalmente distinta.

Esto provoca un desafío a la hora de utilizar herramientas comunes por lo que queremos ver el alcance real que tienen las técnicas mencionadas anteriormente para cada uno de los simuladores tratados en el trabajo.

4. Análisis de las posibilidades de inyección de fallos mediante Cocotb

Como hemos mencionado anteriormente, es de vital importancia para nosotros comprobar la compatibilidad de la biblioteca Cocotb con los simuladores de código libre que se han mencionado a lo largo del trabajo. Esto es una variable para tener en cuenta ya que, como hemos mencionado en capítulos anteriores los simuladores tienen una funcionalidad variable y una implementación distinta, lo que los lleva a tener un comportamiento poco predecible al utilizar herramientas de terceros con ellos.

Para asegurarnos que Cocotb es compatible con nuestros simuladores vamos a realizar un pequeño estudio, donde pondremos a prueba la cobertura real que ofrece la biblioteca en varios de los simuladores que la herramienta afirma soportar.

Utilizaremos como métricas de medición 2 factores. En primer lugar, los tipos de inyección de fallo soportados, evaluando el efecto que causan las órdenes `force`, `deposit` y `release`. En segundo lugar, comprobaremos la inyección por órdenes en distintos tipos de señales, es decir, si son de entrada o salida del componente, o de tipos de datos como booleanos, enumerados, vectores…

4.1 Creación de test de cobertura

Para la medición correcta de los factores mencionados anteriormente crearemos una serie de componentes hardware sin ninguna funcionalidad en concreto, para permitirnos ejecutar pruebas en los distintos escenarios que hemos mencionado:

- *GenericComponent.VHDL/v*: Un pequeño componente jerárquico descrito en VHDL o Verilog (según lenguaje soportado por el simulador). Se compone de un inversor de señal (puerta *NOT* al valor de entrada), y de sí mismo. Este componente tiene entradas de lógica estándar, como lógicas vectoriales para un distinto número de bits, así como salidas del mismo tipo. Posee también, diversas señales internas formadas por distintos tipos de datos con el fin de probar la compatibilidad de la biblioteca, entre ellos: enumerados, enteros, *arrays*…

- *CounterComponent.VHDL/v*: Otro componente simple descrito en VHDL o Verilog en diseño RTL, que trata de probar de manera más minuciosa las diferencias de comportamiento (si existen) entre los tipos de datos enteros con signo o sin signo. Posee una señal de reloj y señales *signed* y *unsigned* que se incrementan por cada ciclo de reloj hasta 10.

- *BooleanComponent.VHDL/v*: En esta ocasión afianzamos la comprobación de los tipos booleanos en un componente hardware que posee una señal de este tipo. Esta cambia de valor *true/false* en cada ciclo de reloj de manera infinita.

Una vez creados los componentes hardware necesarios para nuestras comprobaciones, pasamos a la creación de los test en Python que hacen uso de la herramienta Cocotb para la

inyección por órdenes. Estos test probarán las distintas órdenes de inyección existentes en la biblioteca para los componentes anteriores:

- *ForceTest.py:* En este test al igual que vimos en el ejemplo de la ilustración 11, generamos una serie de comandos basados en el apartado 2.2.2 del documento. Teniendo como objetivo de diseño (DUT) uno de los componentes mencionados anteriormente, podemos utilizar los comandos `force` y `release` para la inyección por órdenes en las señales y observar su comportamiento.

- *DepositTest.py:* Por otra parte, en este test comprobamos el funcionamiento del comando `deposit` para los componentes hardware objetivo. Este comando a diferencia de los anteriores, fuerza su valor en la señal hasta que la propia señal cambie de valor tanto por estímulos externos como por procesos del propio componente sin necesidad de liberarlo.

- *FailTest.py:* Por último, utilizamos este test en Python para la inyección de múltiples fallos consecutivos en el mismo diseño hardware. Utilizando una función genérica, podemos optar por introducir fallos mediante `force y deposit` en distintas señales del componente, evaluando cómo se comportan.

4.2 Evaluación de cobertura por simulador

Una vez vistos los componentes y test que hemos realizado para los simuladores con la herramienta Cocotb, obtenemos los siguientes resultados que plasmamos en las siguientes tablas, donde evaluamos cada simulador por tipo de señal evaluada (fila) y su comportamiento (correcto o no) frente a las distintas ordenes de inyección evaluadas en los test (columnas).

- **GHDL**: En este simulador podemos observar en la Tabla 1 cómo fallan las ordenes de `deposit` y `release` después de inyectar una señal para todos los casos, al no liberar su valor forzado. Esto nos deja solo con la posibilidad de una inyección de fallos permanentes para este simulador.

Tabla 1 Evaluación cobertura GHDL

GHDL	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	NO	NO
SALIDA	SI	NO	NO
INTERMEDIA	SI	NO	NO
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	NO	NO
ARRAY	SI	NO	NO
INTEGER	SI	NO	NO
STD_LOGIC/VECTOR	SI	NO	NO
BOOLEAN	SI	NO	NO
ENUM	SI	NO	NO
MULTIPLE	SI	NO	NO

- **NVC**: Para este simulador de VHDL observamos en la Tabla 2 que la inyección de fallos tanto permanentes como transitorios será posible para la mayoría de los casos ya que las órdenes funcionan como se espera de ellas excepto para las constantes y las variables dentro de un proceso, las cuales el propio simulador no logra forzar su valor. Otro caso particular de fallo en la implementación es la liberación en un valor invalido (XXX) de las señales de entrada del componente en cualquier nivel jerárquico.

Tabla 2 Evaluación cobertura NVC

NVC	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	SI	NO
SALIDA	SI	SI	SI
INTERMEDIA	SI	SI	SI
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	SI	SI
ARRAY	SI	SI	SI
INTEGER	SI	SI	SI
STD_LOGIC/VECTOR	SI	SI	SI
BOOLEAN	SI	SI	SI
ENUM	SI	SI	SI
MULTIPLE	SI	SI	SI

- **ICARUS**: Para este simulador de Verilog observamos en la Tabla 3 un comportamiento correcto de las órdenes excepto para los casos donde se usen variables o constantes dentro de procesos o señales/variables de tipo array propio. En estos casos, el simulador imposibilita el forzado.

Tabla 3 Evaluación cobertura ICARUS

ICARUS	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	SI	SI
SALIDA	SI	SI	SI
INTERMEDIA	SI	SI	SI
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	SI	SI
ARRAY	NO	NO	NO
INTEGER	SI	SI	SI
STD_LOGIC/VECTOR	SI	SI	SI
BOOLEAN	SI	SI	SI
ENUM	SI	SI	SI
MULTIPLE	SI	SI	SI

- **VERILATOR**: El soporte para el simulador de verilator es inconsistente como se muestra en la Tabla 4, y no podemos recomendar su uso en las versiones que utilizamos debido a su comportamiento errático. Encontramos varios casos donde las señales o no pueden ser forzadas (ignoran el valor forzado y se mantienen en su valor original), o ignoran su orden de liberación manteniendo el valor de forzado.

Tabla 4 Evaluación cobertura VERILATOR

VERILATOR	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	NO	NO
SALIDA	SI	SI	NO
INTERMEDIA	NO	NO	NO
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	SI	NO
ARRAY	NO	NO	NO
INTEGER	SI	SI	SI
STD_LOGIC/VECTOR	SI	SI	NO
BOOLEAN	SI	SI	SI
ENUM	SI	NO	SI
MULTIPLE	SI	SI	NO

- **QUESTA-VHDL**: Para el simulador comercial usando lenguaje VHDL, observamos en la Tabla 5 que el forzado se hace imposible para las constantes como las variables en proceso. También, notamos fallos para las señales de tipo booleano, donde su valor de forzado no es correcto. Asumimos que es un problema en la implementación de Cocotb ya que pruebas hechas de manera nativa en el simulador funcionan correctamente.

Tabla 5 Evaluación cobretura QUESTA-VDHL

QUESTA/VHDL	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	SI	SI
SALIDA	SI	SI	SI
INTERMEDIA	SI	SI	SI
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	SI	SI
ARRAY	SI	SI	SI
INTEGER	SI	SI	SI
STD_LOGIC/VECTOR	SI	SI	SI
BOOLEAN	NO	SI	NO
ENUM	SI	SI	SI
MULTIPLE	SI	SI	SI

- **QUESTA-VERILOG**: Como en otros simuladores probados, se puede ver en la Tabla 6 que Questa para Verilog no admite inyección en constantes, variables en proceso ni para array de tipos propios del lenguaje. Al igual que en NVC debido a un fallo en la implementación de Cocotb, la liberación incorrecta de las señales de entrada del componente en cualquier nivel jerárquico dando un valor invalido (XXX).

Tabla 6 Evaluación cobertura QUESTA-VERILOG

QUESTA/VERILOG	FORCE	DEPOSIT	RELEASE
ENTRADA	SI	SI	NO
SALIDA	SI	SI	SI
INTERMEDIA	SI	SI	SI
CONSTANTE	NO	NO	NO
VARIABLE	NO	NO	NO
SEÑAL	SI	SI	SI
ARRAY	NO	NO	NO
INTEGER	SI	SI	SI
STD_LOGIC/VECTOR	SI	SI	SI
BOOLEAN	SI	SI	SI
ENUM	SI	SI	SI
MULTIPLE	SI	SI	SI

Cabe destacar que todas estas conclusiones han sido extraídas y probadas en la versión 2.0.0-dev de la biblioteca Cocotb, y las versiones de los simuladores mencionadas en el apartado 3.2 del trabajo. Es decir, son susceptibles a cambios en un futuro, ya que la compatibilidad de estas herramientas no está bajo un soporte oficial y cualquier cambio puede afectar a la funcionalidad y resultados obtenidos.

4.3 Guía de diseño

Basándonos en los resultados y las conclusiones extraídas sobre la verdadera compatibilidad y funcionalidad que ofrece Cocotb para los simuladores mencionados actualmente, escribimos este apartado. Donde daremos pequeñas indicaciones a la hora de diseñar el hardware para que pueda ser inyectable utilizando la biblioteca.

Estas indicaciones nos servirán como una guía de diseño en el caso que queramos ejecutar una inyección de fallos por órdenes de simulador de manera efectiva. a pesar de las pequeñas incompatibilidades que presentan los simuladores y Cocotb.

- **GHDL**: Si queremos usar inyección de fallos de manera efectiva junto con la biblioteca Cocotb para este simulador tenemos que asumir que los únicos fallos que podremos inyectar en él son los fallos permanentes.

- **NVC**: En este simulador de VHDL, podemos probar tanto fallos permanentes como transitorios. Sin embargo, deberemos tener en cuenta que si queremos inyectar fallos en señales de entrada deberíamos crear una señal auxiliar inyectable. Esta portaría el valor de entrada y lo llevaría a aquellos registros o componentes que la necesiten. Al igual que existen fallos de implementación en las entradas, deberíamos evitar el uso de variables dentro de procesos del componente. También, a la hora de declarar constantes debemos hacerlo como señales normales inicializadas al valor deseado, teniendo en cuenta que no debemos alterar su valor.

- **ICARUS**: Icarus también logra probar su funcionalidad en fallos permanentes y transitorios. Aun así, debido a los fallos de implementación tenemos que evitar el uso de variables internas en procesos. En cuanto a las variables de tipo array y constantes, deberíamos usar los vectores predeterminados y variables de valor fijo en su lugar.

- **VERILATOR**: En Verilator no vemos viable la inyección de fallos en las versiones actuales, el comportamiento para las señales probadas da resultados no esperados y no hay indicaciones de diseño que podamos ofrecer en este caso.

- **QUESTA-VHDL**: El simulador comercial Questa, ofrece el alcance esperado para fallos transitorios y permanentes en VHDL. Debido al fallo en la implementación de Cocotb para las señales booleanas, nos abstendremos de usarlas y en su lugar utilizaremos señales estándar con valores 0/1 para marcar *false/true*. Para el uso de constantes y variables en proceso como en el simulador NVC, deberíamos evitarlas o sustituirlas por su equivalencia en código. También podríamos sopesar la posibilidad de anular las directivas de optimización del lenguaje C en la compilación, para ver si es posible modificar estas señales.

- **QUESTA-VERILOG**: Al igual que en NVC debemos controlar el fallo de implementación a la hora de inyectar en señales de entrada. Del mismo modo, recomendamos el mismo curso de acción, donde generar una señal auxiliar inyectable directamente conectada a las entradas de los componentes. Al igual que en Icarus, debemos evitar el uso de constantes, variables en procesos y los arrays, sustituyéndolos por vectores predeterminados.

5. Metodología para la realización de campañas de inyección de fallos mediante Cocotb

Una vez conocemos los límites que tiene la biblioteca Cocotb y las capacidades que tenemos con nuestros simuladores, avanzamos en una etapa más. Hasta el momento solo nos habíamos planteado la inyección de fallos en componentes simples, con un número reducido de estos.

En la realidad estos casos no son probables, sino que en el ámbito profesional se suelen realizar campañas de inyección. Estas son un conjunto de experimentos de inyección diferentes, que cambian ciertos parámetros, como la duración, tiempo o lugar de los fallos a inyectar.

Con el objetivo de verificar el funcionamiento del hardware y poder sacar conclusiones validas y significativas, estos experimentos se realizan en grandes números, por lo que es necesario una manera de automatizarlos para poder llevarlos a cabo.

Nuestro propósito ahora es utilizar exclusivamente herramientas de código abierto con las que poder conseguir estos resultados, es decir una campaña de inyección efectiva. En este estudio seguiremos el siguiente procedimiento:

1. Ejecutaremos un experimento de referencia en ausencia de fallos para ver el comportamiento base del sistema que vamos a evaluar (*Golden Run*).
2. Ejecutaremos de manera automatizada N experimentos que nos den la confianza necesaria para evaluar el sistema.
3. Haremos un análisis de los resultados obtenidos de los experimentos, lo que nos permitirá observar el comportamiento del sistema, y sus modos de avería.

Cocotb cuando se utiliza sobre un DUT en un *testbench*, solo soporta una única ejecución. Incluso si se definen múltiples test dentro del *testbench* en Python, no hay forma de reiniciar la simulación. Ante esta limitación, se presentan tres opciones posibles:

a. Crear un bucle controlado en el propio *testbench* para ejecutar cada test en Python de manera secuencial.
b. Utilizar una corrutina concurrente con *start_soon* en el test de Python, donde se define el propio *testbench* y se considera el DUT solo como un componente. Este enfoque debe tener en cuenta los fallos permanentes.
c. Desarrollar una infraestructura que ejecute un *testbench* definido en HDL por separado, generando la compilación y simulación del modelo para cada test de manera individual. Aunque este método implica un mayor coste temporal debido a la naturaleza generativa automática, permite generar la "*Golden Run*" aparte.

Los dos primeros métodos son adecuados únicamente si el componente cuenta con una señal de reseteo para su sistema interno o si el componente no tiene un estado que deba ser "conocido" como el inicial.

5.1 Generación de experimentos con bucle

Para este primer caso utilizamos un *testbench* predefinido para el componente hardware a analizar, escrito en el propio lenguaje del sistema. Este *testbench* genera una rutina de funcionamiento para el componente. Nuestro objetivo es hacer que él mismo pueda repetirse de manera infinita en una sola simulación debido a las limitaciones que tiene la biblioteca de Cocotb.

Las implementaciones en VHDL y Verilog son distintas pero el objetivo es el mismo, nombramos como una tarea/procedimiento la rutina del *testbench* y la controlamos con una variable dentro de un bucle infinito. Esto lo hacemos para que su comportamiento sea lo más determinista posible, pudiendo controlar en qué momento se inicia la rutina. Podemos encontrar ejemplos de esto en los archivos *RegisteredALU_tbloop.vhd/v* de nuestro código.

A continuación, creamos nuestro test en Python *LoopTest.py*. Este test al igual que en ocasiones anteriores ejecutará una serie de test de inyección teniendo como objetivo el *testbench* generado anteriormente. Estos vendrán definidos en una lista de experimentos donde enumeraremos los distintos parámetros a modificar para cada experimento.

```
experiments = [
    {"signal_name": "uut.alu.a_i", "fault_type": "permanent", "fault_value": 90, "injection_time": 100, "duration": 20},
    {"signal_name": "uut.alu.a_i", "fault_type": "transient", "fault_value": 90, "injection_time": 100, "duration": 20},
    {"signal_name": "uut.alu.a_i", "fault_type": "transient", "fault_value": 90, "injection_time": 100, "duration": 20},
    {"signal_name": "uut.alu.b_i", "fault_type": "permanent", "fault_value": 20, "injection_interval": (0, 100)},
    {"signal_name": "uut.result_register.d_i", "fault_type": "transient", "fault_value": 18, "injection_interval": (0, 100), "duration": 5},
]
```

Ilustración 15 Lista de experimentos

Como podemos observar en la Ilustración 15 los parámetros a variar son: la señal a inyectar, el tipo de fallo (transitorio o permanente), el valor del fallo, el tiempo de inyección (el cual puede ser fijo o aleatorio entre un rango) y su duración. Para concluir, y poder sacar conclusiones de cada uno de los experimentos, el test guarda en un archivo xml una lista definible por el usuario de variables y su valor como se muestra en la Ilustración 16.

```
−<signals>
  −<signal>
     <nombre>uut.result</nombre>
     <valor>00000000</valor>
   </signal>
  −<signal>
     <nombre>uut.alu.add_result</nombre>
     <valor>01110011</valor>
   </signal>
  −<signal>
     <nombre>uut.result_register.aux</nombre>
     <valor>00000000</valor>
   </signal>
 </signals>
```

Ilustración 16 Lista de señales resultado en formato XML

Como nota final hay que recalcar que, en nuestra experiencia, la utilización de un bucle infinito en el lenguaje Verilog para Cocotb imposibilita la utilización de ordenes de inyección, por lo que para este lenguaje esta forma de generación no es válida.

5.2 Generación de experimentos con corrutinas concurrentes

En el siguiente caso, fusionamos el *testbench* descrito en VHDL/Verilog con los test de inyección en un solo archivo Python para Cocotb. Este archivo tiene como diseño bajo test el propio componente hardware, el cual llamaremos *StartTest.py*.

Al igual que el test en Python del anterior apartado, este test tendrá tanto una lista de experimentos con las variables a modificar, como una lista de señales a observar y escribir en xml. Sin embargo, a diferencia del anterior tendremos definida la corrutina que llevará a cabo el *tesbench* como una función asíncrona mostrada en la Ilustración 17 a continuación.

```
#Corrutina de testbench
async def testbench(dut):
    #Generar reloj
    cocotb.start_soon(Clock(dut.clk_i, 10, units="ns").start())
    #Condiciones iniciales
    dut.en_i.value = 1
    dut.rst_i.value = 0
    dut.a_i.value = 58
    dut.b_i.value = 25
    #Experimento1 Suma
    dut.operation_i.value = 2
    await Timer(10,units='ns')
    #Experimento2 Resta
    dut.operation_i.value = 6
    await Timer(10,units= 'ns')
    #Experimento3 Menor
    dut.operation_i.value = 7
    await Timer(10,units= 'ns')
```

Ilustración 17 Corrutina de Testbench descrita en Python

Esta corrutina, será llamada en cada uno de los test de inyección realizados dentro del mismo archivo con la función `cocotb.start_soon()`,la cual nos permite llamar a corrutinas que se ejecuten de manera concurrente como se muestra en la Ilustración 18.

```
# Golden run (sin fallos)
@cocotb.test(name="golden_run", skip=False)
async def golden_run(dut):
    # Esperar un tiempo suficiente para observar las señales
    cocotb.start_soon(testbench(dut))
    await Timer(40, units='ns')
    # Observar las señales y guardarlas en goldenrun.xml
    observe_signals(dut, "goldenrun.xml")
```

Ilustración 18 Ejemplo de llamada de una corrrutina concurrente en Cocotb

Debido a la propia naturaleza de este test conjunto, debemos aclarar que para la introducción de fallos permanentes debemos de realizar una liberación del forzado de la señal un poco antes del término del test. Esto se debe a que la simulación, como hemos mencionado anteriormente es única para todos los test de Cocotb. Por lo que, si no se libera este forzado, este puede propagarse y afectar a los subsiguientes experimentos.

5.3 Generación de experimentos con estructura

Por último, hablamos de la generación de experimentos mediante una estructura replicada. Este método es el menos eficiente temporalmente, debido a que recompilamos y simulamos el sistema hardware a simular. Sin embargo, es el único método en el que podemos empezar los experimentos desde un estado conocido (inicial).

Esto es de suma importancia, ya que no todos los sistemas poseen un método de reseteo de estado y es vital para que los experimentos no acarreen cambios de estado ejecutados previamente.

Para generar nuestra estructura utilizaremos un script en Shell llamado *infratest.sh.* Este script, generará carpetas distintas por cada experimento a realizar llamadas testN (donde N es el número de test). Cada una de estas carpetas contendrá un archivo de configuración *config.json*, el makefile para poder ejecutar nuestro test en Python y el propio test llamado *ParametricTest.py*.

Este test, funcionará como el test mencionado en el apartado 5.1, pero con sutiles diferencias. En primer lugar, ya no ejecutará varios test de inyección sino uno solo, correspondiente a la carpeta en la que se ubica.

La otra diferencia clave, se basa en la lectura de las variables para la ejecución del experimento. En este caso serán leídas del fichero de configuración mencionado anteriormente. Este fichero, común para todos los experimentos contendrá todas las variables para nuestros experimentos, divididos por nombre del experimento.

```
{
    "exp": {
        "test1": {
            "signal_name": "uut.alu.a_i",
            "fault_type": "transient",
            "fault_value": 90,
            "injection_time": 100,
            "duration": 20
        },
        "test2": {
            "signal_name": "uut.alu.b_i",
            "fault_type": "permanent",
            "fault_value": 4,
            "injection_time": 100
        },
        "test3": {
            "signal_name": "uut.alu.a_i",
            "fault_type": "transient",
            "fault_value": 33,
            "injection_interval": [0, 100],
            "duration": 15
        }
    }
}
```

Ilustración 19 Ejemplo de configuración en JSON de los experimentos.

Estas configuraciones mostradas en la Ilustración 19 serán leídas por el script, y solo serán ejecutadas si su nombre corresponde al del experimento ejecutado en la misma carpeta. Esto nos dejará con archivos de resultados xml independientes para cada experimento, los cuales serán extraídos de cada carpeta para su posterior análisis por el script en Shell mencionado con anterioridad.

5.4 Conclusiones

Tras presentar las distintas metodologías para generar experimentos en una campaña de inyección de fallos, concluimos que cada una es adecuada según la situación requerida. A continuación, se resume las condiciones para la aplicación de cada metodología:

- **Generación en bucle:** Este método de generación de experimentos nos será útil si ya trabajamos sobre un *testbench* en HDL para nuestro sistema hardware. Eso sí, tendremos que estar dispuestos a modificarlo con tal de poder controlarlo externamente. Otro requisito para poder llevar a cabo esta metodología es que el sistema hardware no requiera empezar desde un estado inicial conocido, o tenga presente una señal de reseteo para poder controlarlo.
- **Generación con corrutinas:** Al igual que la metodología anterior, la generación con corrutinas exige que el componente hardware con el que tratamos posea una señal de reset o su funcionamiento empiece siempre desde un estado inicial conocido. Pero al contrario que la metodología anterior, esta no requiere de un *testbench* previo en el lenguaje HDL del sistema a tratar. Esto se debe a que tratamos tanto el test de inyección como el del *workload* del componente en un solo script.
- **Generación con infraestructura:** La metodología con infraestructura es la más costosa computacional y temporalmente de las mencionadas hasta ahora, ya que es la única que, por cada test, compila, analiza y simula el componente entero. Sin embargo, esto trae el beneficio de no necesitar que el sistema tenga una señal de reset o saber su estado interno, ya que siempre empezamos desde el punto de compilación. Eso sí, al igual que la generación en bucle debemos de estar preparados para trabajar tanto con un test de inyección como con un *testbench* por separado en HDL.

En la siguiente Tabla 7 podemos ver un resumen más visual de las conclusiones obtenidas hasta el momento:

Tabla 7 Conclusiones para metodologías de generación de experimentos

CONDICIONES	LOOP	START_SOON	INFRAESTRUCTURA
USO DE TESTBENCH EN VERILOG/VHDL	**SI**	**NO**	**SI**
MODIFICACIÓN DEL TESTBENCH EN VERILOG/VHDL	**SI**	**NO**	**NO**
NECESIDAD DE *RESET* EN EL COMPONENTE HARDWARE	**SI**	**SI**	**NO**
COSTE COMPUTACIONAL	**NORMAL**	**NORMAL**	**ALTO**

6. Resultados de campañas de inyección reales

Basándonos en los modelos de fallo de la sección 2.1, y en las conclusiones para nuestra metodología de generación de experimentos, procedemos a crear nuestras propias campañas de inyección. Esto nos permitirá una mayor comprensión de los simuladores, así como de la biblioteca Cocotb. Además, pondremos a prueba en un escenario realista, todos los conocimientos que hemos adquirido en anteriores capítulos.

6.1 Objetivos de inyección y *workloads*

En primer lugar, presentamos nuestros objetivos de inyección y la carga de trabajo que ejecutan. Los componentes hardware objetivos serán procesadores 8 bits, los cuales son manejables para nuestra capacidad de cómputo, pero lo suficientemente complejos para darnos un entendimiento realista. Los presentamos a continuación:

- **MC8051**: Este procesador es un núcleo IP (*Intellectual Property*) de parte de Oregano Systems [31] y la carga de trabajo seleccionada consta de una multiplicación de matrices. Cuenta con una pequeña RAM y ROM. Almacena su resultado en el array llamado RES. El componente simula su comportamiento durante 2ms con un periodo de reloj de 100MHz.
- **PIC**: Este microcontrolador CMOS de Microchip Technology [32], posee una memoria ROM y ejecuta un *bubblesort* de los números 9 al 0 ordenándolos a la inversa. Este proceso se puede ver en los registros r8-r17 del mismo, durando aproximadamente 9us en simulación con un periodo de reloj de 100MHz.
- **Usimplez**: Este procesador teórico de Gregorio Fernandez [33], calcula la serie de Fibonacci en un bucle infinito hasta desbordar su memoria. Posee una pequeña memoria RAM la cual almacena su resultado desde la posición 101 de manera ascendente hasta desbordar. Nosotros calcularemos la serie Fibonacci 11, lo que dura aproximadamente 5us en simulación con un periodo de reloj de 100MHz.

6.2 Carga de fallos (*faultload*)

Con tal de emular las amenazas reales que pueden sufrir nuestros sistemas objetivo, creamos un conjunto de fallos en base a su duración a través del tiempo. Estos experimentos serán creados utilizando la generación mediante infraestructura presentada en el apartado 5.3:

- **Permanentes**: Dónde inyectamos fallos permanentes, al inicio de la simulación del objetivo, con el valor pertinente hasta el final de la simulación.
- **Transitorios**: Dónde inyectamos fallos transitorios, en un intervalo aleatorio de tiempo de inyección y duración.

Una vez localizado el conjunto de fallos que queremos estudiar , procedemos a crear las cargas de fallos correspondientes para nuestra campaña de inyección en cada procesador. Para ello utilizamos 2 modelos de fallo que son representativos de los fallos mencionados anteriormente.

- ***Stuck-at-(0,1)****:* En esta carga de fallos, forzaremos cada bit de todas las señales influyentes en la carga de trabajo de nuestro objetivo al valor correspondiente durante toda la duración de esta. Esta inyección será representativa de los fallos permanentes que puedan ocurrir en nuestro sistema. Además, cubrimos todas las posibilidades de fallo al ser nuestra muestra de experimentación igual al total posible.

- ***Bit-flip/Pulse***: En esta carga de fallos, forzamos con el valor inverso en el momento de la inyección para cada uno de los bits influyentes de nuestro objetivo. Esta inyección tendrá un tiempo de inicio aleatorio, pero para ser representativos el rango de aleatoriedad propuesto será un poco después del inicio del *workload* y un poco antes de la finalización de este.

  Sobre un 10% del tiempo total del *workload*, esto se hace para omitir los tiempos de inicio y finalización de la carga de trabajo donde la probabilidad de que un fallo afecte al sistema es mínima.

  Debido a que tenemos en cuenta tanto el tiempo de fallo como su localización (bits de las señales), el espacio de fallo es demasiado grande para nosotros, por lo que optamos por una inyección estadística, tomando una pequeña muestra del total de posibilidades que sea representativa. En nuestro caso, tomando en cuenta las limitaciones temporales y de potencia de nuestro sistema, asumimos un margen de error del 5% para un nivel de confianza estándar del 95%.

  Finalmente, consideraremos el diferente comportamiento que presentan los fallos en la lógica combinacional y secuencial. En la lógica combinacional, donde las salidas dependen únicamente de las entradas, los fallos suelen manifestarse como pulsos (Pulse). En cambio, en la lógica secuencial, que incluye registros controlados por señales de reloj, los fallos típicos son del tipo Bit-flip.

  Para el modelo Bit-flip (lógica secuencial) realizaremos inyecciones con el comando de Cocotb `deposit`, lo que permitirá su liberación en cuanto su valor cambie en el ciclo de reloj. Para los fallos del modelo Pulse (lógica combinacional) los forzaremos durante una corta duración para luego liberarlo. Para que esta duración sea representativa, asumimos que su duración sea de 1 ciclo de reloj (10ns).

6.3 Observación

Para poder observar el comportamiento del sistema y determinar su reacción, analizaremos las salidas de nuestros objetivos tras cada inyección. Estas salidas contendrán el resultado de la carga de trabajo, la cual compararemos con un experimento inicial sin fallos (*GoldenRun*). Analizando el comportamiento de los objetivos bajo nuestras campañas, podemos observar sus salidas/resultados y sacar conclusiones sobre sus modos de avería:

- **Fallos enmascarados**: Hemos inyectado un fallo, pero la avería no se ha manifestado, por lo que el estado del sistema puede que no sea correcto pero su funcionamiento no se ha visto afectado y sus salidas son las mismas que en la *GoldenRun* . Para diferenciar estos fallos de los fallos latentes, en el que el estado interno es incorrecto, deberíamos monitorizar el estado de los registros además de las salidas. Debido a la gran carga computacional que resulta para nuestro sistema, solo lo mencionaremos en este apartado y no indagaremos en más profundidad.
- **Averías**: Hemos inyectado un fallo y la avería se ha manifestado, obteniendo un resultado no deseado y un funcionamiento incorrecto.

6.3.1 Consideraciones adicionales

Mencionamos que, a la hora de obtener resultados, hemos intentado utilizar tanto el simulador comercial Questa, como los simuladores de código abierto NVC e Icarus. Los cuales, en las pruebas preliminares nos dieron un mejor resultado. Sin embargo, encontramos ciertos problemas y consideraciones que debemos mencionar al indagar y tener mejor control sobre la herramienta Cocotb.

- Al tratar con modelos mucho más complejos que los usados en nuestras pruebas de compatibilidad, encontramos que con el simulador Icarus, no podemos inyectar en ciertos tipos complejos.
  En particular los tipos que manejan la interfaz común de Cocotb, GPI. Esta interfaz maneja los tipos convencionales de HDL como enteros, vectores, etc., y los convierte en tipos enumerados propios de manera interna. Muchos de ellos no soportan forzado/liberación bit a bit, como el tipo *GPI_NET*. Esto nos imposibilita su uso por el momento, hasta una futura actualización de la biblioteca o el simulador.
- Al mismo tiempo, en nuestro análisis sobre objetivos de mayor complejidad, observamos que Cocotb no puede manejar/llamar objetos de tipo *HierarchyArray*. Estos objetos son el resultado de la simulación de estructuras generativas en HDL. Para poder ser inyectable, hemos modificado nuestro procesador MC8051 para evitar su uso.
- Por último, debemos discutir la liberación de las señales multidimensionales con la función `release`. Según nuestras pruebas, esta liberación en las versiones a nuestro alcance resulta imposible, ya que exhibe un error de segmentación. Esto imposibilita nuestra técnica para la inyección de fallos transitorios para este tipo de señales. Como en el apartado anterior, modificamos los procesadores para evitar el uso de estas señales y que puedan ser inyectables.

A continuación, ofrecemos tablas ilustrativas con los resultados obtenidos en nuestra campaña para cada carga de fallos teniendo en cuenta las técnicas trabajadas y las consideraciones anteriormente mencionadas.

6.3.2 Stuck-at-0

Tabla 8 Resultados faultload (Stuck-at-0)

Lógica	Procesador	Simulador	Tiempo de ejecución (minutos)	Fallos inyectados	Fallos Enmascarados	Averías
Combinacional	MC8051	NVC	611.31	9358	63,52%	36,46%
		QUESTA	726	9358	63,52%	36,46%
	PIC(Verilog)	QUESTA	54.10	718	44,57%	55,43%
	PIC(VHDL)	NVC	27.26	718	44,57%	55,43%
		QUESTA	58.43	718	44,57%	55,43%
	usimplez_16bits	NVC	5.78	151	49,67%	50,33%
		QUESTA	11.56	151	49,67%	50,33%
Secuencial	MC8051	NVC	42.21	613	78,30%	21,70%
		QUESTA	50.31	613	78,30%	21,70%
	PIC(Verilog)	QUESTA	30.20	335	79,70%	20,30%
	PIC(VHDL)	NVC	13	335	79,70%	20,30%
		QUESTA	32.20	335	79,70%	20,30%
	usimplez_16bits	NVC	18.48	509	76,23%	23,77%
		QUESTA	34.80	509	76,23%	23,77%

En la siguiente Tabla 8 se presentan los resultados de la carga de fallos permanentes para los diferentes procesadores evaluados. Aunque todos los procesadores muestran una alta tasa de averías, el MC8051 destaca claramente en términos de fallos enmascarados combinacionales, posicionándose como el más “fiable” de nuestra selección. En la lógica secuencial, las averías han sido mucho menos manifiestas en todos los procesadores, poseyendo un nivel similar en todos ellos.

Es importante señalar que los resultados obtenidos reflejan una inyección exhaustiva de fallos en todas las localizaciones posibles (bits). La obtención del mismo resultado para todos los simuladores nos asegura que los resultados han sido consistentes para todos ellos, confirmando el funcionamiento esperado.

6.3.3 *Stuck-at-1*

Tabla 9 Resultados faultload (Stuck-at-1)

Lógica	Procesador	Simulador	Tiempo de ejecución (minutos)	Fallos inyectados	Fallos Enmascarados	Averías
Combinacional	MC8051	NVC	614.55	9358	59,77%	40,20%
	MC8051	QUESTA	727.89	9358	59,77%	40,20%
	PIC(Verilog)	QUESTA	54	718	29,11%	70,89%
	PIC(VHDL)	NVC	27.23	718	29,11%	70,89%
	PIC(VHDL)	QUESTA	58.26	718	29,11%	70,89%
	usimplez_16bits	NVC	5.80	151	23,18%	76,82%
	usimplez_16bits	QUESTA	11.45	151	23,18%	76,82%
Secuencial	MC8051	NVC	42.31	613	65,91%	34,09%
	MC8051	QUESTA	50.39	613	65,91%	34,09%
	PIC(Verilog)	QUESTA	30.36	335	60,00%	40,00%
	PIC(VHDL)	NVC	12.98	335	60,00%	40,00%
	PIC(VHDL)	QUESTA	32.48	335	60,00%	40,00%
	usimplez_16bits	NVC	18.45	509	22,99%	77,01%
	usimplez_16bits	QUESTA	34.81	509	22,99%	77,01%

En la siguiente Tabla 9 al igual que en la carga de fallos anterior, observamos un elevado nivel de averías en todos los objetivos. El procesador MC8051, sigue su liderazgo como el "más fiable" de nuestros objetivos en la lógica combinacional, incluso más ahora que en esta carga de fallos observamos una tasa de averías aún mayor. En la lógica secuencial, observamos un cambio de tendencia, donde antes los procesadores estaban parejos, ahora existe un gran incremento de averías en el procesador usimplez.

Señalar una vez más que estos resultados son resultado de una inyección exhaustiva, donde se ha inyectado en todas las localizaciones (bits), relacionadas con el *workload* de los objetivos. En vista de que los resultados han sido idénticos para todos los simuladores, confirmamos su funcionamiento esperado una vez más.

6.3.4 *Bit-flip/Pulse*

Tabla 10 Resultados faultload (Bit-flip/Pulse)

Lógica	Procesador	Simulador	Tiempo de ejecución (minutos)	Fallos inyectados	Fallos Enmascarados	Averías
Combinacional/Pulse	MC8051	NVC	289.42	4428	97,38%	2,21%
	MC8051	QUESTA	343.8	4428	97,15%	2,48%
	PIC(Verilog)	QUESTA	53.37	718	65,74%	34,26%
	PIC(VHDL)	NVC	38.21	718	65,60%	33,98%
	PIC(VHDL)	QUESTA	55.10	718	68,25%	31,75%
	usimplez_16bits	NVC	15.32	453	80,79%	19,21%
	usimplez_16bits	QUESTA	29.49	453	83,44%	16,56%
Secuencial/Bit-flip	MC8051	NVC	42.09	613	86,95%	12,07%
	MC8051	QUESTA	50.34	613	89,07%	9,95%
	PIC(Verilog)	QUESTA	20.31	335	68,66%	31,34%
	PIC(VHDL)	NVC	18.95	335	67,46%	32,54%
	PIC(VHDL)	QUESTA	22.18	335	66,57%	33,43%
	usimplez_16bits	NVC	17.31	509	46,37%	53,44%
	usimplez_16bits	QUESTA	33.16	509	49,31%	50,49%

En primer lugar, es importante señalar que los resultados obtenidos son producto de una inyección estadística, por lo que los valores en la tabla presentada pueden variar dentro de un margen de error del 5%. Dado el alto índice de averías, este margen es aceptable. Si no alcanzamos este margen de error con un 95% de confiabilidad, aumentaremos la muestra de experimentos, como en el caso del procesador usimplez en la lógica combinacional.

Para esta carga de fallos, observamos en la Tabla 10 una variabilidad en los resultados para cada uno de los simuladores dentro de nuestras expectativas. También, observamos una fiabilidad claramente superior para el procesador MC8051 en ambas lógicas. Debido al bajo ratio de averías que posee este procesador en su lógica combinacional, aumentamos la precisión en el margen de error al 1%.

6.3.5 Análisis general

Tras analizar todas las cargas de fallo y sus resultados, llegamos a las siguientes conclusiones:

- Aunque el procesador MC8051 mostró mejores resultados en las pruebas, ninguno de los procesadores evaluados es adecuado para su uso en sistemas críticos, como los sistemas de tiempo real o embebidos. Esto se debe a las altas tasas de fallos observadas tras la inyección de errores. Para hacerlos aptos para estos sistemas, se recomienda implementar mecanismos de tolerancia a fallos en ambas lógicas, como la Redundancia Modular, los códigos de detección y corrección de errores, o la duplicación de componentes.

- El simulador de código abierto NVC ejecuta los experimentos de manera significativamente más rápida. Sin embargo, el procesador comercial QUESTA, aunque más lento, ofrece soporte para ambos lenguajes HDL utilizados. La elección entre ambos queda a criterio del usuario según sus necesidades.

7. Conclusiones

Este TFM, ha demostrado la viabilidad de utilizar herramientas *open-source* para la inyección de fallos por órdenes del simulador, en modelos de hardware descritos mediante HDL. A través de la implementación de una metodología basada en la biblioteca Cocotb, se ha logrado dotar a simuladores de código libre de capacidades de inyección para la evaluación de sistemas hardware.

A lo largo del proyecto, se ha probado el alcance y compatibilidad de las herramientas utilizadas. Además, se han explorado diversas metodologías para la generación de experimentos y se ha evaluado su eficacia en diferentes escenarios. Las campañas de inyección de fallos realizadas han permitido el análisis exhaustivo de los sistemas objetivo, pudiendo sacar conclusiones claras sobre los mismos y las herramientas utilizadas en el proceso.

Los resultados obtenidos muestran que es posible realizar pruebas exhaustivas de tolerancia a fallos de manera más económica y accesible. El uso de herramientas *open-source* tiene grandes beneficios, al reducir los costos y dar mayor libertad con una comunidad que da soporte continuo. Por el contrario, también hemos encontrado peculiaridades y errores de compatibilidad debido a la naturaleza abierta de las herramientas y la complejidad creciente de los diseños evaluados a lo largo del trabajo.

Futuras investigaciones podrían expandir este trabajo para abarcar otros lenguajes de descripción de hardware (como SystemC) y explorar nuevas técnicas de inyección de fallos (como saboteadores y mutantes), ampliando así el alcance y la aplicabilidad de las herramientas desarrolladas a más tipo de fallos, sistemas objetivo más complejos, distintos simuladores, etc.

7.1 Trabajo Futuro

Para futuros trabajos, se sugiere explorar las siguientes áreas:

1. **Ampliación de simuladores y herramientas**: Evaluar la compatibilidad y funcionalidad de Cocotb con otros simuladores de código abierto y comerciales no considerados en este trabajo, para ampliar el abanico de herramientas disponibles para la inyección de fallos.
2. **Automatización y optimización**: Desarrollar scripts y herramientas adicionales que automaticen la configuración y ejecución de experimentos de inyección de fallos, optimizando el tiempo y recursos necesarios para realizar campañas extensas.

3. **Experimentación compleja:** Realizar experimentos con sistemas de creciente complejidad para evaluar y afinar las funcionalidades de la biblioteca Cocotb. Esto permitirá identificar limitaciones y áreas de mejora en el manejo de sistemas más avanzados, optimizando su aplicación en escenarios reales de alta complejidad.

4. **Evaluación continua**: Analizar de manera continua las funcionalidades y compatibilidades de la biblioteca de Cocotb con los simuladores de código abierto mencionados. Debido a la naturaleza de las herramientas de código abierto, las cuales no dejan de evolucionar. Su soporte, funcionalidad y compatibilidad pueden variar con el tiempo.

8. Bibliografía

[1] J. C. Laprie, «Dependable computing and fault-tolerance,» *Digest of Papers FTCS-15,* vol. 10, nº 2, p. 124, 1985.

[2] B. W. Johnson, « Design and analysis of fault-tolerant systems for industrial applications,» de *Fehlertolerierende Rechensysteme / Fault-tolerant computing systems*, Springer, 1989.

[3] A. Avizienis, J.-C. Laprie, B. Randell and C. Landwehr, «Basic concepts and taxonomy of dependable and secure computing,» *IEEE Transactions on Dependable and Secure Computing,* vol. 1, nº 1, pp. 11-33, 2004.

[4] NELSON, V.P., & CARROL, «Fault-tolerant computing (A tutorial),» de *AIAA Fault Tolerant Computing Workshop*, 1982.

[5] D. Project, « ETIE2: Dependability Benchmarking for Computer Systems,» 2002.

[6] P. P. Alfredo Benso, «Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation,» de *Frontiers in Electronic Testing Series*, Springer, 2003, p. 256.

[7] G. e. a. Juez Uriagereka, «Design-time safety assessment of robotic systems using fault injection simulation in a model-driven approach,» *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C),* pp. 577-586, 2019.

[8] J. Carreira, H. Madeira, J. G. Silva, « RIFLE: A general purpose pin-level fault injector,» de *Dependable Computing*, 1994.

[9] E. Keren, S. Greenberg, N. M. Yitzhak, D. David, N. Refaeli, A. Haran, «Characterization and mitigation of single-event transients in Xilinx 45-nm SRAM-based FPGA,» *IEEE Transactions on Nuclear Science,* vol. 66, nº 6, pp. 946-954, 2019.

[10] S. Ordas, L. Guillame-Sage, P. Maurine, «Electromagnetic fault injection: The curse of flip-flops,» *Journal of Cryptographic Engineering,* vol. 7, nº 3, pp. 183-197, 2016.

[11] Rodriguez, J., Baldomero, A., Montilla, V., & Mujal, J., «Lateral laser fault injection attack,» IT Labs, Applus+ Laboratories, 2023.

[12] D. Skarin, R. Barbosa, J. Karlsson, «GOOFI-2: A tool for experimental dependability assessment,» de *Proceedings of the International Conference on Dependable Systems & Networks*, 2010.

[13] J. Arlat, J.-C. Fabre, M. Rodriguez, «Dependability of COTS microkernel-based systems,» *IEEE Transactions on Computers,* vol. 51, nº 2, pp. 138-163, 2002.

[14] J. Carreira, H. Madeira, J. G. Silva, «Xception: A technique for the experimental evaluation of dependability in modern computers,» *IEEE Transactions on Software*

Engineering, vol. 24, nº 2, pp. 125-136, 1998.

[15] P. Civera, L. Macchiarulo, M. Rebaudengo, M. Sonza Reorda, M. Violante, «New techniques for efficiently assessing reliability of SOCs,» *Microelectronics Journal,* vol. 34, nº 1, pp. 53-61, 2003.

[16] L. Antoni, R. Leveugle, B. Feher, «Using run-time reconfiguration for fault injection applications,» *IEEE Transactions on Instrumentation and Measurement,* vol. 52, nº 5, pp. 1468-1473, 2003.

[17] G. S. Choi, R. K. Iyer, « FOCUS: An experimental environment for fault sensitivity analysis,» *IEEE Transactions on Computers,* vol. 41, nº 12, pp. 1515-1526, 1992.

[18] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, «Fault injection into VHDL models: The MEFISTO tool,» de *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1994.

[19] J. K. Goswami, «DEPEND: A simulation-based environment for system level dependability analysis,» *IEEE Transactions on Computers,* vol. 46, nº 1, pp. 60-74, 1997.

[20] Gil, D., Baraza, J.C., Gracia, J., Gil, P.J, «VHDL simulation-based fault injection techniques,» de *Fault injection techniques and tools for embedded systems reliability evaluation*, A. &. P. P. Benso, Ed., Springer, 2003.

[21] « Simulación - Wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/Simulaci%C3%B3n. [Último acceso: 23 Junio 2024].

[22] «Wikipedia - Lenguaje de descripción de hardware,» [En línea]. Available: https://es.wikipedia.org/wiki/Lenguaje_de_descripci%C3%B3n_de_hardware. [Último acceso: 23 Junio 2024].

[23] M. Shahdad, «An Overview of VHDL Language and Technology,» de 23rd ACM/IEEE Design Automation Conference, Las Vegas, 1986.

[24] D. Thomas y P. Moorby, The Verilog® Hardware Description Language, Springer Science & Business Media, 2008.

[25] Siemens, «QuestaSim Fact Sheet FS-85329-DS».

[26] T. Gingold y Contributors, «GHDL User Guide,» [En línea]. Available: https://ghdl.github.io/ghdl/. [Último acceso: 30 Junio 2024].

[27] N. Gasson y Contributors, «NVC - VHDL Compiler and Simulator,» [En línea]. Available: https://www.nickg.me.uk/nvc/index.html. [Último acceso: 30 Junio 2024].

[28] S. Williams y Contributors, «Icarus Verilog,» [En línea]. Available: https://steveicarus.github.io/iverilog/index.html. [Último acceso: 30 Junio 2024].

[29] W. Snyder y Contributors, «Veripool - Verilator,» [En línea]. Available: https://veripool.org/verilator/documentation/. [Último acceso: 30 Junio 2024].

[30] S. Hodgson, C. Higgs y Contributors, «COroutine based COsimulation TestBench,» [En línea]. Available: https://docs.cocotb.org/en/latest/. [Último acceso: 30 Junio 2024].

[31] Oregano Systems, «MC8051 IP Core User Guide».

[32] Microchip Technology Inc., «PIC16F87X Data Sheet».

[33] OpenCores, «uSimpleZ: Open-source simple RISC CPU core,» [En línea]. Available: https://opencores.org/projects/usimplez. [Último acceso: 10 Agosto 2024].