Trabajo de fin de grado del Doble Grado en Ingeniería Informática - Matemáticas

---

Universidad Complutense de Madrid

Facultad de Ingeniería Informática

Departamento de Arquitectura de Computadores y Automática

Nessy 7.0: entorno de inyección de fallos para Artix-7

Presentado por

Juan Andrés Claramunt Pérez

Directores

Hortensia Mecha López

Juan Carlos Fabero Jiménez

Curso académico

2015-2016

Autorización de difusión

Autor

Juan Andrés Claramunt Pérez

Fecha

14 de junio de 2016

El autor, alumno del Doble Grado de Matemáticas e Ingeniería Informática en las Facultades de Informática y Matemáticas, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor, el presente Trabajo Fin de Grado: Nessy 7.0: entorno de inyección de fallos para Artix-7, tanto el presente documento como el programa creado durante la realización de este proyecto, realizado durante el curso académico 2015/2016, bajo la supervisión y dirección de Hortensia Mecha López y Juan Carlos Fabero Jiménez en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense, para ayudar a incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Resumen

El hardware reconfigurable es una tecnología emergente en aplicaciones espaciales. Debido a las características de este hardware, pues su configuración lógica queda almacenada en memoria RAM estática, es susceptible de diversos errores que pueden ocurrir con mayor frecuencia cuando es expuesta a entornos de mayor radiación, como en misiones de exploración espacial. Entre estos se encuentran los llamados SEU o Single Event Upset, y suelen ser generados por partículas cósmicas, pues pueden tener la capacidad de descargar un transistor y de este modo alterar un valor lógico en memoria, y por tanto la configuración lógica del circuito.

Por ello que surge la necesidad de desarrollar técnicas que permitan estudiar las vulnerabilidades de diversos circuitos, de forma económica y rápida, además de técnicas de protección de los mismos.

En este proyecto nos centraremos en desarrollar una herramienta con este propósito, Nessy 7.0.

La plataforma nos permitirá emular, detectar y analizar posibles errores causados por la radiación en los sistemas digitales. Para ello utilizaremos como dispositivo controlador, una Raspberry Pi 3, que contendrá la herramienta principal, y controlará y se comunicará con la FPGA que implementará el diseño a testear, en este caso una placa Nexys 4 DDR con una FPGA Artix-7.

Finalmente evaluaremos un par de circuitos con la plataforma.

Palabras clave

FPGA, Artix-7, Raspberry Pi, SEU, bitflip, hardware reconfigurable, inyección fallos.

Abstract

Reconfigurable hardware is an emerging technology in space applications. Due to the characteristics of this hardware, since their logical configuration is stored in static RAM, it is susceptible to various events that can occur more frequently when exposed to high radiation environments, like in space exploration missions. These events are called Single Event Upset or SEU, and are usually generated by cosmic particles, they may be able to discharge a transistor and thus alter a logical value in memory, and therefore the logic circuit configuration.

That is why the need to develop techniques to study the vulnerabilities of various circuits, economically and quickly arises.

In this project we will focus on developing a tool for this purpose, Nessy 7.0.

This platform will allow us to emulate, detect and analyze possible errors caused by radiation in digital systems. We will use as a controller device, a Raspberry Pi 3 that contain the main tool, and control and communicate with the FPGA to implement the design under test, in this case a Nexys 4 DDR board with an Artix-7 FPGA.

Finally we evaluate a couple of circuits with the developed platform.

Keywords

FPGA, Artix-7, Raspberry Pi, SEU, bitflip, reconfigurable hardware, fault injection.

Índice general

Resumen **III**

Lista de figuras **VII**

Lista de tablas **IX**

Lista de códigos **XI**

1. Introducción **1**

1.1. Motivación 1

1.2. Trabajo relacionado 3

1.3. Objetivos 3

2. Entorno de desarrollo Hardware y Software **5**

2.1. Arquitectura Hardware 5

2.1.1. FPGA y placa 6

2.1.2. Raspberry Pi 3 10

2.1.3. Chip PCF8574 13

2.2. Entorno de desarrollo y Software 16

2.2.1. Entorno de desarrollo 16

2.2.2. Software 18

3. Nessy **21**

3.1. Descripción 21

3.2. Carga de archivos y formato 23

3.2.1. DUT o diseño a prueba (Bitstream) 23

3.2.2. Archivo de ubicación lógica 24

3.2.3. Archivo de test 25

3.2.4. Golden 27
3.3. Inyección de fallos 28
3.3.1. Inyección en flipflops 33
3.3.2. Inyección en memoria de configuración 36
3.4. Panel de depuración 40
3.5. Scripts de visualización 41
3.5.1. Script de inyección en memoria 41
3.5.2. Script de inyección en flipflops 42

4. Resultados y conclusiones **45**
4.1. Algoritmo de Checksum de Fletcher de 16 bits 45
4.2. Contador binario de 8 bits 49
4.3. Conclusiones 51
4.4. Trabajo futuro 52

Bibliografía **53**

Anexo A: Bitstream **55**

Anexo B: Archivo .ll **59**

Anexo C: JSON **61**

Índice de figuras

2.1. Esquema de conexiones 6
2.2. Placa Nexys 4 DDR 7
2.3. Conectores PMOD de Nexys 4 DDR 7
2.4. CLB de la familia Artix 7 de FPGAs 9
2.5. Matriz general de enrutamiento 10
2.6. Placa de la Raspberry Pi 3 10
2.7. Pines GPIO de la Raspberry Pi 3 11
2.8. Diagrama de pines del chip PCF8574 13
2.9. Diagrama de bloques del chip PCF8574 14
2.10. Qt Creator 17
2.11. Xilinx ISE 18
2.12. Esquema del proceso de diseño de Xilinx ISE 19
2.13. Esquema de comunicación de aplicaciones 19
2.14. OpenOCD 20

3.1. Diseño del circuito del proyecto 21
3.2. GUI de Nessy 22
3.3. Parte de un módulo .ll de un diseño 24
3.4. Ejemplo de un fichero Testbench usado en este proyecto 26
3.5. Fichero Golden 27
3.6. Diagrama de flujo de la inyección en flipflops 34
3.7. Fichero resultados de inyección en flipflops 35
3.8. Selección de la región de la memoria de configuración sobre la que inyectar. 37
3.9. Diagrama de flujo de la inyección en memoria 38
3.10. Fichero resultados de inyección en memoria 40

4.1. Fallos por inyección en flipflops en cada ciclo del circuito 46
4.2. Éxitos por flipflop en la inyección 47
4.3. Inyección en celda de memoria ($X2$, $Y0$) 48
4.4. Inyección en celda de memoria ($X2$, $Y0$) 50
A.1. Paso de sincronización 56

Índice de tablas

2.2. Funciones de los pines del chip PCF8574 14

3.3. Registros 30
3.5. Descripción de los campos del registro de direccion de *frame* . . 32

4.1. Estadisticas inyección en flipflops 46
4.2. Estadisticas inyección en memoria de configuración 48
4.3. Estadisticas inyección en flipflops 49
4.4. Estadisticas inyección en memoria de configuración 50
A.1. Formato de la cabecera del Bitfile 55
A.2. Formato del bitstream 57

Índice de códigos

2.1. Implementación de la función escritura 15
3.1. Función para analizar el fichero `.ll` 25
3.2. Acceso al pin BCM 4 a través de la biblioteca de Python RPi.GPIO 31
3.3. Uso de las direcciones en memoria de los pines 31
3.4. Fragmento de código del script de visualización en memoria . . . 42
3.5. Fragmento de código del script de visualización en flipflops . . . 42

1 Introducción

1.1. Motivación

Existen principalmente dos métodos en la computación tradicional para ejecutar algoritmos.

El primero es usar un circuito integrado de aplicación específica, o ASIC (Application Specific Integrated Circuit), para realizar operaciones con el hardware. Como estos circuitos están diseñados para realizar unas tareas específicas suelen ser muy rápidos y eficientes realizando el cometido para el que han sido diseñados, sin embargo después de su fabricación, el circuito no puede ser modificado.

Por otro lado los Microprocesadores son una alternativa más flexible. Los procesadores ejecutan un conjunto de instrucciones para computar una tarea y cambiando las instrucciones de software podemos modificar la funcionalidad del sistema sin modificar su hardware. Sin embargo esta flexibilidad se consigue a costa de un decremento en la eficiencia, mucho menor que la conseguida en los circuitos de aplicación específica.

La computación reconfigurable intenta llenar el hueco entre estos dos enfoques, consiguiendo potencialmente mucha mayor eficiencia que una solución por software, a la vez que mantiene una mayor flexibilidad que la solución por hardware específica.

Es por ello que el hardware reconfigurable es una tecnología emergente en aplicaciones espaciales. La base de esta tecnología es la capacidad de implementar diferentes funcionalidades y ser configuradas de forma remota, basándose esta capacidad en tecnología de dispositivos de lógica programable como lo son los CPLDs (Complex Programmable Logic Device), dispositivos con tecnología basada en memoria flash y las FPGA (Field Progammable Gate Arrays).

Una FPGA dispone de una matriz de elementos lógicos, que pueden ser interconectados y configurados para funciones específicas. Todas las definiciones lógicas y las interconexiones entre bloques son controladas por celdas de RAM estática o memoria estática de acceso aleatorio, lo que permite una reconfiguración parcial o total de las funcionalidades del circuito "al vuelo", es decir,

mientras se está ejecutando un circuito.

Sin embargo, el hecho de que la memoria de configuración, la que describe las conexiones de los bloques y las funciones lógicas se base en celdas de memoria SRAM, la hace susceptible de ser alterada por partículas cósmicas, y de esta forma es posible alterar el comportamiento del circuito quizá de forma temporal si solo afecta a un flipflop y no a la configuración del circuito en si, o de forma no transitoria si afecta a elementos de la memoria de configuración.

En el espacio esta situación se ve agravada por los entornos de alta radiación, donde se incrementa la exposición del circuito a sufrir colisiones de partículas solares. Además, estudios recientes muestran que los cinturones de Van Allen, los anillos de radiación que rodean nuestro planeta, son más dinámicos de lo que se pensaba y sus cambios podrían aumentar los daños causados por tormentas solares.

A la alteración del circuito de este modo se le conoce como SEU (Single Event Upset), que son eventos no destructivos causados por colisiones de radiación ionizada en los transistores CMOS, que pueden descargar la carga de elementos de almacenamiento como lo son las celdas de memoria de configuración, la memoria de usuario o los registros. Aunque existen técnicas de protección contra este efecto como escudos o redundancia de circuitos, éstas además de ser costosas no son infalibles y se hace necesaria la existencia de técnicas para evaluar la sensibilidad y vulnerabilidad a estos errores de los diseños para las FPGA.

Los circuitos específicos o ASIC, dependiendo de la tecnología empleada, pueden disponer de cierta tolerancia frente a este tipo de eventos o SEU, porque su configuración lógica no se ve alterada aunque si los flipflops y memoria de datos. Sin embargo, como hemos mencionado, su funcionalidad no puede ser modificada, y como resultado no sólo no podrá ser actualizada o corregida de algún modo, sino que cualquier daño permanente en sus subcircuitos como un evento de tipo SEGR o Single Event Gate Rupture, que a diferencia de los SEU si son eventos destructivos, dejará esa región del circuito desactivada para siempre.

La capacidad de la FPGA para ser reprogramada, es una ventaja en este sentido, además de permitir la actualización de un diseño, permite la detección de estos eventos y su corrección un ilimitado número de veces y si se produce un evento de ruptura de puerta y deja alguna región de la lógica inoperativa, el diseño funcional de esa región puede ser fácilmente reimplementado para no utilizar el área afectada.

Lo anteriormente expuesto nos trae la necesidad de poder estudiar la sensibilidad y vulnerabilidad de los distintos diseños frente a estos tipos de eventos de forma rápida y precisa.

1.2. Trabajo relacionado

Existen varios enfoques para analizar los efectos de los SEU en dispositivos FPGA.

La sensibilidad de un dispositivo a un evento SEU puede ser estimada empíricamente colocando el dispositivo en un flujo de partículas en un acelerador de partículas, lo cual tiene sus ventajas y desventajas. Como ventaja nos proporcionaría una estimación correcta y precisa sobre la sensibilidad del diseño, a cambio de un gran coste y de daños permanentes en los dispositivos utilizados.

El enfoque analítico, se basa en técnicas probabilísticas. Utiliza algoritmos para clasificar los distintos bits de configuración respecto a un diseño, es decir para identificar qué bits de la configuración son esenciales para un determinado circuito. Si un bit esencial se ve alterado, modifica el diseño del circuito, aunque puede no afectar a la función del diseño. Se generan una lista de distintos SEU a simular, y un algoritmo los va seleccionando y modificando la topología del grafo del circuito en consecuencia, evaluando la topología final. En caso de modificación estructural el SEU se clasifica como un error.

Sus ventajas son que utiliza un tiempo reducido, no causa daños permanentes y bajo coste, sin embargo requiere un alto nivel de esfuerzo en las fases de desarrollo.

El enfoque en que nos centraremos son los métodos de inyección de fallos, aprovechando las capacidades que ofrece la FPGA de readback y reconfiguración parcial. Sus principales ventajas son su bajo coste y que no causa daños permanentes en la plataforma, pero puede requerir grandes cantidades de tiempo.

Nos centraremos en estos métodos y desarrollaremos una herramienta con este propósito, con la intención de ser lo menos invasiva posible respecto al diseño del circuito original.

1.3. Objetivos

Como hemos mencionado, nos centraremos en métodos basados en FPGAs y desarrollaremos una herramienta de inyección de fallos, con la intención de ser lo menos invasiva posible respecto al diseño del circuito original.

Se trata de desarrollar una plataforma de inyección de errores para dispositivos del tipo Artix-7, que permita emular, analizar y detectar los posibles errores producidos por la radiación en sistemas digitales, embarcados en misiones espaciales o en entornos de alta radiación.

Esta plataforma permitirá analizar diseños implementados tanto en hardware reconfigurable (inyectando a nivel de memoria de configuración) como en

otras tecnologías no reconfigurables, inyectando en biestables y memoria de datos. También se pretende separar el circuito a testear del resto de la plataforma Nessy.

Este proyecto consta de las siguientes etapas:

1. Estudio de la arquitectura y de la memoria de configuración de las Artix-7.
2. Realización de una plataforma, NESSY, que permita realizar la inyección de bitflips sobre la memoria de configuración de las Artix-7 y la nueva estructura de los biestables y memoria de datos.
3. Medida de la vulnerabilidad de distintos diseños en entornos radiactivos. Se medirá, entre otros, la eficacia de un algoritmo de detección y corrección de errores.

2 Entorno de desarrollo Hardware y Software

En este capítulo se van a detallar tanto los componentes hardware como el software que han sido usados en todo el proyecto. Por un lado se especificarán las características de la FPGA usada y de los chips que han sido necesarios para la conexión de ésta con una Raspberry. Por otro lado, se explicará el OpenOCD, software que nos facilita el uso, la interconexión y manejo de los componentes hardware, y las herramientas usadas para la síntesis y el análisis del diseño.

2.1. Arquitectura Hardware

Comenzaremos explicando la placa utilizada, detallando las especificaciones técnicas que han sido importantes durante este proyecto, y detallaremos las especificaciones tanto de la FPGA como de la Raspberry Pi que han sido usadas, junto con los componentes necesarios para poder establecer su comunicación.

En la figura 2.1 podemos ver un esquema global de las conexiones entre los dispositivos utilizados.

Una Raspberry 3 con un sistema operativo Raspbian, basado en Debian, será la encargada de ejecutar el núcleo de nuestra herramienta escrito en $C++$. Las Raspberry está conectada a un monitor, además de un teclado y ratón, para permitir al usuario interactuar a través de una interfaz gráfica de forma cómoda y sencilla. Haciendo uso de uno de sus puertos USB se conectará con el puerto microusb de la placa Nexys 4, a través de la interfaz JTAG.

Además, gracias a los pines de entrada/salida de propósito general (que se detallan en la sección 2.1.2) que la Raspberry nos proporciona, estableceremos conexiones por un lado, con los conectores PMOD de la placa que contiene la FPGA, para enviar señales de control como pueden ser las señales de reloj y de reset, y por otro lado, haciendo uso del protocolo I2C y con unos chips PCF8574 de intermediarios, estableceremos 16 conexiones más con los conectores PMOD de la FPGA que nos servirán para controlar las señales de datos, la entrada y salida del circuito a testear. También se encarcará de

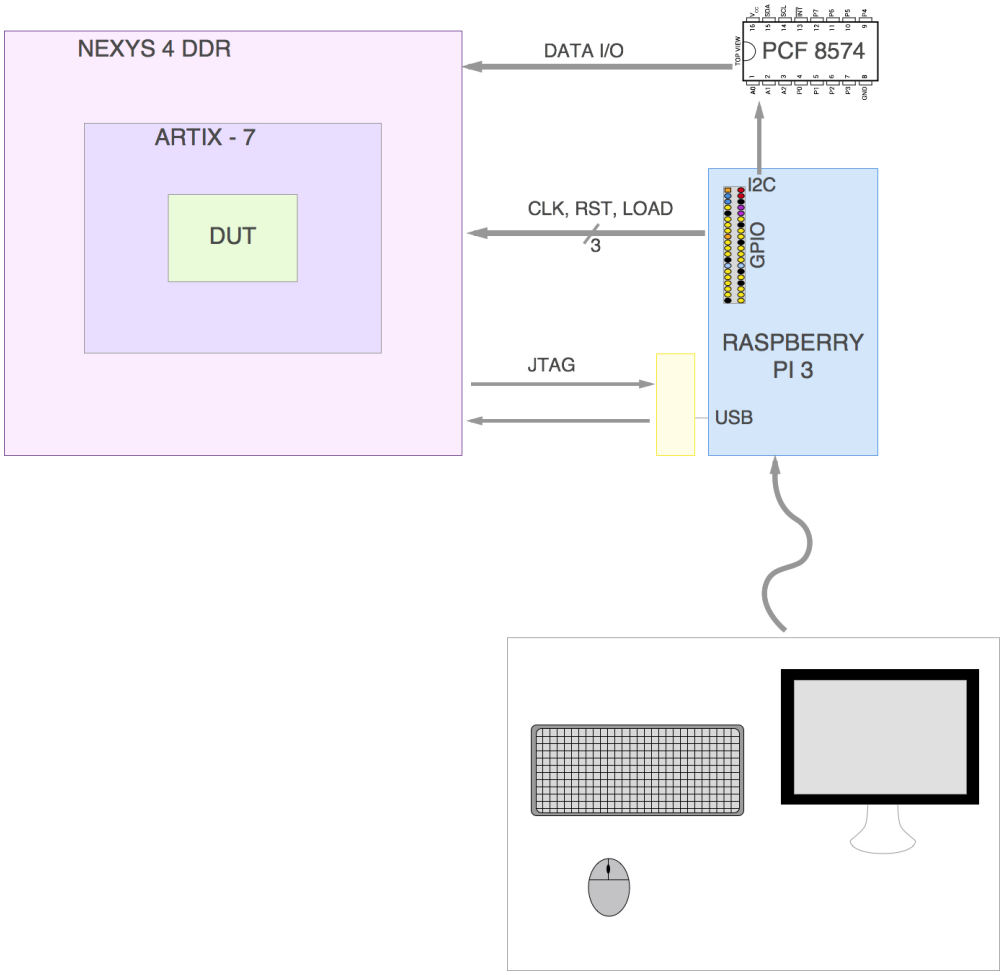


Figura 2.1: Esquema de conexiones

realizar la configuración de la FPGA y las reconfiguraciones parciales que nos permitan simular los fallos.

En la FPGA, una Artix-7 XC7A100T, se implementa el diseño del circuito a testear o DUT (Design Under Test). La herramienta se ha desarrollado con el propósito de no ser invasiva con el circuito que se quiere probar, de forma que la FPGA solamente implementará el circuito a probar y nada más que pueda interferir en las pruebas. Aunque será necesario permitir controlar la señal de reloj de forma externa, en nuestro caso desde la Raspberry.

2.1.1. FPGA y placa

Placa Nexys 4 DDR

La placa Nexys 4 DDR es una placa completa y lista para el desarrollo de plataformas de circuitos digitales basados en la FPGA Artix 7 de Xilinx. Esta placa puede albergar diseños que van desde circuitos combinacionales introductorios a procesadores integrados de gran potencia.

Algunos puertos y periféricos de la placa Nexys 4 DDR son:

- 4,860 Kbits de bloque de RAM

- 16 “LEDs”
- 16 “switches”
- Sensor de temperatura
- Conector de tarjeta MicroSD
- Ocho “displays” de 7 segmentos en dos paneles de 4 dígitos

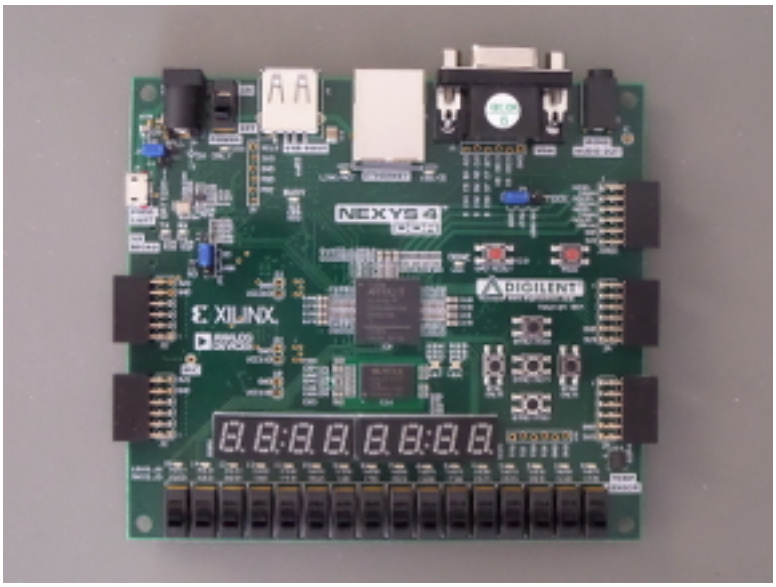


Figura 2.2: Placa Nexys 4 DDR

Entre las características de esta placa, se encuentra la de contener dos memorias externas, una memoria de 1 Gb (128 MB) DDR2 SDRAM y otra memoria de 128Mb (16MB), memoria del dispositivo flash no volátil de serie.

En la figura 2.2, podemos visualizar cómo es físicamente la placa Nexys 4 DDR.

Los conectores que han sido usados en el desarrollo de este proyecto de la placa Nexys 4 DDR han sido los conectores Pmod. Los conectores Pmod están distribuidos en dos filas, y cada fila tiene 6 conectores pines. Por cada 12 pines, los conectores Pmod, proporcionan dos señales de $3.3V$ VCC (correspondientes a los pines 6 y 12), dos señales de tierra (correspondientes a los pines 5 y 11) y ocho señales lógicas, como podemos ver en la figura 2.3.

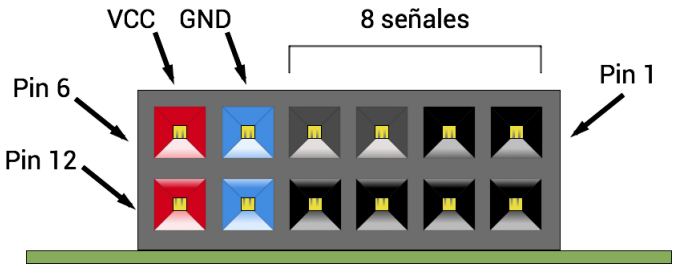


Figura 2.3: Conectores PMOD de Nexys 4 DDR

Por último, comentar que esta placa es compatible con la nueva versión de la herramienta de desarrollo de Xilinx, llamada Vivado.

FPGA

Las FPGAs (Field Programmable Gate Array) son circuitos electrónicos que contienen chips de silicio reprogramables, usando bloques de lógica preconstruidos a los que se les puede configurar una funcionalidad de manera programable mediante una memoria SRAM. Así, las FPGAs son completamente reconfigurables, ya que su memoria de configuración se puede borrar y almacenar una nueva.

Detallemos algunas aplicaciones importantes que tienen las FPGAs hoy en día:

- Sistemas de visión artificial (cámaras de vigilancia, robots ...).
- Sistemas de imágenes médicas (tratamiento de imágenes biomédicas obtenidas mediante rayos X, entre otras).
- Radio definida por software, de manera que puede limitarse la parte analógica a una antena y unos convertidores, sustituyendo así la forma tradicional en la que una antena se encargaba de recibir y enviar la señal y un hardware procesaba esa señal, la filtraba y modificaba su frecuencia.
- Codificación y encriptación.
- Reconocimiento de voz.
- Aeronáutica y defensa.

FPGA Artix 7

La familia de FPGA Artix-7 de Xilinx ha mejorado a sus precedesoras en varios sentidos, donde destaca la mejora del consumo de energía (su consumo es la mitad que las anteriores).

En concreto, en este proyecto se va a utilizar el dispositivo $XC7A100T$ de la FPGA Artix 7.

Algunas de las características de la FPGA Artix 7 $XC7A100T$ son:

- *Celdas lógicas*: 101.440
- *Slices*[1]: 15.850
- Máximo RAM distribuída: 1.188 Kb

[1]Cada 'slice' de las FPGA de la serie 7' contiene cuatro LUTs y ocho 'flip-flops". Sólo algunos "slices" pueden usar sus LUTs como una RAM distribuida.

- Bloques de RAM
  - Bloques de 18 Kb: 270
- Número total de bancos de I/O: 6
- Máximo I/O de usuario: 300
- Un chip de conversión de analógico a digital (XADC)
- La velocidad del reloj interno es superior a 450 MHz

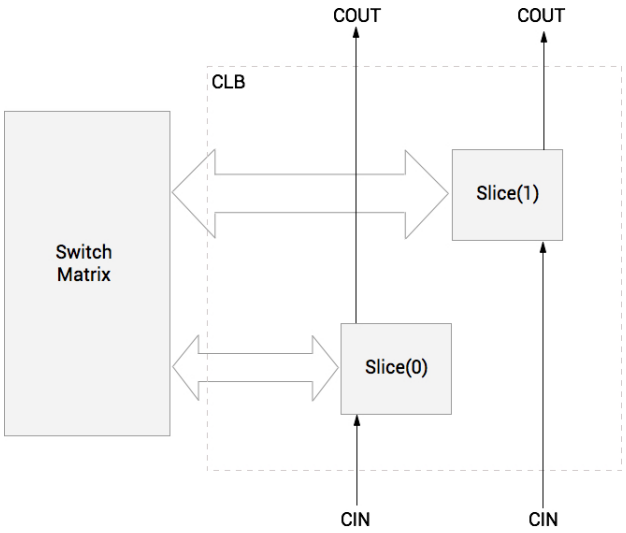


Figura 2.4: CLB de la familia Artix 7 de FPGAs

El CLB de la familia Artix 7 de FPGAs, que puede verse en la figura 2.4, cuenta con algunas de las siguientes características:

- 6 entradas reales a la LUT (look-up table technology)
- Opción de Dual LUT5
- Memoria distribuida y capacidad de *Shift Register Logic*
- *high-speed carry logic* dedicado para funciones aritméticas

Los CLBs son los principales recursos lógicos que implementan tanto circuitos secuenciales como combinacionales. Cada elemento del CLB está conectado con una *matriz de conmutación* para acceder a la matriz general de enrutamiento (*general routing matrix*) como se puede ver en la figura 2.5.

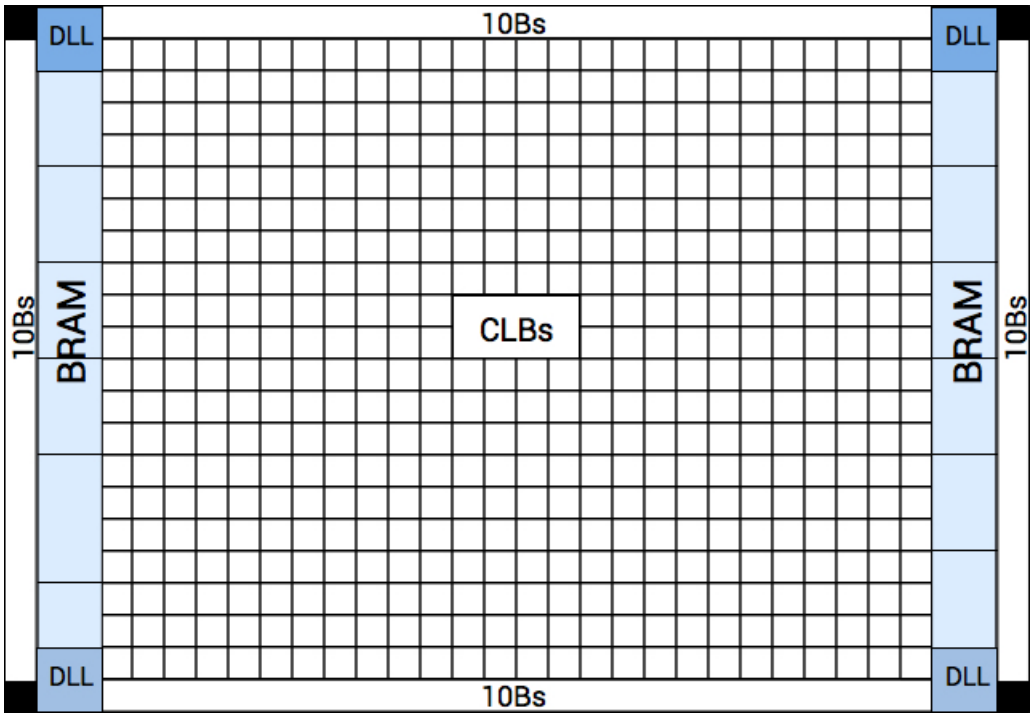


Figura 2.5: Matriz general de enrutamiento

2.1.2. Raspberry Pi 3

En este proyecto, se ha realizado la conexión de la FPGA Nexys DDR 4 con una Raspberry Pi 3 (figura 2.6). Se ha escogido este dispositivo por su gran versatilidad, ya que nos proporciona a la vez un completo sistema operativo basado en Debian y 40 pines de entrada salida de propósito general.

Raspberry Pi es una placa de desarrollo de bajo coste desarrollado en Reino Unido por la Fundación Raspberry Pi.

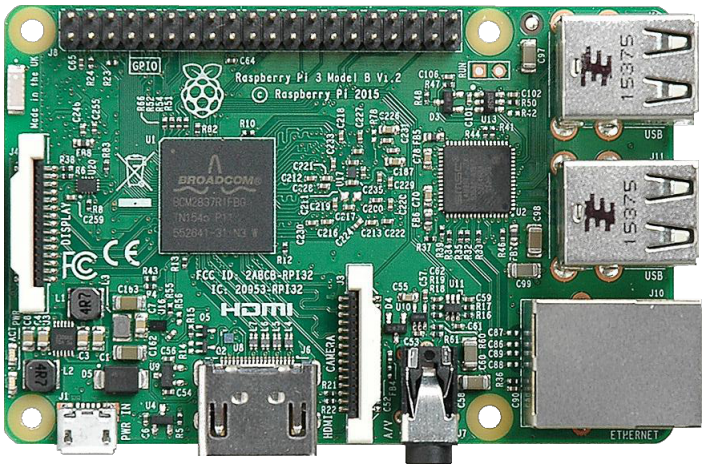


Figura 2.6: Placa de la Raspberry Pi 3

Características de la Raspberry Pi 3

Tiene las siguientes características técnicas:

- CPU ARMv8 de cuatro núcleos de 64 bit a 1.2 GHz
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)
- 1 GB de RAM
- 4 puertos USB
- 40 pins GPIO[2]
- Puerto HDMI
- Puerto Ethernet
- Puerto de tarjeta MicroSD
- Otras características de gráficos e interfaz de cámara

Pines GPIO

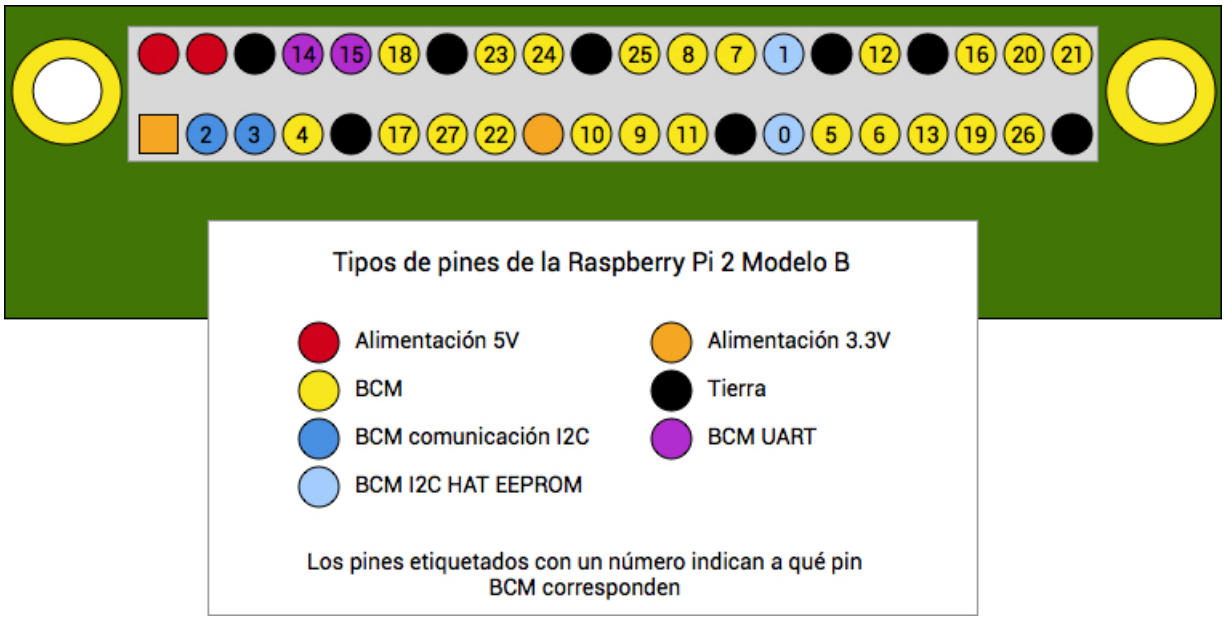


Figura 2.7: Pines GPIO de la Raspberry Pi 3

La Raspberry Pi 3 consta de 40 pines GPIO, como hemos visto en las características. Para ver el uso de cada uno de los pines, expliquemos qué posibilidades tenemos:

- **Alimentación**: Pueden proporcionar, dependiendo del pin elegido, diferentes voltajes:

[2]GPIO (General Purpose Input/Output, Entrada/Salida de Propósito General) es un pin genérico en un chip, cuyo comportamiento se puede programar por el usuario en tiempo de ejecución.

  - 3.3 Voltios: Estos pines tienen una corriente máxima disponible de unos $50\,mA$. Es suficiente para alimentar un par de LEDs.
  - 5 Voltios: Estos pines están conectados directamente a la alimentación de entrada de la Raspberry Pi, y nos pueden llegar a proporcionar la totalidad de la corriente de la fuente de alimentación que el usuario utilice menos la cantidad de corriente que use la Raspberry Pi.

- **Tierra**: Estos pines están todos electricamente conectados entre sí, por lo que es indiferente usar uno u otro.
- **BCM**: *Broadcom pin number*, comúnmente llamados GPIO. Dentro de estos, hay pines que están reservados y otros que tienen usos exclusivos:
  - BCM 0 y BCM 1: reservados para la comunicación I2C con un HAT EEPROM. [3].
  - BCM 2 y BCM 3: SDA y SCL, respectivamente, son los pines I2C en la Raspberry Pi, que son pines útiles para la comunicación con distintos tipos de periféricos externos.
  - BCM 14 y BCM 15: Estos pines tienen dos usos. Por un lado, como pin de transmisión o recepción UART [4], respectivamente. En segundo lugar, también se les conoce como "Serial", y por defecto, te ayudarán a enviar y recibir comandos, de una consola y junto con el cable Serial adecuado, podrás controlar la Raspberry Pi.

Las conexiones que hemos realizado son:

- *BCM Pin 4* Señal de reloj.
- *BCM Pin 17* Señal de reset.
- *BCM Pin 27* Señal de load.
- *BCM Pines 2 y 3* Pines I2C conectados a dos chips pcf8574.

[3]Comunicación I2C (inter-integrated circuit), tipo de comunicación serie definida a mediados de los 80 para trabajar a 100kbit/s y en 1995 se definió un modo rápido de funcionamiento que permitía transferencias de hasta 400kbit/s (algunos dispositivos gráficos llegan hasta 3,4MHz)

[4]Universal Asynchronous Receiver-Transmitter

2.1.3. Chip PCF8574

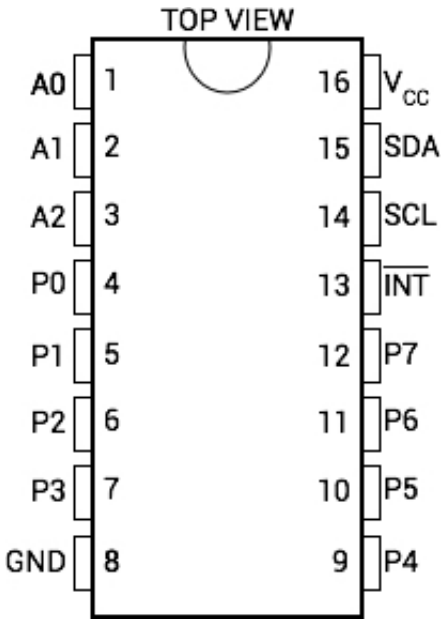


Figura 2.8: Diagrama de pines del chip PCF8574

El chip PCF8574 (figura 2.8) es un expansor de entrada de 8 bits de entrada/salida (I/O) que se comunica mediante dos líneas del bus bidireccional (I2C) y está diseñado para operaciones en las que V_{CC} esté en el rango de $2.5V$ a $6V$.

Este dispositivo PCF8574 proporciona una extensión remota de I/O de propósito general para la mayoría de las familias de microcontroladores vía la interfaz I2C (serial clock (SCL) y serial data (SDA)). Dicho dispositivo se caracteriza por tener un puerto E/S de 8 bits *Quasi-bidireccional* [5](llamados $P0$-$P7$).

El chip PCF8574 es un chip que tiene las siguientes aplicaciones:

- Unidades de filtro en el ámbito de telecomunicaciones
- Servidores
- Routers (Equipamiento de *Switching* en telecomunicaciones)
- Ordenadores propios
- Electrónica propia
- Automatización de la industria
- Productos con Procesadores *GPIO-Limited*, como lo es la Raspberry Pi [6].

[5]Cada E/S quasi-bidirectional puede ser usado como de entrada o salida sin necesitar el uso de una señal de control para la dirección de los datos

[6]La Raspberry Pi proporciona pines digitales de propósito general (llamados pines GPIO, como ya hemos en las características detalladas de la Raspberry Pi 3) que puedes usar para leer niveles de señales lógicas digitales o escribirlos

En la figura 2.9, podemos visualizar el diagrama de bloques del chip PCF8574.

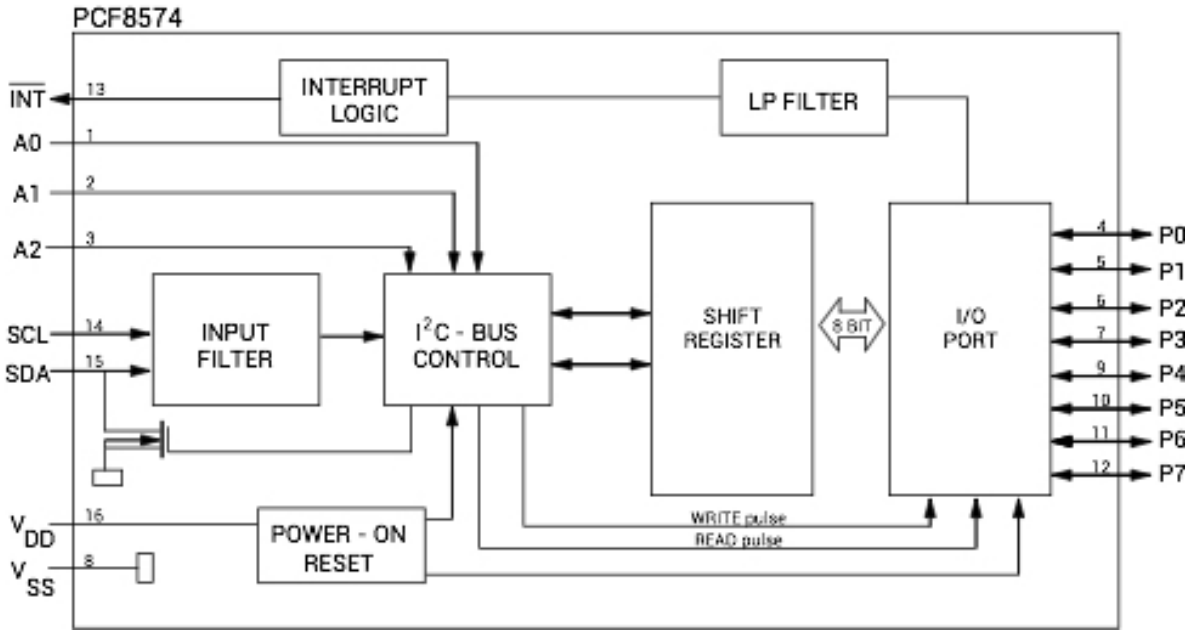


Figura 2.9: Diagrama de bloques del chip PCF8574

Podemos ver las funciones de cada pin (figura 2.8) del chip PCF8574 en la tabla 2.2.

SÍMBOLO	PIN	DESCRIPCIÓN
$A0$	1	Dirección entrada 0
$A1$	2	Dirección entrada 1
$A2$	3	Dirección entrada 2
$P0$	4	Quasi-bidireccional I/O 0
$P1$	5	Quasi-bidireccional I/O 1
$P2$	6	Quasi-bidireccional I/O 2
$P3$	7	Quasi-bidireccional I/O 3
V_{SS}	8	Suministro de tierra
$P4$	9	Quasi-bidireccional I/O 4
$P5$	10	Quasi-bidireccional I/O 5
$P6$	11	Quasi-bidireccional I/O 6
$P7$	12	Quasi-bidireccional I/O 7
$\overline{INT}$	13	Interruptor de salida (activado a baja)
SCL	14	Serial clock line
SDA	15	Serial data line
V_{DD}	16	Suministro de voltaje

Tabla 2.2: Funciones de los pines del chip PCF8574

Usamos un par de chips PCF8574 conectados en paralelo para expandir los pines de la Raspberry Pi, consiguiendo de esta manera 16 pines cuasi-bidireccionales, ocupando únicamente dos pines de la Raspberry correspondientes a las líneas de bus I2C. Estos pines son cuasi-bidireccionales porque antes de realizar una lectura sobre alguno de ellos es necesario escribir un 1 como valor. Estos 16 pines se usarán como una opción de línea de datos de entrada y salida del circuito bajo prueba. Hemos podido conectar los dos chips en paralelo gracias a que cada uno de ellos es direccionable con tres de los pines que trae, por lo que se podrían conectar hasta 8 de estos chips en paralelo proporcionándonos 64 pines útiles. Para la comunicación con los pines hemos usado la biblioteca de *C++* `i2c-dev`, que nos permiten leer y escribir valores en cada uno de los chips dada su dirección. Por ejemplo podemos ver la implementación de la función de escritura en el código 2.1

Código 2.1: Implementación de la función escritura

```
void I2c::I2C_Write(unsigned long address, uint8_t  value)
{
        struct i2c_smbus_ioctl_data argswrite;

        argswrite.read_write = I2C_SMBUS_WRITE;
        argswrite.size = I2C_SMBUS_BYTE;
        argswrite.data = NULL;
        argswrite.command = value;

        int device = open(I2C_BUS, O_RDWR);

        if ((device == -1) && !write_error_flagged) {
                write_error_flagged = 1;
                //printf("ERROR: could not open I2C bus %s
                   for writing\n", I2C_BUS);
                return;
        }

        ioctl(device, I2C_SLAVE, address);
        ioctl(device, I2C_SMBUS, &argswrite);

        close(device);
}
```

I^2C (Inter-Integrated Circuit)

I^2C es un bus de datos en serie desarrollado en 1982 por *Philips Semiconductors*. Su uso principal es la comunicación entre diferentes partes de un circuito, como la conexión entre circuitos periféricos integrados y un controlador.

El I^2C está diseñado como un bus maestro-esclavo. El maestro es el que

inicia siempre la transferencia de datos, y el esclavo lo recibe y debe realizar la siguiente acción. Existe el modo multimaestro en el que hay varios maestros que se pueden comunicar entre sí, haciéndose pasar uno de ellos por esclavo en este caso.

Para terminar de hablar sobre el dispositivo PCF8574, detallemos el **Protocolo de Transferencia de datos I^2C o $I2C$**:

El inicio de la transmisión de datos está indicado por la señal de inicio del maestro a la que sigue la dirección a la que hay que transmitir los datos. El esclavo correspondiente al que le llegará esta señal de inicio, debe confirmar activando el flag ACK (proviene de la palabra *Acknowledgment*). Una vez en este punto, dependiendo del bit de lectura o escritura (R/W Bit), se escriben datos (datos al esclavo) o se leen datos al maestro. De esta manera, el bit ACK es enviado desde el maestro al leer o desde el esclavo al escribir. Para finalizar la transmisión, el maestro realiza la lectura del último byte reconociendo un NACK (*Not Acknowledgment*). Una transmisión se termina cuando se recibe la señal de parada o también puede ser enviada una señal de reset al iniciar una nueva transmisión, para no tener que mandar en este caso dos señales (de parada y de inicio). Los bytes son transferidos siguiendo el orden *Most Significant Bit First*.

2.2. Entorno de desarrollo y Software

En esta sección, explicamos las herramientas Software que se han usado durante el desarrollo de este proyecto. Distinguiremos entre herramientas de desarrollo con las que hemos desarrollado nuestra aplicación y software utilizado propiamente dicho, además del software desarrollado que se explicará en el capítulo 3.

2.2.1. Entorno de desarrollo

Como entorno de desarrollo integrado, principalmente se ha utilizado Qt Creator. Dispone de una herramienta muy útil y sencilla para diseñar interfaces gráficas de usuario. También nos ha ayudado con las tareas de depuración, detección y corrección de errores. Entre las facilidades que ofrece también cuenta con una buena integración con herramientas de control de versiones como son git o subversion. Y además proporciona algunas bibliotecas de desarrollo útiles, por ejemplo para el tratamiento de ficheros entre otros.
Para desarrollar el diseño de los circuitos a testear, sintetizar el diseño y compilarlo generando el bitstream necesario para cargarlo en la FPGA hemos utilizado la herramienta de desarrollo `ISE` que nos proporciona Xilinx.

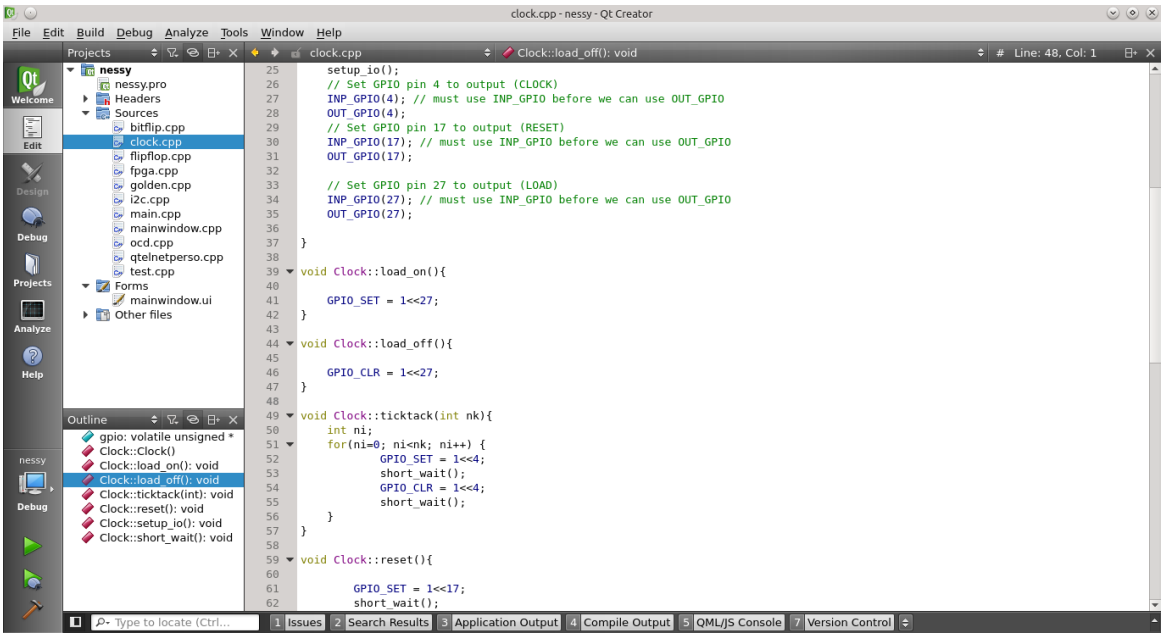


Figura 2.10: Qt Creator

Qt Creator

Qt Creator es un IDE (creado por la compañía Trolltech) que permite el desarrollo de aplicaciones con las bibliotecas Qt. Esta multiplataforma que te permite trabajar con $C++$, *JavaScript* o *QML* (Qt Meta Language o Qt Modeling Language), incluye un depurador visual y tiene integrado un diseñador de GUI y de formas. Además, Qt Creator usa el compilador $C++$.

Qt Creator incluye un editor de código propio e integra Qt Designer para diseñar y construir interfaces GUI a partir de los *widgets* de Qt. Podemos visualizar la interfaz de Qt Creator en la figura 2.10.

Xilinx ISE

Xilinx ISE (Integrated Software Environment) es un entorno de desarrollo que nos proporciona las herramientas necesarias para poder diseñar, implementar y simular circuitos programables en FPGAs. Se ha usado esta herramienta para generar los archivos .bit que sirven para configurar la FPGA con el diseño objetivo. Podemos visualizar la interfaz de Xilinx ISE en la figura 2.11.

Veamos cómo se procesa el diseño que se desea cargar en memoria en la FPGA.

i) **Función de síntesis**: el código inicial en `VHDL` se transforma a una *netlist*[7] escrita en un lenguaje interno que depende del hardware que se utilice. Posteriormente, se realiza la asignación de pines y optimizaciones de los componentes.

ii) **Proceso de implementación**: se enlazan los módulos con las restricciones

[7]Netlist es una descripción de la conectividad de un circuito electrónico

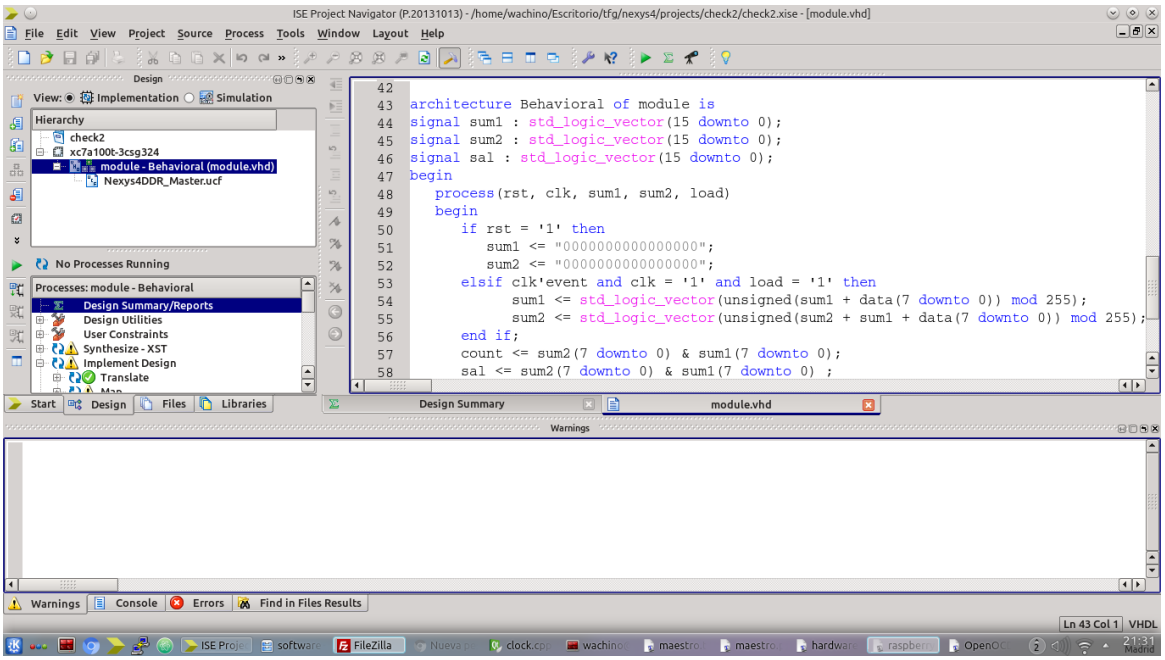


Figura 2.11: Xilinx ISE

especificadas (como puede ser los pines que queremos conectar) y se reducen a primitivas de Xilinx, de manera que el siguiente paso es trasladar el diseño a la FPGA. Este proceso está dividido en tres fases: *translate*, *map* y *place&route*.

iii) **Generación del *bitstream* (secuencia de *bytes*)**: se genera el fichero que contiene toda la información que la FPGA necesita para que se ejecute el diseño, almacenándolo en su memoria.

Podemos observar un esquema del proceso del diseño en la figura 2.12

2.2.2. Software

Para establecer la comunicación a través de la interfaz JTAG entre la Raspberry y la FPGA utilizamos OpenOCD, al que se ha dotado de algunas funcionalidades en cuanto a las operaciones realizables con el PLD, como por ejemplo la función de Readback.

Con OpenOCD montamos un servidor telnet[8] en la máquina local (Raspberry Pi) en el puerto 4444, a través del cual nos comunicamos con la herramienta que desarrollamos, Nessy 7.0, enviándole peticiones de comandos a realizar con la FPGA.

En resumen, Nessy 7.0 está diseñado con el entorno de desarrollo Qt Creator IDE y escrito en el lenguaje de programación $C++$ y corre sobre una placa Raspberry Pi 3. La comunicación entre la Raspberry Pi 3 y la FPGA, se realiza de dos modos. Uno básicamente para configuración, aunque también

[8]Telnet (Telecommunication Network) es el nombre de un protocolo de red que nos permite acceder a otra máquina para manejarla remotamente.

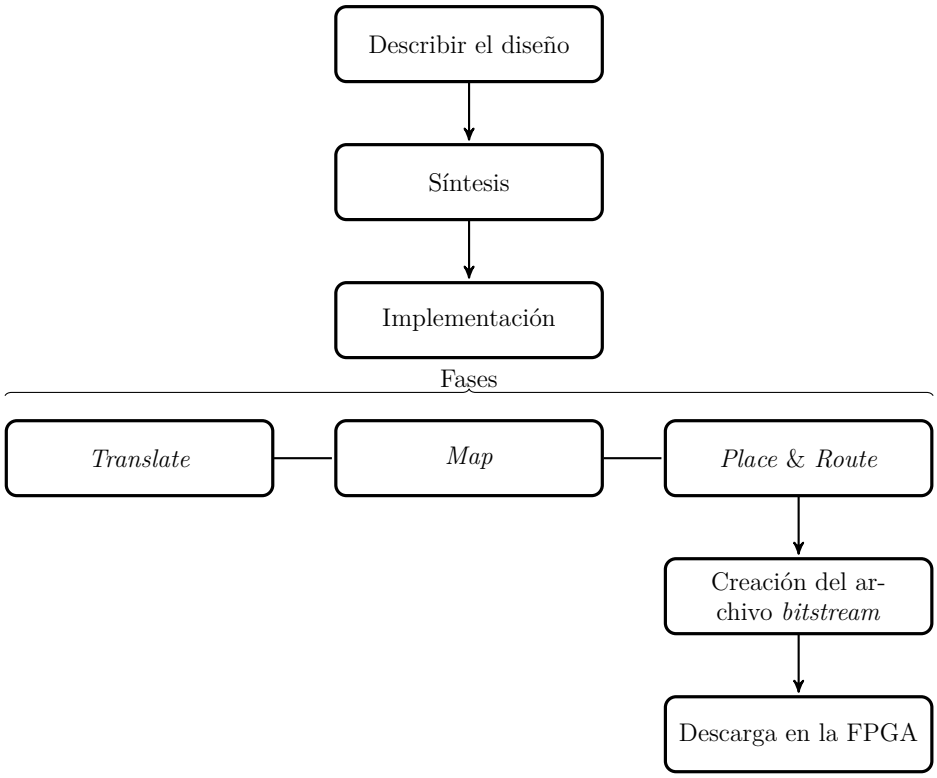


Figura 2.12: Esquema del proceso de diseño de Xilinx ISE

utilizado para la captura de la salida del circuito y el otro para el control del circuito, y envío y recepción de datos.

A través de la interfaz JTAG, mediante el cable microusb, la aplicación Nessy establece comunicación mediante un cliente telnet personalizado con un servidor OpenOCD que también se ejecuta en la Raspberry Pi, de la que se sirve para reconfigurar total y parcialmente la FPGA, capturar y leer su estado o reestablecerlo.

Y los pines GPIO disponibles en la Raspberry Pi, se utilizan para controlar el estado del circuito mediante señales, controlando la señal del reloj, o las señales de reset y load, y enviar los datos de entrada del circuito bajo prueba o capturar la salida de éste, haciendo uso del protocolo I2C, y de un par de chips PCF8574. Entramos más en detalle sobre la herramienta en el capítulo 3 Nessy.

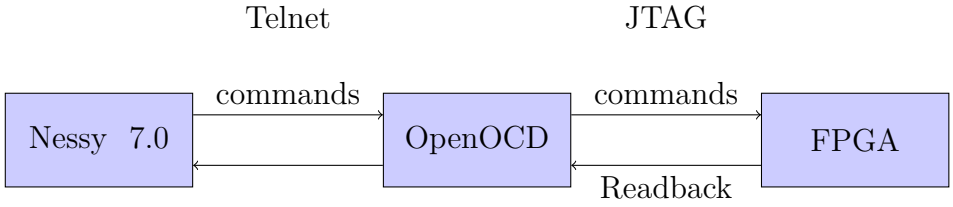


Figura 2.13: Esquema de comunicación de aplicaciones

En la figura 2.13 podemos ver un esquema de la comunicación de las aplicaciones utilizadas.

OpenOCD

OpenOCD (Open On-Chip Debugger) es un software de código abierto que se conecta con un puerto que permite la depuración de hardware, JTAG [9]. Podemos ver su interfaz en la figura 2.14.

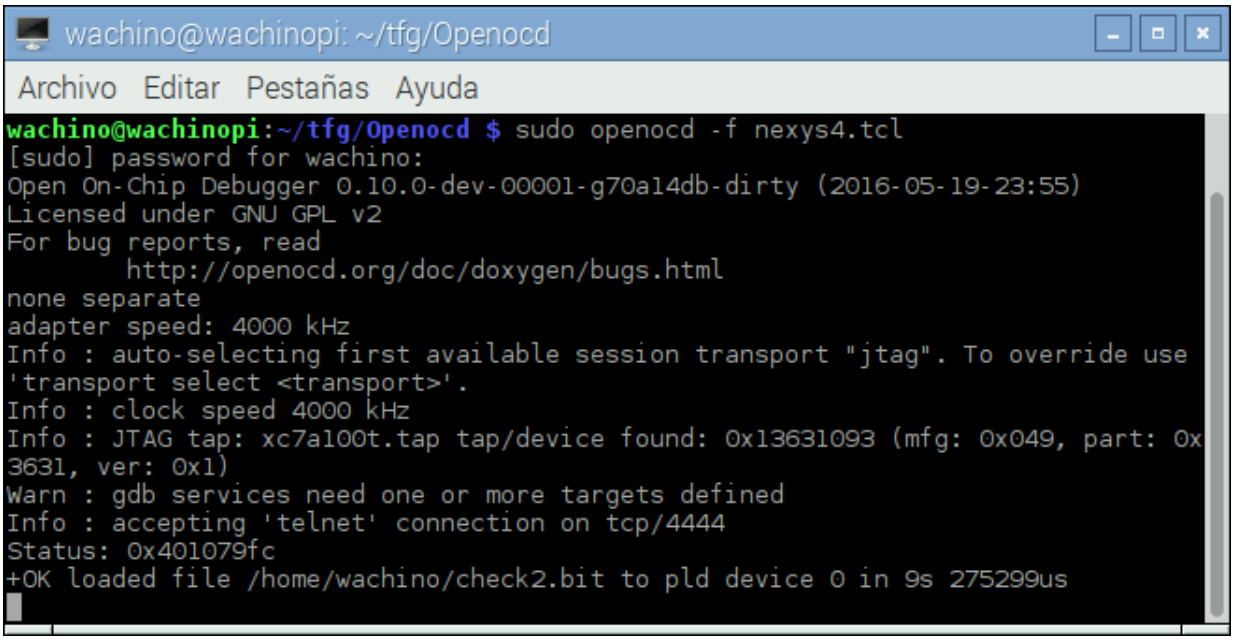


Figura 2.14: OpenOCD

Lo hace con la ayuda de un adaptador de depuración, y éste es un módulo de hardware que ayuda a proporcionar el tipo correcto de la señalización eléctrica del objetivo que está depurando.

En nuestro proyecto hemos hecho uso de las siguientes operaciones que nos ofrece para interactuar con la FPGA, alguna de las cuales hemos tenido que implementar:

- LOAD Envía un bitstream de configuración a la FPGA.
- GCAPTURE Almacena el estado de los registros de usuario de la FPGA en la memoria de configuración.
- SAVE Descarga de la FPGA y almacena en un fichero los frames indicados a partir de una dirección dada.
- PARTIAL Envía a la FPGA un archivo de reconfiguración parcial correspondiente a uno o varios frames consecutivos.
- GRESTORE Inicializa el valor de los registros de usuario con el almacenado en la memoria de configuración.

[9] JTAG (Joint Test Action Group, referring to IEEE Standard 1149.1) es una interfaz de cuatro pines diseñada para probar las conexiones entre chips

3 Nessy

3.1. Descripción

Nessy 7.0 (ver figura 3.1) es una herramienta para la inyección de fallos no intrusiva, que permite simular el efecto producido por la colisión de una partícula cósmica sobre la memoria de configuración, registros y flipflops de una FPGA tipo Artix-7. Aunque la herramienta se ha diseñado con el propósito de ser lo suficientemente genérica para poder trabajar con las FPGAs de la Serie-7, en nuestras pruebas hemos utilizado una placa Nexys 4 DDR con una FPGA Artix-7 XC7A100T.

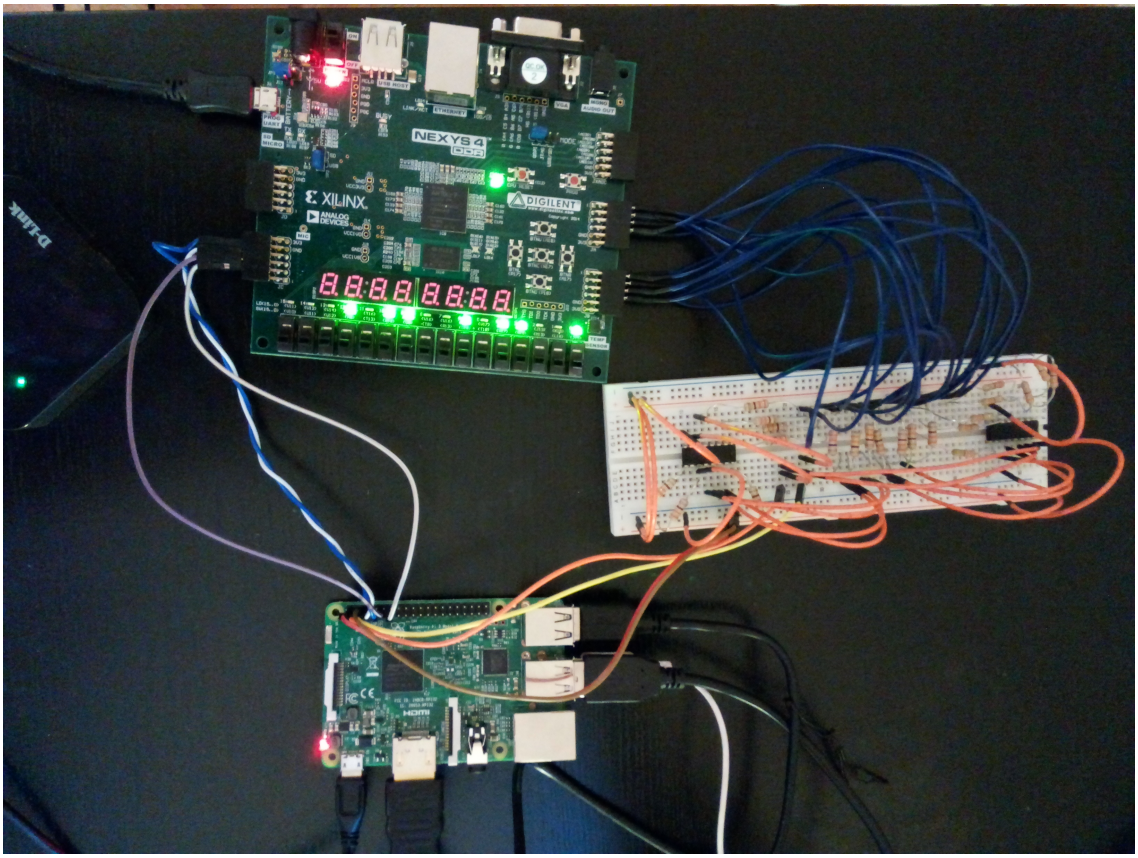


Figura 3.1: Diseño del circuito del proyecto

Para permitir el uso de otras FPGAs, la herramienta inicialmente realiza la carga del fichero `fpga.ls`. En dicho fichero, se pueden introducir otras FPGAs para su uso con la herramienta. Para ello solo será necesario añadir una línea con los datos correspondientes manteniendo el siguiente formato:

```
<name=Artix 7 XC7A100T> <framesize=101> /* Frame size in
    words */ <idcode=0x13631093> /* Artix7 */ <partnumber=
    7a100tcsg324>
```

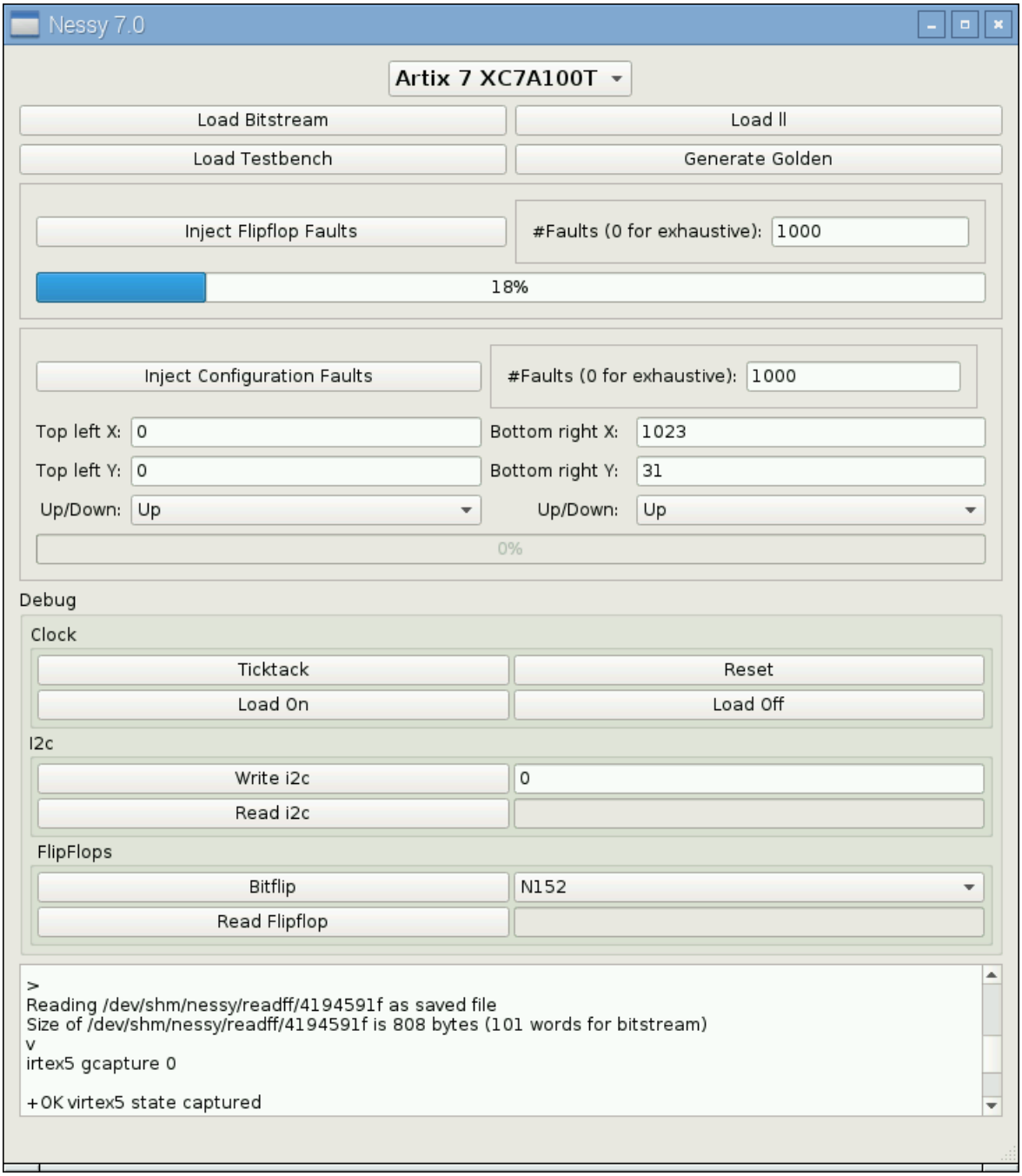


Figura 3.2: GUI de Nessy

Nessy 7.0 dispone de una interfaz gráfica (ver figura 3.2) de usuario que además de permitirnos escoger de una lista (cargada del fichero `fpga.ls`) la

FPGA con la que vamos a trabajar, nos facilita tareas como cargar un archivo de configuración o bitstream (.bit) de un diseño (ver sección 3.3), la carga de la localización de los flipflops utilizados mediante el archivo de Logic location (.ll), cargar un archivo de Testbench y generar un Golden o archivo que contiene la salida correcta de un diseño para un test dado. También permite seleccionar entre distintas opciones sobre inyección de fallos tanto en flipflops (sección 3.3.1), como en memoria de configuración (seccción 3.3.2). Además dispone de un panel de depuración (sección 3.4) que nos permite realizar tareas de más bajo nivel como cargar o leer un dato en la FPGA, enviar un reset o avanzar un ciclo de reloj.

Esta herramienta nos permite realizar la emulación de eventos SEU mediante reconfiguración parcial de los frames. Un frame es la unidad mínima de configuración de la FPGA. En el caso de la FPGA que hemos utilizado cada frame tiene un tamaño de 101 palabras de 32 bits, aunque esto es fácilmente configurable, pues la aplicación dispone de un analizador para cargar distintas configuraciones, correspondientes a distintas FPGAs como hemos mencionado.

Cuando la herramienta se ejecuta, crea en la carpeta del usuario un directorio llamado `nessysession` en el que se almacenarán los archivos generados durante su uso.

Dividimos el capítulo sobre la herramienta en tres secciones, descripción de ficheros y carga, inyección de fallos y metodología y panel de depuración de bajo nivel.

3.2. Carga de archivos y formato

En las siguientes secciones describiremos más detalladamente el uso y funcionamiento de la plataforma Nessy. En particular, en esta sección describimos las cuatro opciones de carga de ficheros que ofrece, a la vez que su descripción y funcionamiento, diferenciando los archivos diseñados a propósito para la herramienta de los archivos propios de Xilinx.

3.2.1. DUT o diseño a prueba (Bitstream)

La opción *cargar Bitstream* nos permite, mediante un cuadro de diálogo, seleccionar un fichero de configuración de la FPGA con el diseño a evaluar o DUT (Design under test) para cargarlo en ésta. El fichero se carga en la FPGA a través de la herramienta que, mediante una conexión por telnet envia el comando `load` al servidor OpenOCD, el cual carga en la FPGA la configuración del circuito, proceso que tarda unos 9 segundos. Es necesario cargar un fichero de Bitstream para utilizar el resto de opciones que nos ofrece la herramienta.

El bitstream es un archivo con un formato diseñado por Xilinx. Al generar el bitstream desde Xilinx ISE, será necesario marcar las siguientes opciones en las características del proyecto.

Si queremos inyectar en flipflops, en *Readback options* habrá que marcar la

opción `-l create Logic Allocation File`, necesaria para localizar los registros de usuario. Este archivo se verá con mayor detalle en la siguiente subsección 3.2.2. En el mismo apartado, en la propiedad de *Seguridad* habrá que escoger *Enable Readback and reconfiguration*, lo que nos permitirá hacer lecturas de la memoria de configuración y reconfiguraciones parciales, necesarias para la inyección de fallos.

Por último, en *configuration options* habrá que desmarcar `-g CRC: Enable Cyclic Redundancy Checking (CRC)` para poder alterar bits con la reconfiguración parcial sin problemas en la generación del CRC.

Esta opción, además de realizar la carga del bitstream en la FPGA, crea en el directorio de la sesión un directorio con nombre el mismo que el del fichero bitstream seleccionado, y además realiza en él una copia del módulo, con nombre `bitstream.bit`.

Veremos más sobre los comandos y registros en la sección 3.3 de inyección y reconfiguración parcial. Para ver más detalles sobre los archivos de bitstream puede consultarse el Anexo A: Bitstream

3.2.2. Archivo de ubicación lógica

Para localizar en la memoria de configuración los registros de usuario necesitaremos el archivo `.ll`. El archivo `.ll` (puede verse un ejemplo en la figura 3.3) o archivo de *logic location* es propio de Xilinx. Lo necesitaremos porque contiene la localización de las variables de estado en el bitstream y en la FPGA. En particular, este archivo indica la posición en el bitstream de los latches, flipflops, entradas y salidas de IOB, y la posición de la programación de las LUT (Look Up Table) y de los bloques de RAM.

```
33  Info STARTSEL0=1
34  Bit   332937 0x0000011f     41 Block=SLICE_X0Y100 Latch=CMUX Net=N152
35  Bit   332982 0x0000011f     86 Block=SLICE_X0Y101 Latch=BMUX Net=sum1[15]_GND_6_o_add_0_OUT<1>
36  Bit   333001 0x0000011f    105 Block=SLICE_X0Y101 Latch=CMUX Net=sum1[15]_GND_6_o_add_0_OUT<2>
37  Bit   333053 0x0000011f    157 Block=SLICE_X1Y102 Latch=BQ Net=sum2<4>
38  Bit   333058 0x0000011f    162 Block=SLICE_X1Y102 Latch=CQ Net=sum2<5>
39  Bit   333075 0x0000011f    179 Block=SLICE_X0Y102 Latch=DMUX Net=Madd_sum1[15]_GND_6_o_add_0_OUT_cy<7>
```

Figura 3.3: Parte de un módulo .ll de un diseño

Como hemos mencionado, este archivo habrá que crearlo al generar el bitstream, marcando la opción necesaria en las opciones de generación del proyecto.

Nota. La opción de creación del archivo `.ll` solo estará disponible cuando la opción de crear los archivos de *Readback* esté seleccionada.

Al seleccionar un archivo de localización lógica, se creará una copia del mismo en el subdirectorio de la sesión del módulo al que corresponde con el nombre `logiclocation.ll`.

Código 3.1: Función para analizar el fichero `.ll`

```
QList<FlipFlop> MainWindow::parseLlFile(QString fileName){
    QRegExp rx("Bit\\s+(\\d+)\\s+(0x[0-9a-f]+)\\s+(\\d+).+
       Net=([^\\s\\n]+)");
    QFile inputFile(fileName);
    QList<FlipFlop> ffs;

    if (inputFile.open(QIODevice::ReadOnly)) {
       QTextStream in(&inputFile);
       while (!in.atEnd()) {
          QString line = in.readLine();
          if( rx.indexIn(line) != -1) {
             int offset = rx.cap(1).toInt();
             int far = rx.cap(2).toUInt(Q_NULLPTR,16);
             int foff = rx.cap(3).toInt();
             QString name = rx.cap(4);
             FlipFlop ff(offset,far,foff,name);
             ffs.append(ff);
          }
       }
       inputFile.close();
    }
    return ffs;
}
```

Además, la herramienta analizará (ver código 3.1) el fichero, ayudándose de expresiones regulares, para capturar todos los campos de cada bit que se referencia, y almacenar así internamente una lista completa de los registros, con toda la información necesaria para localizarlos. Con un listado de todos los registros obtenidos, se rellena en la interfaz gráfica de usuario, en el panel de depuración, un *combobox* con los nombres de los flipflops. (Ver más en la sección 3.4 Panel de depuración)

Se puede consultar más detalles acerca de los ficheros de localización lógica en el capítulo Anexo B: Archivo .ll

3.2.3. Archivo de test

Los archivos de test (`.tb`) han sido diseñados específicamente para este propósito. El archivo contendrá los datos de las señales de entrada que el circuito bajo prueba deberá procesar para producir un resultado, como podemos ver, por ejemplo, en la figura 3.4.

El archivo constará de varias líneas y además se podrán incluir líneas de comentario que la herramienta ignorará. Deberá contener una línea con las señales de entrada en forma de pares

input=<a,b>,<c,d>...

donde para cada par <a,b> representan un dato de entrada concreto y un

```
1  // Input line has the following form:
2  // input$=<a,b>,<c,d>...                Input signals:
3  //                                      - <a,b>: "a" is a specific data entry,
4  //                                               "b" is the number of cycles that is maintained
5  //                                                   the input signal
6  // Type line has the following form:
7  // type=<reg | i2c>                     Output mode. Allowed values:
8  //                                      - i2c:  the output is read through the chips PCF8574
9  //                                              using i2c Protocol.
10 //                                      - reg:  the output is directly read from registers.
11 // Registers line has the following form:
12 // registers=sum2<*>,sum1<1>...         Output registers. If you write the wildcard value "*", you
13 //                                      will select all registers whose name matches with the result
14 //                                      of replace the "*" by some number
15
16 input=<1,1>,<255,1>,<0,1>,<23,1>,<7,1>,<3,1>,<19,1>,<86,1>,<18,1>,<11,1>,<19,1>,<93,1>,<21,1>,<21,7>...
17 type=<reg>
18 registers=sum2<*>,sum1<1>,N251
```

Figura 3.4: Ejemplo de un fichero Testbench usado en este proyecto

número de ciclos que ha de mantenerse esa señal de entrada. Esto es así por que si una señal ha de mantenerse varios ciclos, de este modo ganaremos en tiempo de ejecución, pues el hecho de modificar la señal de entrada al ir a través de los chip PCF8574 está limitada a una velocidad de 100 KHz.

Después, deberá de contener una línea con el tipo del test, representando el modo en el que obtendremos la salida generada por el circuito.

$$\text{type=<reg} \mid \text{i2c>}$$

Los modos pueden ser:

- *i2c*: indicando que la salida será leída a través de los chips PCF8574 mediante el protocolo i2c.
- *reg*, indica que la salida se leerá capturando el estado de los registros de usuario en la memoria de configuración, y descargandose gracias al *readback*.

En caso de haber determinado el tipo del testbench como *reg*, se incluirá una linea más en el archivo. Esta linea debe contener una lista de los registros, separados por coma, que almacenan la salida del circuito bajo prueba.

$$\text{registers=sum2}< * >,\text{sum1}< 0 >,\text{sum1}< 1 >\ldots$$

Para facilitar la tarea se permitirá el uso del carácter especial `*`, asterisco, para indicar todas las señales de un mismo vector, como por ejemplo, el nombre `sum<*>`, que hará referencia a todos los registros del diseño cuyo nombre sean el resultado de sustituir el asterisco por un número, es decir `sum<1>`, `sum<2>`, ... Los valores de estos serán localizados gracias al fichero `.ll` mencionado en la subsección 3.2.2 Archivo de ubicación lógica.

La herramienta analiza el fichero de test de nuevo haciendo uso de expresiones regulares. Al cargar el fichero de testbench, la herramienta guarda una copia en el directorio de la sesión del módulo con nombre `testbench.tb`, además de almacenar internamente los datos necesarios para ejecutarlo sobre el diseño cargado.

3.2.4. Golden

Una vez escogidos y cargados los archivos necesarios (logic allocation y testbench) en la herramienta, y el bitstream en la FPGA, podremos generar un golden, o salida esperada del circuito.

Con el diseño implementado en la FPGA y colocando las entradas en el circuito indicadas por el archivo de test en cada ciclo de reloj, al finalizar la ejecución, el circuito produce una señal de salida, que es el Golden.

Esta salida será leída por la herramienta desde la Raspberry Pi. La lectura puede hacerse de dos modos, por $I2C$ o directamente capturando el valor de los registros que almacenan la señal de salida en la memoria de configuración, descargando el frame adecuado y localizando el valor gracias al fichero de localización lógica (`.ll`). El modo de lectura viene indicado en el fichero de testbench (apartado 3.2.3), y en caso de ser lectura directamente de los registros de usuario, vendrán indicados en el mismo fichero los registros que almacenan la salida.

El Golden así generado, se almacena en la carpeta de la sesión de nessy, en un subdirectorio que lleva por nombre el del fichero bitstream, es decir el del diseño bajo prueba, en el fichero `golden.json`.

```
1  {
2      "type": "reg",
3      "value": [
4          {
5              "frameAddress": 287,
6              "frameOffset": 157,
7              "name": "sum2<4>",
8              "value": true
9          },
10         {
11             "frameAddress": 287,
12             "frameOffset": 162,
13             "name": "sum2<5>",
14             "value": false
15         },
16         ...
```

Figura 3.5: Fichero Golden

El formato del fichero (ver figura 3.5) es de tipo JSON (ver Anexo C: JSON) que almacena un objeto con dos campos, tipo y valor. El tipo indica el tipo de lectura que se ha hecho para obtener el golden, es decir por $i2c$ o por registros. Y el valor almacena la salida capturada, que tiene un formato que depende del tipo del golden. En caso de ser de tipo $i2c$, el valor es un valor numérico de 16 bits, y en el caso de ser de tipo registros, el valor es un array de objetos JSON, donde cada uno de estos objetos representa un registro y contiene toda la información referente a este en sus campos, que se indican a continuación.

- *name* String conteniendo el nombre del registro, si forma parte de un vector de registros su posición vendrá indicada con un número entre < >.
- *frameAddress* Entero representando la dirección del frame que contiene en la memoria de configuración el bit correspondiente al valor del flipflop.
- *frameOffset* Entero que indica la posición absoluta del bit dentro del frame.
- *value* Valor binario del flipflop. Se indica con un booleano.

Se ha escogido este formato de fichero porque es un formato de texto ligero, diseñado para intercambiar datos que goza de una amplia popularidad.

Este golden es usado como referencia de circuito correcto en la fase de inyección de fallos, en cada ejecución con inyección se genera una salida de formato similar que nos permite compararla con el golden para detectar errores producidos en el diseño por una inyección. Al generarse el Golden, se captura además los frames que contienen registros de usuario, de forma que estos sirvan para restaurar el circuito al estado inicial durante el proceso de la inyección.

3.3. Inyección de fallos

Veamos a continuación la metodología utilizada para el proceso de inyección de fallos, que dividiremos en dos tipos. Inyección sobre flipflops, los registros de usuario del circuito bajo prueba, indicados por el archivo de logic location (como vimos en la subsección 3.2.2) y la inyección en memoria de configuración.

Hay técnicas comunes que utilizaremos en ambos procesos. Los dispositivos de la serie 7 de Xilinx permiten a los usuarios leer la memoria de configuración a través de la interfaz JTAG, entre otras. Ofrece dos tipos de *readback*, Readback Verify y Readback Capture.

Con Readback Verify, el usuario lee todas las celdas de la memoria de configuración, incluyendo los valores de todos los elementos de memoria del usuario (LUT RAM, SRL, y bloques RAM).

Readback Capture es un superconjunto de Readback Verify, además de leer todas las celdas de la memoria de configuración, también permite leer el estado actual de todos los registros internos de los CLB y los IOB.

Para leer la memoria de configuración el usuario debe enviar una secuencia de comandos al dispositivo para iniciar el procedimiento de readback. Tras la iniciación, el dispositivo vuelca el contenido de su memoria de configuración a la interfaz JTAG. Para poder efectuar un *readback* son necesarias dos opciones en el bitstream que se seleccionan cuando se generan con ISE. La seguridad no debe prohibir el *readback* (`security:none`) y no se debe usar encriptación del bitstream.

El proceso de leer la memoria de configuración lo haremos escribiendo y leyendo en registros internos de la FPGA, se pueden ver algunos de ellos en la tabla 3.3, los cuales mencionamos a continuación.

Registro CRC	Se usan las escrituras en este registro para realizar comprobaciones de CRC sobre los datos del bitstream. Si el valor escrito coincide con el CRC actualmente calculado, se elimina la señal CRC_ERROR y se permite la inicialización. Como vamos a realizar reconfiguraciones parciales, y por tanto al modificar parte del bitstream el CRC calculado no coincidirá, tenemos desactivada esta comprobación
Registro FAR	Registro de dirección de frame.
Registro FDRI	Las escrituras en este registro configuran los datos del frame cuya dirección está especificada en el registro FAR
Registro FDRO	Este registro de solo lectura proporciona datos de la lectura de los frames de configuración empezando por la dirección indicada en el registro FAR
Registro CMD	El registro de comando o CMD se utiliza para controlar la lógica de configuración y realizar funciones de configuración. El comando presente en el registro CMD se ejecuta cada vez que se carga un nuevo valor en el registro FAR
Registro IDCODE	Cualquier escritura sobre el registro FDRI debe ir precedida por una escritura a este registro. El IDCODE proporcionado debe coincidir con el del dispositivo. Una lectura sobre este registro proporciona el IDCODE del dispositivo

En particular usaremos el registro CMD para enviar comandos, de los cuales podemos ver algunos en la tabla 3.3, y algunos registros más como el FDRO y el FAR (ver el apartado 3.3.2) a través de la interfaz JTAG. Los datos de configuración recibidos a través del registro FDRO pasan a través del frame buffer y por tanto el primer frame de datos del *readback* son datos a descartar.

La secuencia de lectura de memoria de configuración es idéntica para ambos modos de Readback, sin embargo el modo de Captura requiere un paso adicional para muestrear los valores de los registros internos.

Para realizar la captura del estado de los registros de usuario se puede escribir el comando GCACPTURE al registro CMD. Al hacerlo, el dispositivo almacena los valores actuales de los registros de los CLB y los IOB en las celdas de la memoria de configuración, y estos valores se podrán leer después junto al resto de la memoria de configuración mediante el procedimiento de Readback. Los valores de los registros se almacenan en las mismas celdas de memoria que programan la configuración inicial de los registros, por tanto, enviar el comando `GRESTORE` a la FPGA después de una secuencia de Captura, puede causar que los registros queden en un estado involuntario, ya que el efecto del comando GRESTORE es simétrico al de GCAPTURE, inicializa los registros internos de la FPGA con el valor inicial almacenado en la memoria de configuración.

Nombre	**Lectura/Escritura**	**Dirección**	**Descripción**
`CRC`	Lectura/Escritura	00000	Registro *CRC*
`FAR`	Lectura/Escritura	00001	Frame Address Register
`FDRI`	Write	00010	Frame Data Register, Input Register (datos de configuración de escritura)
`FDRO`	Read	00011	Frame Data Register, Output Register (datos de configuración de lectura)
`CMD`	Lectura/Escritura	00100	Command Register
`IDCODE`	Lectura/Escritura	01100	Device ID Register

Tabla 3.3: Registros

También haremos uso de la técnica de reconfiguración parcial. La reconfiguración parcial da un paso más en la flexibilidad que nos proporcionan las FPGA, permitiendo la modificación de un diseño operativo sobre una FPGA cargando un archivo de configuración parcial, normalmente un archivo BIT parcial. Después de configurar la FPGA con un archivo BIT completo, los archivos parciales se pueden descargar para modificar regiones reconfigurables de la FPGA sin comprometer la integridad de las aplicaciones que corren en las partes del dispositivo que no van a ser reconfiguradas.

Nuestra herramienta realiza todas estas operaciones enviando mediante telnet comandos al servidor OpenOCD que es el que las implementa y realiza a través de la interfaz JTAG con la FPGA.

Antes de entrar en detalle sobre la metodología usada para la inyección de fallos describiremos brevemente la generación de la señal de reloj, y el registro de dirección de frame, cuyos campos son esenciales para la inyección en memoria de configuración.

Generación de reloj

Para optimizar la simulación de un error, y dado que la raspberry genera el reloj es importante conseguir proporcionar una buena frecuencia para la señal de reloj, que se emite a través del pin BCM 4 de la Raspberry. Para ello realizamos varias pruebas ayudados de un polímetro [1] y un osciloscopio[2]. Probamos diferentes formas de acceder al pin, con el fin de encontrar una lo suficientemente rápida, limpia y estable. Uno de los métodos investigados fue acceder a los pines GPIO a través del sistema de ficheros de la siguiente forma:

Primero es necesario crear el acceso por fichero a un pin en concreto, para ello hacemos una escritura con el número de pin, por ejemplo el `4`, en el fichero `/sys/class/gpio/export`, a continuación especificamos la dirección del pin (entrada o salida), en nuestro caso `salida` escribiendo el texto `out` en el fichero `/sys/class/gpio/gpio4/direction`. Una vez activado ya podemos emitir nuestra señal escribiendo un `1` o un `0` en el fichero `/sys/class/gpio/gpio4/value`. Con este método no conseguimos superar una frecuencia de 100 Hz.

Para mejorar esta frecuencia, intentamos acceder al pin a través de la biblioteca `RPi.GPIO` del lenguaje de programación Python. Su uso es sencillo, podemos ver un ejemplo en el código 3.2.

Código 3.2: Acceso al pin BCM 4 a través de la biblioteca de Python RPi.GPIO

```
import RPi.GPIO as GPIO
GPIO.setup(4, GPIO.OUT) ## GPIO 4 como salida
GPIO.output(4, True) ## Enciendo el 4
GPIO.output(4, False) ## Apago el 4
...
```

Con el que conseguimos una señal de aproximadamente unos 200kHz, aunque tampoco era suficiente.

Finalmente conseguimos una señal de reloj estable de unos 10 MHz.

Código 3.3: Uso de las direcciones en memoria de los pines

```
#define BCM2708_PERI_BASE        0x3F000000
#define GPIO_BASE                (BCM2708_PERI_BASE + 0
    x200000) /* GPIO controller */
```

Accediendo al pin a través del dispositivo especial en Linux `/dev/mem` utilizando las direcciones, como podemos ver en el código 3.3, en memoria de los pines y escribiendo parte del código en lenguaje ensamblador, sin embargo tuvimos que reducir a alrededor de 1MHz introduciendo espera activa, porque

[1] Un polímetro es un instrumento eléctrico portátil para medir directamente magnitudes eléctricas activas, como corrientes y potenciales, o pasivas, como resistencias, capacidades y otras.

[2] Un osciloscopio es un instrumento de visualización electrónico para la representación gráfica de señales eléctricas que pueden variar en el tiempo.

alguno de los pulsos se perdía. Creemos que esta pérdida de pulsos era debido a las características del cable utilizado y al hecho de mapear de forma externa el reloj de la FPGA.

Llegados a este punto, buscamos una manera de evitar esta pérdida de pulsos al usar una frecuencia mayor que 1 MHz. Así, comprobamos que trenzando el cable con el de tierra, con una frecuencia de 3MHz, seguía sin haber pérdida de pulsos.

Finalmente, la mejor frecuencia que conseguimos en la que todo funcionaba correctamente, fue de 3 MHz. El inconveniente de este enfoque es que para acceder al dispositivo `/dev/mem` y en definitiva a los pines GPIO es necesario ejecutar la herramienta con permisos de superusuario.

Frame Address Register

Expliquemos el registro FAR que vimos en la tabla 3.3.

Los dispositivos de la serie 7 están divididos en dos mitades, la superior y la inferior. Todos los frames tienen un tamaño fijo idéntico de 101 palabras.

Cada mitad está dividida en celdas, que se identifican en el plano por sus coordenadas X e Y. En ambas mitades la coordenada X comienza en el cero y va creciendo hacia la derecha. La coordenada Y, sin embargo, en la mitad superior comienza en el cero y va creciendo hacia arriba, y en la mitad inferior comienza en el cero también y va aumentando su valor conforme las celdas bajan.

Tipo de dirección	**Índices de los bits**	**Descripción**
Tipo de bloque	[25:23]	Los tipos de bloques válidos son CLB, I/O, CLK (0 0 0), contenido de bloques de RAM (0 0 1), y CFG_CLB (0 1 0). Los bitstream normales no incluye el tipo 0 1 1.
Bit superior e inferior	22	Selecciona entre la mitad superior de las filas (0) y la mitad inferior de las filas (1).
Direcciones de fila	[21 : 17]	Selecciona la fila actual. Las direcciones de las filas incrementan desde el centro hacia arriba y entonces resetean e incrementan desde el centro hacia la parte inferior.
Dirección de columna	[16 : 7]	Selecciona una columna como *major*, como una columna de $CLBs$. Las direcciones de la columnas empiezan en 0 desde la izquierda y se incrementan hacia la derecha.
Dirección *minor*	[6 : 0]	Selecciona un *frame* en una columna *major*.

Tabla 3.5: Descripción de los campos del registro de direccion de *frame*

El `FAR` se divide en cinco campos (ver tabla 3.5), tipo de bloque, el bit de superior/inferior, fila, columna, y dirección menor. La dirección puede ser escrita directamente o autoincrementada al final de cada frame. El bitstream típico empieza en la dirección 0 y se autoincrementa hasta el final.

En las dos subsecciones que siguen vemos los dos métodos de inyección de fallos implementados, inyección en flipflops e inyección en memoria de configuración.

3.3.1. Inyección en flipflops

Con esta función realizaremos una serie de bitflips sobre los flipflops del usuario en algún punto de la ejecución de un circuito, de modo que cuando el circuito bajo test finaliza su ejecución y produce una salida, ésta será comprobada con el golden. El resultado de esta comprobación se añadirá a un archivo de resultados, que posteriormente se utilizará para estudiar las vulnerabilidades del diseño.

En cuanto a la metodología, disponemos de dos opciones a la hora de inyectar fallos. **Inyección exhaustiva**, de forma que recorreremos todos y cada uno de los flipflops disponibles en el diseño, a la vez que se efectuará el bitflip explorando también todos los posibles ciclos de reloj en los que se pueda producir. El otro modo será el **modo aleatorio**, elegiremos un número de fallos a inyectar, y se testeará el circuito dicho número de veces, en cada una de las cuales se provocará un bitflip escogido de forma aleatoria en un ciclo de ejecución también aleatorio. Del mismo modo que con la opción anterior, éste producirá un archivo de resultados con el mismo formato.

Entrando un poco más en detalle, en el panel de la interfaz gráfica de usuario dedicado a la inyección de fallos en flipflops, podemos introducir el número de fallos que queremos inyectar en el diseño. Si deseamos realizar una exploración exhaustiva de los registros del diseño, bastará con introducir un cero en dicho campo. El estado del proceso de inyección se verá reflejado en una barra de progreso, que alcanzará el 100 % cuando haya finalizado.

La metodología es la siguiente (ver figura 3.6). Primero se elige un ciclo de ejecución del diseño. En el caso de una inyección exhaustiva se recorrerán todos secuencialmente, y en caso contrario se escogerá uno aleatorio del total de ciclos del circuito para cada inyección. Entonces, desde un estado inicial, se avanza el estado del circuito bajo prueba mediante la señal de reloj, y colocando los datos de entrada correspondientes en cada ciclo, el número de ciclos escogido hasta alcanzar aquel en el que se realizará la inyección. Se escoge un flipflop de la lista de flipflops cargada con el archivo `.ll` (ver sección 3.2.2). De nuevo, en el caso de hacer una inyección exhaustiva, estos se irán recorriendo todos uno a uno secuencialmente, y si no, se escogerá para cada inyección, de entre toda la lista disponible, uno aleatoriamente.

Estando el circuito en el estado deseado, con una llamada a OpenOCD se realiza un `GCAPTURE` sobre la FPGA, que como explicamos anteriormente, captura el estado de los registros de usuario en la memoria de configuración. A continuación una vez capturado el valor del flipflop que queremos alterar,

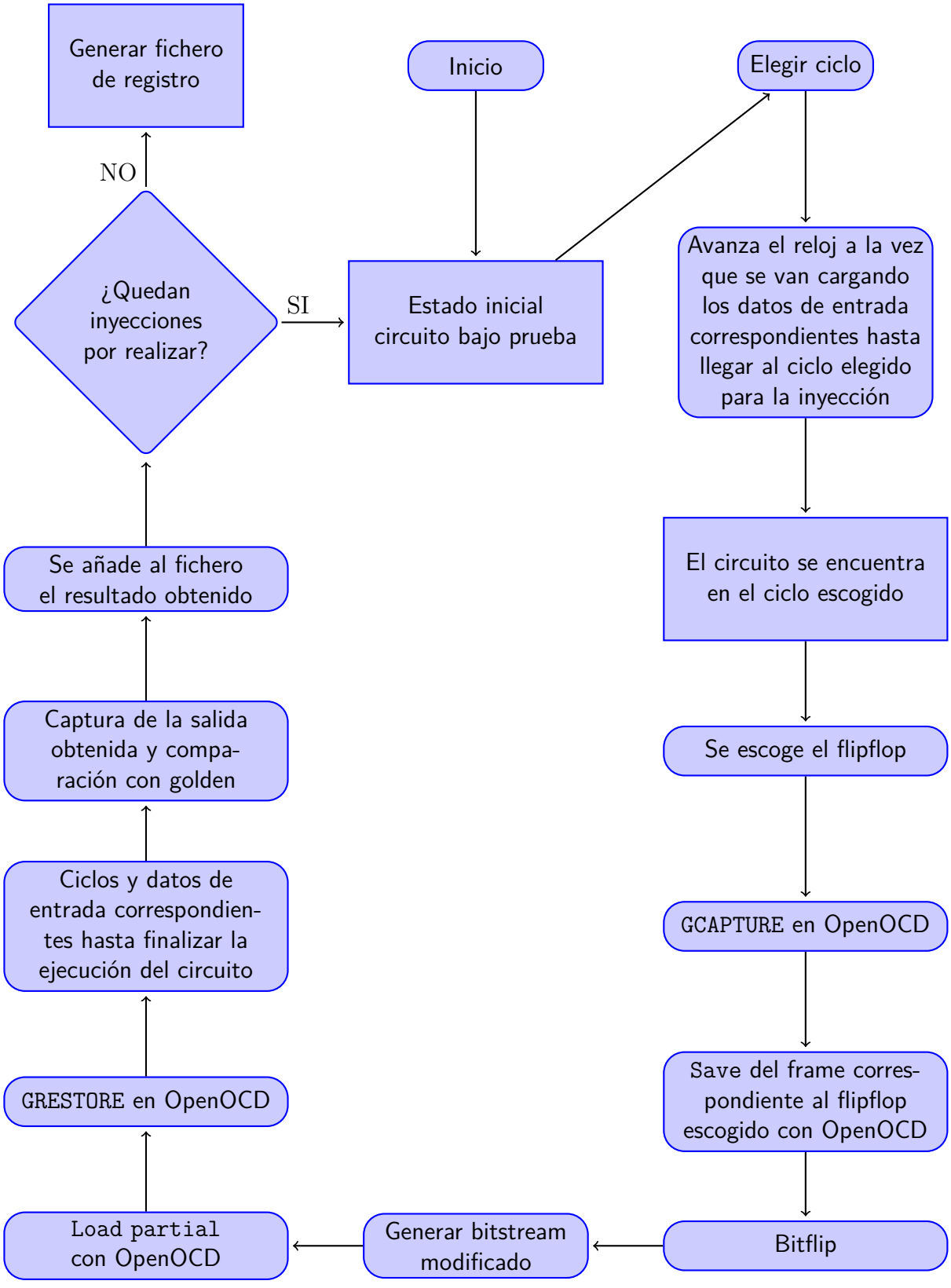


Figura 3.6: Diagrama de flujo de la inyección en flipflops

lo localizamos, y descargamos el frame que lo contiene, enviando un comando `Save` a OpenOcd, con la dirección del frame. La descarga se realiza en

unos 5 milisegundos. Utilizamos para almacenar el archivo el sistema de ficheros `/dev/shm`, que es memoria RAM de la placa Raspberry, de modo que el acceso será más rápido y no cargaremos con sobreescrituras innecesarias la tarjeta SD de la Raspberry Pi.

Dentro de este fichero y gracias al campo `frame_offset` del flipflop, localizamos la palabra que contiene el valor, y en particular el bit exacto.

Ahora, realizamos un *bitflip* sobre él con una `Xor`. El frame así modificado, lo reescribimos creando un bitstream parcial, añadiéndole una cabecera, un *padding* y una cola, necesarios para poder cargarlo de nuevo como archivo de reconfiguración parcial.

La cabecera, entre otros, contiene la palabra de sincronización, el comando de escritura de `id code` y el *idcode* de la FPGA, el comando de escritura de la dirección del frame y la dirección del frame y el tamaño del frame incluyendo un frame de *padding*.

Se carga el fichero bitstream parcial generado en la FPGA mediante el comando `partial` de OpenOCD que también tarda alrededor de unos 5 milisegundos, y a continuación se llama a otro comando de OpenOCD, `GRESTORE`, lo que finalmente modifica el valor del flipflop con el nuevo inyectado en la memoria de configuración, es decir realiza el bitflip buscado.


```
[
    {
        "cycle": 0,
        "flipflop": {
            "frameAddress": 287,
            "frameOffset": 41,
            "name": "N152",
            "value": false
        },
        "success": true
    },
    {
        "cycle": 0,
        "flipflop": {
            "frameAddress": 287,
            "frameOffset": 86,
            "name": "sum1[15]_GND_6_o_add_0_OUT<1>",
            "value": false
        },
        "success": true
    },
    ...
```


Figura 3.7: Fichero resultados de inyección en flipflops

Para finalizar, se envían los ciclos de reloj restantes, también colocando entremedias las señales de datos correspondientes, para que el circuito finalice su ejecución y produzca un valor de salida. Esta salida, es capturada y comparada con el Golden, por el método indicado en el Testbench antes descrito.

El resultado de esta comparación se añade al fichero de resultado (ver figura 3.7).

El circuito entonces se restaura a un estado inicial y se continua con la siguiente inyección iterando hasta finalizar.

El fichero generado por este proceso, se almacena como los demás, dentro del directorio de la sesión, en su módulo correspondiente con el nombre de `flipflopInjection.json`. El formato de este fichero será de nuevo un *array* de objetos (ver figura 3.7). La longitud de dicho *array* se corresponde con el número de inyecciones, es decir se genera un objeto por inyección. Cada inyección contiene los siguientes campos:

- `ciclo` Número del ciclo en el que se realiza la inyección.
- `fliplop` El flipflop es a su vez un objeto JSON que representa el flip-flop sobre el que se inyecta y que contiene los siguientes campos.
  - `frameAddress` Un entero con la dirección del frame en el que se localiza el flipflop.
  - `frameOffset` Un entero con la posición del bit del flipflop dentro del frame.
  - `name` Una cadena con el nombre del flipflop.
  - `value` Un booleano con el valor del flipflop en el momento de la inyección.
- `success` Un valor *booleano* que indica si el circuito ha dado como resultado la salida esperada a pesar de la inyección.

En caso de que como resultado de la inyección, el circuito no haya producido la salida esperada, el objeto JSON correspondiente a la inyección contendrá además dos campos más. El campo `golden`, y el campo `output`. Ambos serán objetos JSON, y su estructura dependerá del tipo de testbench.

En cualquier caso, el `golden`, contendrá una estructura similar al objeto generado en el archivo golden, representando la salida esperada. Y el objeto `output` contendrá los valores de salida obtenidos tras la inyección. Por ejemplo, un *array* de objetos con nombre de flipflop y valor asociado.

3.3.2. Inyección en memoria de configuración

La opción de inyección de fallos en memoria nos permite, como la opción de inyección en flipflops, elegir entre dos modos, inyección exhaustiva y provocar un número determinado de bitflips sobre la memoria de configuración, seleccionando el bit que se alterará y el momento de forma aleatoria. Para elegir el modo exhaustivo bastará, como en el caso de inyección sobre flipflops, con escoger cero como número de fallos a inyectar.

También se muestra una barra de progreso. Es necesario advertir que este proceso puede ser tremendamente duradero, incluso para regiones formadas por una única celda. A modo de ejemplo, para realizar una inyección exhaustiva

en una única celda con un testbench de 53 ciclos, son necesarias algo más de 20 millones de inyecciones.

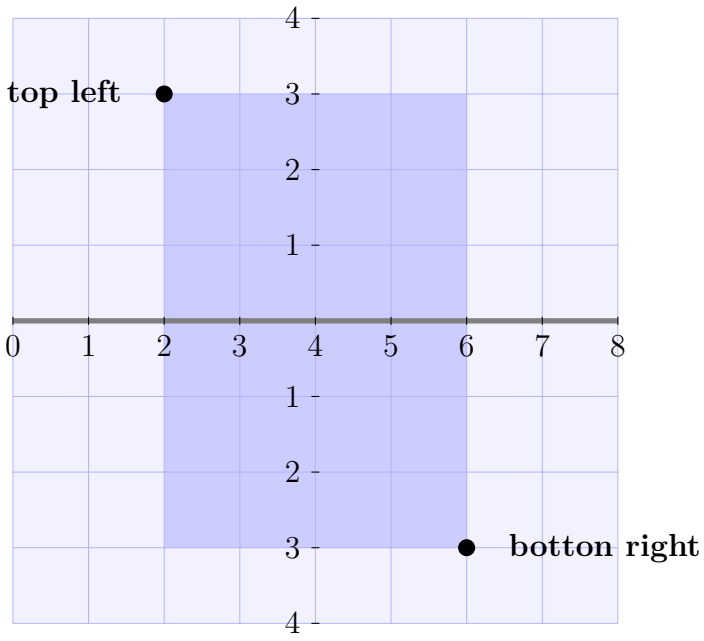


Figura 3.8: Selección de la región de la memoria de configuración sobre la que inyectar.

Además, en esta opción podemos acotar la región de actuación dando cuatro valores que la limitarán: `Xmin`, `Xmax`, `Ymin` e `Ymax` (ver figura 3.8). Dado que se puede limitar la zona de implementación del circuito bajo prueba colocando algunas restricciones en el fichero `.ucf` obligando a la herramienta a colocar el diseño en dicha área. Por ejemplo con el comando:

```
INST"instance_name"LOC=SLICE_X#Y#:SLICE_X#Y#;
```

(donde `X` e `Y` son las coordenadas de los slices mínimo y máximo del rectángulo donde se desea ubicar el circuito). O bien, definiendo un `area group` que puede implicar *clbs*, *dsps*, *RAM*, etc.. del siguiente modo:

```
INST "X " AREA_GROUP=groupname;
```

donde `X` es el nombre de nuestro circuito y `groupname` un nombre para el área donde se va a ubicar.

```
AREA_GROUP "groupname" RANGE= SLICE_X#Y#:SLICE_X#Y#;
```

Esto lo podemos hacer gracias a los campos que componen la dirección de un frame, que detallamos en la sección 3.3 en el apartado Frame Address Register.

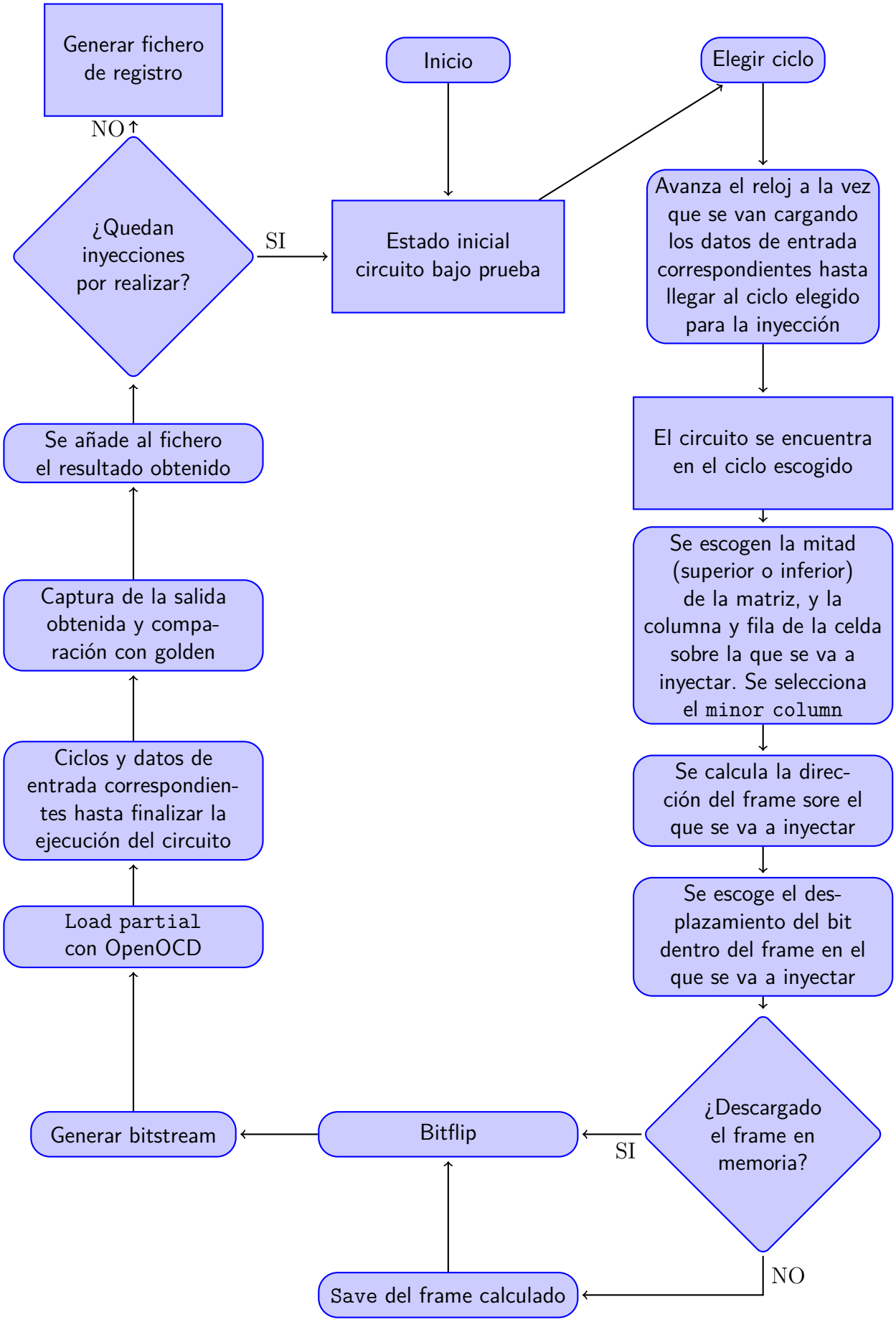


Figura 3.9: Diagrama de flujo de la inyección en memoria

La metodología para la inyección en memoria (que podemos ver en la figura 3.9) será parecida, aunque con diferencias, a la de inyección en flip-flops. En la opción exhaustiva se recorrerán todas las celdas de la región seleccionada y en la opción aleatoria se elegirán celdas del área seleccionada aleatoriamente. Con los ciclos igual, o se explora todo el espacio de ciclos del diseño o se escogen aleatoriamente los momentos en los que se inyecta. Una vez decidido ciclo de inyección, se lleva el circuito al estado correspondiente actuando sobre la señal de reloj y las señales de datos de entrada adecuadamente.

Se escoge primeramente la mitad (superior o inferior) de la matriz seguido de la columna y la fila de la celda sobre la que se va a realizar la inyección siempre dentro del área seleccionada, igual que antes, de forma secuencial en la inyección exhaustiva o aleatoria. Además, se selecciona el `minor column`.

Con estos datos, y utilizando la información de los campos que conforman el `frame address`, se calcula la dirección del frame sobre el que se va a inyectar de la siguiente manera.

```
uint32_t injectFar = (injectTopBottom << 22) | (injectRow <<
    17) | (injectCol << 7) | (injectMinor);
```

Finalmente se escoge el desplazamiento del bit dentro del frame sobre el que se va a inyectar.

Con la dirección del frame y el desplazamiento, se va a realizar el bit-flip. Esta parte difiere también de la inyección sobre flipflops, tenemos ahora algunas ventajas, pues ahora no es necesario el comando `GCAPTURE`, pues no necesitamos capturar el estado de los flipflops. Además, y por esta razón, únicamente necesitaremos descargar cada frame que se vaya a usar una vez, pues al no realizar `GCAPTURE`, éste será valido para cualquier inyección sobre ese frame.

Entonces, si no lo hemos descargado, procedemos a descargarlo con el comando `SAVE` de OpenOCD y realizamos el bitflip sobre el bit indicado por el offset con una `Xor`. Conformamos un bitstream de configuración parcial con su cabecera, cola y padding. Lo cargamos en la FPGA con el comando de reconfiguración parcial de OpenOCD `partial`. En este método tampoco necesitamos realizar un `GRESTORE`.

Finalizamos como en el método de inyección en flipflops, ejecutando el circuito controlando la señal de reloj y los datos de entrada hasta finalmente obtener una salida que se compara con el golden, y se añade el resultado al fichero.

A continuación se restaura el frame modificado, configurando de nuevo la FPGA con el diseño original y se continúa con la inyección hasta finalizar.

El fichero generado en este caso se llama `memoryInjection.json` (ver figura 3.10) y contiene un *array* de objetos con los siguientes campos:

- `ciclo` Entero que referencia al ciclo en el que se inyecta.
- `frameAddress` Un entero con la dirección del frame en el que se localiza el bit inyectado.

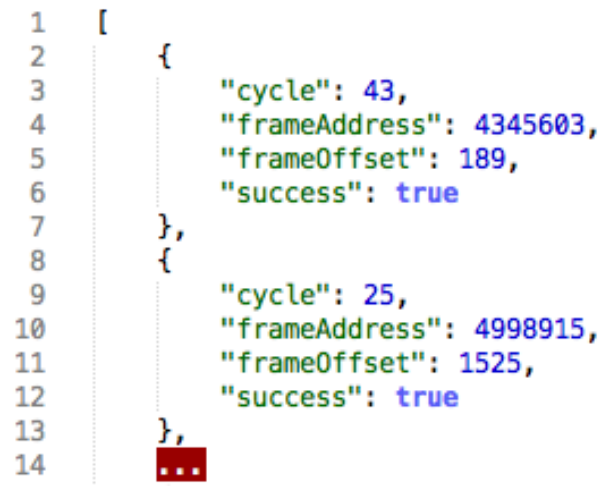


Figura 3.10: Fichero resultados de inyección en memoria

- `frameOffset` Un entero con la posición del bit dentro del frame.
- `success` Un valor *booleano* que indica si el circuito ha dado como resultado la salida esperada a pesar de la inyección.

En caso de que como resultado de la inyección, el circuito haya producido una salida errónea, el objeto JSON correspondiente a la inyección contendrá además dos campos más, el campo `golden`, y el campo `output`, similares a los descritos en la sección anterior 3.3.1

3.4. Panel de depuración

Se ha incluido en la plataforma una sección de depuración, con herramientas y opciones que nos ayudarán a comprobar la herramienta, e incluso a testear manualmente algún circuito. En el panel podemos encontrar 4 subsecciones:

Reloj

En esta sección podemos generar y enviar señales por los pines configurados. Podemos enviar la señal de un ciclo de reloj completo mediante el botón `tickctack`. También es posible enviar un pulso de `reset`. Además tenemos la opción de activar y desactivar la señal de `load`.

I2C

Aquí interactuamos con la interfaz I2C. Podemos escribir en los datos de entrada un valor de 16 bits. Además, podemos capturar un valor de 16 bits de los datos de salida del circuito.

Flipflops

En este apartado podemos interactuar con los valores actuales de los flipflops del circuito. Disponemos de un *combobox* con el listado completo de los flipflops del diseño, extraído del archivo del localización lógica `.ll`. Seleccionando uno de los flipflop podemos leer su valor actual, y también realizar un bitflip sobre él.

Consola registro

Es un cuadro de texto en el que irán apareciendo los mensajes de depuración producidos por la herramienta. Para no saturar la herramienta, estos mensajes se pueden desactivar poniendo a `False` el macro definido en la clase `mainwindow.cpp` de la herramienta.

3.5. Scripts de visualización

En paralelo a la herramienta Nessy 7.0, se han desarrollado dos *scripts* de visualización en lenguaje `Python` versión 2.7 para visualizar gráficamente los resultados obtenidos.

Estos *scripts*, reciben como argumento el nombre del archivo del que se desean procesar los datos para generar las gráficas correspondientes. En un terminal `Unix` se puede ejecutar el siguiente comando para obtener la salida del script:

`python` *nombredelscript.py nombreficheroresultado.json*

3.5.1. Script de inyección en memoria

El procesamiento de los datos en este *script* se ha llevado a cabo generando puntos de la gráfica del siguiente modo:

La coordenada X o columna se corresponde con el valor del *minor* de la columna, que se obtiene de los últimos 7 bits del `frameAddress` mediante una operación de `AND` lógica.

Como coordenada Y o fila, usamos el valor del `frameOffset` dividido entre una constante que nos ha servido para hacer escala en este eje Y.

Finalmente se pinta cada punto de un color, si la inyección causó un error se mostrará un punto rojo, y sin embargo si la ejecución resultó exitosa aparecerá como un punto verde, quedando en blanco los espacios donde no se realizaron inyecciones.

En el fragmento de código 3.4 extraído del *script* de inyección en memoria podemos ver la generación de los puntos de la gráfica y su clasificación, dependiendo de si el valor del campo `success` es `True` o `False`.

Código 3.4: Fragmento de código del script de visualización en memoria

```
# Constants
K = 127
SCALE_Y = 10

for elem in json_data:
    pointY = elem['frameOffset'] / SCALE_Y
    pointX = elem['frameAddress'] & K

    if elem['success']:
        axisX_success[count_success] = pointX
        axisY_success[count_success] = pointY
        count_success += 1
    else:
        axisX_no_success[count_no_success] = pointX
        axisY_no_success[count_no_success] = pointY
        count_no_success += 1
```

3.5.2. Script de inyección en flipflops

En este *script* se generan dos gráficas. En una se agrupan las inyecciones por flipflop, indicando el número de aciertos mediante barras. La otra agrupa por cada ciclo las inyecciones realizadas en todos los flipflops y muestra el número de fallos en cada ciclo. Durante el procesamiento de los datos, se han usado *Diccionarios de* `Python` como estructuras auxiliares para poder ir agrupando tanto por nombre de flipflop como por ciclo. En el fragmento de código 3.5 extraído del *script* de inyección en flipflops podemos ver el proceso de agrupación junto con sus contadores, que serán incrementados dependiendo de si el valor del campo `success` es `True` o `False`, en el caso correspondiente.

Código 3.5: Fragmento de código del script de visualización en flipflops

```
for elem in json_data:
    flipflop_name = elem['flipflop']['name']
    cycle = elem['cycle']

    if elem['success']:
        if flipflop_name in names_dict:
            names_dict[flipflop_name] += 1
        else:
            names_dict[flipflop_name] = 1

        if cycle not in cycles_dict:
            cycles_dict[cycle] = 0
    else: # no success
```

```
        if flipflop_name not in names_dict:
            names_dict[flipflop_name] = 0

        if cycle in cycles_dict:
            cycles_dict[cycle] += 1
        else:
            cycles_dict[cycle] = 1
        count_no_success += 1
```

4 Resultados y conclusiones

Mostraremos a continuación los resultados de utilizar la plataforma desarrollada con un par de diseños creados para tal efecto, el algoritmo de Checksum de Fletcher y un contador binario de 8 bits. Finalmente intentaremos recopilar las conclusiones obtenidas del trabajo y de algún modo indicar cuáles serían los posibles pasos a dar para continuar mejorando y enriqueciendo el proyecto.

4.1. Algoritmo de Checksum de Fletcher de 16 bits

El checksum de Fletcher es un algoritmo que computa un checksum dependiente de posición y fue desarrollado por John G. Fletcher al final de los 1970s. El objetivo del algoritmo era proporcionar una manera de detección de errores del estilo del chequeo de redundancia cíclica (CRC), pero con un coste computacional asociado menor con técnicas acumulativas.

Como en los algoritmos sencillos de checksum, el de Fletcher consiste en dividir los datos para protegerlos de error en pequeños bloques de bits, y calcular la suma modular de esos bloques. En particular los datos se dividen en bloques de 8 bits, que se combinan para formar un checksum de 16 bits.

Con el propósito de probar la herramienta sobre un diseño, se ha implementado este algoritmo en $VHDL$, además de en $C++$, lo que nos da varias opciones a la hora de comprobar el golden generado, por ejemplo. Otra ventaja, es que se ha implementado de forma que los conectores se puedan usar tanto de entrada como de salida, pudiendo así utilizar los 16 pines de los que disponemos por $I2C$.

Con el diseño se han generado además, con la ayuda de Xilinx ISE, los archivos necesarios para testear el circuito, es decir, el fichero de bitstream y el `.ll` de ubicación lógica, el cual contiene los datos de 93 registros.

Hemos creado manualmente dos ficheros de Testbench de 53 ciclos cada uno, que se diferencian en el tipo escogido, para así poder comprobar la lectura de la salidas tanto por $I2C$, como directamente de los registros se-

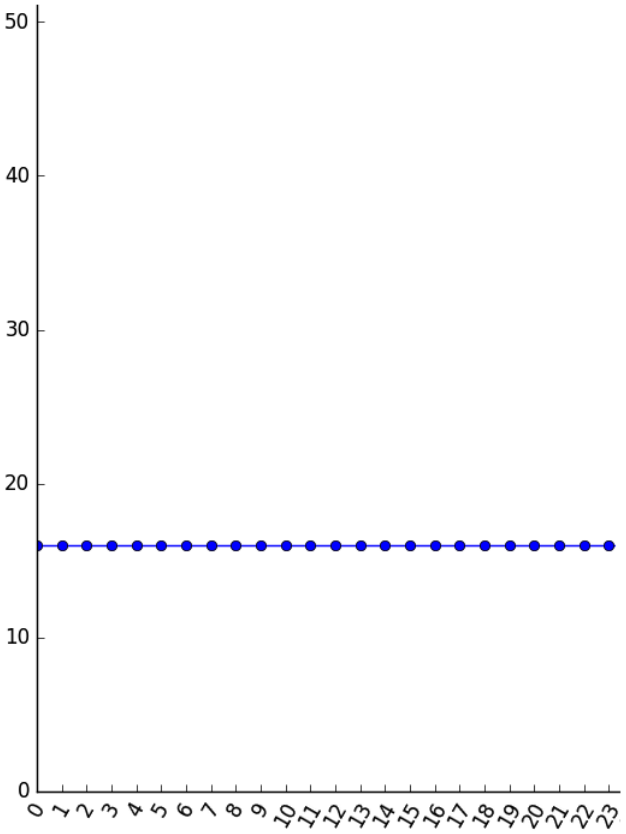


Figura 4.1: Fallos por inyección en flipflops en cada ciclo del circuito

leccionados.

Con la herramienta hemos realizado varias inyecciones. Una inyección exhaustiva sobre los flipflops, del cual se muestran las estadísticas en la tabla 4.1:

Ciclos	53
Flipflops	93
Inyecciones	4929
Configuraciones erroneas	848
Porcentaje de errores	17.20
Tiempo	607 s.

Tabla 4.1: Estadisticas inyección en flipflops

La inyección exhaustiva nos ha llevado unos 10 minutos, lo cual no es demasiado. Esto es debido a que el circuito bajo prueba únicamente hace uso de 93 flipflops, y el testbench utilizado consta de 53 datos que se envían durante 53 ciclos.

Como podemos ver, únicamente un 17 % de las inyecciones han causado error en la salida esperada del circuito, lo cual nos hace pensar que para

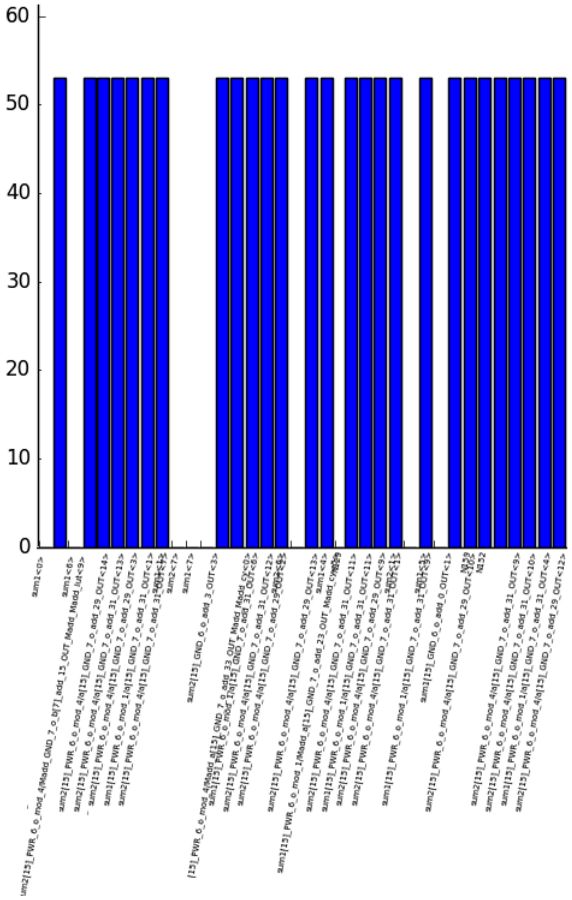


Figura 4.2: Éxitos por flipflop en la inyección

un diseño tan sencillo, el cual parece que hace uso en cada ciclo de forma acumulativa de los mismos flipflops, se han generado registros que no afectan al resultado.

Nuestras sospechas quedan confirmadas al observar las figuras 4.1 y 4.2. Según estas imágenes, generadas con los scripts en *Python* desarrollados en paralelo con nuestra herramienta, parece ser que solo hay 16 de los 93 flipflops esenciales en nuestro diseño.

En la figura 4.1, podemos ver por cada ciclo, cuántos bitflip sobre los flipflop han causado fallo, y comprobamos que el número se mantiene constante a lo largo de la gráfica, es decir, son siempre 16 flipflops los que causan error al hacer un bitflip sobre ellos, independientemente del ciclo en el que se hagan.

La segunda figura 4.2, la cual no se ha incluido completa para mejorar la legibilidad, nos dice quiénes son esos 16 flipflops esenciales en el circuito, pues todas y cada una de las inyecciones sobre ellos, independientemente del ciclo en el que se hayan realizado, han causado una salida errónea.

Estos son los flipflops que almacenan de forma acumulativa la salida ciclo a ciclo. Llama la atención que ninguna de las inyecciones en los 77 flipflop restantes haya causado una salida errónea, en ningún ciclo. Nos preguntamos

entonces, ¿serán innecesarios estos flipflops? Quizá se deba al uso de *buffers* triestado para permitir puertos de entrada y salida simultáneamente en el diseño.

Ciclos	53
Inyecciones	100.000
Configuraciones erroneas	34702
Porcentaje de errores	34.7
Tiempo	5549 s.

Tabla 4.2: Estadisticas inyección en memoria de configuración

Hemos realizado también algunas inyecciones en memoria de configuración sobre el diseño. En particular se han realizado dos rondas de inyecciones de fallos, cada una sobre una celda diferente. Cien mil inyecciones sobre la celda ($X2$, $Y0$), y otras cien mil inyecciones sobre la celda ($X3$, $Y0$) de la FPGA.

Se muestran las estadísticas de la primera de las inyecciones en la tabla 4.2, aunque ambas son similares.

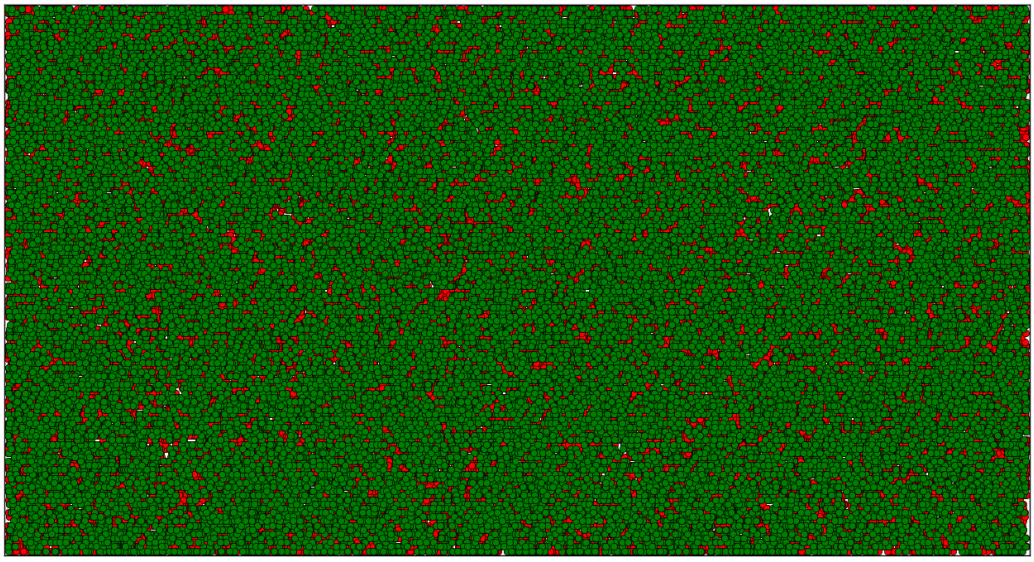


Figura 4.3: Inyección en celda de memoria ($X2$, $Y0$)

Estas inyecciones ya han tardado algo más de hora y media en realizarse. Para realizar una inyección exhaustiva sobre esta única celda he calculado que serían necesarias algo más de 20 millones de inyecciones, lo que proporcionalmente nos da aproximadamente 300 horas, o lo que es lo mismo 12 días y medio, con lo que para realizar estudios completos sobre circuitos más complejos podría volverse inviable.

Aún así hemos obtenido un porcentaje de error del 34,7 %, que podría

considerarse alto en este caso. Con la herramienta auxiliar desarrollada en *Python* generamos la imagen de la figura 4.3.

En ella intentamos visualizar las zonas de la celda de configuración, cuya modificación se hace notar en el diseño. Aunque las inyecciones se han hecho en ciclos concretos del diseño, en este caso aleatorios, no se han tenido en cuenta para la generación de la imagen. En rojo se ven las zonas donde al realizar un bitflip dentro de la celda han causado fallo. En verde vemos las zonas en las que se ha provocado bitflips que aparentemente no han alterado el funcionamiento normal del diseño, dicho de otro modo, que no han podido ser detectados con la comprobación de la salida producida, pues esta era correcta. Y en blanco se ven zonas de la celda sobre las que no se han realizado inyecciones.

4.2. Contador binario de 8 bits

Un contador es un circuito secuencial construido a partir de flipflops y puertas lógicas capaz de almacenar y contar los impulsos (a menudo relacionados con una señal de reloj), que recibe en la entrada destinada a tal efecto, asimismo también actúa como divisor de frecuencia.

Además del algoritmo de Fletcher, hemos implementado en *VHDL* el circuito de un contador binario de 8 bits para testear. En este caso no serán necesarios datos de entrada, más que la señal de reloj generada. Consideraremos en cada ejecución la generación de 255 pulsos de reloj para pasar por todos los estados intermedios del contador.

Con el diseño se han generado igual que en el caso anterior, los archivos necesarios para testear el circuito, es decir, el fichero de bitstream y el `.ll` de ubicación el cual contiene los datos de 8 registros.

Con la herramienta hemos realizado varias inyecciones. Una inyección exhaustiva sobre los flipflops, del cual se muestran las estadísticas en la tabla 4.3:

`Ciclos`	255
`Flipflops`	8
`Inyecciones`	2040
`Configuraciones erroneas`	2040
`Porcentaje de errores`	100
`Tiempo`	68 s.

Tabla 4.3: Estadisticas inyección en flipflops

La inyección exhaustiva nos ha llevado poco más de un minuto. En comparación con la prueba anterior, podemos ver que hemos realizado aproximadamente la mitad de inyecciones, y sin embargo hemos tardado unas 10 veces menos. Esto se debe a que en el circuito anterior, en cada ciclo de reloj teníamos que colocar una señal de datos distinta, lo que hace el proceso más lento, ahora sin embargo solo emitíamos la señal de reloj.

Como podemos ver, el 100 % de las inyecciones han causado error en la salida esperada del circuito, lo cual puede ser fácilmente justificable, pues el diseño solo consta de 8 bits, los 8 bits que en cada momento almacenan el estado completo del contador, si en cualquier momento de la ejecución se modifica cualquiera de ellos, el resultado final no será el esperado. No se han incluido figuras con estos datos por la sencillez del resultado.

Desde el punto de vista alternativo, el ciclo en el que se inyecta, es evidente también que en cualquiera de los ciclos que se inyecte, serán los 8 flipflops los que alteren el resultado final.

Ciclos	255
Inyecciones	200.000
Configuraciones erroneas	15
Porcentaje de errores	7.5e-05
Tiempo	2581 s.

Tabla 4.4: Estadisticas inyección en memoria de configuración

Llevamos a cabo también algunas inyecciones en memoria de configuración sobre el diseño. En este caso, doscientas mil inyecciones sobre la celda ($X2$, $Y0$) de la FPGA.

Podemos ver las estadísticas de la inyección en la tabla 4.4.

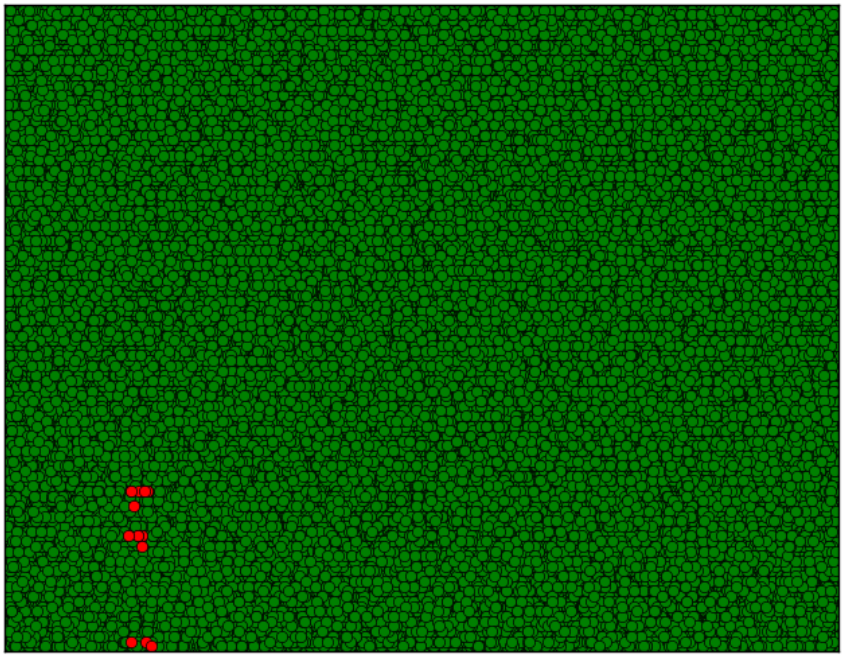


Figura 4.4: Inyección en celda de memoria ($X2$, $Y0$)

De nuevo, hemos inyectado el doble de fallos que en circuito de Fletcher, sin embargo, hemos tardado menos de la mitad del tiempo. Esto nos refleja

que quizá merecería la pena investigar formas más rápidas de colocar los datos de entrada del circuito.

También observamos que el porcentaje de error obtenido es casi nulo, y está muy localizado (ver figura 4.4). Es un circuito muy sencillo que no requiere apenas lógica, y basado mayormente en los flipflops. Quizá sería interesante una inyeccíon exhaustiva más localizada dentro de la celda.

4.3. Conclusiones

A lo largo del proyecto hemos desarrollado una plataforma libre y no intrusiva, Nessy 7.0, para testear diseños hardware implementados en FPGAs mediante inyección de fallos. Durante el proceso, y probando la herramienta, hemos ido conociendo sus limitaciones y sus ventajas.

Una de las limitaciones con la que nos enfrentamos inicialmente fue conseguir proporcionarle al circuito una buena frecuencia de reloj a través de los pines de la Raspberry, aunque finalmente conseguimos unos valores aceptables.

Como comentamos en opciones de trabajo futuro, también sería conveniente permitir introducir los datos de entrada del circuito a través de la memoria de configuración, pues actualmente solo se permite a través de las conexiones que hacemos con la Raspberry mediante los chips PCF8574 y la interfaz I2C. Aunque por un lado esta mejora nos daría más libertad, por ejemplo, en cuanto al tamaño de los datos de entrada, que actualmente se encuentra por este motivo limitado a 66 señales lógicas, por otro lado esta alternativa sería algo más lenta, pues requeriría la reconfiguración parcial de uno o varios frames en la FPGA. Esta opción sí se encuentra implementada para capturar los datos de salida del diseño.

En cuanto a las virtudes de la plataforma, encontramos la posibilidad de realizar inyección de fallos exclusivamente en los flipflops del diseño y no únicamente en la configuración en sí misma. Además esta opción viene mejorada al haber añadido una dimensión temporal en el espacio de inyección, es decir, al permitir inyectar fallos en determinados estados de un circuito, dependiendo del ciclo de reloj en el que se encuentre, lo que abre posibilidades nuevas en el estudio de vulnerabilidades de un circuito.

Además hemos podido comprobar que una inyección exhaustiva en memoria de configuración de un diseño no necesariamente grande puede llevar varias semanas, lo que quizá nos diga que podría ser necesario buscar para esos casos otras alternativas o realizar estudios estadísticos con inyecciones aleatorias localizadas.

Finalmente, mencionar que la plataforma nos permite realizar el estudio de vulnerabilidades, conociendo sus limitaciones, de una forma económica pues el hardware necesario es mínimo.

4.4. Trabajo futuro

El trabajo de desarrollo de la plataforma podría continuar de distintas formas. Por ejemplo se podría dar soporte a inyectar los datos de entrada del circuito directamente sobre los registros usando archivos de configuración parcial de forma similar a como actualmente se permiten leer las salidas de circuitos. Esto haría mas sencillo testear circuitos que requieran datos de entrada de gran tamaño.

Además podría resultar útil ofrecer la posibilidad de comparar estados intermedios del circuito con estados intermedios del Golden, por ejemplo para generar una batería de tests en un único fichero.

Otra posibilidad sería diseñar archivos de inyección selectiva de fallos, que la herramienta pudiese cargar, y en los que se indicase ciclos y la localización de los bitflips a realizar.

Respecto a los ficheros de resultado de inyección, que pueden llegar a ser muy grandes y dificiles de manejar. En este proyecto se han desarrollado un par de scripts en python que nos ayudan a sacar estadisticas y visualizar gráficamente los resultados, pero quizá podría ser interesante escribirlos directamente en una base de datos de tipo no SQL, como por ejemplo MongoDB que da muchas facilidades para trabajar con grandes cantidades de datos en formato JSON.

También se podría estudiar la implementación de técnicas de protección de circuitos a partir de los datos de vulnerabilidad obtenidos con la herramienta.

Bibliografía

[1] Carl Carmichael, Earl fuller, Phil Blain, Michael Caffrey, *SEU Mitigation Techniques for Virtex FPGAs in Space Applications*, Los Alamos National laboratory, Novus Technologies, Inc, Xilinx, Inc.

[2] Katherine Compton y Scott Hauck, *An Introduction to Reconfigurable Computing*,

[3] F. Corno, M. Reorda y G. Squillero, *RT-Level ITC'99 Benchmarks and First ATPG Results, Design Test of Computers*, IEEE, vol. 17, no. 3, pp. 44 –53, July/Sept. 2000.

[4] Nexys4 DDR FPGA Board Reference Manual, Revised September 11, 2014

[5] 7 Series FPGAs Configuration, *User Guide*, UG470 (v1.10) June 24, 2015.

[6] Virtex-5 FPGA, *Congiguration User Guide*, UG191 (v3.11) October 19, 2012.

[7] Ken Chapman, *SEU Mitigation Techniques for SRAM based FPGAs*, 30th September 2015, TWEPP2015.

[8] 7 Series FPGAs Configurable Logic Block, *User Guide*, UG474 (v1.7) November 17, 2014.

[9] Partial Reconfiguration, *User Guide*, UG702 (v14.1) April 24, 2012.

[10] Haissam Ziade, Rafic Ayoubi, Raoul Velazco, y Tarek Idriss *A New Fault Injection Approach to Study the Impact of Bitflips in the Configuration of SRAM-Based FPGAs* , Faculty of Engineering I, Lebanese University, Lebanon Department of Computer Engineering, University of Balamand, Lebanon. TIMA Laboratory, INPG, France, April 2011.

[11] Standard ECMA-404, *The JSON Data Interchange Format* , First Edition, October 2013.

Anexo A: Bitstream

Los diseños de las FPGAs se cargan como un bitstream, cuyo formato se muestra en la tabla A.2, que se carga a través de la interfaz para configurar la FPGA con el diseño. Cada bitstream dependiendo del dispositivo, es de un tamaño fijo, en el caso de la FPGA que hemos usado el tamaño es de $30,606,304$ bits y es cargado a través de la interfaz JTAG, leyendo y escribiendo sobre los registros de configuración, utilizando un $IDCODE$ en hexadecimal de 32 bits como por ejemplo $X3631093$ donde el valor X representa el campo de revisión $IDCODE[31:28]$ que puede variar.

El archivo .bit es un archivo binario de configuración que contiene una cabecera, que puede verse en la tabla A.1, aunque no es necesario descargar esta a la FPGA, y puede ser utilizada para programar el dispositivo desde la herramienta iMPACT.

Campo	**Longitud**	**Explicación**
0	13	Número mágico: $00090FF00FF00FF00FF0000001$
a	XXXX	Nombre del diseño; UserID=$0xFFFFFFFF\backslash 0$
b	XXXX	Part number (p.e $5vlx110tff1136)\backslash 0$
c	XXXX	Fecha $AAAA/MM/DD\backslash 0$
d	XXXX	Hora $HH{:}MM{:}SS\backslash 0$
e	XXXXXXXX	Bitstream (la longitud, en bytes, incluye los comandos)

Tabla A.1: Formato de la cabecera del Bitfile

El bitstream contiene comandos para la configuración lógica de la FPGA y datos. En general tiene tres secciones.

- Autodetección del ancho de bus
- Palabra de sincronización
- Configuración de la FPGA

El patrón de autodetección de ancho de bus se inserta al comienzo de cada bitstream, se usa en modos de configuración paralela para detectar automáticamente la configuración del ancho de bus. Como aparece antes de la palabra de sincronización, los modos de configuración en serie la ignoran.

La palabra de sincronización ($0xAA995566$) se usa para permitir a la configuración lógica alinearse con una palabra de 32 bits. La FPGA no procesa ningún paquete hasta que encuentra la palabra de sincronización. Después de la sincronización, la configuración lógica procesa cada palabra de 32-bit como un paquete de configuración o como un componente de una configuración de varias palabras.

La configuración de las FPGA está organizada en frames. Los frames son los segmentos direccionables más pequeños del espacio de la memoria de configuración y por tanto todas las operaciones deben actuar sobre frames de configuración completos. En el caso de las Artix-7 cada frame consta de 101 palabras de 32-bits.

Todos los comandos del bitstream se ejecutan, leyendo o escribiendo, sobre los registros de configuración (ver 3.3).

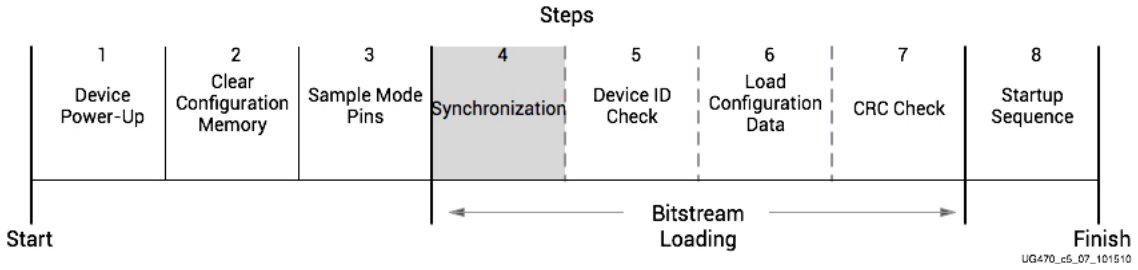


Figura A.1: Paso de sincronización

La secuencia de configuración (ver figura A.1) se puede describir en tres fases divididas en 7 pasos básicos, que son los mismos para todos los modelos de la serie 7. Estos pasos son:

1. Encendido del dispositivo
2. Resetear memoria de configuración
3. Muestreo de los pines de modo
4. Sincronización
5. Comprobación Id de dispositivo
6. Carga de los datos de configuración
7. Comprobación de CRC
8. Secuencia de inicialización

Times	Configuration data word (Hex)	Description
8	$FFFFFFFF$	DUMMY
1	$000000BB$	Bus Width Sync Word
1	11220044	Bus Width Detect
2	$FFFFFFFF$	DUMMY
1	$AA995566$	SYNC
1	20000000	NOOP
1	30008001	Type 1 Write 1 word to CMD
1	00000007	RCRC command
2	20000000	NOOP
1	30022001	Type 1 Write 1 word to TIMER
1	00000000	NOOP
1	30026001	Type 1 Write 1 word to CBC
1	00000000	
1	30012001	Type 1 Write 1 word to COR0
1	$02003FE5$	Ignore it doing OR with $0x10000000$
1	$3001C001$	Type 1 Write 1 word to COR1
1	00000000	
1	30018001	Type 1 Write 1 word to IDCODE
1	$02AD6093$	ID code
1	30008001	Type 1 Write 1 word to CMD
1	00000009	SWITCH command
1	20000000	NOOP
1	$3000C001$	Type 1 Write 1 word to MASK
1	00400000	
1	$3000A001$	Type 1 Write 1 word to CTL0
1	00400000	
1	$3000C001$	Type 1 Write 1 word to MASK
1	00000000	
1	30030001	Type 1 Write 1 word to CTL1
1	00000000	
8	20000000	NOOP
1	30002001	Type 1 Write 1 word to FAR
1	00000000	
1	30008001	Type 1 Write 1 word to CMD
1	00000001	WCFG command (Write Configuration Data)
1	20000000	NOOP
1	30004000	Type 1 Write 0 words to FDRI
1	$0x500ED5A0$	Type 2 Write ED5A0 words to FDRI (size, in words, of configuration memory)
Siguen los bits de la memoria de configuración		
1	30000001	Type 1 Write 1 word to CRC
1	$F7EFDE50$	CRC value. It is ignored when it has the $0x0000DEFC$ value
1	30008001	Type 1 Write 1 word to CMD
1	$0000000A$	GRESTORE command
1	20000000	NOOP
1	30008001	Type 1 Write 1 word to CMD
1	00000003	DGHIGH/LFRM command (Last Frame)
100	20000000	NOOP
1	30008001	Type 1 Write 1 word to CMD
1	$0000000A$	GRESTORE command
30	20000000	NOOP
1	30008001	Type 1 Write 1 word to CMD
1	00000005	START command
1	20000000	NOOP
1	30002001	Type 1 Write 1 word to FAR
1	$00EF8000$	
1	$3000C001$	Type 1 Write 1 word to MASK
1	00400000	
1	$3000A001$	Type 1 Write 1 word to CTL0
1	00400000	
1	30000001	Type 1 Write 1 word to CRC
1	$0C90449E$	CRC value. It is ignored when it has the $0x0000DEFC$ value
1	30008001	Type 1 Write 1 word to CMD
1	$0000000D$	DESYNCH command
31	20000000	NOOP

Tabla A.2: Formato del bitstream

Anexo B: Archivo .ll

El archivo .ll consta de varias líneas, que pueden ser de dos tipos distintos, las líneas de Bit y las de Info.

Las líneas de Bit tienen el siguiente formato:

<offset>	Contiene la posición absoluta del bit dentro del Bitstream.
<frame address>	Contiene la dirección del frame de configuración que contiene al bit al que hace referencia.
<frame offset>	Se corresponde con el desplazamiento del bit dentro del frame.
<information>	Que pueden ser cero o más pares $< kw >=< value >$.
Block=<blockname>	Especifica el bloque asociado con esta celda de memoria.
Latch=<name>	Especifica el *latch* asociado con esta celda de memoria.
Net=<netname>	Especifica el nombre asociado al componente en el diseño.
COMPARE=[YES | NO]	Especifica si es o no apropiado comparar esta posición del bit entre el bitstream de programación y el de *readback*. Si no está presente, el valor por defecto es *No*.

Ram=< *r*am id>:<bit>	Se usa en casos donde hay un generador de función de CLB usado como RAM (o ROM). ▪ <Ram id>: podrá ser 'F', 'G' o 'M' indicando que es parte de una función simple F, o un generador de función G, o como una simple Ram (o ROM) construida por ambos, F y G. ▪ <Bit>: es un número decimal.

Las líneas de Info son de la siguiente forma:

Info <name>=<value>	Especifica un bit asociado con las opciones de configuración de la celda y el valor del bit. El nombre de estos bits pueden tener significado especial para programas que analizan los archivos .ll.

Anexo C: JSON

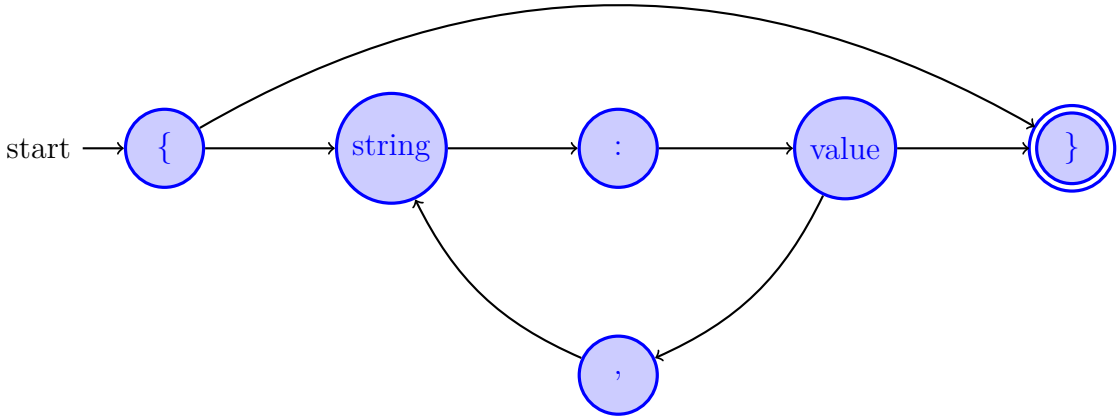
JSON (JavaScript Object Notation) es un formato de intercambio de datos que es fácil de leer y escribir para los seres humanos, y simple de analizar y generar para las máquinas. Está basado en un subconjunto del lenguaje de programación *Javascript*, como su nombre indica. JSON es un formato de texto que es independiente del lenguaje usado, pero su uso implica seguir una serie de convenciones.

JSON está constituido por:

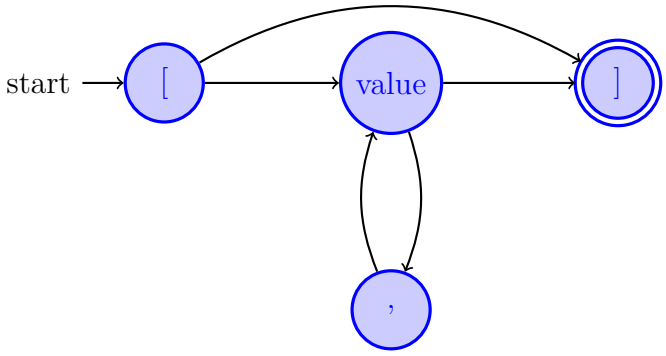
- Una estructura que está basada en una colección de pares de nombre/valor. En algunos lenguajes esto se puede asemejar a un objeto, diccionario, tabla *hash* ...
- Una estructura basada en una lista ordenada de valores. En algunos lenguajes esto se puede asemejar a un vector, un array, una lista ...

Veamos las diferentes formas que se pueden representar en JSON:

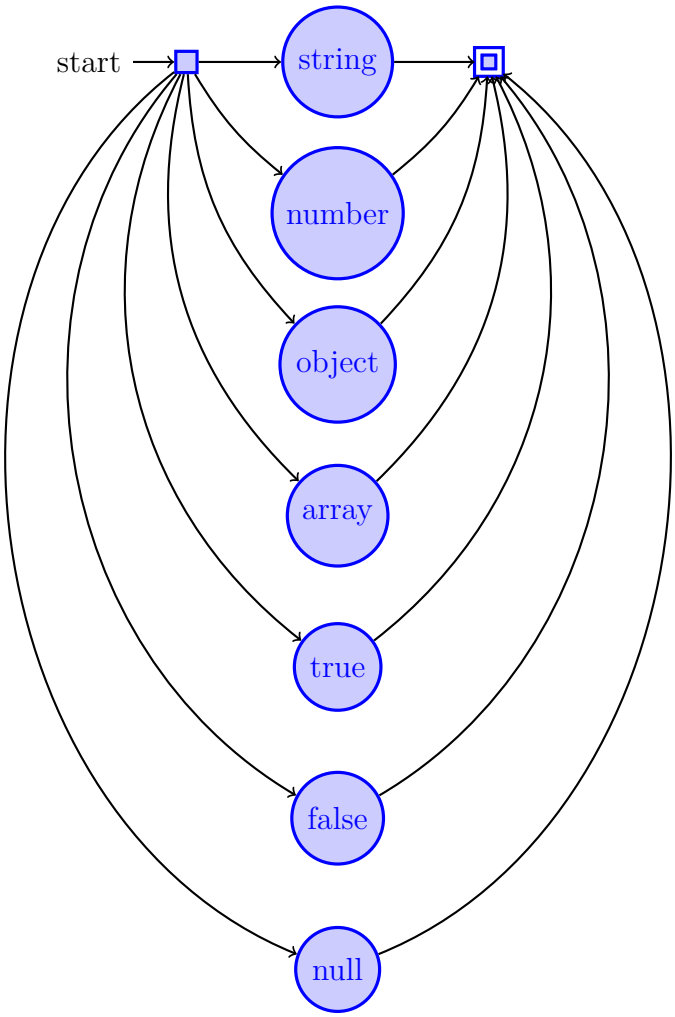
- **Objeto**: Es un conjunto desordenado de pares nombre/valor. Estos pares siempre están encerrados entre los símbolos { (comienzo de objeto) y } (final de objeto). Cada nombre es seguido del símbolo : seguido del valor, y estos pares están separados por comas (,).



- **Array**: Es una colección de valores separados por comas (,). Estos valores están limitados por los símbolos [(comienzo del array) y] (final del array).

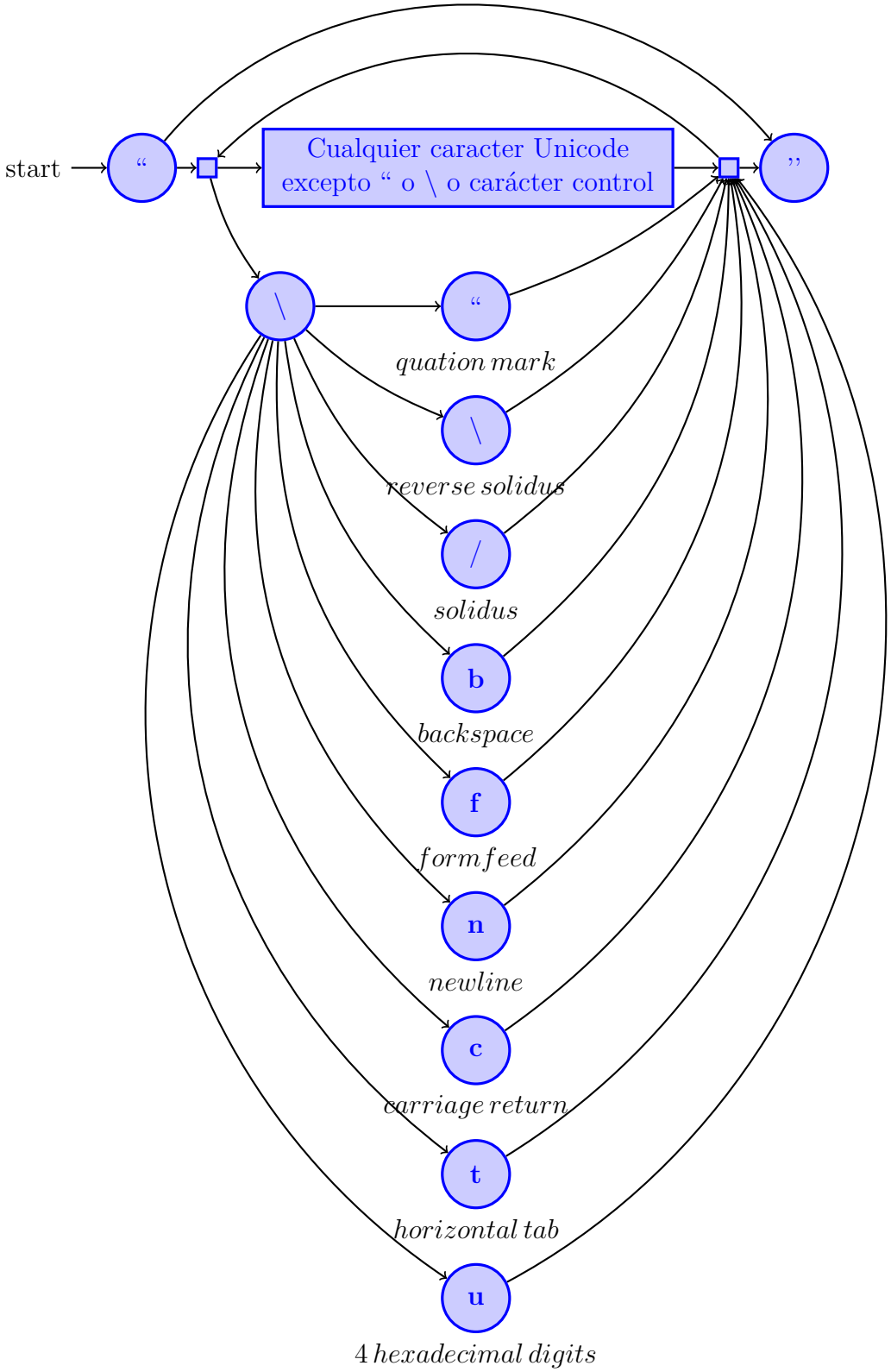


- **Valor**: Puede ser de varias formas: una cadena de caracteres encerrada entre comillas dobles, un número, true o false, null, un objeto o un array.



- **Cadena de caracteres (string)**: Es una colección de cero o más carac-

teres *Unicode*[1], limitados por el comienzo y fin de comillas dobles, en la que se puede hacer uso de barra divisoria invertida como escape.



[1]Unicode es un estándar de codificación de caracteres diseñado para que el tratamiento informático de textos de múltiples lenguajes y disciplinas técnicas sea más fácil

- **Número**: Está representado en base 10. Este puede ser precedido del signo menos (-). Puede tener un punto (.) indicando que tiene parte decimal. Además, puede tener un exponente de diez, prefijado por *e* o *E* y opcioinalmente seguido de los símbolos + (más) o - (menos). Resaltamos que no se permite el uso de representación octal y hexadecimal.