Proyecto Fin de Carrera
Grado de Ingeniería en Tecnologías Industriales

# Estudio sobre inyección de fallos en simulaciones Verilog/VHDL utilizando Cocotb y software libre

Autor: Jose Luis Delgado Delgado
Tutor: Hipólito Guzmán Miranda

**Dpto. de Ingeniería Electrónica**
**Escuela Técnica Superior de Ingeniería**
**Universidad de Sevilla**

Sevilla, 2020

Proyecto Fin de Carrera
Ingeniería Industrial

# Estudio sobre inyección de fallos en simulaciones Verilog/VHDL utilizando Cocotb y software libre

Autor:

Jose Luis Delgado Delgado

Tutor:

Hipólito Guzmán Miranda

Profesor titular

Dpto. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2020

Proyecto Fin de Carrera: Estudio sobre inyección de fallos en simulaciones Verilog/VHDL utilizando Cocotb y software libre

Autor: Jose Luis Delgado Delgado

Tutor: Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

*A mi familia*

# Agradecimientos

Mis más sinceros agradecimientos a mis padres, por haberme aguantado todos estos años de carrera, animándome en los momentos difíciles y estando siempre ahí para cualquier cosa. Sin su apoyo este trabajo jamás hubiera sido posible.

También quiero expresar mi agradecimiento al cuerpo docente, en especial al departamento de ingeniería electrónica y a Hipólito, mi tutor. Ellos me descubrieron una vocación y me señalaron por donde seguir en mi carrera profesional. Un saludo para todos ellos.

*Jose Luis Delgado Delgado*

*Sevilla, 2020*

# Resumen

En este trabajo vamos a realizar una prueba de concepto en la que determinaremos si el programa Cocotb se presenta como una buena opción para el ámbito de la inyección de fallos. Para trabajar con Cocotb nos apoyaremos siempre en programas de software libre.

Comenzaremos explicando una serie de conceptos previos necesarios para el correcto entendimiento del proyecto, pasando por el diseño digital y a inyección de errores y cerrando con las plataformas de software libres y el pilar fundamental del proyecto, Cocotb.

Continuaremos con un capítulo de instalación y uso de los programas y scripts usados, para que todo aquel que quiera usar esta plataforma de inyección de fallos encuentre una curva de aprendizaje menos pronunciada, a la vez que explicamos cómo vamos a usar dichas herramientas en nuestro proyecto.

A todo esto le seguirá el capítulo de puesta a prueba del inyector de errores de Cocotb. En este capítulo propondremos una serie de sistemas a los que le realizaremos inyecciones de fallos específicas y aleatorias. Nuestro objetivo será la realización de un análisis objetivo del funcionamiento de Cocotb como inyector, siendo siempre críticos con los resultados.

Esto último se recogerá en las conclusiones finales, donde discutiremos si es una herramienta verdaderamente útil para la inyección de fallos o si requiere de un mayor desarrollo para poder ser usada.

# Abstract

In this work we are going to carry out a proof of concept in which we will determine if the program “Cocotb” is presented as a good option for the field of fault injection. To work with Cocotb we will always rely on free software programs.

We will begin by explaining a series of previous concepts necessary for the correct understanding of the project, going through digital design and fault injection itself, closing with free software platforms and the fundamental pillar of the project, Cocotb.

We will continue with a chapter on the installation and use of the programs and scripts used, so that anyone who wants to use this platform finds a less pronounced learning curve, at the same time that we explain how we are going to use these tools in our project.

This will be followed by the Cocotb fault injector test chapter. In this chapter we will propose a series of systems to which we will perform injections of specific and random faults. Our objective will be to carry out an objective analysis of the operation, always being critical with the results.

All this will be reflected in the final conclusions, where we will discuss whether it is a truly useful tool for fault injection or whether it requires more development.

# Índice

**Agradecimientos** **9**

**Resumen** **11**

**Abstract** **13**

**Índice** **15**

**Índice de Tablas** **18**

**Índice de Figuras** **20**

**Capítulo 1: Introducción y conceptos previos** **23**
1.1 Orígenes del proyecto 23
1.2 Diseño Digital 24
1.2.1 Conocimientos básicos 24
1.2.2 Lenguajes de descripción hardware 25
1.3 Inyección de fallos 26
1.3.1 ¿Qué es la inyección de fallos? 26
1.3.2 Estudios en la industria 29
1.4 Herramientas de sowftware libre 30
1.4.1 ¿Qué es el software libre? 30
1.4.2 Cocotb 31

**Capítulo 2: Instalación** **35**
2.1 Instalación 35
2.1.1 Programas y requisitos 35
2.1.2 Comandos de instalación 36
2.1.3 Uso básico de programas 37
2.1.3.1 Icarus Verilog 37
2.1.3.2 GHDL 40
2.1.3.3 Gtkwave 41
2.1.3.4 Vcddiff 42
2.1.3.5 Cocotb 43
2.1.3.5.1 Verilog 43
2.1.3.5.1 VHDL 45
2.1.3.5.1 Testbenches sin Python 46

**Capítulo 3: Uso del inyector** **49**
3.1 Uso básico 49
3.2 Uso de herramientas auxiliares proporcionadas 51
3.2.1 Funcionamiento conjunto: Requisitos y modos 52
3.2.2 Funcionamiento específico 52
3.2.2.1 Config 53
3.2.2.2 Test.py 55
3.2.2.3 Gen.py 58
3.2.2.4 Org.py 59

**Capítulo 4: Inyección en distintos dispositivos** **62**
4.1 DUTs y objetivos 62
4.2 Inyección de fallos 63
4.2.1 Shifter 63
4.2.2 UART 65
4.2.3 Delay Timer 67
4.2.4 FIFO 68

**Capítulo 5: Conclusiones finales** **72**

**Referencias** **75**
**Anexo** **77**

# ÍNDICE DE TABLAS

Tabla 1: Ventajas e inconvenientes de los diferentes tipos de inyección de fallos 28
Tabla 2: Beneficios de los principales proveedores de software 31

# ÍNDICE DE FIGURAS

Ilustración 1: Ejemplo diseño digital 24
Ilustración 2: Comparacion de uso Verilog/VHDL 25
Ilustración 3: Esquema radiación iones pesados 29
Ilustración 4: Esquema interferencia electromagnética 29
Ilustración 5: Esquema FT-UNSHADES 30
Ilustración 6: Esquema de funcionamiento de Cocotb 32
Ilustración 7: Uso de programas: Código Icarus Verilog 38
Ilustración 8: Uso de programas: Comandos Icarus Verilog 38
Ilustración 9: Uso de programas: Código (corregido) Icarus Verilog 39
Ilustración 10: Uso de programas: Resultados Icarus Verilog 39
Ilustración 11: Uso de programas: Código GHDL 40
Ilustración 12: Uso de programas: Comandos GHDL 40
Ilustración 13: Uso de programas: Resultados GHDL 41
Ilustración 14: Uso de programas: Comandos Gtkwave 41
Ilustración 15 Uso de programas: Resultados Gtkwave 42
Ilustración 16: Uso de programas: Comandos y Resultados Vcddiff 42
Ilustración 17: Uso de programas: Makefile básico para simulaciones con Cocotb en Verilog 43
Ilustración 18: Uso de programas: Test básico para simulaciones con Cocotb en Verilog 44
Ilustración 19: Uso de programas: Resultados de la simulación de ejemplo (Cocotb y Verilog) 45
Ilustración 20: Uso de programas: Makefile básico para simulaciones con Cocotb en VHDL 45
Ilustración 21: Uso de programas: Test básico para simulaciones con Cocotb en VHDL 45
Ilustración 22: Uso de programas: Resultados de la simulación de ejemplo (Cocotb y VHDL) 46
Ilustración 23: Uso de programas: Makefile básico para simulaciones con Cocotb sin test en Python 46
Ilustración 24: Uso de programas: Test en Python (para testbenches en HDL) 47
Ilustración 25: Uso de programas: Resultado de la simulación de ejemplo (Cocotb sin test en Python) 47
Ilustración 26 : Uso de programas: Funciones de inyección de fallos de Cocotb 49
Ilustración 27: Uso de programas: Test del ejemplo de inyección de fallos 50
Ilustración 28: Uso de programas: Resultado del ejemplo de inyección de fallos 50
Ilustración 29: Uso de programas: Resultado del ejemplo de inyección de fallos (señal interna forzada) 51
Ilustración 30: Archivos requeridos y generados por Cocotb 53
Ilustración 31: Esquema de funcionamiento Config 54
Ilustración 32: Variables del test.py prediseñado 55
Ilustración 33: Esquema funcionamiento corutinas 56

Ilustración 34: Esquema funcionamiento del test.py prediseñado 57
Ilustración 35: Esquema funcionamiento gen.py 59
Ilustración 36: Esquema funcionamiento org.py 60
Ilustración 37: Variables introducidas en la simulación del Shifter 64
Ilustración 38: Resultado (sin inyección) de la simulación del Shifter 64
Ilustración 39: Resultado (con inyección) de la simulación del Shifter 64
Ilustración 40: Variables introducidas en la simulación de la UART 65
Ilustración 41: Resultado (sin inyección) de la simulación de la UART 66
Ilustración 42: Resultado (con inyección) de la simulación de la UART 66
Ilustración 43: Variables introducidas en la simulación del Delay Timer 67
Ilustración 44: Resultado (sin inyección) de la simulación del Delay Timer 67
Ilustración 45: Resultado (con inyección) de la simulación del Delay Timer 68
Ilustración 46: Variables introducidas en la simulación de la FIFO 69
Ilustración 47: Resultado (sin inyección) de la simulación de la FIFO 69
Ilustración 48: Resultado (con inyección) de la simulación de la FIFO 69

# CAPÍTULO 1: INTRODUCCIÓN Y CONCEPTOS PREVIOS

En este primer capítulo de nuestro proyecto, haremos una breve introducción de todos los temas que vamos a tocar a lo largo del mismo. Explicaremos los orígenes y objetivos de nuestro trabajo, hablaremos brevemente del diseño digital y de lenguajes de descripción hardware, pues serán la base de este proyecto de evaluación de capacidades. Seguiremos con una breve descripción de lo que es la inyección de fallos y cómo se trabaja en la universidad, y finalizaremos hablando de las plataformas de software libres y de porqué nuestro trabajo está basado sobre estas, haciendo especial hincapié en Cocotb, pilar de este proyecto.

## 1.1 Orígenes del Proyecto

Este proyecto surge de unas dos ideas que se mezclaron en el momento justo. En un principio, el objetivo de este tfg era realizar un estudio sobre simulación mixta VHDL y Verilog usando como intermediario Cocotb. Este concepto, surgido de una conversación casual entre dos profesionales del sector en un chat público se convirtió en una gran idea para realizar un tfg de investigación. Sin embargo, tras un mes de trabajo con el tema, quedó claro que la infraestructura estaba demasiado por desarrollar para que fuera realizable en este tipo de trabajo.

Ante dicha situación, y para no desperdiciar el trabajo realizado, se decidió seguir trabajando con Cocotb, pero el tema del proyecto quedaba por determinar. Este paréntesis de incertidumbre no duró demasiado, pues en una tutoría improvisada se planteó el concepto de la inyección de fallos. Cocotb es un programa de software libre con unas características únicas, tal y como veremos a lo largo del trabajo, y la oportunidad de inyectar fallos mediante el mismo abría un sinfín de posibilidades para desarrolladores. Tanto es así que esta idea se convirtió en el tema central del tfg. Como era de esperar, surgieron una serie de problemas a la hora de intentar realizar esto. El concepto de "inyección de fallos" en Cocotb era algo no implementado en ese momento, por lo que el primer objetivo se convirtió en conseguir que Cocotb pudiera realizar este tipo de simulaciones. Esto derivó en una serie de meses de trabajo de investigación puro en los que se intentó por todos los medios inyectar un error "single-event" mediante un script de Cocotb. La situación alcanzo el clímax cuando los propios desarrolladores de Cocotb incluyeron esta función que estábamos intentando desarrollar en una de las frecuentes actualizaciones de su programa, dejando nuestro trabajo completamente obsoleto.

Volvíamos a tener otro proyecto no presentable. Aun así, se decidió seguir trabajando con Cocotb, y la solución al problema consistió en poner a prueba dicha actualización de inyección de fallos. Con los conocimientos previos de Cocotb nos propusimos mejorar el propio funcionamiento del programa y comprobar si de verdad es algo que se pueda utilizar para el desarrollo de diseños digitales. Este trabajo es el resultado de dicha proposición.

## 1.2 Diseño Digital

### 1.2.1 Conocimientos básicos

Los sistemas digitales son dispositivos de conmutación que operan en solo uno de los dos estados posibles en un momento dado, pero que pueden cambiarse de un estado a otro a muy alta velocidad (millones de veces por segundo). Los dos estados son alto voltaje (HV) y bajo voltaje (LV). Los niveles de LV y HV generalmente se toman como 0 V y 2 a 5 V, respectivamente (dentro de un circuito lógico CMOS común al menos). Desarrollando este principio básico se ha conseguido alcanzar a diseñar circuitos de una complejidad inimaginable hace unas décadas. Todo aparato electrónico digital actual funciona con este concepto, aunque la ejecución del mismo varía [1]:

Actualmente, un diseñador puede implementar un diseño digital para una aplicación en concreto mediante dos métodos distintos: programando un microprocesador o creando un circuito digital personalizado, lo que es conocido como “Diseño digital”. Esto es fácilmente entendible mediante un ejemplo. Pongamos que queremos diseñar una aplicación simple que consiste en encender una lampara cada vez que se detecta movimiento en una habitación oscura. Este ejemplo tan simple solo necesita de 3 señales distintas. Llamemos “a” a la señal del detector de movimiento, que marca 1 cuando ha detectado algo y 0 si no lo ha hecho. La señal del sensor luminoso la definiremos como “b”, y marcara un 1 cuando haya luz en la habitación y 0 cuando esta esté oscura. Por último, tenemos una salda “F”, encargada de encender o apagar la lampara.

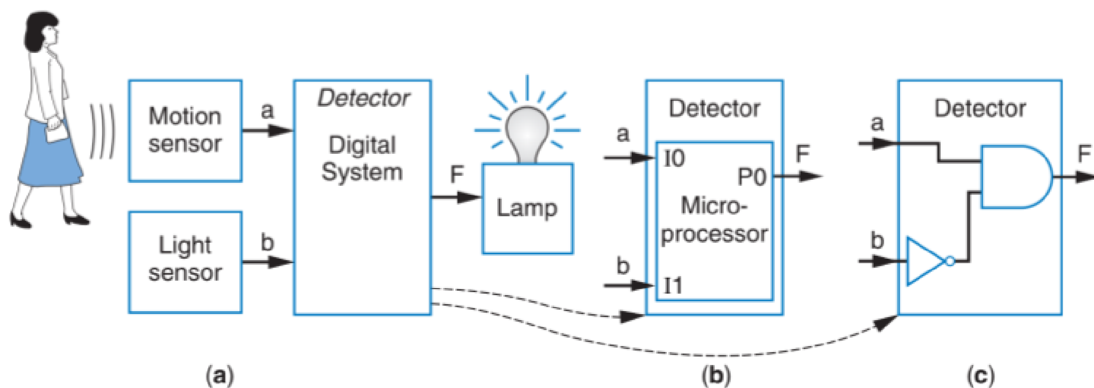


Ilustración 1: Ejemplo diseño digital [1]

El problema de diseño aparece cuando tenemos que definir que poner dentro del bloque “Detector”. Este bloque recibe las señales “a” y “b” como entradas y genera un valor de “F” como salida, de forma que la luz debería encenderse cuando se detecte movimiento y la habitación esté oscura. En este ejemplo, este circuito es fácilmente implementable mediante un diseño digital, pues, como se ve en la figura es simplemente una implementación de una puerta AND. Aun así, un diseñador siempre tiene la opción de realizar nuestra aplicación programando un microprocesador.

### 1.2.2 Lenguajes de descripción hardware

Entender el concepto del diseño digital es algo muy sencillo a niveles básicos, pero los circuitos actuales son infinitamente más complejos que una puerta AND, e imposibles de diseñar a nivel de puertas lógicas. Esto no es un concepto nuevo de esta década. Ya en 1990 los diseñadores descubrieron que eran mucho más productivos si trabajaban en un nivel más alto de abstracción, especificando solo la función lógica y permitiendo que una herramienta de diseño asistido por computadora (CAD) produjera las puertas optimizadas. Las especificaciones

generalmente se dan en lo que se conoce como un lenguaje de descripción de hardware (HDL), siendo los dos principales lenguajes de descripción de hardware de nuestra era SystemVerilog y VHDL, de principios similares, pero sintaxis diferentes. Verilog fue desarrollado por Gateway Design Automation como un lenguaje propietario para la simulación lógica en 1984 mientras que VHDL fue desarrollado originalmente en 1981 por el Departamento de Defensa, para describir la estructura y función del hardware. En comparación con SystemVerilog, VHDL es más detallado y engorroso, pero ambos idiomas son totalmente capaces de describir cualquier sistema de hardware, y ambos tienen sus peculiaridades [2].

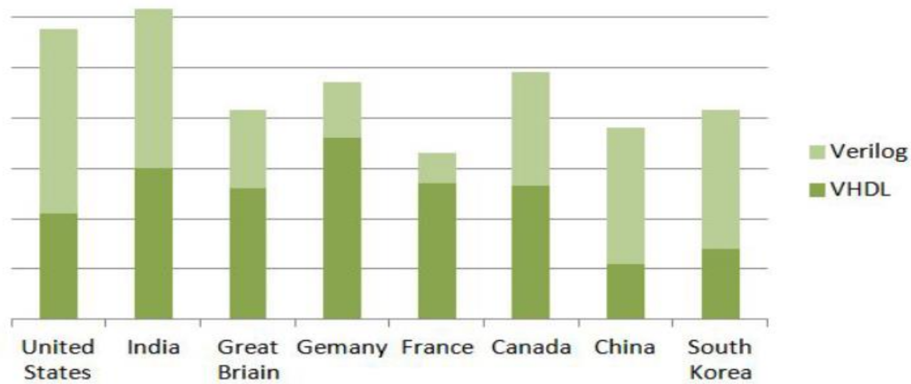


Ilustración 2: Comparacion de uso Verilog/VHDL [3]

El mejor lenguaje para usar sería el que se estuviera utilizando en su área de trabajo o el que exijan los clientes. A pesar de esto, la mayoría de las herramientas CAD actuales permiten mezclar los dos idiomas, de modo que los diferentes módulos se pueden describir en diferentes idiomas. Esta elección entre un lenguaje y otro afecta hasta a las universidades, las cuales están divididas casi por igual en cuál de estos idiomas debe enseñarse. La industria tiende hacia SystemVerilog, pero muchas compañías aún usan VHDL y muchos diseñadores necesitan tener fluidez en ambos. Debido a esta situación, a lo largo de este trabajo usaremos Verilog y VHDL por igual, dando ejemplos y explicaciones para ambos lenguajes.

Dejando de lado la controversia entre VHDL y Verilog, estos dos HDL son lenguajes modulares, es decir, basan su funcionamiento en la descripción y conexión de bloques de hardware con diferentes inputs y outputs conocidos como módulos. La funcionalidad de un módulo se puede describir en "comportamiento" y "estructura" (behavioral y structure). Los modelos de comportamiento describen lo que hace un módulo y los modelos estructurales describen cómo se construye un módulo a partir de piezas más simples. Dentro del comportamiento de cada módulo podemos definir una serie de procesos, ya sean combinacionales o de reloj. Lo más importante que debes recordar cuando estás escribiendo HDL es que el código que estás describiendo es hardware real, no un programa de ordenador.

Los dos propósitos principales de los leguajes de descripción hardware son la simulación lógica y la síntesis. Durante la simulación, las entradas se aplican a un módulo y las salidas se verifican para verificar que el módulo funcione correctamente. Durante la síntesis, la descripción textual de un módulo se transforma en puertas lógicas. La simulación lógica es una buena forma de probar un sistema en un ordenador antes de convertirlo en hardware. Los simuladores permiten verificar los valores de las señales dentro del propio sistema, cosa que puede ser imposibles de medir en un hardware físico. Es por esto que el concepto de inyección de fallos se convierte en algo de vital importancia en el momento que hablamos de HDLs. La facilidad de su implementación y las ventajas que presentan junto con el uso de HDLs es algo que se manifestará a lo largo de este proyecto [2].

## 1.3 Inyección de Fallos

### 1.3.1 ¿Qué es la inyección de fallos?

Todo sistema imaginable en la actualidad funciona con un software determinado. Ya sea dentro del ámbito de la salud, el transporte o las comunicaciones, todo sistema electrónico con un mínimo de complejidad se basa en software. Es por esto que la confiabilidad de los sistemas de software es algo de vital importancia a la hora de diseñar un dispositivo. Este concepto algo que generalmente pasa desapercibido ante la multitud, pues todo el mundo asume que un dispositivo electrónico estropeado ha fallado debido a un problema con el hardware. Aun así, los fallos de software, coloquialmente llamados "*bugs*", están reconocidos por una serie de estudios como una causa recurrente en los fallos de computadores. Tanto es esto que muchos de estos bugs provocaron grandes pérdidas económicas e incluso la pérdida de vidas humanas. Este problema de los fallos con el software es algo que se va agravando a medida que pasa el tiempo. Los sistemas son cada vez más complejos, realizan más funciones y son comercializados cada vez más rápido, esto aumenta en gran medida una posible eliminación de errores.

Es por esto que asegurar un sistema robusto ante fallos es recomendado por los estándares de la industria e iniciativas de investigación varias. La inyección de fallos es una de las soluciones planteadas a la hora de garantizar estos sistemas robustos [4].

Podemos definir la inyección de fallos como una técnica de validación de la robustez/confiabilidad de los sistemas tolerables a fallos que consiste en la realización de experimentos controlados donde la observación del comportamiento del sistema en presencia de errores es inducida explícitamente por la introducción de fallos en el sistema. Como la introducción a este apartado da a entender, a lo largo de este proyecto trabajaremos con inyección de fallos en software. Pero la inyección de fallos es un concepto mucho más amplio. Las técnicas principales se recogen en cinco categorías distintas [5]:

- Inyección de fallos basada en hardware: Se realizan alteraciones a nivel físico. Perturbando el hardware con factores del entorno (radiación de iones, interferencias electromagnéticas, etc.), inyectando caídas de voltaje en los rieles de potencia del hardware (perturbaciones de la fuente de alimentación), modificación del valor de los pines del circuito, etc.
- Inyección de fallos basada en software: Como su nombre indica, el objetivo de esta técnica consiste en reproducir a nivel de software una serie de errores que provoquen el mal funcionamiento del sistema.
- Inyección de fallos basada en simulación: Puede definirse también como una inyección de errores en software. En esta categoría es donde incluimos la inyección de errores realizada en este proyecto. Esta técnica consiste en inyectar los errores en modelos de alto nivel (con mayor frecuencia, modelos VHDL/Verilog). Permite evaluar tempranamente la confiabilidad del sistema cuando solo hay disponible un modelo del sistema. Aborda diferentes niveles de abstracción mediante el uso de distintos lenguajes de descripción. Debe proporcionarse un entorno coherente para favorecer la interoperabilidad entre los sucesivos niveles de abstracción e integrar la validación en el proceso de diseño.
- Inyección de fallos basada en emulación: Esta técnica se ha presentado como una solución alternativa para reducir el tiempo dedicado a las campañas de inyección de fallas basadas en simulación. Se basa en la exploración del uso de FPGAs para acelerar la simulación de fallas y explota dichos FPGAs para una emulación de circuito efectiva. Esta técnica puede permitir al diseñador estudiar el comportamiento real del circuito en el entorno de la aplicación, teniendo en cuenta las interacciones en tiempo real.
- Inyección de fallos híbrida: En esta categoría se mezclan la inyección de fallos en hardware y la inyección de fallos en software.

Cada una de estas cinco categorías tiene sus ventajas en inconvenientes. Este es un tema muy amplio de abarcar que da de por si para un proyecto nuevo, por lo que, en este caso, las resumiremos en esta tabla:

| Técnicas | Ventajas | Inconvenientes |
| --- | --- | --- |
| **Hardware** | - Puede acceder a ubicaciones a las que es difícil acceder por otros medios.<br>- Alta resolución temporal para activación y monitorización de hardware.<br>- Muy adecuado para modelados de fallos de bajo nivel.<br>- No intrusivo.<br>- Los experimentos son rápidos.<br>- No se requiere desarrollo o validación del modelo.<br>- Capaz de modelar errores a nivel de pin. | - Tiene un alto riesgo de daños para el sistema inyectado.<br>- El alto nivel de integración de dispositivos, los circuitos híbridos de múltiples chips y las tecnologías de empaquetado limitan la accesibilidad a esta técnica.<br>- Baja portabilidad y observabilidad.<br>- Conjunto limitado de puntos de inyección y conjunto limitado de fallos inyectables.<br>- Requiere hardware de propósito especial para realizar los experimentos |
| **Software** | - Puede ser dirigido a aplicaciones y sistemas operativos.<br>- Los experimentos se pueden ejecutar casi en tiempo real.<br>- No requiere ningún hardware especial. Baja complejidad, poco desarrollo y bajo coste de implementación.<br>- No se requiere desarrollo o validación del modelo.<br>- Se puede ampliar para nuevas clases de fallos. | - Conjunto limitado de instantes para inyectar.<br>- No puede inyectar fallos en ubicaciones inaccesibles para el software.<br>- Requiere una modificación del código fuente para admitir la inyección de fallos.<br>- Observabilidad y controlabilidad limitada.<br>- Muy difícil de modelar fallos permanentes. |
| **Simulación** | - Puede soportar todos los niveles de abstracción del sistema.<br>- No intrusivo.<br>- Permite control total de los modelos de errores y los mecanismos de inyección.<br>- No requiere ningún hardware | - Requiere de un desarrollo extenso.<br>- Consumo de tiempo (duración del experimento).<br>- El modelo no está disponible fácilmente.<br>- La precisión de los resultados |

| | | |
|---|---|---|
| | especial.<br>- Permite el máximo de observabilidad y controlabilidad.<br>- Permite realizar evaluaciones de confiabilidad en diferentes etapas del proceso de diseño.<br>- Capaz de modelar fallos transitorios y permanentes. | depende del modelo utilizado.<br>- No es posible la inyección de fallos en tiempo real en un prototipo.<br>- El modelo puede no incluir ninguna de los errores de diseño que pueden estar presentes en el hardware real. |
| **Emulación** | - Tiempo de inyección más rápido comparado con las técnicas basadas en simulación.<br>- El tiempo de experimentación puede reducirse implementando el "input pattern" en el FPGA. Estos patrones ya se conocen cuando el circuito a analizar es sintetizado. | - La descripción inicial de VHDL debe ser sintetizable y estar optimizada para evitar la necesidad de un emulador demasiado grande y costoso y para reducir el tiempo total de ejecución durante la campaña de inyección.<br>- El coste de un sistema de emulación de hardware general es muy alto.<br>-Una placa FPGA dedicada a emulación es muy compleja.<br>- La emulación solo se usa para analizar las consecuencias funcionales de un fallo.<br>- Cuando se utiliza una placa de desarrollo basada en FPGA, la principal limitación se convierte en el número de E / S del hardware programable.<br>- Necesidad de un enlace de comunicación de alta velocidad entre la computadora host y la placa de emulación |

Tabla 1: Ventajas e inconvenientes de los diferentes tipos de inyección de fallos [5]

### 1.3.2 Estudios en la industria

Como bien hemos dicho en el apartado anterior, la inyección de errores es un tema de vital importancia en lo que al diseño web se refiere. Es por esto que a lo largo de los años se han ido desarrollando más y más técnicas para asegurar que los diseños testeados estén a prueba de fallos. La complejidad de algunas técnicas de inyección de errores alcanza la complejidad de muchos diseños en sí mismo. En este apartado, veremos algunos ejemplos, tanto para hardware como software, de la inyección de errores en la industria [6]:

• **Radiación de iones pesados:** Los experimentos de inyección de fallos con radiación de iones pesados (HI, por sus siglas en inglés) se llevaron a cabo en la Universidad Tecnológica de Chalmers en Göteborg, Suecia. Como su nombre indica, este método de inyección de fallos por hardware basa su funcionamiento en la radiación. Los iones pesados de una fuente Californium-252 se puede usar para inyectar perturbaciones “single-event” en ubicaciones internas en circuitos integrados (CI) utilizando una cámara de vacío en miniatura. En la siguiente figura podemos observar la sección transversal de dicha cámara de vacío en miniatura. Los pines del CI objetivo se extienden a través de la placa inferior de la cámara de vacío, para que la cámara junto con el circuito pueda ser enchufados directamente en el zócalo del CI bajo prueba. La cámara de vacío contiene un obturador controlado eléctricamente, que se utiliza para proteger el circuito bajo prueba de la radiación durante el arranque.

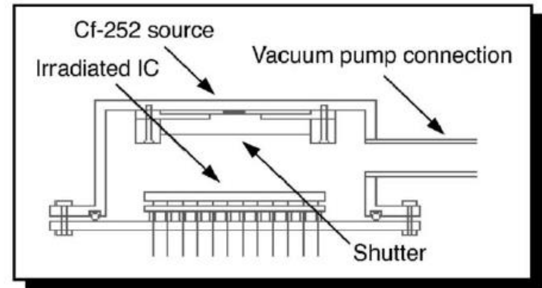


Ilustración 3: Esquema radiación iones pesados [6]

• **Interferencias electromagnéticas:** Esta es una técnica que se usa ampliamente para poner a prueba los equipos digitales. Cualquier dispositivo electrónico está sujeto a una normativa de interferencia electromagnética por el simple hecho de que todo dispositivo es sensible al electromagnetismo. La inyección de fallos por interferencia electromagnética basa su funcionamiento en este hecho.

Los experimentos que dieron lugar al nacimiento de este tipo de inyección de errores se llevaron a cabo en la Universidad Técnica de Viena, Austria mediante el uso de un generador de ráfagas comercial (lo cual aumenta exponencialmente su implementación). Se consideraron dos formas diferentes de aplicación. En la primera forma, la placa del DUT se montó entre dos placas de metal conectadas al generador de ráfaga. De esta manera, todo el nodo se vio afectado por las ráfagas generadas. Debido a que los transceptores Ethernet resultaron ser más sensibles a las ráfagas que el nodo bajo prueba en sí, se configuró una segunda configuración que utilizaba una sonda especial que se colocaba directamente sobre el circuito objetivo. De esta manera, las ráfagas generadas afectaron solo el circuito objetivo (y algunos otros circuitos ubicados cerca de la sonda).

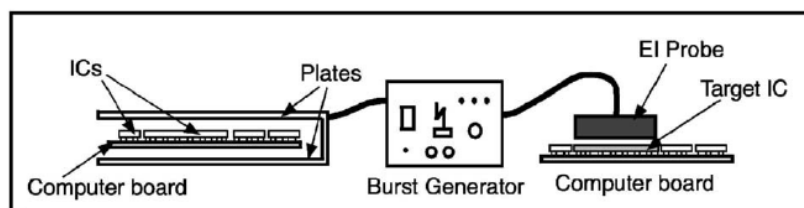


Ilustración 4: Esquema interferencia electromagnética [6]

• **FT-UNSHADES:** El sistema FT-UNSHADES es la plataforma de inyección de fallos usada por el Departamento de Ingeniería Eléctrica de la universidad de Sevilla. Es una plataforma basada en FPGA originalmente dedicada al estudio de la confiabilidad de las listas de redes mediante técnicas de reconfiguración parcial. Se basa en un conjunto de herramientas informáticas y una plataforma de hardware dedicada basada en un FPGA Xilinx de la familia Virtex-II. En la forma original del sistema FT-UNSHADES, los fallos (single-event) se inyectan como “bit-flips” en uno o más registros de usuario del diseño. Los paquetes de bits de configuración que contienen el valor real de un registro de usuario particular se cargan desde la plataforma de hardware, se procesan y descargan con nuevos valores. El software FT-UNSHADES se alimenta con el archivo de asignación de bits obtenido a través de un diseño estándar de Xilinx. En este archivo, la asignación física dentro del FPGA de cada registro de diseño se informa y se asocia a su nombre lógico, como resultado del proceso de síntesis de alto nivel. El registro de destino se selecciona del conjunto completo de registros de usuario y memorias que componen el diseño, antes de la inyección. Otras herramientas de inyección basadas en FPGA realizan la inyección de manera ciega, porque el proceso de inyección se realiza registro por registro y la identificación de cada falla se realiza cuando finaliza la campaña de inyección [7].

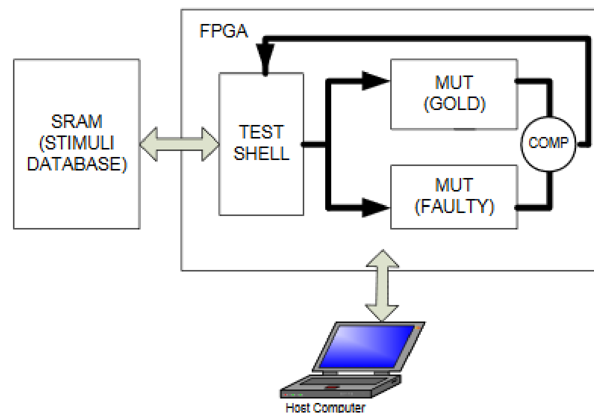


Ilustración 5: Esquema FT-UNSHADES [7]

## 1.4 Herramientas de software libre

### 1.4.1 ¿Qué es software libre?

Podemos definir el software libre como aquel que dé libertad a los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. Esta libertad puede ser clasificada en los siguientes tipos [8]:

• **Libertad 0:** La libertad de ejecutar el programa, para cualquier propósito.

• **Libertad 1:** La libertad de estudiar cómo funciona el programa y adaptarlo a sus necesidades. (El acceso al código fuente es una condición previa para esto).

• **Libertad 2:** La libertad de redistribuir copias para que pueda ayudar a su vecino.

• **Libertad 3:** La libertad de mejorar el programa y publicar sus mejoras al público, para que toda la comunidad se beneficie. (El acceso al código fuente vuelve a ser una condición previa para esto).

Siempre hay que recordar que "Software libre" no significa "no comercial". Un programa gratuito debe estar disponible para uso comercial, desarrollo comercial y distribución comercial. Puedes haber pagado para obtener copias de software gratuito o haberlas obtenido sin cargos, pero independientemente de cómo obtuvieras una copia, siempre se tiene la libertad de modificarla.

Ante estos argumentos uno puede ver de una manera sencilla porqué el software libre puede alcanzar un desarrollo tan elevado como el del software de pago. Si cada uno que utiliza un programa de software libre, lo modificara para incluir una mejora sin ánimo de lucro alguno, en muy poco tiempo podríamos tener un software

con una infinidad de capacidades en comparación con la versión que se ofreció por primera vez al público. Esto es una ventaja clara para los usuarios, pero un problema si tu objetivo es desarrollar software con tu empresa. El único objetivo de una empresa es producir beneficio, y el concepto de software libre choca directamente con este hecho. Lo cierto es que desde cualquier ámbito empresarial no hay ningún argumento que pueda usarse para apoyar el software libre contra software de pago, y menos cuando uno ve los beneficios que obtienen los principales proveedores de software del mundo.

| Puesto Global | Compañía | Región | Ingresos por software (€M) | Ingresos totales (€M) | Ingresos por software (%) |
|---|---|---|---|---|---|
| **1º** | Microsoft | US | 32.686 | 42.504 | 77% |
| **2º** | IBM | US | 14.429 | 68.660 | 21% |
| **3º** | Oracle | US | 13.854 | 16.758 | 83% |
| **4º** | SAP | DE | 8.111 | 10.672 | 76% |
| **5º** | EMC | US | 4.244 | 10.057 | 42% |
| **6º** | Symantec | US | 3.696 | 4.234 | 94% |
| **7º** | HP | US | 3.065 | 89.807 | 4% |
| **8º** | CA | US | 2.825 | 3.080 | 92% |
| **9º** | Intuit | US | 2.299 | 2.340 | 98% |
| **10º** | Adobe | US | 2067 | 2127 | 97 |

Tabla 2: Beneficios de los principales proveedores de software [9]

Aun con todo esto, el software libre sigue apostando por estas libertades. Esta idea casi utópica ha cogido mucha fuerza a lo largo de los años. En la actualidad quizás el mayor ejemplo de todo lo alcanzable mediante herramientas de software libre sea Linux, el sistema operativo por defecto a la hora del desarrollo de software, capaz de mantenerse a flote compitiendo con Microsoft, que como hemos podido ver en la tabla, es la empresa que más beneficios obtiene de la venta de software a nivel mundial.

Para todo aquel que desee meterse de lleno en el mundo de software libre, le recomiendo encarecidamente que mire la página web de “GNU Project” como los escritos de su fundador “Richard Matthew Stallman”.

### 1.4.2 Cocotb

Cocotb es un entorno Test Bench de cosimulación basado en corutinas usado para verificar VHDL y SystemVerilog RTL usando Python. Cocotb es completamente gratuito, de código abierto y alojado en GitHub. Requiere un simulador para simular el diseño HDL y ha sido probado con una variedad de simuladores en Linux, Windows y macOS.

Es un entorno muy diferente a los disponibles en el mercado y presenta unas características únicas [10].

- Fomenta la misma filosofía de reutilización de diseño y pruebas aleatorias que UVM (Universal Verification Methodology), sin embargo, se implementa en Python.
- Con cocotb, VHDL o SystemVerilog normalmente solo se usan para el diseño en sí, no para los Test Benches
- Tiene soporte incorporado para integrarse con sistemas de integración continua, como Jenkins, GitLab, etc. a través de formatos de informes de prueba estandarizados y legibles por máquina.
- Diseñado específicamente para reducir la sobrecarga de crear una prueba.
- Descubre automáticamente las pruebas para que no se requiera ningún paso adicional para agregar una prueba a una regresión.
- Toda la verificación se realiza utilizando Python, que tiene varias ventajas sobre el uso de SystemVerilog o VHDL para la verificación

    - Escribir Python es rápido: es un lenguaje muy productivo
    - Es fácil interactuar con otros lenguajes desde Python.
    - Python tiene una gran biblioteca de código existente para reutilizar
    - Python es interpretado: las pruebas se pueden editar y volver a ejecutar sin tener que volver a compilar el diseño o salir de la GUI del simulador.
    - Python es popular: muchos más ingenieros conocen Python que SystemVerilog o VHDL.

Su funcionamiento tampoco es especialmente difícil entender, aunque puede llegar a ser muy complicado de manejar este programa al 100% de sus capacidades. Un Test Bench típico de Cocotb no requiere código RTL adicional. El DUT (Device Under Test) se instancia “toplevel” en el simulador sin ningún código por encima. Cocotb dirige los estímulos de las entradas al DUT (o más abajo en la jerarquía), monitorizando las salidas directamente desde Python.

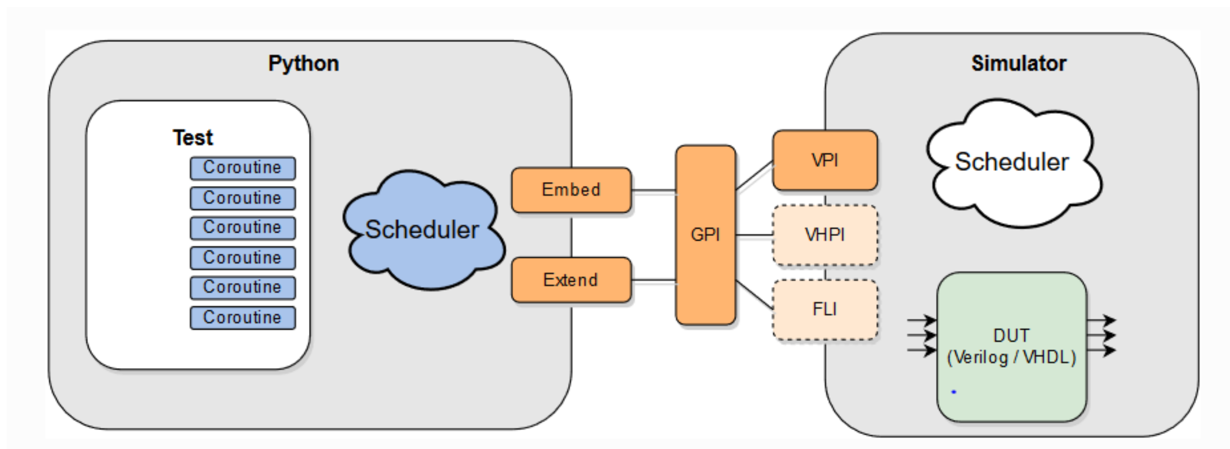


Ilustración 6: Esquema de funcionamiento de Cocotb [10]

Un test es simplemente una función de Python. Para cualquier momento dado, el simulador está avanzando o el código Python se está ejecutando. Se usan una serie de palabras clave para indicar cuándo pasar el control de ejecución al simulador. Sin embargo, lo más destacable de Cocotb son las corutinas. Los Test Benches construidos con cocotb usan corutinas. Mientras se está ejecutando dicha rutina, la simulación se detiene. Una corutina puede bloquear la ejecución de otra corutina o pasar el control de ejecución al simulador cuando lo desee, permitiendo que el tiempo de simulación avance.

Para poder llevar a cabo un primer ejemplo de Cocotb necesitas un archivo Makefile, un test en Python un DUT definido en Verilog o VHDL. De todas formas, tanto el uso básico de Cocotb como un uso avanzado basado en corutinas se verá más adelante en apartados posteriores.

# CAPÍTULO 2: INSTALACIÓN Y USO DEL INYECTOR

Como hemos explicado a lo largo del Capítulo 1, en este proyecto usaremos una serie de programas y utensilios de software libre para realizar simulaciones de diversos dispositivos, corroborando así el correcto funcionamiento de la inyección de errores. Conocidos los conceptos previos, en este Capítulo 2 pasaremos a explicar tanto la instalación de dichos programas, como de los programas previos requeridos para la instalación. También hablaremos de cómo vamos a usar dichos programas y de sus utilidades básicas. Para finalizar, en los últimos apartados, explicaremos en profundidad el código realizado para facilitar una rápida visualización de resultados cuando estemos simulando errores.

## 2.1. Instalación

### 2.1.1 Programas y requisitos

Antes de empezar, hay que mencionar que todo el proyecto está realizado para sistemas basados en Ubuntu. Más concretamente, el sistema operativo utilizado ha sido "*Linux Mint 18.3 Cinnamon (64-bit)*". Para otros sistemas operativos que no estén basados en Ubuntu, será siempre posible realizar la instalación de los programas, al igual que ejecutarlos. Sin embargo, no será posible apoyarse en los scripts que usaremos en apartados siguientes para facilitar el proceso. Tanto la guía de instalación como la guía básica para el uso de los programas están pensadas para estos sistemas operativos, por lo que los comandos usados para instalarlos y ejecutarlos pueden variar en caso de que se use otro tipo de sistema.

Teniendo todo esto en cuenta, la lista de programas es la siguiente:

- Cocotb 1.4.0. dev0
- Icarus Verilog 10.1
- GHDL 0.37
- Gtkwave 3.3.86
- Vcddiff 0.04c

Cocotb ya lo conocemos del capítulo 1. Icarus Verilog y GHDL son los simuladores de software libre en los que se apoyará Cocotb para realizar las simulaciones. Gtkwave y Vcddiff son dos programas de apoyo que nos permitirán visualizar de manera más cómoda los resultados obtenidos. Ambos trabajan con archivos .vcd. De manera resumida y sin meternos en profundidad, los .vcd serán el principal resultado de las simulaciones, pues contienen los valores de las señales, tanto internas como reales. Gtkwave nos presenta una interfaz bastante intuitiva en la que podremos ver las señales de nuestra simulación, mientras que Vcddiff es un programa usado para comparar distintos .vcd. En nuestro caso, lo usaremos para comparar los resultados de simulación con errores y sin errores. Todos estos programas requieren que tengamos instalados otros para que funcionen correctamente. Estos requisitos previos son los siguientes:

- Python3
    - Setup tools, Wheel y Python-dev-packages
- Pip
- Build Essential
- Git
- Gnat
- Zlib1g-dev

Como ya sabemos, Python es una de las bases del funcionamiento de Cocotb, por lo que su instalación es más que necesaria si vamos a trabajar con él. Aparte de Python necesitaremos las librerías correspondientes: Setup tools, Wheel y Python-dev-packages. Pip es un programa que nos permitirá instalar librerías en Python, además que es el programa usado en la instalación de cocotb. Build Essential es un paquete de programas para compilación y programación que contienen lo necesario para trabajar con cocotb y con los scripts que usaremos en apartados posteriores. Git, Gnat y Zlib1g-dev son todos requisitos de GHDL.

### 2.1.2 Comandos de instalación

Conocidos con los programas que vamos a utilizar, pasamos ahora a cómo instalarlos en nuestro sistema. Como hemos mencionado previamente, todo este apartado está pensado para sistemas basados en Ubuntu. Para instalar todos los programas en nuestro sistema, simplemente habrá que introducir los comandos propuestos a continuación, siguiendo el mismo orden que se especifica en esta guía. Este apartado está pensado para aquellos que no han trabajado con Ubuntu/Linux nunca. Para aquel que tenga unos conocimientos previos, puede instalar los componentes por cuenta propia.

*sudo apt-get update*

*sudo apt-get upgrade*

*sudo apt install python3*

*sudo apt install build-essential*

*sudo apt install python3-pip*

*pip install setuptools*

*pip install wheel*

*pip install python3-dev*

*sudo apt-get install Verilog*

*sudo apt install -y git make gnat zlib1g-dev*

*git clone https://github.com/ghdl/ghdl*

*cd ghdl*

```
. /configure --prefix=/usr/local
make
make install

sudo apt-get install gtkwave
git clone https://github.com/veripool/vcddiff
cd vcddiff
make
cp -p vcddiff /usr/local/bin/vcddiff

pip install cocotb
```

Si al utilizar pip nos diera algún problema, los más probable es que se deba a la propia versión de pip, por lo que con actualizar a la última versión debería de solucionarse. Si el problema persistiera, puede probar a ejecutar el mismo comando con *"pip3"* en lugar de *"pip"* o utilizar el comando *"--user"* junto con pip/pip3.

### 2.1.3 Uso básico de programas

Este apartado está pensado para aquellos que deseen realizar sus propias simulaciones sin usar las herramientas proporcionadas o para aquel que no los haya usado nunca. A lo largo de esta sección explicaremos el funcionamiento básico de los programas con los que trabajaremos. Hay que tener en cuenta que todo lo expresado en esta sección depende de la versión de los programas utilizados en el momento de realizar esta guía, por lo que para versiones distintas el funcionamiento puede variar.

#### 2.1.3.1 Icarus Verilog

Icarus Verilog es el simulador que usaremos cuando tengamos que usar DUTs definidos en Verilog/SystemVerilog. Es un simulador muy sencillo de utilizar, pero cuenta con algunas exigencias para usarse, tal y como veremos en este apartado. Aun así, está soportado por cocotb y tiene una wiki muy completa para aquellos que decidan adentrarse más en el manejo de esta aplicación [11].

Su so básico es bastante sencillo. Simplemente necesitamos un dispositivo definido en verilog, junto con su respectivo testbench [12].

```
module up_counter(input clk, reset, output[3:0] counter
    );
reg [3:0] counter_up;

// up counter
always @(posedge clk or posedge reset)
begin
if(reset)
 counter_up <= 4'd0;
else
 counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

```
module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;

up_counter dut(clk, reset, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end

initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

Ilustración 7: Uso de programas:  Código Icarus Verilog

Una vez tengamos un diseño a compilar, simplemente hay que llamarlo con Icarus. Desde *"Terminal"* nos situamos en la carpeta donde tengamos almacenado el diseño y ejecutamos lo siguiente:

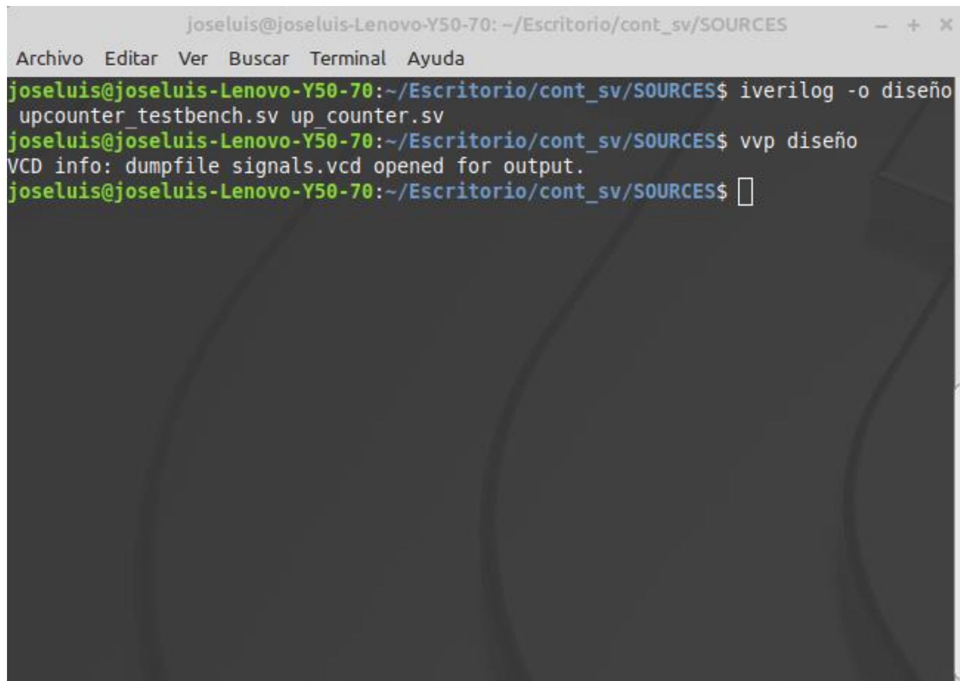


Ilustración 8: Uso de programas:  Comandos Icarus Verilog

"Diseño" es un nombre cualquiera para la compilación, puede ser cambiado a voluntad. En un principio, puede parecer que todo ha salido correctamente, podemos seguir trabajando en la terminal y tenemos a disposición el archivo .vcd de la simulación. Sin embargo, esto ocurre debido a unas modificaciones necesarias a lo estipulado hasta ahora. Icarus verilog no genera los archivos .vcd a menos que se lo pidamos, de la misma forma que no termina la simulación nunca. Por lo que antes de ejecutar el proceso previo hemos hecho unos cambios al código.

```
module upcounter_testbench();
reg clk, reset;
wire [3:0] counter;

up_counter dut(clk, reset, counter);
initial begin
clk=0;
forever #5 clk=~clk;
end

initial begin
   #150; // Wait a long time in simulation units (adjust as needed).
   $display("Simulación Terminada");
   $finish;
 end

initial begin
    $dumpfile("signals.vcd");
    $dumpvars(0,upcounter_testbench);
end

initial begin
reset=1;
#20;
reset=0;
end
endmodule
```

Ilustración 9: Uso de programas:  Código (corregido) Icarus Verilog

En el primer proceso creado simplemente pondremos el tiempo límite que queramos, mientras que en el segundo especificaremos un nombre para nuestro .vcd e indicaremos a qué modulo estamos haciendo referencia, esto es, el testbench. Con esto, nuestro proceso debería de terminar, dejándonos la terminal libre de nuevo a la vez que nos genera un .vcd con el siguiente resultado.

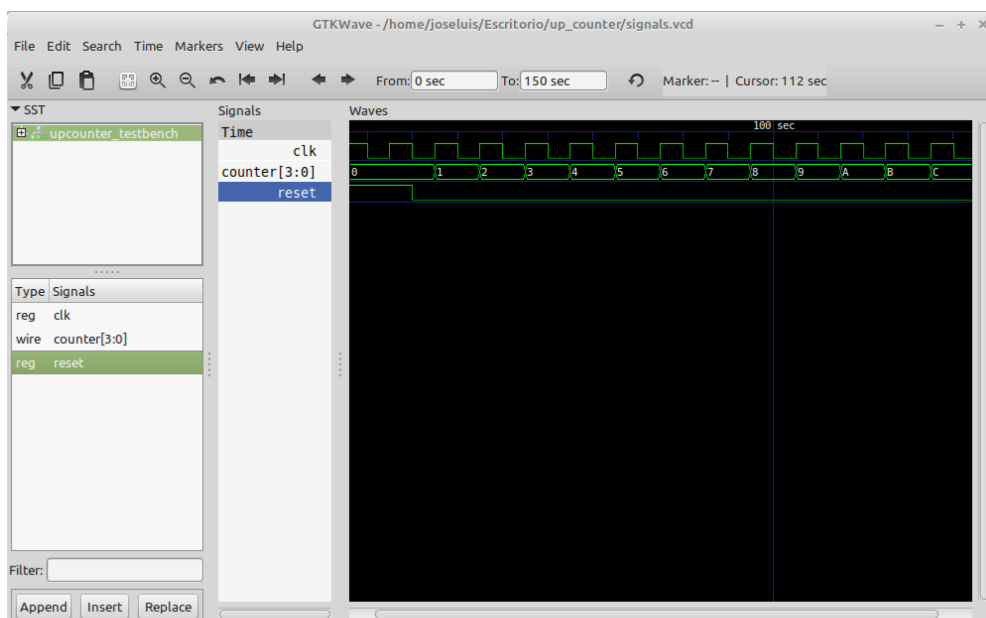


Ilustración 10: Uso de programas:  Resultados Icarus Verilog

#### 2.1.3.2 GHDL

Como contraparte de Icarus Verilog tenemos a GHDL. Como su nombre indica, este compilador está pensado para dispositivos definidos en VHDL. Su uso es relativamente más incómodo, pues requiere de más comandos a la hora de realizar una simulación, pero es muy rápido en comparación con otros compiladores/simuladores. Al igual que con Icarus Verilog, necesitaremos un diseño con su respectivo testbench, esta vez en VHDL.

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


entity cont_dig is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : in  STD_LOGIC;
           cuenta : out  STD_LOGIC_VECTOR (3 downto 0));
end cont_dig;

architecture Behavioral of cont_dig is

signal cuenta_signal,p_cuenta:std_logic_vector(3 downto 0);
begin


    sync:process (clk,reset)
    begin

        if reset='1' then
            cuenta_signal<= (others =>'0');
        elsif rising_edge(clk)then
            cuenta_signal<=p_cuenta;
        end if;
        end process sync;

    comb:process (cuenta_signal,enable,p_cuenta)
    begin


    p_cuenta <= cuenta_signal;
    if enable='1' then
        p_cuenta<=std_logic_vector(unsigned(cuenta_signal)+1);
        if cuenta_signal = "1001" then
        p_cuenta<="0000";
        end if;
    end if;

    end process comb;

    cuenta<=cuenta_signal;


end Behavioral;
```

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

entity cont_dig_tb is
    end cont_dig_tb;

architecture behavior of cont_dig_tb is
    component cont_dig
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : in  STD_LOGIC;
           cuenta : out  STD_LOGIC_VECTOR (3 downto 0));
    end component;

    signal clk : std_logic := '0';
    signal reset : std_logic := '0';
    signal enable : std_logic := '0';

    signal cuenta : std_logic_vector (3 downto 0);

    constant clk_period : time := 10 ns;

Begin
    uut: cont_dig
    Port map (
             clk => clk,
             reset => reset,
             enable => enable,
             cuenta => cuenta    );

    sync :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    comb: process
    begin
        reset <='1';
        enable <='0';
        wait for 50 ns;
        reset <='0';
        wait for 20 ns;
        enable <='1';

        wait;
    end process;

end;
```

Ilustración 11: Uso de programas:  Código GHDL

Para ejecutar la simulación simplemente ejecutamos los siguientes comandos desde "Terminal" situándonos en la carpeta donde tengamos nuestros archivos:

```
joseluis@joseluis-Lenovo-Y50-70: ~/Escritorio/cont_vhd
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -s cont_dig.vhdl
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -s cont_dig_tb.vhdl
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -a cont_dig.vhdl
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -a cont_dig_tb.vhdl
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -e cont_dig_tb
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$ ghdl -r cont_dig_tb --vcd=signal.vcd --stop-time=300ns
ghdl:info: simulation stopped by --stop-time @300ns
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/cont_vhd$
```

Ilustración 12: Uso de programas:  Comandos GHDL

Al igual que con Icarus Verilog, GHDL no tiene tiempo límite de ejecución, y no genera .vcd automáticamente. Sin embargo, en este aspecto es mucho más cómodo que Icarus Verilog, pues nos permite establecer estos parámetros desde los comandos de ejecución tal y como se puede apreciar en la foto. Esta simulación generó los siguientes resultados:

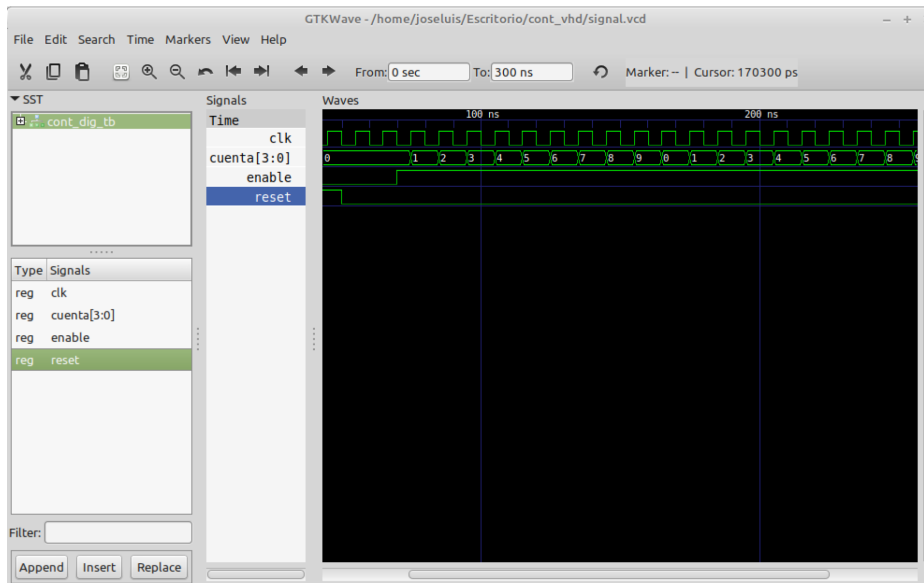


Ilustración 13: Uso de programas:  Resultados GHDL

#### 2.1.3.3 Gtkwave

A lo largo de los dos últimos apartados hemos estado viendo graficas de gtkwave sin saber cómo generar dichas gráficas. Cómo ya sabemos, esto se consigue con Gtkwave, cuyo uso es el más sencillo de los presentados en esta guía. Una vez instalado, basta con definir el programa como predeterminado a la hora de abrir archivos .vcd, cosa que ya hace automáticamente al instalarse. A pesar de esto, siempre podemos llamar a gtkwave desde terminal mediante el siguiente comando.

```
joseluis@joseluis-Lenovo-Y50-70: ~/Escritorio/up_counter
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/up_counter$ ls
diseño  signals.vcd  up_counter.sv  upcounter_testbench.sv
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/up_counter$ gtkwave signals.vcd
```

Ilustración 14: Uso de programas:  Comandos Gtkwave

Con esto podemos llamar a cualquier .vcd que queramos. Esto será bastante útil cuando llamemos a terminal desde los respectivos scripts para lanzar automáticamente el .vcd de los resultados obtenidos tras la simulación. Independientemente de esto, dentro del propio gtkwave el manejo es muy sencillo, simplemente desplegamos el dispositivo en cuestión y arrastramos las señales que queramos observar al display.

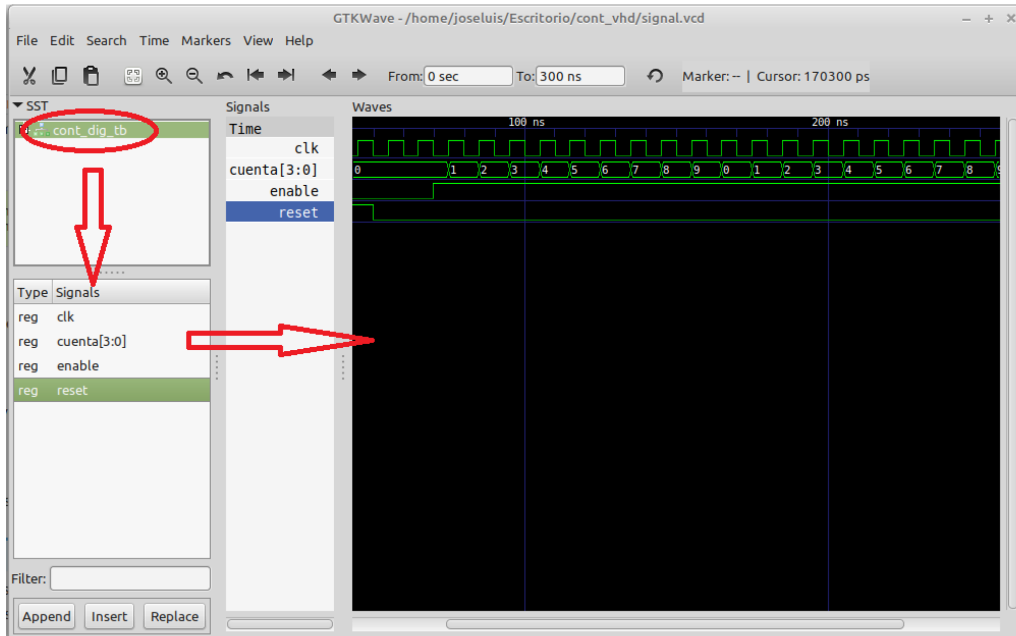


Ilustración 15 Uso de programas: Resultados Gtkwave

#### 2.1.3.4 Vcddiff

El uso de vcddiff, al igual que con gtkwave es muy simple. Si bien solo se puede lanzar desde terminal, es un programa con muy pocas opciones de configuración y variables. Teniendo dos .vcd, Podemos compararlos mediante esta herramienta introduciendo los siguientes comandos:

```
joseluis@joseluis-Lenovo-Y50-70: ~/Escritorio/up_counter
Archivo Editar Ver Buscar Terminal Ayuda
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/up_counter$ vcddiff -w 10s.vcd 15s.
vcd
> test.c1.reset (#) at time 10 next occurence at time 15
> #10   (10)
> #15   (10)
test.c1.reset (#) Never found in file 2 at time 15
< test.c1.reset (#) at time 15 next occurence at time 100
< #15   (10)
< #100  (0-)
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/up_counter$ vcddiff -s 10s.vcd 15s.
vcd
> test.c1.reset (#) at time 10 next occurence at time 15
> #10   0
> #15   0
test.c1.reset (#) Never found in file 2 at time 15
< test.c1.reset (#) at time 15 next occurence at time 100
< #15   0
< #100  0
joseluis@joseluis-Lenovo-Y50-70:~/Escritorio/up_counter$
```

Ilustración 16: Uso de programas: Comandos y Resultados Vcddiff

Si queremos imprimir los cambios en los valores directamente, introduciremos el argumento “-s” junto con vcddiff, mientras que si queremos ver los valores de pico utilizaremos el argumento “-w”. En lo que respecta a la simulación efectuada, es simplemente un contador para el cual el reset se desactiva a los 10 y a los 15 segundos. Como se puede observar, vcddiff nos muestra esa variación por pantalla.

#### 2.1.3.5 Cocotb

Como hemos mencionado previamente, Cocotb es un programa de uso complejo, que requiere de muchas horas para dominar todas las opciones que nos pone a disposición. Aun así, comenzar a usar la herramienta es relativamente sencillo. Como bien sabemos, Cocotb necesita de tres piezas fundamentales para poder realizar una simulación, esto es:

- Makefile
- Test en Python
- DUT en Verilog o VHDL

Puesto que el DUT ya estará previamente hecho (sino no estaríamos buscando una herramienta para simularlo), esto nos deja con dos objetos a fabricar, el Makefile y el test en Python. Para aquellos que no hayan trabajado nunca ni con makefiles ni con Python, esto puede parecer bastante complicado a primera vista, pero lo cierto es que uno puede ponerse a simular sin conocimientos previos ni de GNU Make ni de Python. Además de que la wiki de Cocotb pone a disposición toda la información existente sobre el programa, en caso de que surjan dudas o las simulaciones no funcionen. Volviendo al tema que nos atañe, en este apartado explicaremos dos modelos básicos para simular en Cocotb, estos modelos serán explicado para DUTs en VHDL como en Verilog. Hay que mencionar que no entraremos en corutinas, puesto que es un tema que debe de quedar fuera de esta guía básica y con el que nos meteremos en más detalle al avanzar con este proyecto.

El primer punto con el que vamos a trabajar será realizar una simulación simple con un test en Python. Para ello usaremos los dispositivos que hemos usado anteriormente en la guía con GHDL e Icarus Verilog. Empezaremos con Verilog, seguido de VHDL. Terminaremos con el segundo punto, que es la realización de simulaciones usando testbenches no diseñados en Python:

##### 2.1.3.5.1 Verilog

```
include $(shell cocotb-config --makefiles)/Makefile.sim

VERILOG_SOURCES = $(PWD)/up_counter.sv

TOPLEVEL=up_counter

MODULE=test
```

Ilustración 17: Uso de programas: Makefile básico para simulaciones con Cocotb en Verilog

Esta figura representa el modelo estándar de un Makefile para Icarus Verilog. Solo hay que incluir la cabecera de cocotb, indicarle cuáles son las fuentes que queremos usar (Si hay más de un archivo Verilog, simplemente se añaden como si del ejemplo se tratase, usando += en vez de =. Si se conoce el uso de $(PWD), se puede indicar el directorio directamente), decirle cual es el toplevel de todos los archivos, y darle el nombre de nuestro test de Python. En este caso, el test creado se llama directamente "test.py". Definir el archivo como Makefile es tan fácil como guardarlo como "Makefile".

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer


@cocotb.test()
def test(dut):
    dut._log.info("Starting clock manager")
    c = Clock(dut.clk, 2, 'sec')
    cocotb.fork(c.start())

    dut._log.info("Running test!")
    dut.reset = 1
    yield Timer(4, units='sec')
    dut.reset = 0
    yield Timer(100, units='sec')
```

Ilustración 18: Uso de programas:  Test básico para simulaciones con Cocotb en Verilog

Para crear nuestro test de Python simplemente hay que tener en cuenta las señales que vamos a modificar. Para crear un reloj recomiendo copiar directamente las primeras líneas de nuestro test y poner la especificación temporal deseada (La especificación temporal hace referencia al ciclo del reloj). A partir de ese punto solo hay que modificar las señales de entrada llamándolas con "dut.nombre_de_la_señal" y asignarle un valor. El comando yield dejara que pase el tiempo que se ha especificado dentro de la simulación. El nombre "test" no puede ser alterado, al igual que "dut".

Teniendo definido ya todo, solo queda lanzar Cocotb, para ello solo hay que abrir un terminal, situarse en la carpeta donde se encuentre el Makefile y nuestro test en Python, e introducir el comando "make". Cocotb realizará la simulación y podremos acceder a los datos de ella mediante el propio terminal y una serie de archivos que nos aparecerán en la carpeta seleccionada. Si hemos seguido la guía de Icarus Verilog, también podemos usar el apartado de los .vcd para generar este tipo de archivos junto con el resto. Poniendo de ejemplo este mismo test, vamos a lanzar una simulación con el reset a 1 durante 4 segundos, y después lo ponemos a 0 durante 100 segundos.

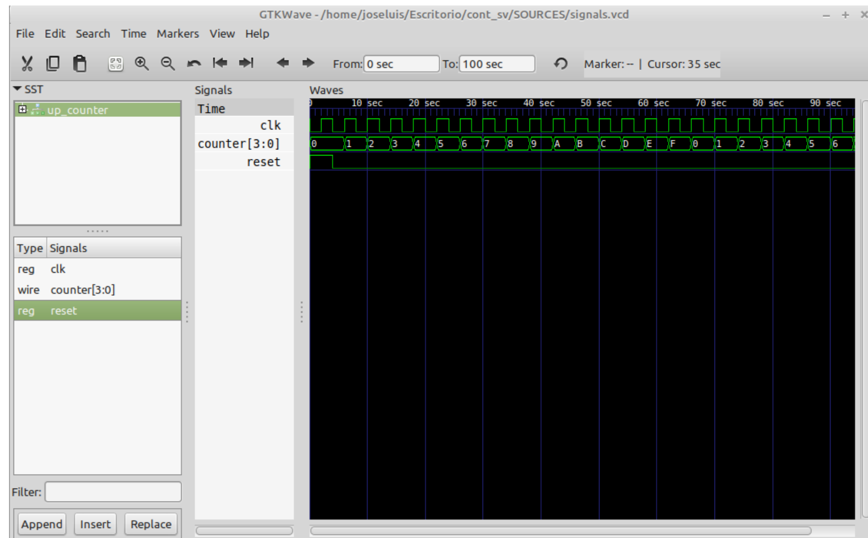


Ilustración 19: Uso de programas:  Resultados de la simulación de ejemplo (Cocotb y Verilog)

#### 2.1.3.5.2 VHDL

Las simulaciones en VHDL serán muy similares a las de Verilog, pero tendrán alguna característica particular que hay que tener en cuenta:

```
include $(shell cocotb-config --makefiles)/Makefile.sim

SIM_ARGS ?= --vcd=signals.vcd
TOPLEVEL_LANG=vhdl

VHDL_SOURCES += $(PWD)/cont_dig.vhdl

TOPLEVEL=cont_dig

MODULE=test
```

Ilustración 20: Uso de programas:  Makefile básico para simulaciones con Cocotb en VHDL

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer


@cocotb.test()
def test(dut):
    dut._log.info("Starting clock manager")
    c = Clock(dut.clk, 2, 'ns')
    cocotb.fork(c.start())

    dut._log.info("Running test!")
    dut.reset = 1
    dut.enable=0
    yield Timer(4, units='ns')
    dut.reset = 0
    yield Timer(10, units='ns')
    dut.enable=1
    yield Timer(200, units='ns')
```

Ilustración 21: Uso de programas:  Test básico para simulaciones con Cocotb en VHDL

En lo que respecta al Makefile, será exactamente igual que para Verilog, solo que esta vez habrá que indicar el lenguaje de programación y, en caso de querer un .vcd, indicarlo mediante SIM_ARGS tal y como viene en el ejemplo. Aparte de todo esto habrá que lanzar la simulación con "make SIM=ghdl" o nos dará un error. El test de Python funciona exactamente igual que para Icarus Verilog. El resultado del test presentado en la figura anterior puede verse en la siguiente gráfica:

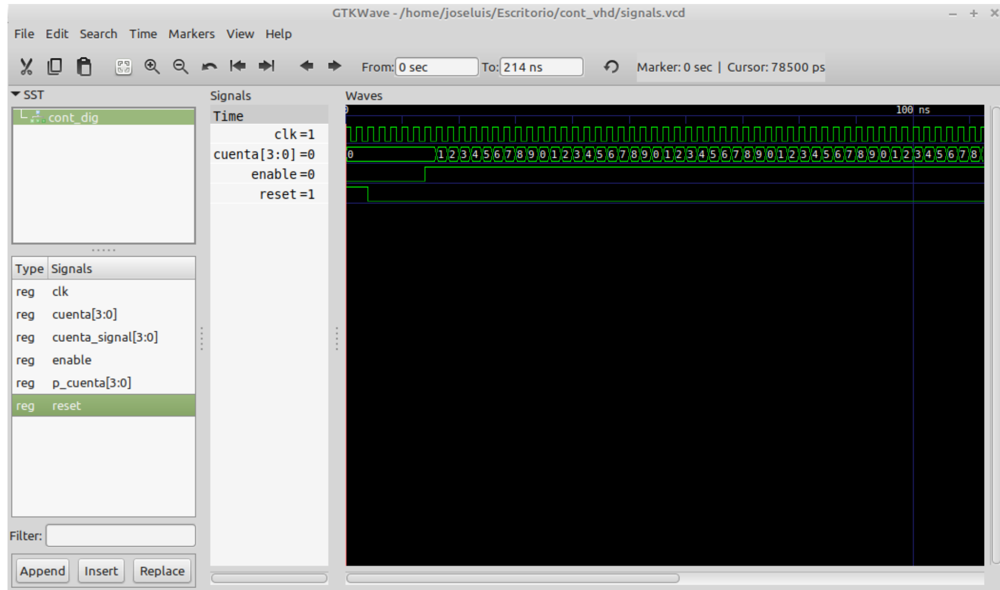


Ilustración 22: Uso de programas: Resultados de la simulación de ejemplo (Cocotb y VHDL)

### 2.1.3.5.3 Testbenches sin Python

Una de las funcionalidades más desconocidas y más básicas de cocotb es su capacidad para realizar la simulación sin especificar nada en Python mediante el uso de un testbench clásico (en Verilog o VHDL). Es cierto que necesitaremos sí o sí un test en Python para simular, pero este estará completamente vacío. En las siguientes imágenes explicaremos como generar esta clase de simulaciones en VHDL, aunque el proceso es completamente extrapolable a Verilog.

```
include $(shell cocotb-config --makefiles)/Makefile.sim

SIM_ARGS ?= --vcd=signals.vcd
TOPLEVEL_LANG=vhdl

VHDL_SOURCES += $(PWD)/cont_dig.vhdl
VHDL_SOURCES += $(PWD)/cont_dig_tb.vhdl

TOPLEVEL=cont_dig_tb

MODULE=test
```

Ilustración 23: Uso de programas: Makefile básico para simulaciones con Cocotb sin test en Python

```python
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import Timer


@cocotb.test()
def test(dut):
    yield Timer(200, units='ns')
```

Ilustración 24: Uso de programas:  Test en Python (para testbenches en HDL)

Como se puede apreciar a simple vista, el proceso es muy fácil, simplemente hay que incluir el testbench dentro de las fuentes que vamos a usar e indicar que el testbench es el toplevel. En lo que respecta a el test en Python, es un test vacío. Lo único que posee es un tiempo para que corra la simulación, pues todas las variables vienen definidas en el testbench. Todo lo que hemos definido en el apartado 2.1.3.5.2 es aplicable de también aquí.

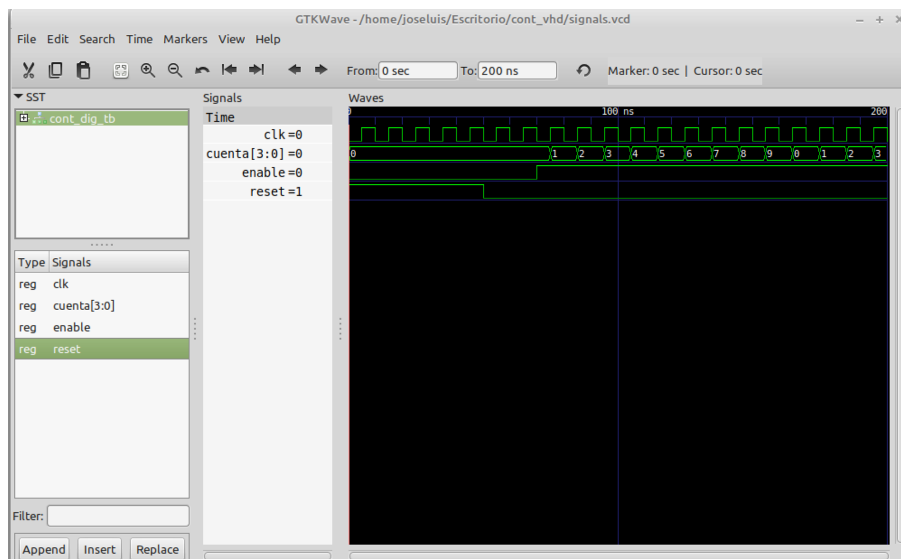


Ilustración 25: Uso de programas:  Resultado de la simulación de ejemplo (Cocotb sin test en Python)

Cómo bien se aprecia en la gráfica, la simulación funciona perfectamente. Esto puede parecer algo bastante trivial, pero puede volverse una herramienta muy útil como veremos más adelante, pues nos permite probar dispositivos con test ya definidos si necesidad de crear test de Python para cada uno.

# CAPÍTULO 3: USO DEL INYECTOR

En este capítulo hablaremos tanto de la inyección de fallos básica en Cocotb, como de la inyección de fallos mediante las herramientas desarrolladas. Todos y cada uno de los scripts desarrollados será explicado con detenimiento, haciendo hincapié en su funcionamiento individual.

## 3.1 Uso básico

Una vez conocidos y probados los programas con los que ejecutaremos las simulaciones, es hora de explicar cómo se usa el inyector de errores de Cocotb. Su funcionamiento es muy sencillo, el inyector de errores está definido por completo en cuatro funciones muy simples que importaremos directamente sobre nuestro test.py desde las librerías de Cocotb. Estas funciones son las siguientes:

```
# Deposit action
dut.my_signal <= 12
dut.my_signal <= Deposit(12)  # equivalent syntax

# Force action
dut.my_signal <= Force(12)    # my_signal stays 12 until released

# Release action
dut.my_signal <= Release()    # Reverts any force/freeze assignments

# Freeze action
dut.my_signal <= Freeze()     # my_signal stays at current value until released
```

Ilustración 26 : Uso de programas: Funciones de inyección de fallos de Cocotb

Como se puede apreciar en la ilustración, "Deposit" tiene la misma función que asignar un valor a un input cualquiera, "Force", como su nombre indica, fuerza un valor determinado a una señal en concreto, ya sea input, output o señal interna, "Release" libera un valor forzado y "Freeze" tiene la misma función que "Force", solo que fuerza el valor de la variable en el momento de la llamada. Conociendo estas funciones uno puede técnicamente hacer simulación de fallos para cualquier diseño que desee. Para poner esto a prueba vamos a usar nuestro contador en Verilog y veremos cómo responde el inyector.

Introduciremos un fallo "single-event" para la señal "counter_up". El código es básicamente el mismo que en los apartados anteriores (2.1.3.5.1), solo alteraríamos el test.py para incluir la rutina de inyección de fallos, quedando de la siguiente forma:

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


@cocotb.test()
def test(dut):
    dut._log.info("Starting clock manager")
    c = Clock(dut.clk, 2, 'sec')
    cocotb.fork(c.start())

    dut._log.info("Running test!")
    dut.reset = 1
    yield Timer(4, units='sec')
    dut.reset = 0
    yield Timer(10, units='sec')
    dut.counter_up <= Force(10)
    yield Timer(2, units='sec')
    dut.counter_up <= Release()
    yield Timer(80, units='sec')
```

Ilustración 27: Uso de programas: Test del ejemplo de inyección de fallos

Realizando la misma simulación que en el apartado anterior para este test en Python, obtenemos el siguiente archivo .vcd con los resultados de nuestra inyección de fallos.

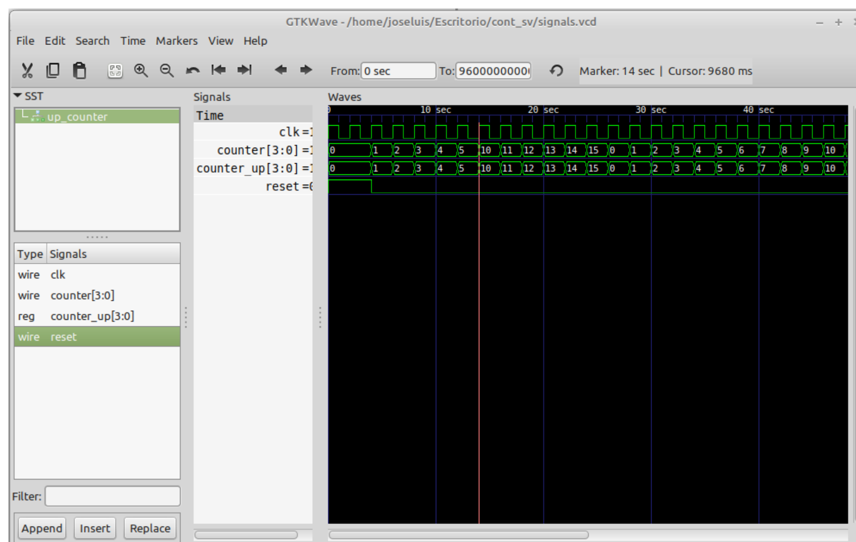


Ilustración 28: Uso de programas: Resultado del ejemplo de inyección de fallos

Como se puede apreciar en la gráfica, vemos como la señal interna “counter_up” es forzada al valor “10” en el instante T=14s, tal y como hemos introducido en el test.py. Ese valor es forzado hasta el siguiente ciclo, momento en el que liberamos la señal. Hemos decidido forzar la señal interna porque es la que propaga el error dentro de la simulación. Si hubiéramos alterado el output (“counter”), la simulación no habría visto propagación alguna, tal y como se ve en el siguiente .vcd.

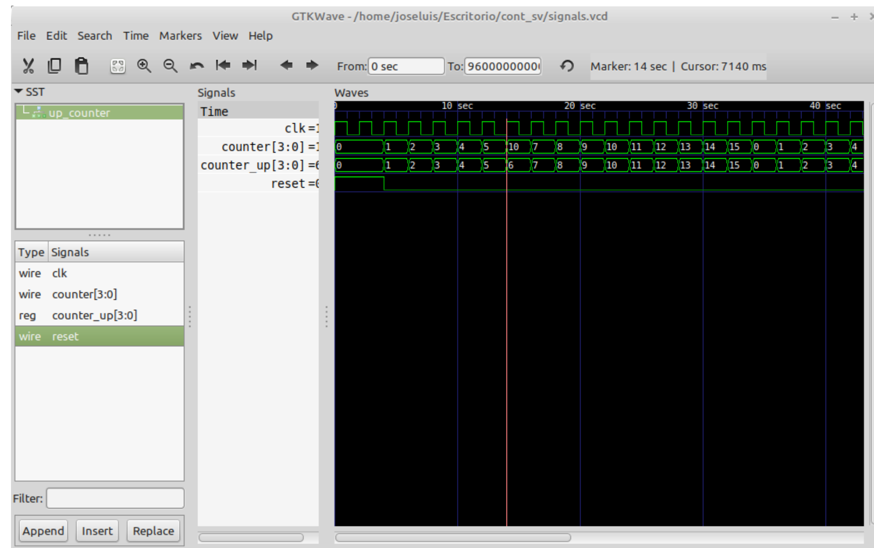


Ilustración 29: Uso de programas: Resultado del ejemplo de inyección de fallos (señal interna forzada)

Esto es uno de los principios de lo que definiremos en un futuro como “campaña de inyección de errores”, probar todas las señales posibles para ver cuál puede llegar a ser problemática en caso de fallo.

## 3.1 Uso de herramientas auxiliares proporcionadas

En este apartado explicaremos todos y cada uno de los scripts desarrollados para expandir las capacidades de Cocotb. Estas herramientas de desarrollo serán utilizadas en el capítulo siguiente para realizar pruebas de inyección con distintos dispositivos. Puesto que el uso de Cocotb es relativamente limitado (todo el inyector de fallos se resume en 4 funciones) y requiere de ajustar test.py y Makefile para cada dispositivo, hemos desarrollado una serie de scripts que, además de facilitar la tarea de simulación restando tiempo de preparación de la herramienta, aportan una serie de características que expanden las opciones del propio Cocotb. Todo siempre queda por encima del propio programa, por lo que no será necesario modificar ningún archivo interno. Los scripts son los siguientes:

- Test.py
- Makefile
- Config
- Gen.py
- Org.py

Test.py y Makefile son dos plantillas personalizadas para trabajar junto con el resto de scripts. Puesto que vamos a trabajar un nivel por encima del programa (sin modificar archivos internos), será lógico que nos apoyemos en los propios archivos de Cocotb para lograr nuestros objetivos, simplemente por el hecho de que al no alterar nada, el programa técnicamente sigue un funcionamiento normal, por lo que seguirá requiriendo tanto el Makefile como un test en Python. En lo que respecta al resto de archivos, son relativamente independientes de Cocotb, pero esenciales si queremos usar las herramientas.

### 3.2.1 Funcionamiento conjunto: Requisitos y modos

- Requisitos

Para poder utilizar los scripts de apoyo vamos a tener que cumplir una serie de requisitos previos. En primer lugar, hay que decir que todo lo proporcionado hasta ahora está pensado para funcionar con un testbench como toplevel (2.1.3.5.1). Esta decisión fue tomada por el simple hecho de que, en principio, en la etapa de diseño de un sistema, este se ha tenido que probar mínimamente para comprobar su funcionamiento, por lo que la gran mayoría de diseños cuentan ya con su testbench asociado. Que el toplevel sea un testbench nos permite usar esos diseños, además de ahorrarle al usuario de dichos diseños la necesidad de traducir todo su código a un test en Python si quiere usar la herramienta de inyección de errores.

Estas herramientas están pensadas tanto para circuitos con señal de reloj, como para circuitos completamente combinacionales, independientemente de si usan señal de reset o no. Dentro del testbench asociado será necesario indicar si quieres usar el propio reloj y reset del testbench en HDL (o si no poseen ninguno). Esto es así debido a que tanto el proceso de reloj como el reset inicial pueden ser reemplazados desde el propio test.py.

Estos son los requisitos previos que se deben cumplir a la hora de utilizar estas herramientas. Dentro del archivo Config y del propio test.py se especifican particularidades respecto al uso de las mismas, por lo que recomiendo encarecidamente leer el archivo Config antes de empezar a inyectar errores.

- Modos

Dentro del archivo “Config” se puede observar que se especifican dos modos de uso para las herramientas proporcionadas. El modo normal y el modo campaña.

> - Modo Normal: El modo normal está pensado para simulaciones concretas de inyección de fallos, en los que se sepa perfectamente qué es lo que se quiere hacer. Permite controlar en su totalidad una simulación, permitiendo inyectar un número infinito de errores concretos en dicha prueba. Cada error puede ser inyectado a voluntad en un instante determinado, para una señal cualquiera. La duración del error también es especificada por el usuario. Aparte de la simulación con los errores, la herramienta realiza la misma simulación sin forzar ningún fallo. Después de ambas simulaciones, se lanza un protocolo automático de comparación entre los resultados de ambas mediante vcddiff mientras se comparan también con Gtkwave. Todos los resultados quedan almacenados en dos carpetas separadas mientras que el propio terminal donde hemos ejecutado las herramientas mostrará los resultados de Cocotb.
>
> - Modo Campaña: El modo campaña está pensado para poner a prueba nuestro dispositivo con el mayor número de pruebas posibles. En el modo campaña se realizarán una serie de simulaciones de inyección de fallos “single-event” en nuestro dispositivo una detrás de otra, favoreciendo siempre la velocidad a la hora de simular. El objetivo de esto es realizar todas las simulaciones posibles y comparar los resultados de estas con una simulación normal del dispositivo. Todos los archivos obtenidos de las pruebas serán almacenados y ordenados en una carpeta, permitiendo fácil acceso al usuario para su posterior comparación. Dentro del modo campaña existen dos modos de operación, campaña exhaustiva y campaña aleatoria. Esto será explicado en el apartado siguiente junto con los scripts.

### 3.2.2 Funcionamiento específico

A lo largo de este apartado vamos a explicar cómo funciona el sistema de manera más específica, pasando por

todos los archivos disponibles. Debido a la complejidad de los mismos, resulta ineficiente poner el código directamente para comentarlo, por lo que en este apartado cambiaremos un poco la estrategia vista en el manual de uso de programas y basaremos nuestras explicaciones en una serie de esquemas de funcionamiento. Aun así, el código estará siempre disponible junto con este documento.

Antes de empezar recordemos a grandes rasgos el funcionamiento de Cocotb, pues es esencial entenderlo bien antes de explicar los scripts individualmente. Cocotb inicia su funcionamiento con una llamada a terminal (make). Esta lanza la simulación que tengamos definida en el test y genera tanto datos por terminal como una serie de archivos de resultados. Como bien sabemos requiere de un test en Python, un Makefile y un DUT en VHDL o Verilog. Esto queda recogido en el siguiente esquema:

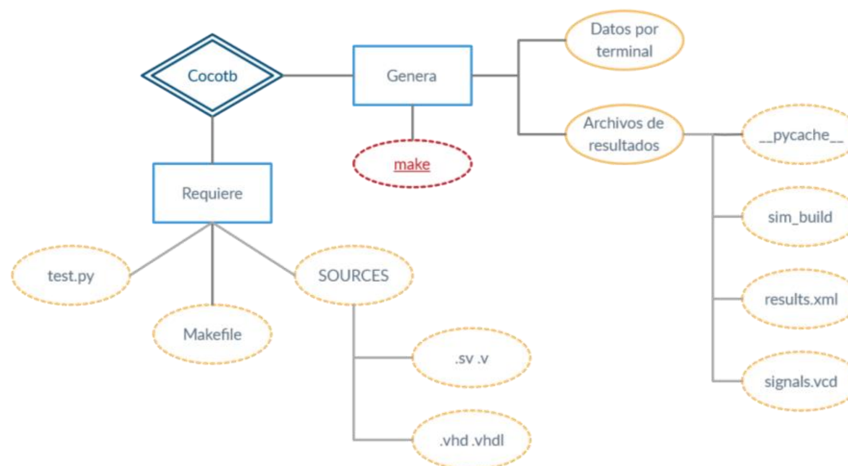


Ilustración 30: Archivos requeridos y generados por Cocotb

#### 3.2.2.1 Config

El archive Config es el centro de todas las herramientas. Es el encargado de comunicar la selección del modo de funcionamiento a los demás archivos y de pasar todas las variables necesarias a Cocotb. Como hemos mencionado antes, basaremos nuestra explicación en un esquema de funcionamiento.

En el esquema podemos ver los principios básicos de Config. Config es tanto un archivo ejecutable como un archivo de datos. Dentro de Config podemos encontrar una pequeña guía básica de cómo se usa la herramienta junto con los requisitos de uso de la misma (no confundir con los requisitos previos). Es por esto que en este apartado volvemos a recomendar su lectura antes de continuar.

Config también contiene las variables necesarias para poder ejecutar el Makefile de Python. Como podemos ver en las ramas de ejecución, esas variables son entregadas al Makefile predefinido mediante una llamada recursiva de make, y es este Makefile el que lanza Cocotb. Estas variables son las mismas que usaríamos para un Makefile básico de Cocotb. Dentro de Config vienen estipuladas las mínimas a rellenar si queremos que el sistema funcione.

Sin embargo, el principal funcionamiento de Config no es como archivo de datos, sino como archivo ejecutable. Config es el archivo a llamar para lanzar la herramienta de inyección de fallos. Config se lanza como un Makefile cualquiera con los argumentos necesarios para seleccionar cualquiera de los modos disponibles, tal y como se ve en el esquema.

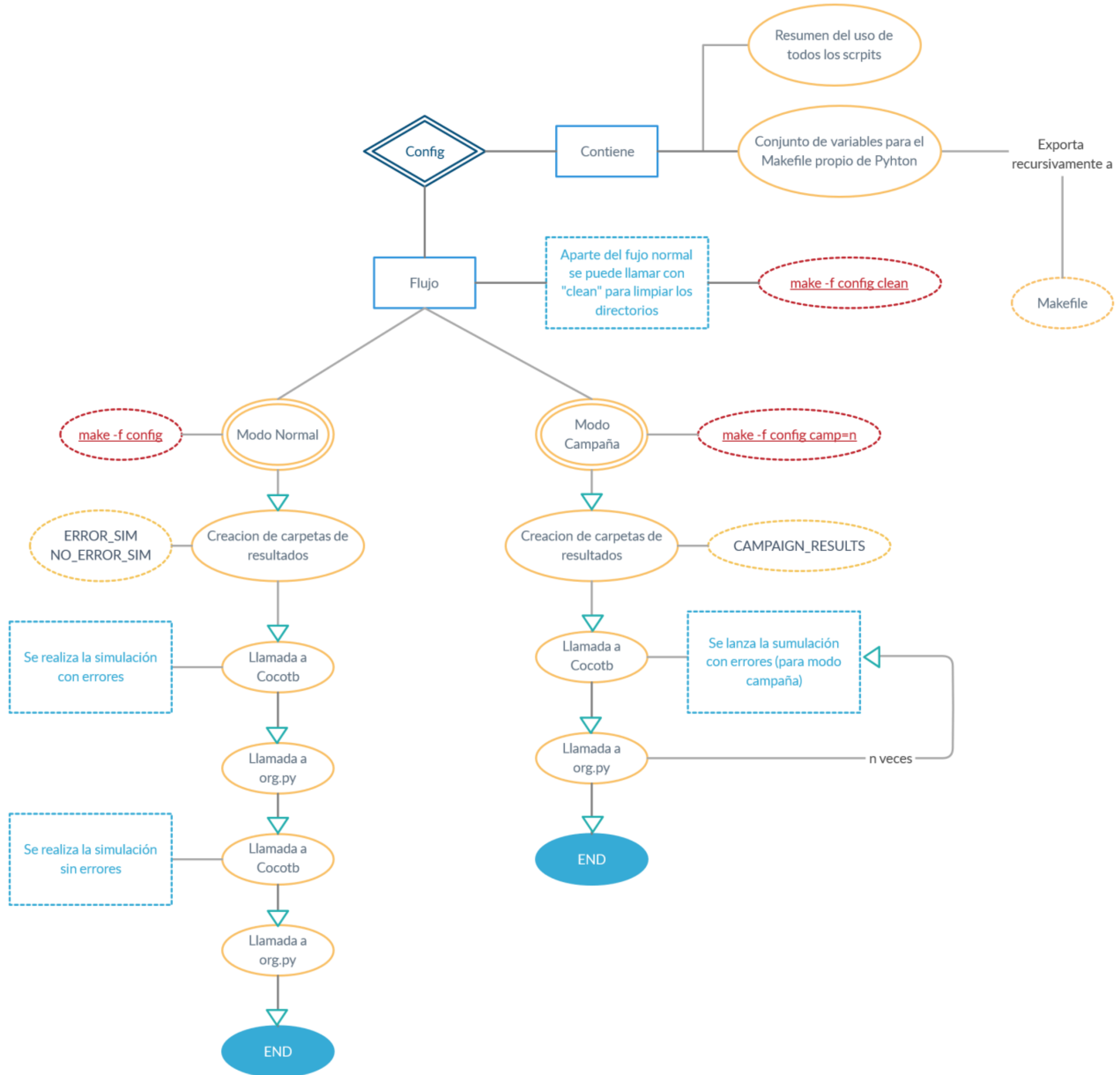


Ilustración 31: Esquema de funcionamiento Config

Si seleccionamos el modo normal, se crearán una serie de carpetas de resultados (ERROR_SIM y NO_ERROR_SIM), seguidas de la primera llamada a Cocotb. Esta primera llamada se lanzará la simulación con inyección de fallos (estos fallos estarán definidos en test.py como veremos más adelante). En el momento en el que termine la simulación, Config llamará a org.py para que renombre y agrupe los resultados en su carpeta correspondiente, (ERROR_SIM). Todo este proceso volverá a repetirse para una nueva simulación sin inyección de fallos, tal y como se puede apreciar en la figura.

Para el modo campaña el funcionamiento es bastante similar. Config creará la carpeta de resultados CAMPAIGN_RESULTS y procederá a lanzar Cocotb seguido de org.py "n" veces. Este número "n" introducido como argumento en la propia llamada de Config representa el número de simulaciones del archivo CAMPAIGN. Este nuevo archivo se explicará con más detalle junto con gen.py y test.py. Por ahora basta con saber que el modo campaña requiere de un archivo llamado CAMPAIGN para poder ejecutarse. Esto se debe a que las variables para inyectar fallos no están definidas dentro de test.py como pasa dentro del modo normal.

Volviendo a Config, una vez que termine con las "n" simulaciones, terminará la ejecución del mismo.

#### 3.2.2.2 Test.py

Al igual que Config, test.py también funciona como archivo de datos. Debido al hecho de que estamos trabajando por encima de Cocotb, pasar datos al propio test en Python es algo que queda fuera de nuestro alcance. Es por esto que dentro de test.py el usuario tendrá que rellenar una serie de variables indispensables para el correcto funcionamiento de las herramientas. Las variables vienen explicadas tanto en Config como en el propio test.py, aun así, podemos ver cuáles son en la siguiente figura:

```
#VARIABLES
SIMULATION_CYCLES=

TIME_UNIT=""            #sec, ns, etc
CLK_PERIOD=
CLK_NAME=""             #CLOCK SIGNAL NAME
COM_P=                  #0 FOR AN ONLY COMBINATIONAL DUT

T_RESET=                #INITIAL RESET DURATION
RESET_VALUE=            #RESET ACTIVE AT "1" OR "0"
RESET_NAME=""           #RESET SIGNAL NAME


#NORMAL MODE            (IF YOU ARE USING CAMPAIGN MODE, YOU CAN FILL THIS RANDOMLY)
VAR=[""]
CI=[]
DU=[]

VAL=[]
BIT=[""]                #NULL IF YOU DONT WANT TO USE THIS FEATURE
```

Ilustración 32: Variables del test.py prediseñado

La mayoría de las variables son auto explicativas, como las que hacen referencia al reset o el reloj del circuito. Estas variables tienen que ser rellenadas tanto para el modo campaña como para el normal. Las que son más difíciles de entender son aquellas exclusivas del modo normal, las cuales vienen explicadas brevemente en el archivo Config. En este apartado las analizaremos en mayor profundidad:

- VAR: "Var" es la lista en la que se definen las señales a forzar en modo normal. Pueden ser señales distintas o puede ser la misma señal que ha sufrido x fallos en la misma simulación.
- CI: "Ci" hace referencia al ciclo de inicio de los fallos, y se corresponde con las señales de "Var", la señal en la 5º posición de "Var· verá un fallo en el ciclo situado en la 5º posición de "Ci"
- DU: "Du" hace referencia a los ciclos en los que la señal debe estar bloqueada en el valor de "Var". Puedes bloquear una señal durante toda la simulación o realizar un fallo "single-event"
- VAL: "Val" es el valor a asignar a las señales en la inyección del fallo. Tiene la misma estructura que las variables anteriores. No se pueden forzar valores si esto no caben en la propia variable (forzar un 15 en una variable de 2 bits)
- BIT: "Bit" nos permite trabajar directamente sobre los bits de una variable. Funciona a la vez que "Val", por lo que solo puedes usar una de las dos. Con "Bit" podemos inyectar un error en un único bit de una variable. Por ejemplo, si tenemos la variable 1010, e inyectamos un fallo con "Bit" a 2, se forzará un valor 1110, si ponemos "Bit" a 0 se forzará 1011.

Dejando de lado la función de archivo de datos del script, test.py sigue cumpliendo la misma función para una simulación simple de Cocotb, conteniendo funciones, corutinas y el propio test para la simulación. En apartados anteriores hemos mencionado las corutinas, pero no nos hemos metido nunca en el concepto de qué es una corutina y cómo se usan en Cocotb. Pues bien, una corutina es un concepto similar a un hilo o proceso, pero con una característica única. Mientras que un hilo/proceso es algo que se está ejecutando en paralelo a una simulación, las corutinas no se ejecutan de esta forma. Una corutina detiene la simulación para iniciar su propia ejecución, y solo le devuelve el control a la misma al finalizar dicha corutina o mediante el uso de palabras

claves o llamadas. Las corutinas no solo se comunican con la propia simulación también pueden saltar entre otras corutinas, por lo que son una herramienta muy útil para aquellos que sepan utilizarlas. Además de esto, Cocotb te da la opción de que las corutinas que definas funcionen como procesos en paralelo, abriendo aún más las posibilidades a la hora de construir un test en Python. Para aquel que le interese hacer sus propias simulaciones, la wiki de Cocotb alberga una gran cantidad de información y ejemplos sobre el tema.

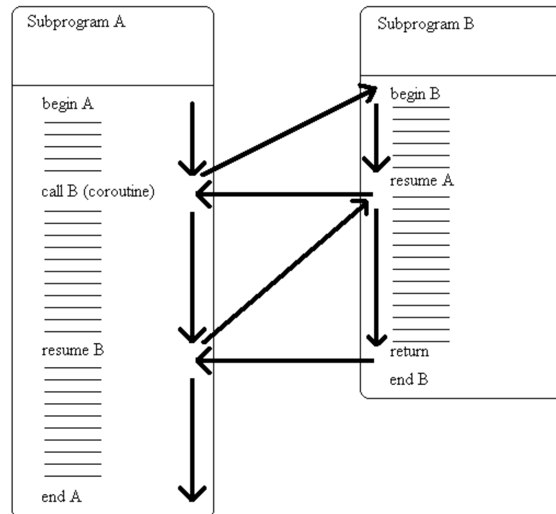


Ilustración 33: Esquema funcionamiento corutinas [13]

Volviendo a nuestro test, en el esquema podemos ver que contamos con cuatro funciones/corutinas definidas por el usuario (aparte de estas cuatro, usamos una rutina definida por el propio Cocotb para lanzar el reloj). Más concretamente, "campaign" y "modifyBit" son funciones y "forced" y "reset_f" son corutinas. De los propios nombres ya se puede suponer su funcionamiento a rasgos generales, pero se requiere una explicación más en detalle.

- Campaign: Como su nombre indica, es la función que lee los datos del archivo CAMPAIGN. Más concretamente, almacena los datos en un archivo temporal, y va cogiendo datos del propio CAMPAIGN para las simulaciones hasta que este está vacío. El número de simulaciones no es controlado por el test.py, sino por el propio Config, tal y como sabemos del apartado anterior. Al final de la campaña, es también Config el que se encarga de organizar tanto el temporal como el original.
- ModifyBit: Es una función utilizada para cambiar ese único bit definido en la variable con su mismo nombre ("BIT"). La componen simplemente una serie de operaciones lógicas.
- Forced: Es una corutina que funciona como proceso en paralelo a la simulación. Como argumentos, recibe todos los valores de las variables definidas en el propio test.py (las que hemos explicado anteriormente: VAR, CI, DU, VAL, BIT). Si se está ejecutando el modo campaña recibe los datos del propio archivo CAMPAIGN. El funcionamiento de forced es muy sencillo, es la corutina de inyección de errores, controla los tiempos y valores del propio fallo.
- Reset_f: Al igual que "Forced", es una corutina que se ejecuta como proceso en paralelo. Su funcionamiento es bien simple, recibe las variables correspondientes al reset (RESET_NAME, RESET_VALUE y T_RESET) y ejecuta un reset inicial.

Con esto queda explicada la rama de funciones y corutinas de nuestro test.py, solo queda entender cómo funciona todo en conjunto dentro de la propia simulación. En @cocotb.test se realizan dos distinciones nada más iniciarse,

se diferencia entre simulación con errores y sin errores. La simulación sin errores presenta un funcionamiento muy sencillo. Simplemente se lanzan clk y el reset inicial y se deja correr la simulación. Puesto que estamos trabajando con un testbench como toplevel, este proceso generará los resultados de haber simulado dicho testbench, o lo que es lo mismo, la simulación sin fallos.

Para la simulación con errores la cosa se complica. Al igual que el caso anterior, se empieza lanzando clk y el reset inicial (si se ha especificado), pero en esta rama no se deja correr la simulación sin más. Si estamos dentro del modo campaña, tal y como hemos visto en la propia explicación de la función "campaign", se seleccionan los datos de uno en uno, y se realiza una simulación a la vez hasta que CAMPAIGN sea un archivo sin datos. Todo este proceso está definido en Config.

Dentro del modo normal, el proceso es parecido al modo campaña, pero con una estructura distinta. Como bien sabemos, el modo campaña tiene n simulaciones distintas para una única inyección de errores en cada simulación, mientras que para el modo normal, se realiza una única simulación controlada con las inyecciones de errores que se quieran definir. Esto cambia la estructura del modo normal frente a la rama del modo campaña. Para el modo normal no se lanza x veces Cocotb hasta que CAMPAIGN está vacío, sino que se lanzan n corutinas de forzados siendo n el número de variables a forzar en la misma simulación. Este valor n viene definido del tamaño de las variables introducidas en test.py, más concretamente del tamaño de "VAR".

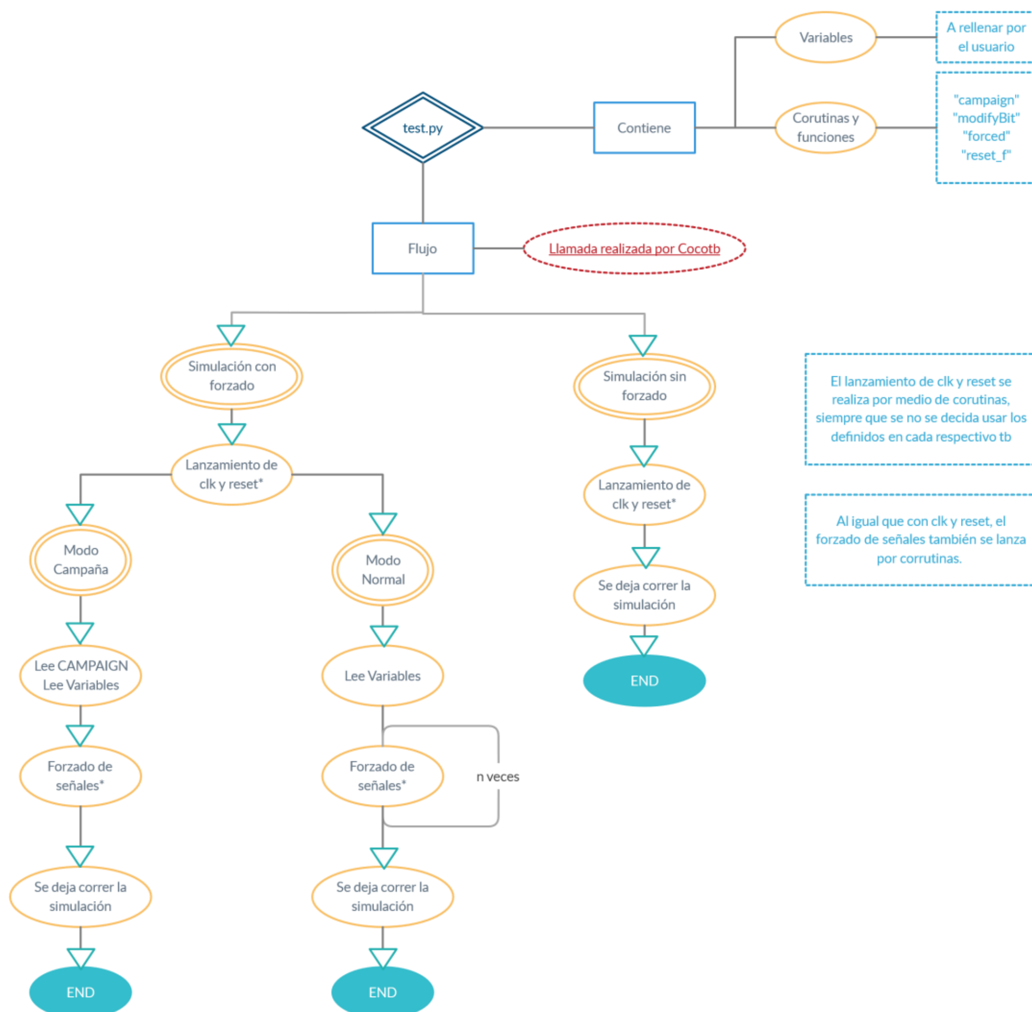


Ilustración 34: Esquema funcionamiento del test.py prediseñado

#### 3.2.2.3 Gen.py

A lo largo de los dos últimos apartados hemos hablado del archivo CAMPAIGN sin meternos en su composición. Sabemos que es un archivo de datos que contiene una serie de simulaciones para realizar una campaña y poco más. Pues bien, en este apartado explicaremos CAMPAIGN en su totalidad además de un script de generación de campañas automático conocido como “gen.py”. Comencemos por el archivo CAMPAIGN.

El archivo CAMPAIGN es un archivo de texto de x líneas que presentan la siguiente estructura:

VAR BIT CI

VAR BIT CI

Siendo “VAR”, “BIT” y “CI” variables con el mismo significado que las definidas para el modo normal de test.py (2.2.2.2.2). Cada una de estas líneas representa una simulación con un fallo “single-event” para la variable seleccionada. El archivo CAMPAIGN no tiene una longitud estipulada, y puede alcanzar la longitud que el usuario desee. Es por esto que diseñar campañas de tamaños considerables puede volverse una tarea muy pesada. Para estos casos usamos el script gen.py.

Este script es básicamente un generador de archivos CAMPAIGN aleatorios. Como podemos ver en el esquema, requiere de un archivo SIGNAL_LIST para generar una campaña. Este archivo solo será necesario definirlo una única vez y podrá ser reutilizado para generar infinitas campañas. SIGNAL_LIST estará formado por todas las señales en las que se inyectaran los fallos “single-event”. Si no quieres que una señal se vea afectada por errores, basta con que no la definas. Este archivo presenta la siguiente estructura:

VAR MSB LSB

VAR MSB LSB

Siendo MSB y LSB el bit más significativo y el bit menos significativo de la variable (esto se exige para asegurar que la generación aleatoria de “BIT” no genere error. Ejemplo: intentas cambiar el 5º bit de una variable de 2 bits). Con un archivo SIGNAL_LIST definido podemos empezar a generar campañas. Para ello solo hay que llamar desde terminal al propio script junto con los argumentos necesarios de ejecución, tal y como se ve en el esquema. Estos argumentos son:

- CYCLE: Este argumento hace referencia a los ciclos totales de duración de la simulación. Necesitamos introducir este valor como argumento para asegurarnos que en la generación aleatoria no se intenté inyectar un error fuera de la propia simulación (que CI sea mayor que los ciclos totales).
- -a: Si introducimos el argumento “-a” se nos creará lo que hemos definido como una campaña aleatoria. Es decir, se generará un archivo CAMPAIGN con n simulaciones con señales aleatorias. Esto es fácilmente entendible con un ejemplo. Pongamos que tenemos un archivo SIGNAL_LIST con dos señales definidas, “s1” y “s2”. Si lanzamos gen.py para generar una campaña aleatoria con un valor de n=5, obtendremos un archivo CAMPAIGN de 5 líneas, en las que se alternarán las señales “s1” y “s2” de forma completamente aleatoria.
- -e: Introduciendo este argumento se creará una campaña exhaustiva. Una campaña exhaustiva consiste en una campaña con n simulaciones para cada señal disponible. Si imitamos el ejemplo anterior seleccionando el modo exhaustivo en lugar del aleatorio, el resultado será un archivo CAMPAIGN de 10 líneas, 5 simulaciones para “s1” y 5 simulaciones para “s2”
- n: Número de simulaciones. Su valor cobra diferente significado para campañas exhaustivas o aleatorias.

En lo que respecta al funcionamiento del propio script, gen.py no es especialmente complicado. Simplemente lee SIGNAL_LIST, genera una serie de datos aleatorios y los escribe en un archivo CAMPAIGN, distinguiendo entre el modo aleatorio y el modo exhaustivo.

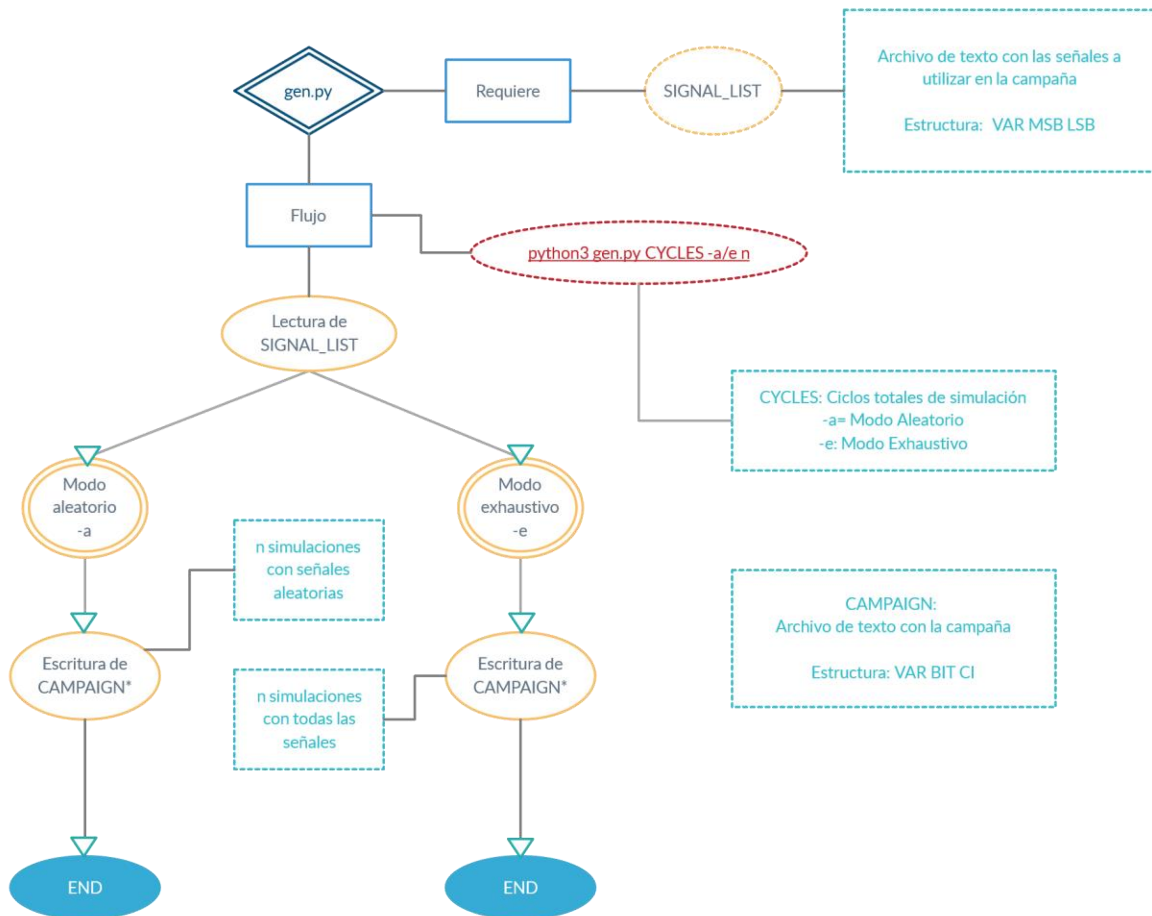


Ilustración 35: Esquema funcionamiento gen.py

### 3.2.2.4 Org.py

Org.py es el script de Python encargado de organizar todos los archivos de resultados de Cocotb. Si realizas dos simulaciones seguidas de Cocotb los resultados de la segunda se sobrescriben sobre la primera, eliminando la opción de acceder a dichos datos. Org.py es el encargado de que esto no ocurra, ya sea renombrando dichos archivos o almacenándolos en distintos directorios para evitar que se sobrescriban.

Como vimos en el apartado 2.2.2.2.1 dónde explicamos el funcionamiento de Config. Org.py es un script llamado cada vez que se realiza una simulación, ya sea dentro del modo normal o del modo campaña. Aun así, como es lógico, presenta un funcionamiento distinto dependiendo del modo en cuestión.

Para el modo campaña realiza una reestructuración más sencilla que para el modo normal. Cada vez que se termina una simulación, org.py intenta almacena los resultados en la carpeta CAMPAIGN_RESULTS. Cada conjunto de archivos de resultados recibe un cambio de nombre (se la añade un subíndice) mediante el cual podemos organizar los resultados de todas las simulaciones de la campaña de manera sencilla y en orden de ejecución.

Para el modo normal se distingue entre simulación con fallos y sin fallos. Si ejecutamos org.py después de la simulación con errores del modo normal, nos almacenará los resultados de dicha simulación en ERROR_SIM. Si por el contrario llamamos a org.py tras la simulación sin fallos, nos almacenará los resultados en NO_ERROR_SIM. Una vez se han realizado las dos simulaciones, org.py lanza un protocolo de autocomparación de resultados entre ambas, llamando a Gtkwave de forma automática, a la vez que se comparan los .vcd mediante Vcddiff.

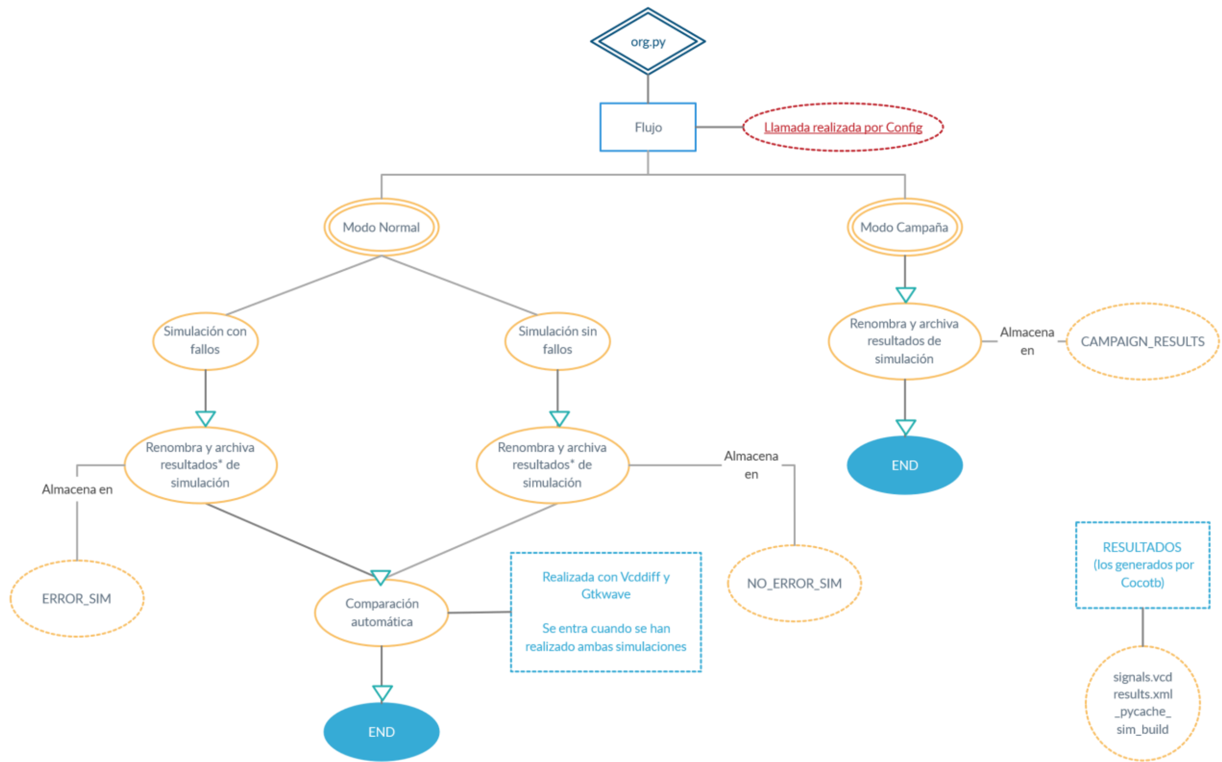


Ilustración 36: Esquema funcionamiento org.py

# CAPÍTULO 4: INYECCIÓN EN DISTINTOS DISPOSITIVOS

## 4.1 DUTs y objetivos

A lo largo de esta memoria hemos visto muchos ejemplos de uso de herramientas y del propio Cocotb, pero todas estas pruebas se han realizado en un entorno controlado y usando dispositivos testeados previamente (contadores muy sencillos). En este capítulo inyectaremos fallos en una serie de dispositivos distintos a ese contador usado anteriormente, tanto para ver la respuesta a fallos de los mismos como para ver si Cocotb es capaz de generar resultados lógicos en la simulación. Todas las simulaciones de este capítulo no han sido probadas previamente, por lo que errores catastróficos que impidan la simulación son una posibilidad. Aun así, puesto que esto es un estudio de las capacidades de Cocotb, podremos sacar información de los propios errores para nuestras conclusiones finales. Para realizar este estudio usaremos las herramientas proporcionadas, y simularemos una inyección de fallos mediante el modo normal, seguida de una campaña de errores.

Realizaremos cuatro pruebas distintas, o lo que es lo mismo, testearemos cuatro dispositivos distintos. Dos de estos dispositivos estarán definidos en Verilog, mientras que los otros dos lo estarán en VHDL (para poder probar al 100% Cocotb necesitamos ver cómo funciona para los diferentes HDL). Los dispositivos estudiados han sido seleccionados de entre dispositivos realizados por la propia universidad como de diseños realizados por otras entidades [12]:

- Shifter (VHDL): Este dispositivo combinacional permite mover y girar datos, lo que es usado en numerosos cifradores. A pesar de su simpleza, nos permitirá estudiar el funcionamiento de Cocotb y las herramientas de desarrollo ante un sistema completamente combinacional.

- UART (VHDL): Este dispositivo es una implementación de un UART de 8 bits de datos, 0 paridad y 1 bit de stop. Relativamente complejo y con numerosas entradas y salidas, nos permitirá realizar una inyección de errores en modo campaña muy completa.

- Delay Timer (Verilog): Un dispositivo que nos permite programar delays. Bastante sencillo en lo que respecta al funcionamiento y sin ninguna característica especial, es un dispositivo más a la hora de la inyección de fallos.

- FIFO (Verilog): Un diseño en Verilog de una memoria FIFO. Nos permitirá jugar inyectando fallos en las variables que indican cuando la pila está llena o vacía. Además de esto, es un dispositivo con numerosos módulos y relativamente complejo, por lo que será interesante ver cómo reacciona Cocotb para este tipo de DUTs

Debido a la cantidad de simulaciones realizadas (especialmente las campañas), los resultados de las simulaciones serán comentados en la memoria a grandes rasgos. Solo las gráficas de un solo dispositivo pueden llegar a duplicar la longitud total de este documento. Se analizarán los .vcd de las simulaciones en modo normal, mientras que en las simulaciones modo campaña se comentarán brevemente: Si existe una simulación en concreto de una campaña que presente características únicas, esta sí se comentará más a fondo. Para aquel que quiera ver en su totalidad los resultados, como hemos mencionado anteriormente, todo el código realizado para este proyecto estará disponible junto con la memoria. En el caso de este capítulo 4, recomendamos tenerlo a mano para comprender en su totalidad los resultados obtenidos.

## 4.2 Inyeccion de fallos

Conociendo los dispositivos que vamos a usar y los objetivos de este estudio, pasamos ahora a la parte de la simulación. Como bien sabemos, realizaremos una simulación en modo normal seguida de una simulación en modo campaña. La simulación en modo normal será una simulación controlada, en la que intentaremos inyectar un error conocido para el dispositivo para ver si este se simula correctamente mediante Cocotb. Las variables modificadas serán definidas en la propia memoria, seguidas de los resultados de simulación. La simulación en modo campaña será menos controlada. Haremos una selección de señales para inyectar errores y definiremos una campaña aleatoria de unas veinte simulaciones aproximadamente. Si dentro de estas simulaciones encontramos alguna destacable, se analizará más en profundidad.

### 4.2.1 Shifter

El shifter se presenta como el DUT más sencillo de este apartado. Con solo tres variables, sin reloj ni reset, nos deja pocas opciones a la hora de realizar una inyección de fallos. Aun así, puesto que el objetivo de esta simulación es ver el funcionamiento del sistema para DUTs completamente combinacionales, el shifter se convierte en la mejor opción.

Puesto que solo tenemos tres señales disponibles, y todas pueden presentar posibles errores propagables en el funcionamiento del sistema, en la inyección de errores controlada (modo normal) vamos a inyectar para la misma simulación en las 3 variables. Las siguientes ilustraciones recogen tanto las variables como los resultados de la simulación:

```
#VARIABLES
SIMULATION_CYCLES=50

TIME_UNIT="ns"          #sec, ns, etc
CLK_PERIOD=10
CLK_NAME="NULL"          #CLOCK SIGNAL NAME
COM_P=0

T_RESET=0               #INITIAL RESET DURATION
RESET_VALUE=0           #RESET ACTIVE AT "1" OR "0"
RESET_NAME="NULL"           #RESET SIGNAL NAME
RESET_P=0

#NORMAL MODE        (IF YOU ARE USING CAMPAIGN MODE, YOU CAN FILL THIS RANDOMLY)
VAR=["SHIFTINPUT","SHIFTOUT","SHIFT_Ctrl"]
CI=[15,30,45]
DU=[5,10,1]

VAL=[1,1,1]
BIT=[0,3,1]       #NULL IF YOU DONT WANT TO USE THIS FEATURE
```

Ilustración 37: Variables introducidas en la simulación del Shifter

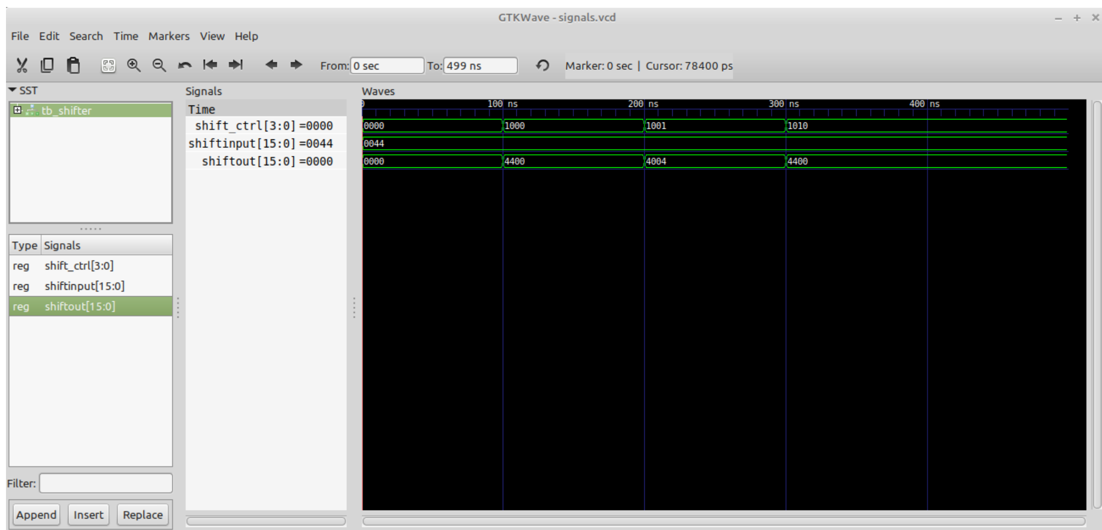


Ilustración 38: Resultado (sin inyección) de la simulación del Shifter

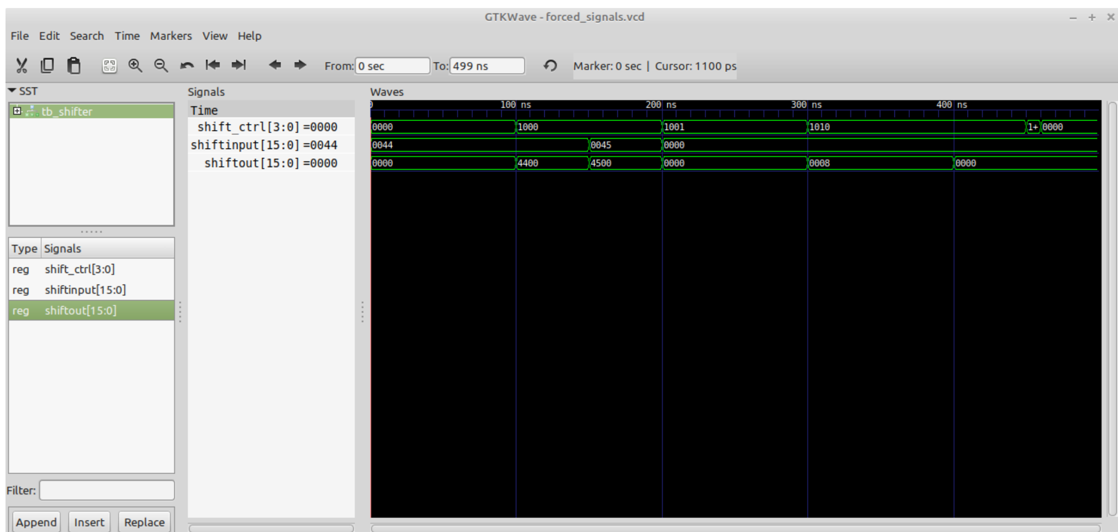


Ilustración 39:  Resultado (con inyección) de la simulación del Shifter

Como se puede ver en las ilustraciones, el testbench de la simulación sin errores es muy sencillo. Ante un valor de "shift_ctrl", el shifter reorganiza la entrada de una forma distinta. Sin embargo, algo que parece sencillo nos ha presentado un fallo catastrófico en el momento de la inyección. Los fallos inyectados en "SHIFTOUT" y "SHIFT_crtl" no han presentado ningún error propagable dentro del sistema, pero el error en "SHIFTIMPUT" se ha propagado de manera que ha bloqueado completamente el DUT. Al liberar el valor forzado ("0045"), el sistema combinacional debería mantenerlo, pues en el testbench en ningún momento se actualiza esta señal. Sin embargo, como se puede ver en la gráfica, la señal se pone a "0000". Con esto no queremos poner de manifiesto el mal funcionamiento del DUT, sino la utilidad de Cocotb. Una vez conocida la existencia del error, el diseñador del dispositivo podrá cambiar su diseño para que esto no ocurra.

Para la simulación en modo campaña hemos realizado las veinte simulaciones aleatorias entre las diferentes señales. Al realizar la campaña no se dieron errores a la hora de simular, ni se observaron más fallos catastróficos quitando el que ya hemos observado. Como siempre, todos los resultados se encuentran junto con la memoria.

### 4.2.2 UART

El protocolo de comunicaciones UART es algo sustancialmente más complicado que el shifter. Es un estándar conocido en la industria y se usa en infinidad de dispositivos. Es por esto por lo que se ha elegido para en este capítulo. El dispositivo presenta una cantidad de señales considerable, de las cuales vamos a realizar una inyección sobre dos, al menos en el modo normal. Como característica especial, hay que decir que para este dispositivo se ha utilizado el reloj y el reset inicial definido por la herramienta, en contraste con el dispositivo plenamente combinacional del apartado anterior. Señales y resultados vienen recogidos en las siguientes gráficas:

```
#VARIABLES
SIMULATION_CYCLES=40000

TIME_UNIT="ns"         #sec, ns, etc
CLK_PERIOD=10
CLK_NAME="I_clk"          #CLOCK SIGNAL NAME
COM_P=1

T_RESET=100              #INITIAL RESET DURATION
RESET_VALUE=1          #RESET ACTIVE AT "1" OR "0"
RESET_NAME="I_reset"          #RESET SIGNAL NAME
RESET_P=1

#NORMAL MODE        (IF YOU ARE USING CAMPAIGN MODE, YOU CAN FILL THIS RANDOMLY)
VAR=["O_txRdy","O_rxSig"]
CI=[6600,15500]
DU=[436,552]

VAL=[1,1]
BIT=["NULL","NULL"]      #NULL IF YOU DONT WANT TO USE THIS FEATURE
```

Ilustración 40: Variables introducidas en la simulación de la UART

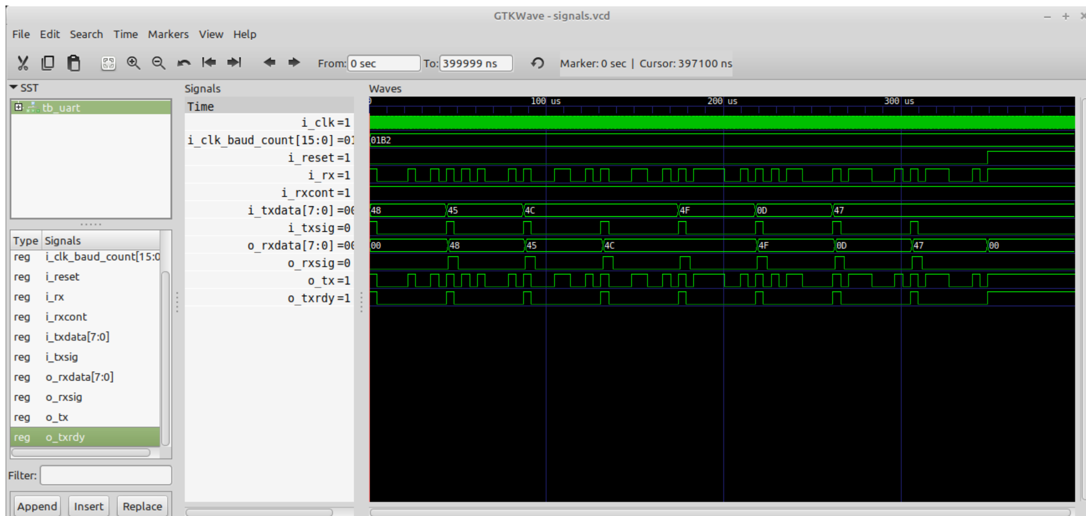


Ilustración 41: Resultado (sin inyección) de la simulación de la UART

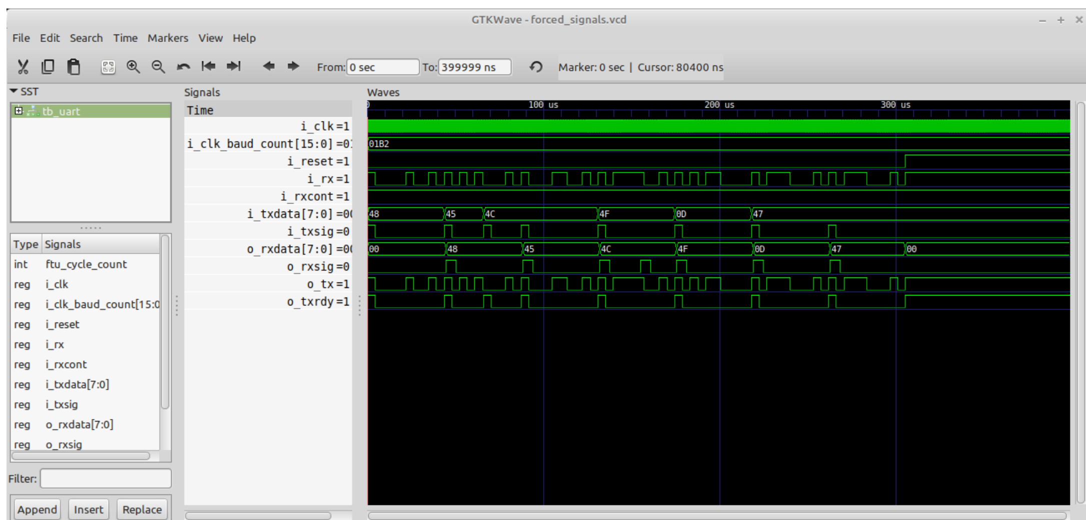


Ilustración 42: Resultado (con inyección) de la simulación de la UART

Para aquello que desconozcan el funcionamiento del protocolo UART, lo único que estamos haciendo en este testbench es enviar un mensaje "HELLO", el cual viene representado por los valores de la señal "o_rxdata". Para esta simulación hemos introducido un fallo conocido en "I_txSig", señal que controla cuando debe transmitirse la siguiente pieza de información. Este error debería provocar que en vez de enviar "HELLO", el dispositivo enviase "HELO". Como se ve en la gráfica de la simulación con fallos, esto ha sido exactamente lo que ha ocurrido. Se ha introducido también un error a través "O_rxSig" para ver si provocaba algún malfuncionamiento, pero no se ha propagado de ninguna manera. A pesar de esto, con la primera inyección ya hemos conseguido nuestro objetivo, ver que Cocotb ha simulado un error conocido, presentando los datos esperados ante dicho error, volviendo a demostrar su utilidad y fiabilidad como inyector de errores.

En lo que respecta al modo campaña se han vuelto a realizar veinte simulaciones con las señales previas utilizadas y alguna más que nos ha parecido destacable. Las simulaciones no han presentado errores de funcionamiento de Cocotb ni han generado resultados interesantes para los errores inyectados.

### 4.2.3 Delay Timer

Empezamos las pruebas con Verilog mediante este dispositivo temporizador. Este DUT no es para nada complejo, ni presenta ninguna característica especial, pero es un dispositivo interesante de simular en lo que a inyección de errores se refiere. En lo que respecta a las señales en las que vamos a utilizar, solo vamos a realizar inyección en la señal “trigger”, buscando alterar brevemente el funcionamiento del delay, lo cual puede llevar a propagar errores si este dispositivo forma parte de otro más complejo. Señales y resultados en las siguientes ilustraciones:

```
#VARIABLES
SIMULATION_CYCLES=250

TIME_UNIT="ns"          #sec, ns, etc
CLK_PERIOD=500
CLK_NAME="clk"           #CLOCK SIGNAL NAME
COM_P=1

T_RESET=35                #INITIAL RESET DURATION
RESET_VALUE=0            #RESET ACTIVE AT "1" OR "0"
RESET_NAME="reset"            #RESET SIGNAL NAME
RESET_P=0

#NORMAL MODE        (IF YOU ARE USING CAMPAIGN MODE, YOU CAN FILL THIS RANDOMLY)
VAR=["trigger"]
CI=[85]
DU=[1]

VAL=[1]
BIT=["NULL"]      #NULL IF YOU DONT WANT TO USE THIS FEATURE
```

Ilustración 43: Variables introducidas en la simulación del Delay Timer

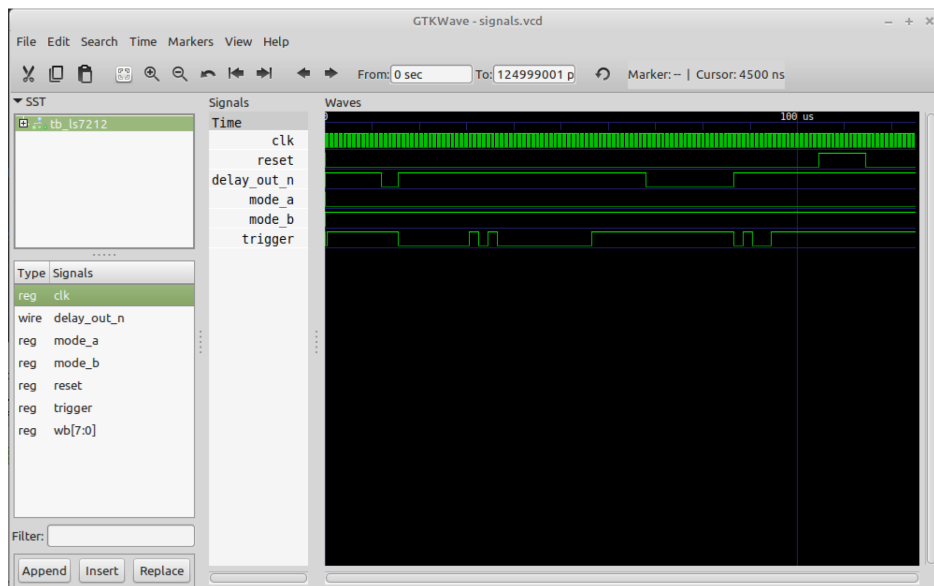


Ilustración 44: Resultado (sin inyección) de la simulación del Delay Timer

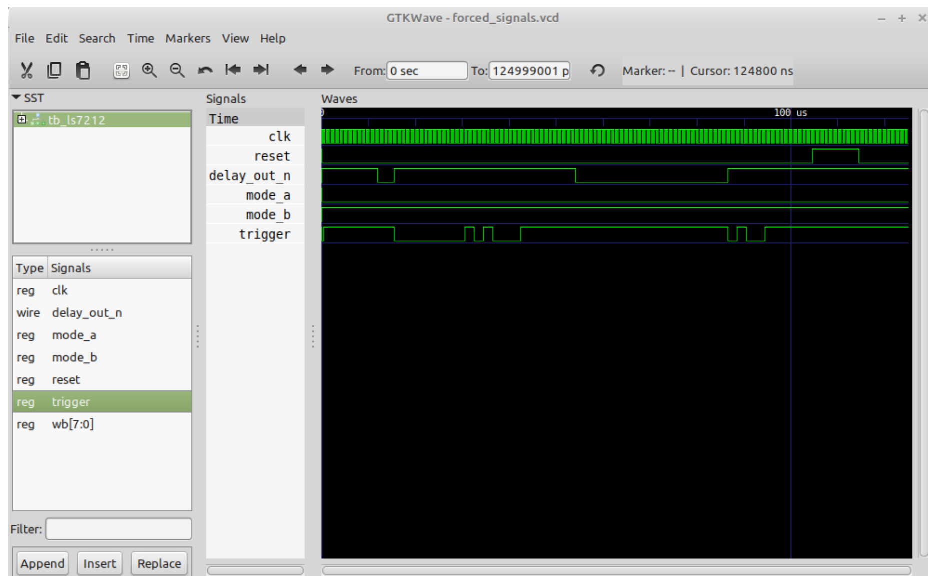


Ilustración 45: Resultado (con inyección) de la simulación del Delay Timer

En la simulación sin fallos podemos comprobar como seria el correcto funcionamiento de este delay. En lo que a nosotros nos interesa, tenemos una señal “delay_out_n” que es el propio delay generado y una señal “trigger”, que como su nombre indica, activa el delay (El resto del código está disponible junto con los resultados como en todas las simulaciones). Comparando el funcionamiento de la señal forzada y la señal sin forzar se puede apreciar como una pequeña alteración en el funcionamiento de “trigger” puede alterar de manera notable la señal de salida. Con esta prueba lo que estamos intentando demostrar es cómo con Cocotb pueden simularse esta clase de fallos simples, ya no solo para cambiar el propio DUT, sino para mejorar un diseño mayor del que forme parte, evitando futuros problemas.

Como siempre, en el modo campaña se han realizado veinte simulaciones distintas. Aparte de la señal “Trigger”, se han alterado las señales responsables de los modos. Como viene siendo habitual en este capítulo, no ha habido ningún problema que haya impedido realizar la campaña de inyección de errores, la cual ha arrojado los resultados de simulación esperados. “Trigger” generando problemas a la salida y los cambios de modo repentinos generando errores similares a los causados por “Trigger”.

### 4.2.4 FIFO

Pasamos ahora al último ejemplo de este capítulo 4, la inyección de errores en una memoria FIFO. Al igual que la UART en protocolos de comunicación, la FIFO es un estándar de la industria en lo que a almacenamiento de datos se refiere, por lo que inyectar errores en algo de estas características puede llegar a ser muy interesante. A parte de todo esto, se ha elegido este dispositivo en cuestión debido al gran número de módulos que lo forman. La UART, a pesar de ser compleja, solo se compone de un módulo (sin incluir el testbench). Que la memoria FIFO presente cinco módulos nos puede ser muy útil para comprobar si Cocotb funciona correctamente inyectando errores en sistemas con estas cualidades.

En lo que a la propia inyección se refiere, vamos a inyectar errores en las variables “fifo_full” y “fifo_empty”, variables que se encargan de indicar cuando la memoria está llena o vacía respectivamente. Inyectando este error esperamos conseguir que el DUT pare de sacar o meter datos en memoria. Las señales y resultados se muestran en las siguientes ilustraciones:

```
#VARIABLES
SIMULATION_CYCLES=100

TIME_UNIT="ns"          #sec, ns, etc
CLK_PERIOD=20
CLK_NAME="clk"           #CLOCK SIGNAL NAME
COM_P=1

T_RESET=35                #INITIAL RESET DURATION
RESET_VALUE=0            #RESET ACTIVE AT "1" OR "0"
RESET_NAME="rst_n"            #RESET SIGNAL NAME
RESET_P=1


#NORMAL MODE        (IF YOU ARE USING CAMPAIGN MODE, YOU CAN FILL THIS RANDOMLY)
VAR=["fifo_empty","fifo_full"]
CI=[80,20]
DU=[5,10]

VAL=[1,1]
BIT=["NULL","NULL"]      #NULL IF YOU DONT WANT TO USE THIS FEATURE
```

Ilustración 46: Variables introducidas en la simulación de la FIFO

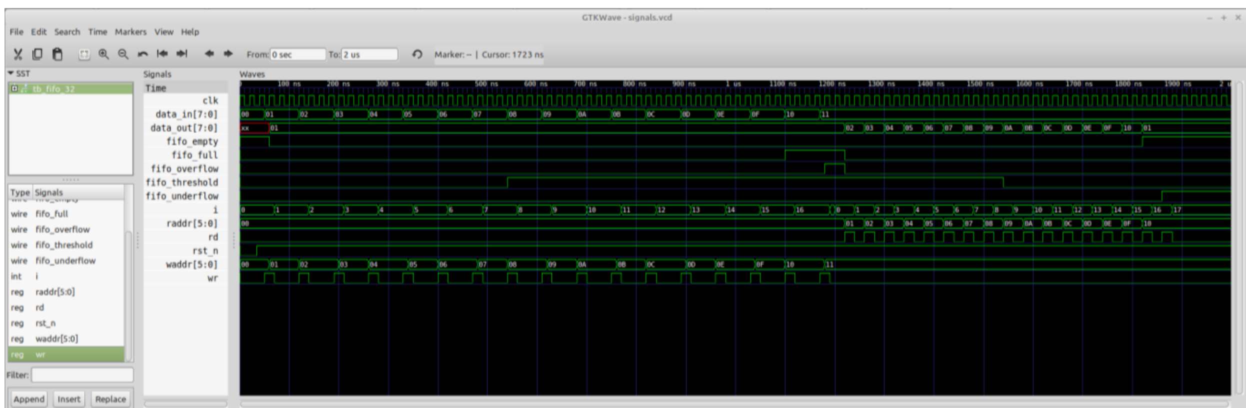

Ilustración 47:  Resultado (sin inyección) de la simulación de la FIFO

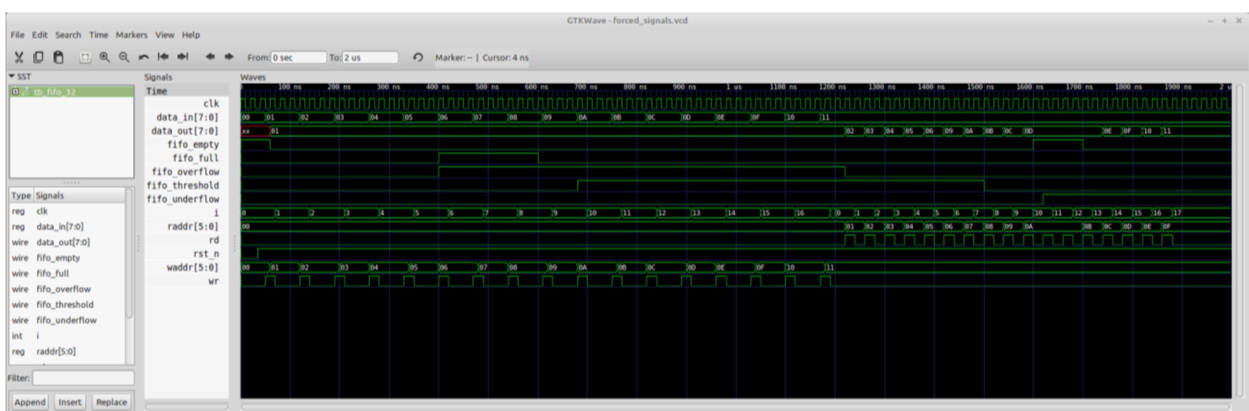

Ilustración 48: Resultado (con inyección) de la simulación de la FIFO

Como siempre, en la primera simulación podemos comprobar cómo funciona el testbench diseñado sin errores. El funcionamiento de este es bastante sencillo, en primer lugar, la llenamos hasta el límite, y después la vaciamos. En lo que respecta a la simulación con errores, Cocotb ha conseguido inyectar los fallos sin problema alguno, aunque en lo que respecta al propio diseño ha ocurrido algo inesperado. Al cambiar el valor de

"fifo_empty" se ha parado de leer de la propia FIFO, lo cual es el funcionamiento lógico. Sin embargo, al cambiar "fifo_full" la FIFO ha seguido su funcionamiento normal y nos ha permitido seguir metiendo valores en memoria. La conclusión de esto es la misma que para las opciones anteriores. Gracias a Cocotb y su inyección de errores hemos conseguido descubrir un fallo de diseño en el dispositivo, lo cual en un proceso de diseño, propiciaría su solución.

En la simulación en modo campaña tampoco ha habido problemas a la hora de la inyección. Hemos seguido probando la teoría previa y las simulaciones lo han corroborado.

# CAPÍTULO 5: CONCLUSIONES FINALES

A lo largo de este trabajo hemos explorado las capacidades de Cocotb y el resto de herramientas de software libre (GHDL, Icarus, Gtkwave, etc), demostrando que se puede realizar una inyección de fallos en diseños VHDL y Verilog exclusivamente con software libre. Además de esto, hemos desarrollado nuestras propias herramientas, desarrollando aún más las capacidades de Cocotb, incluyendo la capacidad de realizar campañas de inyección de fallos sin la necesidad de hardware externo. Todo esto queda demostrado en las pruebas de concepto realizadas a lo largo de la memoria, siendo todas y cada una de ellas exitosas.

Con todo esto podemos decir que Cocotb es una herramienta muy útil para un diseñador digital, ya sea en inyección de errores o en cualquier etapa del diseño. Sin embargo, requiere de unos conocimientos previos y de un nivel de entendimiento de la herramienta relativamente alto para poder utilizarla de forma eficiente.

Cocotb es capaz de realizar simulaciones simples de inyección de errores para cualquier dispositivo de manera rápida y eficaz. Con las herramientas de desarrollo utilizadas en esta memoria, no se tarda más de diez minutos en preparar un DUT para inyectarle errores, independientemente de la complejidad del propio DUT. Incluso una persona ajena al funcionamiento del dispositivo puede realizar este proceso, aunque como es lógico, no sería capaz de analizar los resultados del mismo. Esta es una de las grandes ventajas de Cocotb a la hora de compararlo con otros simuladores, el hecho de que, con las herramientas adecuadas, es un simulador rápido y eficaz capaz de proporcionar datos utilizables por los diseñadores, permitiéndoles corregir posibles errores en el diseño de su DUT.

Sin embargo, esta velocidad viene con su contraparte. Al utilizar las herramientas proporcionadas estamos desaprovechando una de las mejores características a la hora de usar Cocotb por el mero hecho de optimizar todo el proceso, los test en Python. Durante todo este proyecto hemos trabajado usando testbenches en HDL como toplevel de nuestras simulaciones. Esto ha optimizado todo el proceso, permitiéndonos realizar simulaciones de inyección de errores casi sin ningún trabajo previo, pero quitándole a Cocotb la propia esencia de su funcionamiento.

Un test en Python es infinitamente más flexible que un test en HDL, solo hace falta un rápido vistazo a los ejemplos que proponemos en esta memoria o a los ejemplos disponibles en la propia wiki de Cocotb. Al utilizar el test en HDL estamos perdiendo capacidades. No solo en el hecho de flexibilidad, sino también en el ámbito de los resultados obtenidos. Si volvemos atrás hasta la guía de instalación, y observamos los resultados de las simulaciones básicas realizadas con el contador, podemos observar que presentan una característica que no podemos apreciar en el Capítulo 4. Las simulaciones realizadas con un testbench en Python nos permiten acceder a las señales internas de los dispositivos. Esto ya lo mencionamos a la hora de explicar el funcionamiento del inyector, cuando comparamos a forzar una señal interna y una salida del contador. Se podía observar perfectamente como el fallo introducido en la salida no se propagaba. Esta característica en si misma es para muchos, más útil que la posibilidad de realizar simulaciones a grandes velocidades como hemos ido realizando a lo largo de este proyecto, pero como es lógico, requiere de mayor preparación.

Si tenemos un dispositivo complejo y queremos poder acceder a las señales internas, no podemos usar los testbenches en HDL que se han ido definido en la propia etapa de diseño del DUT. Tendremos que usar un test

en Python. Si queremos probar todos los testbenches del DUT, necesitaremos replicar todos y cada uno de los testbenches del dispositivo en Python, cosa que puede ser incluso imposible si los diseñadores del dispositivo desconocen el funcionamiento de este lenguaje. Para simular diseños complejos se requeriría del uso de corutinas y elementos avanzados del propio Cocotb, sin mencionar que todo lo desarrollado para un diseño, probablemente no fuera transferible para otro dispositivo. Este simple hecho puede provocar que muchos diseñadores que escuchen de Cocotb lo consideren un programa con una cantidad de opciones y posibilidades extremadamente amplia, que probablemente no utilizarán jamás.

Y es con esto con lo que vamos a terminar este proyecto y esta conclusión. Cocotb presenta una serie de opciones que la competencia no presenta, pero que requiere de unos conocimientos que pueden quedar fuera del alcance de muchos. Si lo utilizamos sin llegar a aprovechar todas sus opciones, o intentando maximizar la velocidad o la flexibilidad (tal y como hemos intentado a lo largo de todo este trabajo), estaremos perdiendo las características que lo hacen único ante otros simuladores/inyectores de fallos. Con todo, es cierto que es capaz de realizar simulaciones de inyección de fallos exclusivamente con software libre, y si usamos las herramientas desarrolladas, podemos realizar campañas enteras de simulación sin hardware externo.

# REFERENCIAS


[1] **Digital Design and Computer Architecture**, David Harris and Sarah Harris, *"Book", 2012*

[2] **Engineering Digital Design,** Richard F. Tinder, *"Book", 2000*

[3] **Verilog para diseñadores de VHDL,** Hipólito Guzmán Miranda, *"Departamento de Ingeniería Electrónica", sin fecha*

[4] **Assessing Dependability with Software Fault Injection: A Survey,** Roberto Natella and Domenico Cotroneo, *ACM Comput. Surv. 48, 3, Article 44 (February 2016), 55 pages*

[5] **A Survey on Fault Injection Techniques**, Haissam Ziade, Rafic Ayoubi, and Raoul Velazco, *The International Arab Journal of Information Technology, Vol. 1, No. 2, July 2004*

[6] **Comparison of Physical and Software-Implemented Fault Injection Techniques**, Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs and Gu¨ nther H. Leber, *IEEE transactions on computers, Vol 52, No 9, September 2013*

[7] **FT-UNSHADES-uP: A platform for the analysis and optimal hardening of embedded systems in radiation environments,** H. Guzmán-Miranda, J. N. Tombs and M. A. Aguirre, *Conference: Industrial Electronics, 2008. ISIE 2008*

[8] **Free Software, Free Society: Selected Essays of Richard M. Stallman,** Richard M. Stallman, "*Book*", *2002*

[9] **Busqueda Web,** https://www.dinero.com/negocios/articulo/estos-lideres-mundiales-del-mercado-del-software/112147

[10] **Cocotb´s documentation,** https://docs.cocotb.org/en/stable/

[11] **Icarus Verilog´s Fandom,** https://iverilog.fandom.com/wiki/Main_Page

[12] **fpga4student,** https://www.fpga4student.com/

[13] **Busqueda Web**, https://commons.wikimedia.org/wiki/File:Coroutine.png

# ANEXO

Todo el código está disponible en un repositorio propiedad del departamento de Ingeniería Electrónica de la Universidad de Sevilla. Para usuarios no pertenecientes a la entidad será necesario comunicarse con dicho departamento para poder acceder al repositorio.

usuario@woden.us.es:/var/git/fisim.git

Una vez se disponga de acceso, recomendamos descargar el archivo comprimido "tools", que contiene las plantillas de las herramientas desarrolladas como los ejemplos del capítulo 4.