Trabajo Fin de Máster
Ingeniería de las Tecnologías Industriales

Estudio de viabilidad y desarrollo de infraestructura para plataforma de inyección de fallos a través de PCI express en Kintex Ultrascale

Autor: Javier González Moreno

Tutor: Hipólito Guzmán Miranda

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Máster
Ingeniería de las Tecnologías Industriales

Estudio de viabilidad y desarrollo de infraestructura para plataforma de inyección de fallos a través de PCI express en Kintex Ultrascale

Autor:
Javier González Moreno

Tutor:
Hipólito Guzmán Miranda
Profesor Titular

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2022

Trabajo Fin de Máster: Estudio de viabilidad y desarrollo de infraestructura para plataforma de inyección de fallos a través de PCI express en Kintex Ultrascale

Autor: Javier González Moreno
Tutor: Hipólito Guzmán Miranda

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Grow as we go.

Ben Platt

En primer lugar quisiera expresar mi agradecimiento a Hipólito, director de este TFM (Trabajo Final de Máster), ya que ha sido su valiosa experiencia, orientación y motivación lo que ha permitido su realización. Siempre es un placer trabajar al lado de un gran profesional que, además, es una grandísima persona.
En segundo lugar, pero no menos importante, quiero agradecer la ayuda, cariño y paciencia que me han brindado aquellos que, desde el anonimato, han aportado mucho a este trabajo. Con esto me refiero a mis padres, grandes revisores y correctores, pero sobre todo un apoyo incondicional. Fueron ellos lo que, desde pequeño, motivaron mi interés por entender el funcionamiento de todo lo que me rodeaba y gracias a los cuales he llegado hasta donde estoy hoy.
También me gustaría recordar en este momento a las personas que han formado una parte muy importante de mi vida, pero que a día de hoy no pueden compartir este momento conmigo. A mi abuelo Manolo, por enseñarme todo lo que puede llegar a transmitir el silencio de una mirada. A mi abuela Pilar, por su habilidad para ver todo bueno que había dentro de mí hasta el final. A mi abuelo Manolo, por la unión tan fuerte que teníamos, la cantidad de momentos compartidos y cómo me dejaba ser lo que el llamaba "el báculo de su vejez". Y a mi tío Gabriel, una persona incansable en el amor y entrega hacia los demás y un gran ejemplo a seguir.
Tampoco me quiero olvidar de aquellos que forman parte de mi rutina: mis hermanos, Emma e Ignacio, mi abuela María Luisa y toda mi familia completa. Son personas muy importantes para mí y me han aportado muchísimas cosas a lo largo de todos estos años.
De mis amigos y compañeros de carrera también me gustaría acordarme, por los ratos compartidos y por todo el camino recorrido. Del magnífico grupo que nos dedicamos a organizar todo tipo de eventos tampoco me quiero olvidar, pues son ellos los que son capaces de conseguir que la rutina nunca nos absorba y que todos salgamos siempre a flote.
Por último, no me olvido de la persona más importante para mí a día de hoy. Gracias María por tanto amor, tanta paciencia y, sobretodo, por sacar siempre lo mejor de mí.

Javier González Moreno
Alumno Interno de la Universidad de Sevilla

Sevilla, 2020

Resumen

Este Trabajo Fin de Máster parte de la oportunidad de participar en el diseño de una plataforma de inyección de fallos en la tarjeta ADM-XRC-KU1 de Alpha Data, así como del estudio de la misma realizado en mi Trabajo Fin de Grado [5].
Concretamente, se centra en desarrollar la infraestructura mínima necesaria para demostrar la viabilidad de implantar un sistema de inyección de fallos completo en la FPGA bajo estudio (XCKU060).

Para ello, se comienza resolviendo las comunicaciones entre el ordenador en el que se conecta la tarjeta y las interfaces de entrada y salida de un sistema de inyección de fallos.

En paralelo, se trabaja para mejorar la infraestructura que maneja y prepara las instrucciones (comandos y datos) a enviar a través del sistema de comunicaciones.

Por otro lado, se diseña un intérprete de comandos sencillo (a nivel *hardware*) capaz de interpretar y ejecutar varios comandos propios de un sistema de inyección de fallos.

Por último, se genera un proyecto que permite la reconfiguración parcial con el objetivo de obtener *bitstreams* parciales funcionales, ya que son necesarios para poder inyectar fallos.

El objeto de este documento es presentar toda la información recabada durante el desarrollo del trabajo, así como los resultados de las pruebas y arquitecturas diseñadas con el fin de obtener los elementos mínimos necesarios para demostrar la viabilidad de un sistema de inyección de fallos.

Abstract

This Master's Thesis begins when given the opportunity to work in the design of a fault injection platform on the ADM-XRC-KU1 board (developed by Alpha Data). It is also based on the FPGA study carried out in Bachelor's Thesis.

Specifically, it focuses on developing the minimum necessary infrastructure to demonstrate the feasibility of implementing a complete fault injection system on the FPGA under study (XCKU060).

To achieve this, the communications between the computer to which the card is connected and the input and output interfaces of a fault injection system are first resolved.

In the meantime, the infrastructure that handles and prepares the instructions (commands and data) to be sent through the communications system is improved.

Furthermore, a simple command interpreter (at hardware level) is designed. This interface must be able to interpret and execute several commands of a fault injection system.

Finally, a project that allows partial reconfiguration is generated. This makes it possible to obtain functional partial bitstreams, as they are necessary to inject faults.

The purpose of this document is to collect all the information obtained during the development of this project, as well as the results of the tests and architectures designed in order to obtain the minimum necessary infraestructure to demonstrate the viability of a fault injection plataform.

Índice Abreviado

Lista de tareas pendientes

Índice

Glosario

API
: Application Programming Interface

ARM
: Advanced RISC Machine

AMBA
: Advanced Microcontroller Bus Architecture

AVR
: Familia de microcontroladores

AXI, AXI4
: Advanced eXtensible Interface

bRAM
: Block RAM

COTS
: Commercial Off The Shelf

DDR4 SDRAM
: Double Data Rate 4 Synchronous Dynamic RAM

DMA
: Direct Memory Access

DSP
: Digital Signal Processor

FIFO
: First In, First Out

FPGA
: Field-Programmable Gate Array

FT-UNSHADES (FTU)
: Fault Tolerance Universidad de Sevilla Hardware Debugging System

GIE
: Grupo de Ingeniería Electrónica

JTAG
: Joint Test Action Group

KU
: Kintex Ultrascale

LUT
: Look-Up Table

MPTL
: Modular Plug Terminated Link

OCP
: Open Core Protocol

PCIe
: Peripheral Component Interconnect Express

RAM
: Random Access Memory

SRAM
: Static RAM

USB
: Universal Serial Bus

VPWR
: Voltage Input Power

V5
: Virtex 5

XMC
: Switched Mezzanine Card

1 Introducción

Todo se andará hijito.

PILAR VICENCE RECUERO

El ser humano ha demostrado desde siempre un gran interés por investigar, descubrir y conocer su entorno. Últimamente, esa mirada va mucho más allá del planeta Tierra: las galaxias cercanas, el resto de planetas del sistema solar y el espacio exterior cada vez toman más relevancia.

Gracias a los avances científicos y al desarrollo en la carrera espacial, se ha conseguido reducir los costes de lanzamiento, tanto de los satélites puramente científicos, como de aquellos que proporcionan servicios. Nos encontramos ante una nueva era espacial, en la que los grandes proyectos exclusivos han dejado de ser la norma.

Esta nueva visión del sector espacio, conocida como *New Space*, se caracteriza por un crecimiento cada vez mayor de misiones *low-cost*, en la que la cantidad pasa a ser mucho más importante que la la calidad. Un ejemplo muy ilustrativo de esta nueva estrategia son las constelaciones de satélites, como *Starlink* de *SpaceX* [11] y *OneWeb* [1]. Entre las dos tienen más de 2700 satélites en órbita.

Debido a esto, enviar satélites al espacio plantea retos cada vez más complejos, debido a la necesidad de emplear componentes *low-cost*. La filosofía de utilizar elementos de bajo coste se conoce como COTS (*Commercial Off The Shelf*) y va ligada al concepto de *New Space*.

El espacio es un entorno muy agresivo, por lo que siempre se han sometido todos los dispositivos a pruebas muy estrictas antes de ponerlos en órbita. Concretamente, destacan el test térmico y de vacío (se suelen hacer a la vez), los ensayos de vibraciones y el test de radiación.

Aun así, algunas de estas pruebas dejan de tener sentido cuando introducimos los COTS, dejando sitio a otros métodos tales como las plataformas de inyección de fallos. Estas toman cada vez más importancia, para mejorar la robustez en diseños implementados en Field-Programmable Gate Arrays (FPGAs) COTS.

1.1 Inyección de fallos: FT-UNSHADES

Esta técnica consiste en la introducción de fallos en el sistema y el posterior estudio del comportamiento del sistema en presencia de estos, comparando la salida del sistema antes y después del test. De esta forma, se pueden validar diseños tolerantes a fallos, al estudiar la robustez y fiabilidad de los mismos durante el proceso de diseño [12].

Si se tienen resultados de radiación, normalmente no son necesarios los resultados de inyección de fallos. Es por ello que la inyección de fallos se utiliza como ayuda al diseño de sistemas robustos, ya que hacer inyección de fallos a un diseño y trabajar para mejorar su robustez en función de los resultados de dicha inyección es mucho más económico en tiempo, esfuerzo y coste que fabricar un diseño, irradiarlo y, si se quiere mejorar su robustez o modificarlo, volverlo a fabricar y volverlo a irradiar.

Por otro lado, los resultados de inyección de fallos se pueden combinar con resultados de sensibilidad de la tecnología para poder obtener la sensibilidad de un diseño, ver [10] y [13].

1.1.1 FT-Unshades

FT-Unshades (FTU) es un sistema de inyección de fallos híbrido (utiliza conceptos de inyección por hardware y software) basado en FPGA y no instrumentado [2]. Esto se traduce en que no es necesario modificar el circuito para poder realizar la inyección de los fallos.

Fue desarrollado por el grupo de investigación GIE (Grupo de Ingeniería Electrónica) del departamento de Ingeniería Electrónica de la Universidad de Sevilla.

La segunda versión de este sistema (FT-Unshades2) utiliza el *Universal Serial Bus* para comunicarse con las FPGAs y realizar la inyección de fallos. Este método empieza estar obsoleto, ya que el Peripheral Component Interconnect Express (PCIe) ofrece más prestaciones y mayor velocidad. Es por ello que las nuevas líneas de investigación están enfocadas a este tipo de interfaces.

1.2 Tarjeta ADM-XRC-KU1

Se trata de una tarjeta de desarrollo distribuida por Alpha Data (imagen 1.1) cuyas características podrían permitir la integración de una plataforma de inyección de fallos aprovechando el PCIe.

Está recomendada por Xilinx para el desarrollo de sistemas aeroespaciales, ya que, al tratarse de una tarjeta muy completa, permite diseñar sin restricciones. De esta forma, se pueden recortar gastos y facilitar el diseño, ya que no se fabrica la tarjeta final hasta que no se ha validado por completo el diseño.

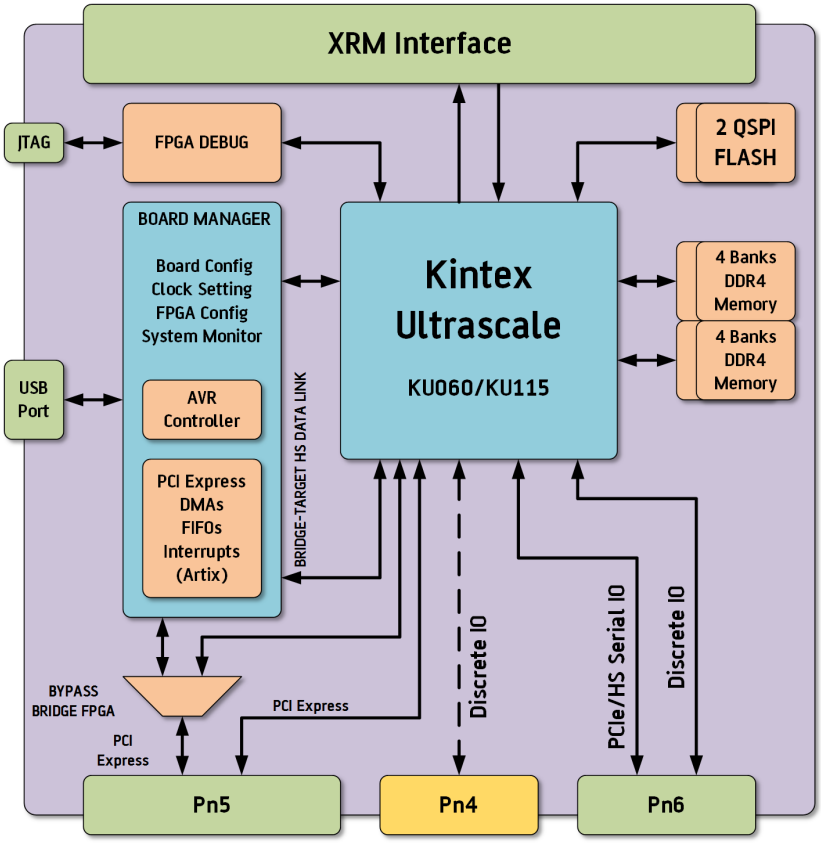

Figura 1.1 Tarjeta ADM-XRC-KU1 [3].

En cuanto a las características, cabe destacar las siguientes:

- Se trata de una Switched Mezzanine Card (XMC) de alto rendimiento basada en la familia Xilinx Kintex Ultrascale de FPGAs, concretamente cuenta con una XCKU060 (*target* FPGA).
- Posee una interfaz PCIe Gen2, conectable a un PC mediante una *Carrier Board*.
- Cuenta con 4x2GB DDR4-2400 de memoria externa, puertos de E/S de alta densidad y un sistema de monitoreo del sistema.
- Además, cuenta con memoria flash de arranque.
- Por otro lado, la placa se gestionada mediante la combinación de una FPGA Artix (*bridge* FPGA) y un microcontrolador AVR los cuales permiten administrar la tarjeta a través del PCIe o USB.
- Por último, posee una Application Programming Interface (API) integral multiplataforma con soporte para Microsoft Windows, Linux y VxWorks, desarrollada por Alpha Data, que proporciona acceso a la funcionalidad completa de todas las características del hardware.

1.2.1 PCIe

El principal atractivo de esta tarjeta, tal y como se menciona al principio de la sección, es la interfaz PCIe ya que, además de ofrecer mejores prestaciones que el JTAG o USB, permite nuevas formas de trabajo a nivel del software que se ejecuta en el PC.

Todo eso gracias a la API proporcionada por Alpha Data, ya que ofrece, entre otras cosas, mapear la memoria interna de la FPGA al mapa de memoria del PC.

2 Alcance del Proyecto

Tanto Cuanto.

José Luis González Moreno

2.1 Antecedentes

Este trabajo parte de la oportunidad de continuar trabajando con la tarjeta ADM-XRC-KU1, investigando la viabilidad y las diferentes opciones para conseguir un sistema de inyección de fallos completo.

Se parte de los conocimientos adquiridos durante la realización del Trabajo de Fin de Grado [5], que investiga la interfaz ADM-XRC-KU1-HSAXI (proporcionada por Alpha data) y propone varios sistemas para adaptarla a un sistema de inyección de fallos.

Concretamente, se decide continuar con uno de los trabajos propuestos tras realizar el estudio, que consiste en el desarrollo de una interfaz que permita la interpretación y la ejecución de comandos, así como la gestión de datos, basada en la tecnología bRAM [5].

2.2 Plataforma de Inyección de Fallos: FTU-Lite

Los elementos mínimos necesarios a implementar para estudiar la viabilidad de un sistema de inyección de fallos en la tarjeta ADM-XRC-KU1, se basan en la arquitectura propuesta por el Grupo de Ingeniería Electrónica (GIE) de la Universidad de Sevilla para el FTU-Lite [6] (imagen 2.1).

Por un lado, es importante tener un sistema de comunicaciones que permita tanto enviar y recibir datos a la FPGA, como inyectar fallos a través del *SelectMap*. Además, debe contener una interfaz que controle el sistema (*controller* en la figura 2.1), encargada de manejar los datos y realizar las inyecciones.

El resto de sistemas que aparecen en la imagen 2.1 no se consideran esenciales para demostrar la viabilidad de diseñar una plataforma de inyección de fallos en la tarjeta ADM-XRC-KU1, por lo que no se considerarán en este proyecto.

2.2.1 Infraestructura mínima necesaria

A partir de los mencionado en el párrafo anterior, se deduce que para poder implementar un sistema de inyección de fallos en la tarjeta bajo estudio se ha de comenzar resolviendo al completo las comunicaciones entre el ordenador y la FPGA (interfaces de entrada y salida de la plataforma de inyección de fallos).

Una vez se han resuelto las comunicaciones, se ha de trabajar en la arquitectura propia de un sistema de inyección de fallos. Concretamente, se ha de desarrollar una interfaz que permita controlar todo el sistema (*core_fsm* o intérprete de comandos).

A continuación, se ha de investigar cómo inyectar los fallos en la FPGA, generalmente a través del *SelectMap*. Para ello, es necesario contar un con *bitstream* que permita reconfigurar parcialmente la FPGA, ya que a partir de este, se pueden obtener las cabeceras necesarias para inyectar los fallos.

Por último, es necesario desarrollar un sistema que permita leer y traducir los *bitstreams* parciales (desensamblador) para obtener tanto las instrucciones como los frames que se envían a la FPGA.

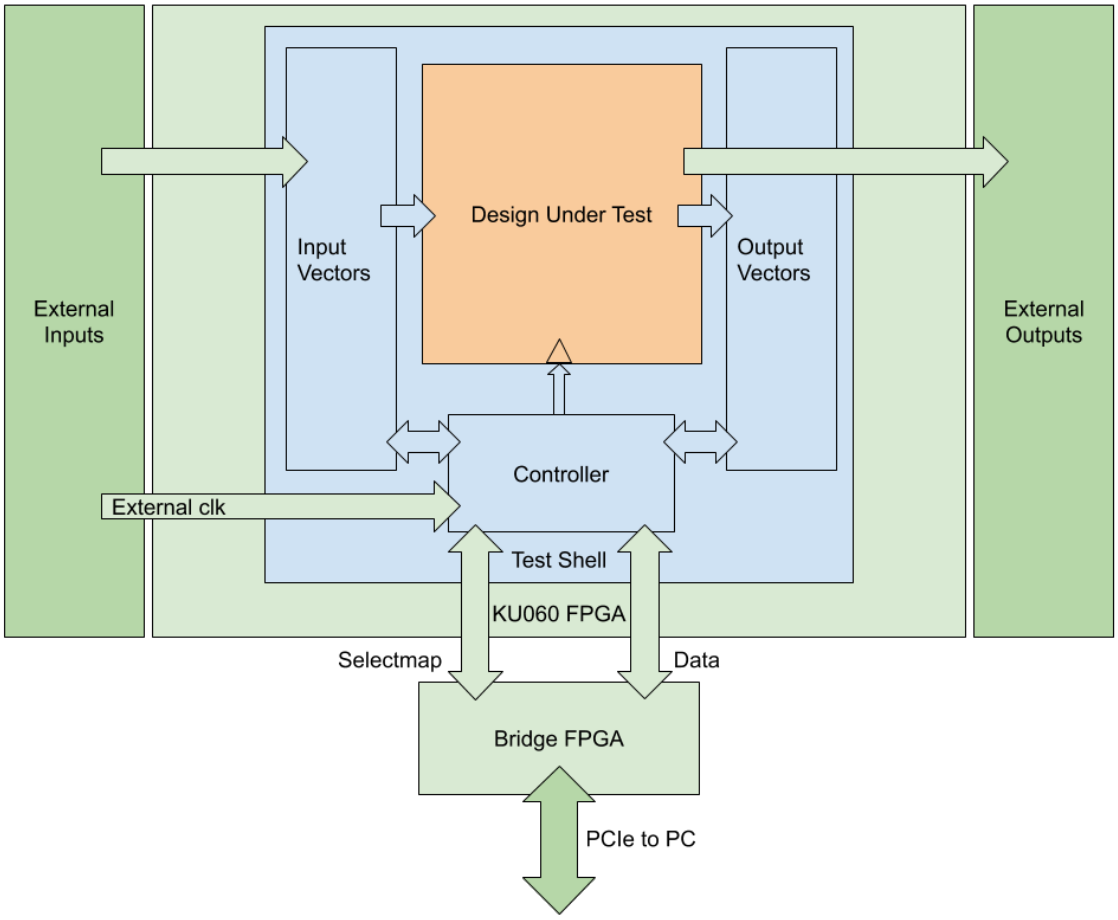

Figura 2.1 Arquitectura FTU-Lite [6].

2.3 Alcance del proyecto

La finalidad principal de este trabajo es estudiar la viabilidad y desarrollar la infraestructura necesaria para implementar una plataforma de inyección de fallos en la tarjeta ADM-XRC-KU1 (descritos en la sección 2.2). Para ello, se proponen los siguientes objetivos:

1. Estudiar la viabilidad de implementar una plataforma de inyección de fallos en la tarjeta ADM-XRC-KU1 mediante el diseño y verificación de la infraestructura mínima necesaria.

2. Diseñar un sistema de comunicaciones completo, que permita transferencias de instrucciones (comandos y datos) desde el ordenador al que la tarjeta se encuentra conectado (*host computer*) hasta las interfaces de entrada y salida de un sistema de inyección de fallos (*ftu_streams*), implementado dentro de la arquitectura de la FPGA.

3. Preparar el *software* (programa que se ejecuta en el ordenador al que está conectado la tarjeta) para manejar grandes transacciones de datos.

4. Diseñar un intérprete de comandos que reciba las instrucciones, las procese, las ejecute y envíe (a través de las interfaces de entrada y salida del sistema de inyección de fallos) los resultados de las mismas al *host*.
5. Obtener un bitstream parcial funcional, que permita reconfigurar parcialmente el dispositivo. Este ha de seguir funcionando sin ningún error tras la reconfiguración.
6. Obtener un sistema que permita desensamblar *bitstreams* de la familia Kintex Ultrascale (KU), identificando las instrucciones y *frames* que este contiene.

3 Interfaz de Comunicaciones

Descanso para el alma y la balanza vuelve a estar en su sitio.

María de Luján

El objetivo de este primer capítulo es estudiar la viabilidad de un sistema de comunicaciones que permita transferir información entre las interfaces de entrada/salida del sistema de inyección de errores y el *host*, basado en la tecnología bRAM.

Esto es debido a que se cuenta con un sistema que permite enviar datos desde el PC hasta dos memorias bRAM, desarrollado en el Trabajo de Fin de Grado [5] a partir de los ejemplos incluidos en el SDK de la tarjeta [4]. Este utiliza la interfaz ADM-XRC-KU1-HSAXI y aprovecha las funciones de la API (proporcionada por Alpha Data) para enviar datos desde el *host* hasta las bRAMs a través de dos Direct Memory Access (DMA) Engines (presentes en la interfaz ADM-XRC-KU1-HSAXI).

Todo esto ocurre a través del PCIe, ya que la comunicación entre el *host* y la tarjeta se realiza por medio de este, mediante las funciones de la API.

Por tanto, a partir de todo eso, se diseña un test que estudia si es factible emplear direcciones específicas de una memoria bRAM como señales que indiquen el inicio o final de una transacción entre PC y FPGA.

Una vez comprobado que el sistema es viable, se diseña un test basado en el anterior para demostrar que es posible transferir datos desde una bRAM a un *ftu_stream*. Estas interfaces se encuentran descritas en profundidad en la sección 5.3 del TFG [5].

Por último, se diseña la interfaz de comunicaciones completa, capaz de enviar comandos y datos al sistema de inyección de fallos, así como de recibir los resultados y enviarlos a través de la infraestructura existente hasta el *host*.

Es importante mencionar, que en este documento los test se enumeran del 7 en adelante, ya que el último test del TFG [5] fue el número 6.

3.1 Estudio de viabilidad de direcciones bRAM como señales de inicio/final de transmisión: Test 7

El objetivo de este test es comprobar si es posible utilizar dos direcciones específicas de una bRAM (bRAM0) como señales (o *flags*) que indiquen el comienzo o final de una transmisión. Concretamente, se va a utilizar la primera dirección para indicar que la escritura del *host* en la bRAM ha terminado y la última para avisar al *host* de que se ha recibido la *flag*.

En cuanto al dato específico a escribir en estas direcciones para notificar a la otra parte (*ack*), se decide llenar esa dirección de unos (`[0xffffffffffffffffffffffffffffffff]`), ya que las memorias están inicializadas con todos los datos a cero (`[0x00000000000000000000000000000000]`).

3.1.1 Arquitectura

El diseño de la arquitectura de este test parte de la proporcionada por Alphadata en el ejemplo 3 del *Development Kit* (utilizada en los test 0, 1 y 2) [5].

De este diseño, se elimina todo lo referente al *Direct Slave*: el *AXI 4 Crossbar*, 3 de las interfaces *AXI4 to bRAM* y la bRAM1 (tal y como se puede ver en la figura 3.1), quedando los siguientes elementos:

- La interfaz ADM-XRC-KU1-HSAXI diseñada por Alpha Data, aprovechando únicamente el *DMA Engine* 0.
- Una bRAM de 512kB en la que se escriben y de la que se leen únicamente la primera y última dirección.
- Una interfaz que conecta el bus AXI del *DMA Engine* 0 con la bRAM0.
- La interfaz diseñada para este test, que espera a que el host escriba el *ack* en la primera dirección (0x0000) de la bRAM y a continuación escribe en la última dirección (0x7FFF) el mismo *ack*.

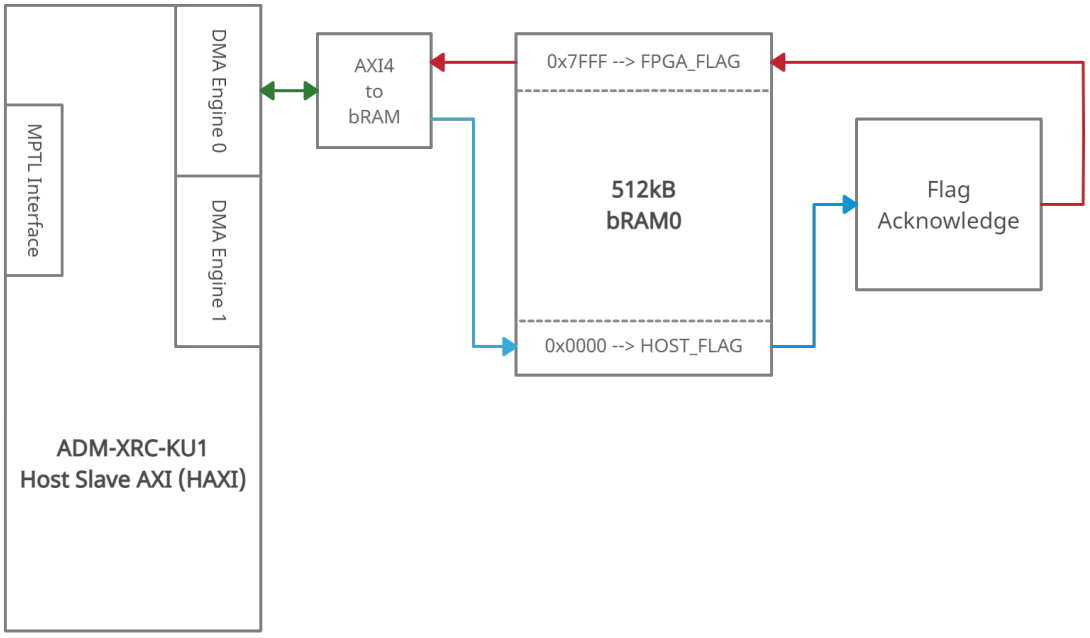

Figura 3.1 Arquitectura Test 7.

3.1.2 Host Program

El *Host Program* (*software* que se ejecuta en el PC al que está conectada la tarjeta, emplea varias funciones de la API que proporciona Alphadata [4], y realiza los siguientes pasos:

1. Abre el dispositivo utilizando la función ADMXRC3_Open(), programa la FPGA con la función ADMXRC3_ConfigureFromFile(), mapea la bRAM en la memoria del host con la función ADMXRC3_MapWindow() y bloquea el buffer para las transferencias a través del *DMA Engine* con la función ADMXRC3_Lock().
2. Utiliza la función ADMXRC3_WriteDMALockedEx() para escribir la dirección 0x0000 de la bRAM0 el *ack* (`[0xffffffffffffffffffffffffffffffff]`).
3. A continuación, lee el dato almacenado en la última dirección de la bRAM cada 1 segundo, hasta que obtiene el *ack* de la FPGA.

4. Por último, desbloquea el buffer del *DMA Engine* con la función ADMXRC3_Unlock(), desmapea la bRAM de la memoria del host y cierra el dispositivo mediante la función ADMXRC3_Close().

3.1.3 Resultados y conclusiones

Los resultados que devuelve este test son los siguientes:

Código 3.1 Resultados Test 7.

```
Target configured
ack:      [ffffffffffffffffffffffffffffffff]
buffer:   [ffffffffffffffffffffffffffffffff]
Writing to fpga
Reading fpga (Reset buffer first)
buffer:   [00000000000000000000000000000000]
buffer:   [ffffffffffffffffffffffffffffffff]
Test succeeded
```

La conclusión que se obtiene es que el protocolo de comunicación investigado (basado en *flags*) para avisar de que los datos están listos funciona acorde a lo esperado. Esto permite continuar trabajando en un sistema de comunicaciones entre el *host* y el sistema de inyección de fallos que siga en esa misma línea.

3.2 Transmisión de datos desde bRAM a *ftu_stream*: Test 8

Una vez se ha comprobado que el sistema de *flags* (con *acks*) funciona correctamente, se comienza a trabajar en una interfaz que sea capaz de transmitir los datos almacenados en una bRAM (enviados por *host*) hasta un *ftu_stream*.

Para poder realizar esto, se modifica el sistema de *flags* utilizado en el test anterior (sección 3.1), de forma que ahora no solo contiene el *ack*, sino también el tamaño (en bytes) de los datos. Concretamente, se utiliza el byte más significativo como *ack* y el resto de bytes indicando el tamaño de los datos escritos en la memoria bRAM listos para leer (en bytes), tal y como se puede ver en la figura 3.2. Las direcciones de las *flag*s no cambian en este test: 0x000 para la *flag* del *host* y 0x7FFF para la *flag* de la FPGA.

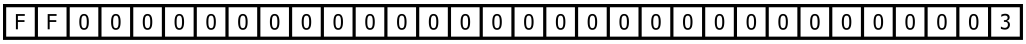

Figura 3.2 Flag.

En cuanto al orden de los datos, el *host* los escribe a partir de la dirección 0x0001 (incluida), colocando el primer dato (byte) en los bits menos significativos, tal y como se puede ver en la imagen 3.3.

3.2.1 Arquitectura

La arquitectura de este test parte de la utilizada en el anterior (3.1: Test 7).

Se sustituye la interfaz encargada de manejar los *flags* por una un poco más avanzada, que permite gestionar tanto los *acks* como el número de datos a transmitir. Además, este bloque es capaz de escribir en un *ftu_stream* los datos enviados por el *host*, tal y como se puede ver en la figura 3.4.

De esta forma, la arquitectura de este test contiene lo siguiente:

- La interfaz ADM-XRC-KU1-HSAXI, aprovechando únicamente el DMA Engine 0.
- Una bRAM de 512kB (bRAM0) en la que escriben y leen tanto el *host* como la FPGA.
- Una interfaz que conecta el bus AXI del DMA Engine 0 con la bRAM0.
- La interfaz diseñada para este test, que espera a que el host escriba la *flag* con el *ack* y el número de datos en la dirección 0x0000 de la bRAM, lee los datos correspondientes, los escribe en el *ftu_stream* y por último escribe en la última dirección el *ack*.

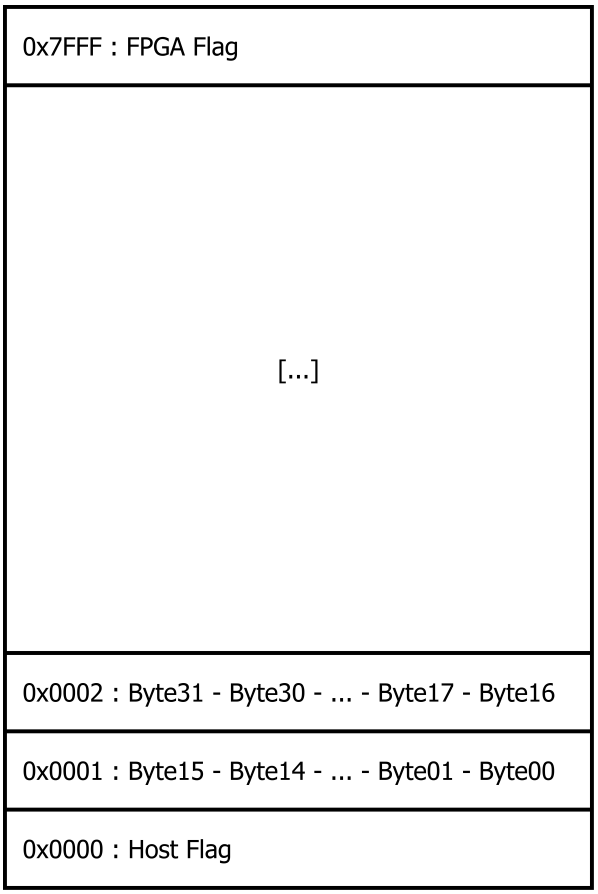

Figura 3.3 Posición de los datos en la bRAM.

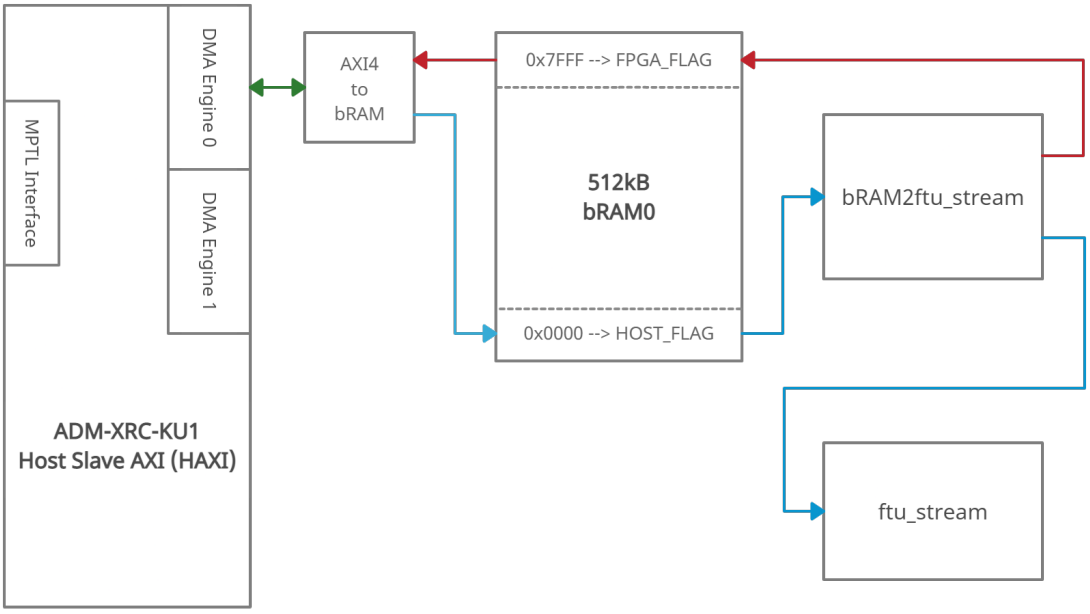

Figura 3.4 Arquitectura Test 8.

FSM: bram2ftu_stream

Para enviar los datos almacenados en la bRAM a un *ftu_stream* se diseña una máquina de estados cuyo diagrama de bloques simplificado se puede ver en la figura 3.5.

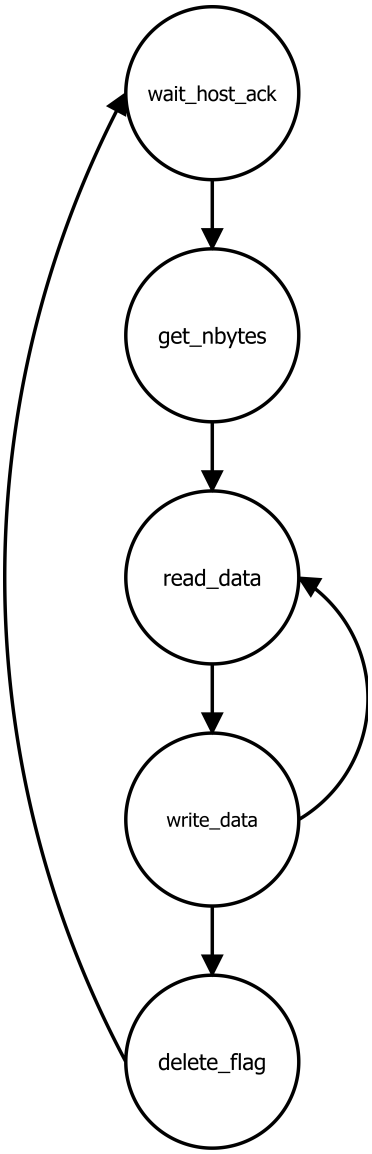

Figura 3.5 FSM: bram2ftu_stream.

Esta máquina de estados hace lo siguiente:

1. Espera a que el *host* termine de escribir los datos y la *flag* correspondientes.
2. Lee de la dirección 0x0000 los el número de datos a procesar.
3. Empieza a leer los datos de la bRAM (empezando en la dirección 0x0001) y los va escribiendo en el *ftu_stream*.
4. Cuando no quedan datos que leer, borra la flag del *host*, escribe la suya propia en la dirección 0x7FFF y vuelve al punto inicial.

3.2.2 Host Program

El *host program* de este test es, de nuevo, una versión más avanzada del anterior, ya que no solo envía el *ack* en la *flag*, sino también el tamaño. Además, llena de unos (`[0xFF]`) los bits de la dirección 0x0001,

simulando la escritura de datos, antes de enviar la *flag*.
A continuación se enumera todo lo que realiza el *host*:

1. Abre el dispositivo, mapea la bRAM en la memoria del host y bloquea el buffer para las transferencias a través del *DMA Engine*.
2. Escribe en la dirección 0x0001 de la bRAM0 (a través del DMA Engine 0) los datos.
3. Escribe en la dirección 0x0000 de la bRAM0 el *ack* y el número de datos enviados (16 bytes en este test).
4. A continuación, lee el dato almacenado en la última dirección de la bRAM cada 1 segundo, hasta que obtiene el *ack* de la FPGA.
5. Por último, desbloquea el buffer del *DMA Engine*, desmapea la bRAM de la memoria del host y cierra el dispositivo.

3.2.3 Resultados y conclusiones

Los resultados que devuelve este test son los siguientes:

Código 3.2 Resultados Test 8.

```
Target configured
data buffer: [ffffffffffffffffffffffffffffffff]
Writing to fpga
flags buffer: [ff00000000000000000000000000001]
Reading fpga (Reset buffer first)
flags buffer: [00000000000000000000000000000000]
flags buffer: [ff00000000000000000000000000000]
Test succeeded
```

De este test se concluye que el sistema de comunicaciones entre la bRAM y el *ftu_stream* funciona correctamente y acorde a lo esperado. Lo único que no se comprueba es si los datos han sido correctamente escritos en el *ftu_stream*, por lo que se simula y se observa que la escritura funciona correctamente.

3.3 Interfaz de comunicaciones entre bRAM y *ftu_stream*: Test 9

Una vez se tiene una interfaz capaz de transferir datos desde el *host* (que escribe en una bRAM mediante un *DMA Engine*) hasta la interfaz de entrada de un sistema de inyección de fallos (*ftu_stream*), se procede a diseñar el sistema de comunicaciones completo.

El objetivo de este test es, a partir del trabajo realizado en los tests 7 y 8, completar el sistema de comunicaciones de forma que se pueda enviar información (comandos y datos) desde el *host* hasta la FPGA y viceversa.
Para ello, se diseña una nueva interfaz que permita enviar datos desde un *ftu_stream* a una bRAM y se integra dentro de la arquitectura del test anterior.

3.3.1 Arquitectura

Para poder probar a fondo el sistema de comunicaciones, se modifica la arquitectura del test anterior (test 7: 3.1), quedando tal y como se puede ver en la figura 3.6. De esta forma, la arquitectura contiene los siguientes elementos:

- La interfaz ADM-XRC-KU1-HSAXI, aprovechando tanto *DMA Engine* 0 como el *DMA Engine* 1.
- Una bRAM de entrada de 512kB (bRAM0) en la que escribe los datos el *host* para la FPGA.
- Una bRAM de salida de 512kB (bRAM1) en la que escribe los datos la FPGA para el *host*.
- Dos interfaces que conectan el bus AXI del los *DMA Engines* con las bRAMs.

- Dos *ftu_streams*, uno de entrada (ftu_stream_in) y otro de salida (ftu_stream_out) del sistema de inyección de fallos.
- Una interfaz que se encarga de coger los datos escritos por el *host* de la bRAM0 y enviarlos al *ftu_stream_in*.
- Una interfaz que coge los datos del *ftu_stream* de entrada y los envía (copia) al *ftu_stream* de salida, diseñada en el Test 7 (sección 3.1).
- La interfaz diseñada para este test, que espera a que el *ftu_stream* de salida tenga datos, los va escribiendo en la bRAM1 y cuando acaba, escribe el *flag* correspondiente en la bRAM1, indicando el número de bytes transferidos y el *ack*.

Cabe destacar que al utilizar dos bRAMs, una de entrada y otra de salida, se modifica un poco el protocolo desarrollado anteriormente para avisar de que los datos están listos. Concretamente, se decide únicamente usar la dirección 0x0000 de ambas memorias para avisar de que los datos están listos para ser leídos.

De esta forma, si el *host* quiere enviar datos a la FPGA, los escribe en la bRAM0 y escribe la *flag* correspondiente. A continuación, la FPGA los lee y, cuando acaba, borra la *flag* que había escrito el *host*, avisándole que puede iniciar una nueva transacción cuando quiera. El envío de datos en el otro sentido funciona exactamente igual.

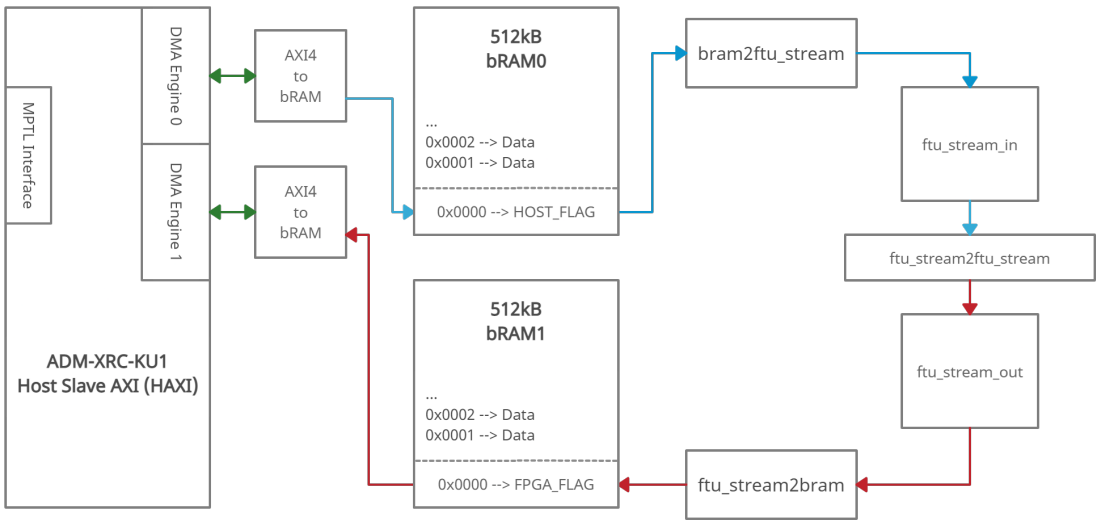

Figura 3.6 Arquitectura Test 9.

FSM: ftu_stream2bram

Para enviar los datos en el *ftu_stream* de salida se diseña una máquina de estados cuyo diagrama de bloques simplificado se puede ver en la figura 3.7.

Esta máquina de estados hace lo siguiente:

1. Espera a que el *ftu_stream_out* tenga datos listos para procesar.
2. Le pide datos al *ftu_stream_out* y los escribe en la bRAM1 hasta que ya no queda espacio en la misma, o el *ftu_stream* está vacío durante varios ciclos.
3. Escribe la *flag* correspondiente a la transmisión efectuada (número de datos y *ack*) .
4. Espera a que el *host* borre la *flag* y vuelve al punto inicial.

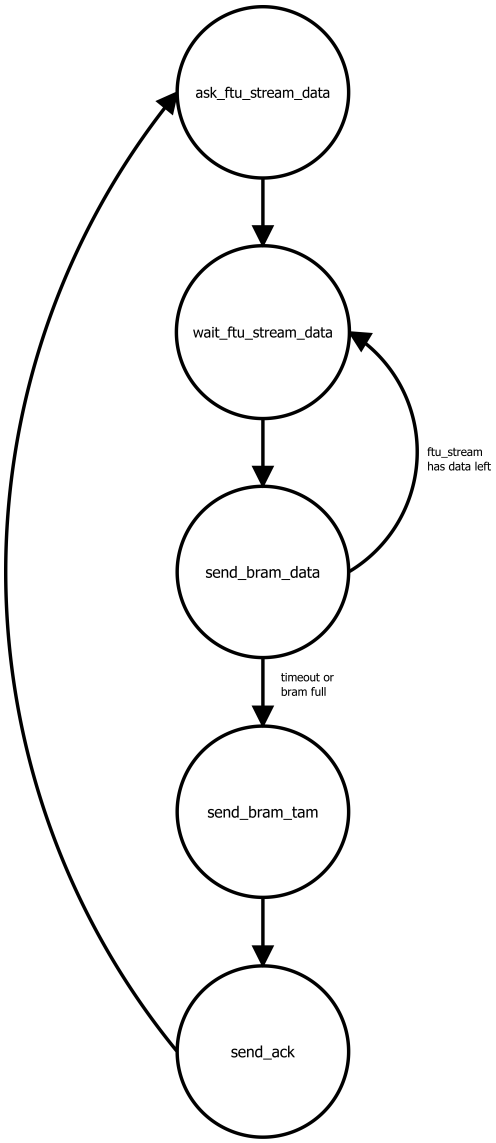

Figura 3.7 FSM: ftu_stream2bram.

3.3.2 Host Program

Para este test, se aprovecha el diseñado en el anterior (descrito en la sección 3.2.2, modificándolo para que realice lo siguiente:

1. Abre el dispositivo, mapea las bRAMs en la memoria del host y bloquea el buffer para las transferencias a través de los *DMA Engines*.
2. Envía un patrón pseudoaleatorio (*data*) a la bRAM0 a través del *DMA Engine* 0 (empezando en la dirección 0x0001).
3. Escribe en la dirección 0x0000 de la bRAM0 el *ack* y el número de datos enviados (12 bytes en este test).
4. A continuación, lee el dato almacenado en la última dirección de la bRAM1 cada 1 segundo, hasta que obtiene el *ack* de la FPGA.
5. Lee los datos escritos por la FPGA en la bRAM1 (empezando en la dirección 0x0001) y comprueba que son iguales a los datos originales (escritos en la bRAM0 por el *host*) al comienzo del test.
6. Por último, desbloquea el buffer del *DMA Engine*, desmapea la bRAM de la memoria del host y cierra el dispositivo.

3.3.3 Resultados y conclusiones

Los resultados que devuelve este test son los siguientes:

Código 3.3 Resultados Test 9.

```
Number of bytes to send: 12
data:        [020009010203040506070809]
Data reversed
data_buffer: [090807060504030201090002]
Data prepared to be inserted to ftu_stream_in (8 byte reversed)
data_buffer: [060708090200090102030405]
------------------------------------------------------------
Target configured
------------------------------------------------------------
Writing to bram0 via dma0
flag_buffer: [ff000000000000000000000000000000c]
------------------------------------------------------------
Reading bram1 via dma1 (reset buffers first)
Buffers reset
flag_buffer: [00000000000000000000000000000000]
data_buffer: [000000000000000000000000]
FPGA ACK Received
flag_buffer: [ff000000000000000000000000000000c]
Data received
data_buffer: [090807060504030201090002]
------------------------------------------------------------
Data matched: Test SUCCEEDED
```

Observando los resultados obtenidos, se concluye que se ha cumplido con el primer objetivo de este proyecto, ya que se diseñado un sistema de comunicaciones que permite enviar y recibir datos desde el *host* hasta las interfaces de entrada y salida de una plataforma de inyección de fallos (*ftu_stream*).

4 Mejoras Software

Neither ever, nor never, goodbye.

Apparat, Goodbye

Una vez se ha demostrado que las interfaces de comunicaciones diseñadas funcionan correctamente, se decide invertir tiempo en mejorar la infraestructura del *host program* (programa que se ejecuta para enviar y recibir datos a través del sistema de comunicaciones) con el objetivo de enfocarlo a una futura plataforma de inyección de fallos. Es por ello que se decide trabajar en dos líneas distintas.

Por un lado, se crea un *script* de *python* que permite controlar todos los parámetros del sistema, ya que prepara los datos a enviar, configura la profundidad del *logger* y ejecuta el test con los parámetros correspondientes.

Por otro, se mejora el propio programa en C++ (utilizado para los tests 7, 8 y 9), implementando un *logger* y un sistema de entrada y salida de información (datos y comandos) a enviar y recibir a la FPGA.

4.1 Implementación del sistema de control con *python*

4.1.1 Sistema de control mediante argumentos

Como se comentaba en el párrafo anterior, lo primero en lo que se trabaja es en implementar un *script* de *python* que permita controlar todos los parámetros necesarios para efectuar los tests. Para ello, se utiliza la librería *getopt*[8], incluida en la biblioteca estándar de *python*.

De esta forma, el *script* acepta los siguiente argumentos:

- `"-h"`: Imprime por el terminal los diferentes argumentos posibles y cómo utilizarlos.
- `"-t"`: Selecciona el número concreto de test a realizar.
- `"-l"`: Selecciona el nivel del *logger* (*debug*, *info*, *warning*, *error* o *critical*).

A continuación se muestra el resultado de ejecutar el *script* con el comando -h (código 4.1).

Código 4.1 Python arguments example.

```
src git:(fit) python3 main.py -h
main.py -l <level> -t <test>
```

4.1.2 Preparación de instrucciones para el programa en C++

Se implementa una función que se encarga de coger las instrucciones almacenadas en el fichero *inst* (escritas por la persona que ejecuta el test), analizarlas y prepararlas para que el código en C++ sea capaz de realizar la transmisión sin problema.

Esta preparación de los datos consiste en la alineación y separación en líneas con los datos que caben en cada dirección de la bRAM de entrada. Una vez preparadas, la función las escribe en el fichero *data_in*, que leerá el programa en C++.

Es importante destacar que estas instrucciones son comandos para la FPGA, ya que es el firmware de la FPGA el debe responder a los comandos enviados desde el *host*.

4.1.3 Logger de python

En paralelo, se añade un sistema de *logging* que se encarga de guardar en un *log* todos los resultados impresos por el terminal. Para su desarrollo se emplea la librería *logging* [9], incluida en a biblioteca estándar de *python*.

El nombre del fichero de *log* contiene la fecha en la que se ha llevado a cabo el test y el formato del mismo se puede ver en el código 4.2

Código 4.2 Python logger example: 2022-07-04.log.

```
src git:(fit) sudo python3 main.py -t test -l DEBUG
2022-07-04 15:40:54: INFO        - ------------------------------------------------------
2022-07-04 15:40:54: INFO        - | STARTING NEW TEST: 2022-07-04 15:40:54           |
2022-07-04 15:40:54: INFO        - ------------------------------------------------------
2022-07-04 15:40:54: INFO        - Logger created and configured
2022-07-04 15:40:54: INFO        - ------------------------------------------------------
2022-07-04 15:40:54: INFO        - Instructions successfully prepared -> ./data_in
2022-07-04 15:40:54: INFO        - ------------------------------------------------------
[...]
C++ logger outputs info about the test here
[...]
2022-07-04 15:40:55: INFO        - ------------------------------
2022-07-04 15:40:55: INFO        - -- END OF FIRST TRANSACTION --
2022-07-04 15:40:55: INFO        - ------------------------------
```

4.1.4 Ejecución del test

Por último, el *script* de *python* ejecuta el test correspondiente y espera pacientemente a que el programa en C++ termine. Una vez ha terminado, procede a dar por finalizado el test por completo (código 4.2).

4.2 Mejora del *Host Program*

4.2.1 Gestión de datos entrada y salida

Lo primero que se implementa en el *host program* es un sistema que se encarga de la gestión de datos de entrada y salida del programa. Este funciona de la siguiente forma:

1. Coge las instrucciones preparadas por el *script* de *python*, contenidas en el fichero *data_in*.
2. Las divide en trozos del tamaño del máximo número de datos posible que caben en la bRAM de entrada.
3. Envía el el primer grupo de instrucciones a la FPGA.
4. Recibe los datos de la FPGA y los escribe en un fichero de salida *data_out*.
5. Repite el proceso hasta que no queden datos que enviar, recibiendo los datos correspondientes después de cada envío.

4.2.2 Logger de C++

Por último, se implementa un *logger* que trabaja simultáneamente con el de *python*. El formato de la información que este maneja es exactamente igual que el descrito en el código 4.2. A continuación se puede observar el resultado final (código 4.3).

Código 4.3 Both loggers example: 2022-07-04.log.

```
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - | STARTING NEW TEST: 2022-07-04 15:40:54          |
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - Logger created and configured
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - Instructions successfully prepared -> ./data_in
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - STARTING 1º TRANSACTION
2022-07-04 15:40:54: INFO       - Number of bytes to send: 2
2022-07-04 15:40:54: DEBUG      - Data reversed
2022-07-04 15:40:54: DEBUG      - data_buffer_in:   [1513]
2022-07-04 15:40:54: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-04 15:40:54: DEBUG      - data_buffer_in:   [1315]
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - Writing to bram0 via dma0
2022-07-04 15:40:54: DEBUG      - flag_buffer:      [ff00000000000000000000000000002]
2022-07-04 15:40:54: INFO       - ------------------------------------------------
2022-07-04 15:40:54: INFO       - Reading bram1 via dma1 (reset buffers first)
2022-07-04 15:40:54: DEBUG      - Buffers reset
2022-07-04 15:40:54: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-04 15:40:54: DEBUG      - data_buffer_in:   [0000]
2022-07-04 15:40:55: INFO       - FPGA ACK Received
2022-07-04 15:40:55: DEBUG      - flag_buffer:      [ff00000000000000000000000000001]
2022-07-04 15:40:55: INFO       - Number of bytes to read: 1
2022-07-04 15:40:55: DEBUG      - Data received
2022-07-04 15:40:55: DEBUG      - data_buffer_out:  [01]
2022-07-04 15:40:55: INFO       - Data recieved written -> ./data_out
2022-07-04 15:40:55: INFO       - Flag cleared
2022-07-04 15:40:55: DEBUG      - Testing if flag was cleared
2022-07-04 15:40:55: DEBUG      - flag_buffer:      [01010101010101010101010101010101]
2022-07-04 15:40:55: DEBUG      - Reading bram1
2022-07-04 15:40:55: DEBUG      - flag_buffer:      [ff00000000000000000000000000001]
2022-07-04 15:40:55: INFO       - ------------------------------
2022-07-04 15:40:55: INFO       - -- END OF FIRST TRANSACTION --
2022-07-04 15:40:55: INFO       - ------------------------------
```

4.3 Conclusiones

Gracias a las mejoras implementadas en este capítulo, se ha obtenido una mejor infraestructura con la que realizar tests fácilmente, cumpliendo, por tanto, con el objetivo del mismo.
Esto es fundamental de cara a al futuro desarrollo de la plataforma de inyección de fallos completa, ya que se ha facilitado el envío de instrucciones desde el *host* hasta la FPGA a través del sistema de comunicaciones.

5 Intérprete de comandos en la FPGA

Me voy a subir al palo.

MANUEL MORENO BORRÁS

Una vez se tiene un sistema de comunicaciones plenamente funcional y preparado para un sistema de inyección de fallos, se procede a implementar un intérprete de comandos y datos en la arquitectura de la FPGA.

Esta interfaz es de vital importancia para la viabilidad del proyecto, ya que se trata del cerebro del sistema de inyección de fallos. Ha de ser capaz de entender y ejecutar todos los comandos necesarios para realizar la inyección de fallos, así como de depurar cualquier error en el sistema.

Debido a esto, se parte del código utilizado en FTU-VEGAS [7], proporcionado por el tutor del TFM (Hipólito Guzmán Miranda). Este se modifica casi al completo, adaptándolo a la arquitectura desarrollada en el capítulo 3.

5.1 Comandos

Una de las decisiones más importantes en este punto del trabajo es elegir qué comandos ha de ser capaz de manejar la interfaz, ya que el objetivo del mismo es desarrollar las herramientas necesarias para inyectar fallos en la tarjeta, no el sistema completo. Por tanto, teniendo como referencia los comandos del *instruction set* del FTU-VEGAS [7], se decide comenzar implementando tres comandos sencillos: *echo*, *bit_up* y *bit_get*.

El primero de ellos se utiliza para testear que la interfaz funciona correctamente y es capaz de leer, procesar y manejar todo tipo de datos.

El segundo y el tercero son inversiones a futuro, ya que controlarán un contador, el cual se utilizará más adelante para probar que la reconfiguración parcial funciona correctamente.

5.1.1 Comando *echo*

Este comando consiste en enviar una serie de datos que el sistema devuelve sin modificar. Para ello se emplea la siguiente estructura:

- **Identificador del comando**: i8 code → 0x02
- **Argumentos**: i32 size, i8[size] data → Indican el tamaño dato y el dato en sí.

Código 5.1 Ejemplo comando *echo*.

```
Comando enviado:  02 00000004 01020304
Dato recibido:    00000004 01020304
```

5.1.2 Comandos *bit_up* y *bit_get*

Como se menciona al comienzo de esta sección, estos comandos son muy importantes de cara a probar que la reconfiguración parcial funciona (tema desarrollado en el capítulo 7), por lo que se implementan junto a un contador de 8 bits.

La estructura de los mismos es muy sencilla, ya que únicamente constan de un byte que indica el identificador del comando, tal y como se puede ver en la tabla 5.1.
En cuanto a la funcionalidad de estos comandos, el primero activa el la señal *enable* del contador, haciendo que este incremente su cuenta en uno.

Aquí es importante destacar que normalmente (por ejemplo, en el FTU-VEGAS) este comando reconfiguraría un bit, de forma que pasara de valer 0 a valer 1. Sin embargo, al no estar implementada la reconfiguración parcial (todavía) y al tratarse de un prototipo, se decide que simplemente incremente el valor del contador. Es por ello que, realmente, se parece más al comando *step* del FTU-VEGAS, que hace que el diseño avance un ciclo de reloj.

El segundo obtiene el valor de la cuenta almacenada en el contador y la escribe en el *ftu_stream* de salida, de forma que se envía de vuelta al *host*.

Tabla 5.1 Comandos Implementados.

Comando	Código	Argumentos	Devuelve	Descripción
echo	0x02	i32 size, i8[size] data	i32 size, i8[size] data	Devuelve a través del ftu_stream los datos recibidos
bit_up	0x13	-	-	Activa la señal "enable" del contador, haciendo que incremente su cuenta
bit_get	0x15	-	-	Devuelve a través del ftu_stream_out el valor almacenado en el contador

5.2 Intérprete de comandos: Test 10

Para comprobar que el sistema desarrollado funciona, se diseña un test en el que se envían las instrucciones necesarias para probar los distintos comandos. Para ello, se divide el Test 10 en dos partes diferentes, las cuales comparten la misma arquitectura de FPGA y *host program*, pero cada parte envía instrucciones diferentes.

En la primera (Test 10a) se envían varios comandos *echo* seguidos y se comprueba que la FPGA devuelve los mismos datos, a través de la interfaz de comunicaciones (capítulo 3).

En la segunda (Test 10b) se envían varios *bit_up* seguidos de un *bit_get*. De esta forma se puede comprobar, por un lado, que el contador cuenta correctamente y, por otro, que ambos comandos funcionan bien.

5.2.1 Arquitectura

La arquitectura utilizada en este test parte de la utilizada en el Test 9 (sección 3.3.1). Lo único que se sustituye es la interfaz que copia los datos del *ftu_stream_* al *ftu_stream_out* por el intérprete de comandos desarrollado en este capítulo. De esta forma, los elementos de esta arquitectura son los siguientes:

- La interfaz ADM-XRC-KU1-HSAXI, aprovechando tanto *DMA Engine* 0 como el *DMA Engine* 1.
- Una bRAM de entrada de 512kB (bRAM0) en la que escribe los datos el *host* para la FPGA.
- Una bRAM de salida de 512kB (bRAM1) en la que escribe los datos la FPGA para el *host*.
- Dos interfaces que conectan el bus AXI del los *DMA Engines* con las bRAMs.
- Dos *ftu_streams*, uno de entrada (ftu_stream_in) y otro de salida (ftu_stream_out) del sistema de inyección de fallos.

- La interfaz que se encarga de coger los datos escritos por el *host* de la bRAM0 y enviarlos al *ftu_stream_in*.
- La interfaz que se encarga de coger los datos escritos por la *FPGA* del *ftu_stream_out* y enviarlos a la bRAM1.
- La interfaz diseñada para este test, que coge las instrucciones del *ftu_stream_in*, las interpreta, las ejecuta y escribe los resultados en el *ftu_stream_out*.

En la figura 5.1 se presenta la arquitectura simplificada utilizada para este test.

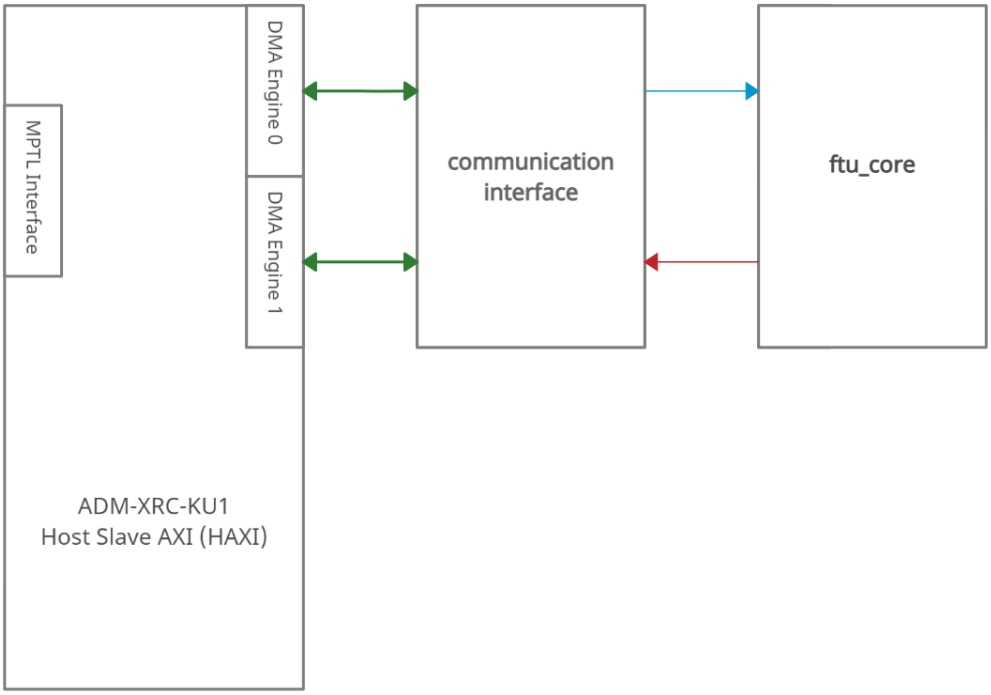

Figura 5.1 Arquitectura Test 10.

En esta imagen, las bRAMs, las interfaces que conectan los buses Advanced eXtensible Interface (AXI) de los DMA Engines a las bRMAs y las interfaces que transfieren las instrucciones de las bRAMs a los *ftu_streams* de entrada y salida se encuentran dentro del bloque *communications interface*.

Por otro lado, el contador, los *ftu_streams* de entrada y salida y la interfaz que interpreta los comandos (*core_fsm*) se encuentran dentro del bloque *ftu_core*.

Sistema de inyección de fallos: ftu_core

El bloque *ftu_core* emula un sistema de inyección de fallos y, aunque se trate solo de un prototipo, contiene la infraestructura mínima necesaria. A continuación se listan los elementos que lo componen:

- Las interfaces de entrada y salida (*ftu_streams*), las cuales están conectadas a la interfaz de comunicaciones (*communication interface*).
- El intérprete de comandos (*core_fsm*), encargado de manejar todo el sistema y ejecutar las instrucciones correspondientes.
- El contador implementado para comprobar el funcionamiento del intérprete de comandos y la reconfiguración parcial (capítulo 7).

Todo esto se puede ver representado en la imagen 5.2.

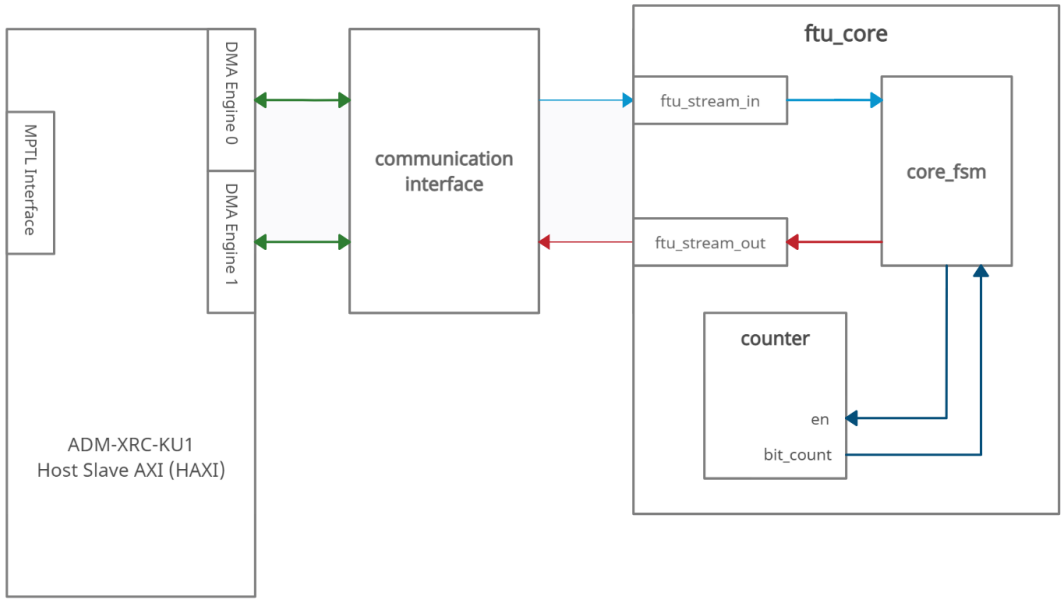

Figura 5.2 Arquitectura Test 10.

Intérprete de comandos: core_fsm

Se trata de la interfaz desarrollada para este test y se encuentra dentro del bloque ftu_core, tal y como se puede ver en la imagen 5.2.

Lo más importante de esta es la máquina de estados que controla todo el sistema. Concretamente, pasa por los siguientes estados:

1. ***idle***: Espera que haya instrucciones disponibles en el *ftu_stream* de entrada.
2. ***parse_cmd***: Interpreta el comando a ejecutar y salta al estado correspondiente para ejecutarlo.
3. ***echo***: Pide 4 bytes (tamaño de los datos a devolver) al *ftu_stream* de entrada.
4. ***echo_1***: Escribe los datos recibidos (tamaño en la primera iteración, datos en las siguientes) en el *ftu_stream* de salida.
5. ***echo_2***: Comprueba si quedan datos sin leer y, en ese caso, se los pide al *ftu_stream* de entrada. Si no quedan datos por leer, vuelve a *idle*.
6. ***bit_up***: Activa la seña *enable* del contador, haciendo que este incremente el valor de la cuenta. A continuación, vuelve a *idle*.
7. ***bit_get***: Devuelve a través del *ftu_stream* de salida el valor de la cuenta almacenada en el contador y vuelve a *idle*.

5.2.2 Host Program

Para este test, se mantiene el *host program* desarrollado en el capítulo 4, ya que permite el envío de instrucciones simplemente escribiéndolas en el fichero *inst* y ejecutando el test mediante el *script* de *python*.

Para el Test 10a, este fichero contiene las instrucciones de tres comandos *echo*, tal y como se puede ver en el código 5.4.

Código 5.2 Test 10a: Instruciones enviadas a la FPGA.

```
02 00000009 010203040506070809
02 0000000a 12131415161718191a1b
02 0000000d 232425262728292a2b2c2d2e2f
```

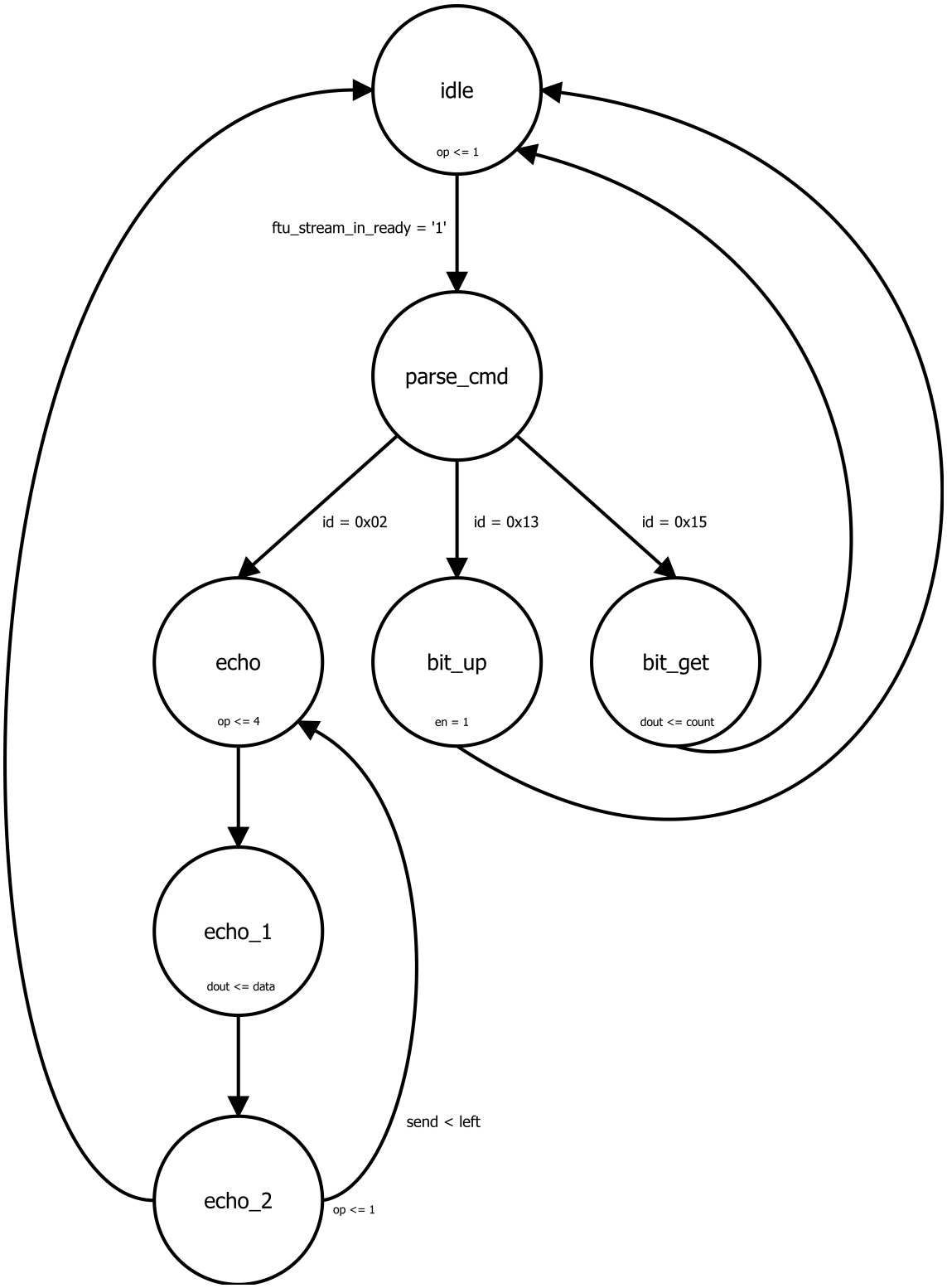

Figura 5.3 Arquitectura Test 10.

Para el Test 10b, se añaden tres comandos *bit_up* y un *bit_get* (ver código 5.6).

Código 5.3 Test 10b: Instruciones enviadas a la FPGA.

```
13 13 13 15
```

5.2.3 Resultados y conclusiones

El Test 10a devuelve lo siguiente:

Código 5.4 Test 10a.

```
2022-07-05 20:24:35: INFO       - ------------------------------------------------
2022-07-05 20:24:35: INFO       - | STARTING NEW TEST: 2022-07-05 20:24:35          |
2022-07-05 20:24:35: INFO       - ------------------------------------------------
2022-07-05 20:24:35: INFO       - Logger created and configured
2022-07-05 20:24:35: INFO       - ------------------------------------------------
2022-07-05 20:24:35: INFO       - Instructions successfully prepared -> ./data_in
2022-07-05 20:24:35: INFO       - ------------------------------------------------
2022-07-05 20:24:35: INFO       - Starting FPGA Test
2022-07-05 20:24:37: INFO       - ------------------------------------------------
2022-07-05 20:24:37: INFO       - Target configured
2022-07-05 20:24:37: INFO       - ------------------------------------------------
2022-07-05 20:24:37: INFO       - STARTING 1º TRANSACTION
2022-07-05 20:24:37: INFO       - Number of bytes to send: 47
2022-07-05 20:24:37: DEBUG      - Data reversed
2022-07-05 20:24:37: DEBUG      - data_buffer_in:   [0000020b0a19181716151413120a0000]
2022-07-05 20:24:37: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-05 20:24:37: DEBUG      - data_buffer_in:   [1718190a0b02000000000a1213141516]
2022-07-05 20:24:37: INFO       - ------------------------------------------------
2022-07-05 20:24:37: INFO       - Writing to bram0 via dma0
2022-07-05 20:24:37: DEBUG      - flag_buffer:      [ff000000000000000000000000002f]
2022-07-05 20:24:37: INFO       - ------------------------------------------------
2022-07-05 20:24:37: INFO       - Reading bram1 via dma1 (reset buffers first)
2022-07-05 20:24:37: DEBUG      - Buffers reset
2022-07-05 20:24:37: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-05 20:24:37: DEBUG      - data_buffer_in:   [00000000000000000000000000000000]
2022-07-05 20:24:38: INFO       - FPGA ACK Received
2022-07-05 20:24:38: DEBUG      - flag_buffer:      [ff000000000000000000000000002c]
2022-07-05 20:24:38: INFO       - Number of bytes to read: 44
2022-07-05 20:24:38: DEBUG      - Data received
2022-07-05 20:24:38: DEBUG      - data_buffer_out:  [00000009010203040506070809000000]
2022-07-05 20:24:38: INFO       - Data recieved written -> ./data_out
2022-07-05 20:24:38: INFO       - Flag cleared
2022-07-05 20:24:38: DEBUG      - Testing if flag was cleared
2022-07-05 20:24:38: DEBUG      - flag_buffer:      [01010101010101010101010101010101]
2022-07-05 20:24:38: DEBUG      - Reading bram1
2022-07-05 20:24:38: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-05 20:20:39: INFO       - ------------------------------------------------
2022-07-05 20:20:39: INFO       - Test Finished
```

Para comprobar que el sistema devuelve los datos de entrada, es decir, que ha ejecutado bien el comando *echo* es necesario ver el contenido de *data_out*, ya que el *host program* solo imprime 32 caracteres como máximo.

Código 5.5 Test 10a: fichero data_out.

```
00000009010203040506070809000000
    a12131415161718191a1b0000000d232425262728292a2b2c2d2e2f
```

Como se puede observar en la tabla 5.2, se han obtenido los datos esperados, ya que estos coinciden con los enviados a la FPGA al principio del test.

Tabla 5.2 Test 10a: Datos a la salida.

Tamaño	Dato
00000009	010203040506070809
0000000a	12131415161718191a1b
0000000d	232425262728292a2b2c2d2e2f

De esta forma se concluye que el sistema funciona correctamente.

El resultado del Test 10b se puede ver a continuación:

Código 5.6 Test 10b.

```
2022-07-05 20:20:20: INFO       - ------------------------------------------------------
2022-07-05 20:20:20: INFO       - | STARTING NEW TEST: 2022-07-05 20:20:20           |
2022-07-05 20:20:20: INFO       - ------------------------------------------------------
2022-07-05 20:20:20: INFO       - Logger created and configured
2022-07-05 20:20:20: INFO       - ------------------------------------------------------
2022-07-05 20:20:20: INFO       - Instructions successfully prepared -> ./data_in
2022-07-05 20:20:20: INFO       - ------------------------------------------------------
2022-07-05 20:20:20: INFO       - Starting FPGA Test
2022-07-05 20:20:22: INFO       - ------------------------------------------------------
2022-07-05 20:20:22: INFO       - Target configured
2022-07-05 20:20:22: INFO       - ------------------------------------------------------
2022-07-05 20:20:22: INFO       - STARTING 1º TRANSACTION
2022-07-05 20:20:22: INFO       - Number of bytes to send: 4
2022-07-05 20:20:22: DEBUG      - Data reversed
2022-07-05 20:20:22: DEBUG      - data_buffer_in:   [15131313]
2022-07-05 20:20:22: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-05 20:20:22: DEBUG      - data_buffer_in:   [13131315]
2022-07-05 20:20:22: INFO       - ------------------------------------------------------
2022-07-05 20:20:22: INFO       - Writing to bram0 via dma0
2022-07-05 20:20:22: DEBUG      - flag_buffer:      [ff000000000000000000000000000004]
2022-07-05 20:20:22: INFO       - ------------------------------------------------------
2022-07-05 20:20:22: INFO       - Reading bram1 via dma1 (reset buffers first)
2022-07-05 20:20:22: DEBUG      - Buffers reset
2022-07-05 20:20:22: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-05 20:20:22: DEBUG      - data_buffer_in:   [00000000]
2022-07-05 20:20:23: INFO       - FPGA ACK Received
2022-07-05 20:20:23: DEBUG      - flag_buffer:      [ff000000000000000000000000000001]
2022-07-05 20:20:23: INFO       - Number of bytes to read: 1
2022-07-05 20:20:23: DEBUG      - Data received
2022-07-05 20:20:23: DEBUG      - data_buffer_out:  [03]
2022-07-05 20:20:23: INFO       - Data recieved written -> ./data_out
2022-07-05 20:20:23: INFO       - Flag cleared
2022-07-05 20:20:23: DEBUG      - Testing if flag was cleared
2022-07-05 20:20:23: DEBUG      - flag_buffer:      [01010101010101010101010101010101]
2022-07-05 20:20:23: DEBUG      - Reading bram1
2022-07-05 20:20:23: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-05 20:20:39: INFO       - ------------------------------------------------------
2022-07-05 20:20:39: INFO       - Test Finished
```

La conclusión tras este test es doble, ya que se ha comprobado que el contador de 8 bits funciona correctamente y que las instrucciones enviadas para controlarlo han sido procesadas y ejecutadas sin ningún problema.

6 Reconfiguración Parcial

We remember the good times and the bad ones, forgetting that most times are neither good nor bad. They just are.

Brandon Sanderson, The Way of Kings

La finalidad de este capítulo es obtener un *bitstream* funcional que permita reconfigurar parcialmente la FPGA.

Para ello, se cuenta con la guía de reconfiguración parcial de Vivado (*Vivado Design Suite User Guide*) [16], así como el tutorial de reconfiguración parcial (*Vivado Design Suite Tutorial*) [17], ambos desarrollados por *Xilinx*. Estos documentos explican cómo generar un proyecto en Vivado que permita la reconfiguración parcial.

Se procede a estudiar los elementos necesarios y los pasos a seguir, se modifica el Test 10 (sección 5.2) de forma que se pueda reconfigurar el contador y se adapta un desensamblador de *bitstreams* (desarrollado y proporcionado por Hipólito Guzmán Miranda, tutor del TFM) de forma que procese *bitstreams* de la familia Kintex Ultrascale (KU).

6.1 Reconfiguración Parcial en Vivado

Para desarrollar un diseño que permita reconfigurar parcialmente la FPGA, es necesario realizar lo siguiente:

- Obligar a los pines del *SelectMap* a que persistan, para que la FPGA admita nuevos *bitstreams* después de ser configurada por primera vez.
- Establecer las regiones estáticas (no reconfigurables) y las regiones en las que se va a realizar la reconfiguración parcial.
- Definir las diferentes configuraciones (elementos diferentes a colocar en las regiones reconfigurables)
- Sintetizar, implementar y crear los *bitstreams* para cada una de las configuraciones establecidas.

El segundo punto de esta lista es de vital importancia para FTU-Lite (la plataforma cuya viabilidad se está estudiando), ya que al convivir el diseño bajo prueba con el firmware de la plataforma de inyección de fallos en la misma FPGA, es crítico separar bien uno del otro, con el objetivo de evitar que los fallos inyectados afecten al firmware de la plataforma, ya que sólo se quiere que afecten al diseño bajo test.

6.2 Reconfiguración Parcial: Test 11

Una vez se conocen los requisitos necesarios, se procede a modificar el Test 10, de forma que permita reconfigurar parcialmente el contador.

Con ese objetivo, se diseña otro contador de 8 bits casi igual que el original, con la única diferencia en que este comienza en `[10000000]` (`0x80 en hexadecimal`) en vez de en `[00000000]` (`0x00` en hexadecimal).

A continuación, se crean dos configuraciones, una con el contador a cero (*counter_low*) y otra con el contador a 0x80 (*config_high*).

Por último se añaden las *constraints* necesarias para obligar a los pines del *SelectMap* a que persistan.

6.2.1 Arquitectura

Para este test, se utilizan dos arquitecturas diferentes, tal y como se explicaba en el párrafo anterior. La primera es exactamente igual a la del test anterior (sección 5.2) y en la segunda solo cambia el contador, ya que se coloca el que comienza contando por 0x80.

A la primera configuración se le llamará a partir de ahora *counter_low* y a la segunda *counter_high*.

6.2.2 *Host Program*

El *Host Program* parte del desarrollado en el capítulo 4. Este se modifica de forma que se realicen dos transmisiones: la primera antes de reconfigurar parcialmente la FPGA y la segunda justo después. De esta forma, se pueden estudiar los efectos de la reconfiguración parcial.

Esta es la versión final del *host program*, por lo que a continuación se describe todo lo que este realiza (incluyendo lo descrito en los capítulos anteriores):

1. El *script* de *python* configura el *logger*, coge las instrucciones del fichero inst, las prepara para que el código en C++ pueda procesarlas y las escribe en el fichero *data_in*.
2. A continuación, ejecuta el código en C++.
3. Este coge los datos preparados de *data_in* y los divide en paquetes (del tamaño máximo admisible por el sistema de comunicaciones).
4. Abre el dispositivo, mapea las bRAMs en la memoria del host y bloquea el buffer para las transferencias a través de los *DMA Engines*.
5. Escribe, a partir de la dirección 0x0001 de la bRAM0, el primer paquete de instrucciones, empezando la primera transmisión.
6. Escribe en la dirección 0x0000 de la bRAM0 el *ack* y el número de datos enviados.
7. A continuación, lee el dato almacenado en la última dirección de la bRAM1 cada 1 segundo, hasta que obtiene el *ack* de la FPGA.
8. Lee los datos escritos por la FPGA en la bRAM1 (empezando en la dirección 0x0001) y los escribe en el fichero de salida *data_out*.
9. Vuelve al punto 5, si todavía quedan paquetes por enviar.
10. A continuación, reconfigura la FPGA con el bistream parcial con la configuración *config_low* y repite el envío de instrucciones (segunda transmisión).
11. Una vez ha acabado, desbloquea el buffer del *DMA Engine*, desmapea la bRAM de la memoria del host y cierra el dispositivo.

Por último, cabe mencionar que en este test se envían dos instrucciones *bit_up* (0x13) seguidas de una *bit_get* (0x15).

6.2.3 Resultados

A la hora de realizar este test, es muy importante comprobar que el sistema sigue funcionando correctamente tras la reconfiguración parcial, por lo que se ejecuta el test 3 veces.

La primera vez hace exactamente lo mismo que se especificaba en la sección anterior (6.2.2). La segunda y la tercera, omiten ambas configuraciones: tanto la del principio, como la reconfiguración parcial.

Es importante mencionar que en la segunda ejecución del test, se cambian las instrucciones, enviando solo una del tipo *bit_up* y otra de *bit_get*. En la tercera se envía un comando *echo*.

Esto se hace así para comprobar que el sistema sigue funcionando correctamente tras la reconfiguración parcial. Todo esto queda recogido en la tabla 6.1

Tabla 6.1 Instrucciones Test 11.

Ejecución	Instrucciones	Programación Inicial	Reconfiguración Parcial
Primera	13 13 15	SI	SI
Segunda	13 15	NO	NO
Tercera	02 00000003 010203	NO	NO

Resultado Primera Ejecución (con reconfiguración parcial)

Código 6.1 Test 11: Primera Ejecución (con reconfiguración parcial).

```
2022-07-07 18:26:09: INFO       - ------------------------------------------------
2022-07-07 18:26:09: INFO       - | STARTING NEW TEST: 2022-07-07 18:26:09          |
2022-07-07 18:26:09: INFO       - ------------------------------------------------
2022-07-07 18:26:09: INFO       - Logger created and configured
2022-07-07 18:26:09: INFO       - ------------------------------------------------
2022-07-07 18:26:09: INFO       - Instructions successfully prepared -> ./data_in
2022-07-07 18:26:09: INFO       - ------------------------------------------------
2022-07-07 18:26:09: INFO       - Starting FPGA Test
2022-07-07 18:26:11: INFO       - Target configured
2022-07-07 18:26:11: INFO       - ------------------------------------------------
2022-07-07 18:26:11: INFO       - STARTING 1º TRANSACTION
2022-07-07 18:26:11: INFO       - Number of bytes to send: 3
2022-07-07 18:26:11: DEBUG      - Data reversed
2022-07-07 18:26:11: DEBUG      - data_buffer_in:   [151313]
2022-07-07 18:26:11: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-07 18:26:11: DEBUG      - data_buffer_in:   [131315]
2022-07-07 18:26:11: INFO       - ------------------------------------------------
2022-07-07 18:26:11: INFO       - Writing to bram0 via dma0
2022-07-07 18:26:11: DEBUG      - flag_buffer:      [ff00000000000000000000000000003]
2022-07-07 18:26:11: INFO       - ------------------------------------------------
2022-07-07 18:26:11: INFO       - Reading bram1 via dma1 (reset buffers first)
2022-07-07 18:26:11: DEBUG      - Buffers reset
2022-07-07 18:26:11: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-07 18:26:11: DEBUG      - data_buffer_in:   [000000]
2022-07-07 18:26:12: INFO       - FPGA ACK Received
2022-07-07 18:26:12: DEBUG      - flag_buffer:      [ff00000000000000000000000000001]
2022-07-07 18:26:12: INFO       - Number of bytes to read: 1
2022-07-07 18:26:12: DEBUG      - Data received
2022-07-07 18:26:12: DEBUG      - data_buffer_out:  [02]
2022-07-07 18:26:12: INFO       - Data recieved written -> ./data_out
2022-07-07 18:26:12: INFO       - Flag cleared
2022-07-07 18:26:12: INFO       - ------------------------------
2022-07-07 18:26:12: INFO       - -- END OF FIRST TRANSACTION --
2022-07-07 18:26:12: INFO       - ------------------------------
2022-07-07 18:26:14: DEBUG      - Starting FPGA Partial Reconfiguration
2022-07-07 18:26:16: INFO       - Target reconfigured
2022-07-07 18:26:16: INFO       - ------------------------------------------------
2022-07-07 18:26:16: INFO       - STARTING 2º TRANSACTION
2022-07-07 18:26:16: INFO       - Number of bytes to send: 3
2022-07-07 18:26:16: DEBUG      - Data reversed
2022-07-07 18:26:16: DEBUG      - data_buffer_in:   [151313]
2022-07-07 18:26:16: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
```

```
2022-07-07 18:26:16: DEBUG       - data_buffer_in:   [131315]
2022-07-07 18:26:16: INFO        - ------------------------------------------------
2022-07-07 18:26:16: INFO        - Writing to bram0 via dma0
2022-07-07 18:26:16: DEBUG       - flag_buffer:      [ff00000000000000000000000000003]
2022-07-07 18:26:16: INFO        - ------------------------------------------------
2022-07-07 18:26:16: INFO        - Reading bram1 via dma1 (reset buffers first)
2022-07-07 18:26:16: DEBUG       - Buffers reset
2022-07-07 18:26:16: DEBUG       - flag_buffer:      [00000000000000000000000000000000]
2022-07-07 18:26:16: DEBUG       - data_buffer_in:   [000000]
2022-07-07 18:26:17: INFO        - FPGA ACK Received
2022-07-07 18:26:17: DEBUG       - flag_buffer:      [ff00000000000000000000000000001]
2022-07-07 18:26:17: INFO        - Number of bytes to read: 1
2022-07-07 18:26:17: DEBUG       - Data received
2022-07-07 18:26:17: DEBUG       - data_buffer_out:  [82]
2022-07-07 18:26:17: INFO        - Data recieved written -> ./data_out
2022-07-07 18:26:17: INFO        - Flag cleared
2022-07-07 18:26:17: INFO        - ------------------------------
2022-07-07 18:26:17: INFO        - -- END OF SECOND TRANSACTION --
2022-07-07 18:26:17: INFO        - ------------------------------
2022-07-07 18:26:17: INFO        - ------------------------------------------------
2022-07-07 18:26:17: INFO        - Test Finished
```

Resultado Segunda Ejecución (sin reprogramar la tarjeta ni reconfigurar parcialmente)

Código 6.2 Test 11: Segunda Ejecución.

```
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - | STARTING NEW TEST: 2022-07-07 18:27:39          |
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - Logger created and configured
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - Instructions successfully prepared -> ./data_in
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - Starting FPGA Test
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - STARTING 1º TRANSACTION
2022-07-07 18:27:39: INFO        - Number of bytes to send: 2
2022-07-07 18:27:39: DEBUG       - Data reversed
2022-07-07 18:27:39: DEBUG       - data_buffer_in:   [1513]
2022-07-07 18:27:39: DEBUG       - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-07 18:27:39: DEBUG       - data_buffer_in:   [1315]
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - Writing to bram0 via dma0
2022-07-07 18:27:39: DEBUG       - flag_buffer:      [ff00000000000000000000000000003]
2022-07-07 18:27:39: INFO        - ------------------------------------------------
2022-07-07 18:27:39: INFO        - Reading bram1 via dma1 (reset buffers first)
2022-07-07 18:27:39: DEBUG       - Buffers reset
2022-07-07 18:27:39: DEBUG       - flag_buffer:      [00000000000000000000000000000000]
2022-07-07 18:27:39: DEBUG       - data_buffer_in:   [000000]
2022-07-07 18:27:40: INFO        - FPGA ACK Received
2022-07-07 18:27:40: DEBUG       - flag_buffer:      [ff00000000000000000000000000001]
2022-07-07 18:27:40: INFO        - Number of bytes to read: 1
2022-07-07 18:27:40: DEBUG       - Data received
2022-07-07 18:27:40: DEBUG       - data_buffer_out:  [83]
2022-07-07 18:27:40: INFO        - Data recieved written -> ./data_out
2022-07-07 18:27:40: INFO        - Flag cleared
2022-07-07 18:27:40: INFO        - ------------------------------
2022-07-07 18:27:40: INFO        - -- END OF FIRST TRANSACTION --
2022-07-07 18:27:40: INFO        - ------------------------------
```

Resultado tercera Ejecución (sin reprogramar la tarjeta ni reconfigurar parcialmente)

Código 6.3 Test 11: Tercera Ejecución (sin configuración de ningún tipo).

```
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - | STARTING NEW TEST: 2022-07-07 18:28:19       |
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - Logger created and configured
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - Instructions successfully prepared -> ./data_in
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - Starting FPGA Test
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - STARTING 1º TRANSACTION
2022-07-07 18:28:19: INFO       - Number of bytes to send: 8
2022-07-07 18:28:19: DEBUG      - Data reversed
2022-07-07 18:28:19: DEBUG      - data_buffer_in:   [0302010300000002]
2022-07-07 18:28:19: DEBUG      - Data prepared for ftu_stream_in (8 byte reversed)
2022-07-07 18:28:19: DEBUG      - data_buffer_in:   [0200000003010203]
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - Writing to bram0 via dma0
2022-07-07 18:28:19: DEBUG      - flag_buffer:      [ff000000000000000000000000000008]
2022-07-07 18:28:19: INFO       - ------------------------------------------------
2022-07-07 18:28:19: INFO       - Reading bram1 via dma1 (reset buffers first)
2022-07-07 18:28:19: DEBUG      - Buffers reset
2022-07-07 18:28:19: DEBUG      - flag_buffer:      [00000000000000000000000000000000]
2022-07-07 18:28:19: DEBUG      - data_buffer_in:   [0000000000000000]
2022-07-07 18:28:20: INFO       - FPGA ACK Received
2022-07-07 18:28:20: DEBUG      - flag_buffer:      [ff000000000000000000000000000007]
2022-07-07 18:28:20: INFO       - Number of bytes to read: 7
2022-07-07 18:28:20: DEBUG      - Data received
2022-07-07 18:28:20: DEBUG      - data_buffer_out:  [00000003010203]
2022-07-07 18:28:20: INFO       - Data recieved written -> ./data_out
2022-07-07 18:28:20: INFO       - Flag cleared
2022-07-07 18:28:20: INFO       - ------------------------------
2022-07-07 18:28:20: INFO       - -- END OF FIRST TRANSACTION --
2022-07-07 18:28:20: INFO       - ------------------------------
```

Conclusiones

Tras observar los resultados obtenidos se concluye que el *bitstream* parcial compilado funciona correctamente.

Por otro lado, se ha comprobado (mediante la segunda y tercera ejecución) que tanto el el contador como el resto del sistema siguen funcionando perfectamente tras la reconfiguración parcial.

7 Desensamblador de bitstreams

Nunca dejarás de sorprenderme.

Emma Moreno Socías

La principal motivación de este capítulo es que, para poder hacer inyección de fallos usando reconfiguración parcial, no se basta con hacer sólo una operación de escritura (que es lo que se consiguió en el capítulo 7).

Por tanto, se tiene que hacer lo que se conoce como un *read-modify-write* (si no, se machacan todos los bits de usuario que estén en ese frame con el valor de INIT que haya en el bitstream, y no se busca eso, sólo cambiar 1 bit), lo que implica hacer *readback* del *frame* de la FPGA que se quiere modificar.

Para poder hacer *readback*, el software de Xilinx no genera nada, por lo que, en el futuro cercano será necesario comparar las cabeceras de *readback* y reconfiguración que se tienen de la versión anterior de la plataforma (en Virtex-5) con las cabeceras de reconfiguración de la KU060 (generadas por Vivado).

De esta forma, se pueden determinar (con la ayuda de la documentación de *V*ivado [15] y reconfiguración de la familia Kintex Ultrascale) los comandos necesarios para hacer *readback*.

7.1 Adaptación del desensamblador de *bitstreams* a KU

Una vez se ha conseguido obtener un *bitstream* parcial, se procede a adaptar el desensamblador de *bitstreams* (para *bistreams* de la familia Virtex-5) a la familia a la que pertenece la tarjeta bajo estudio: Kintex Ultrascale (KU).

Es importante mencionar que este desensamblador se ha creado específicamente para este proyecto, a partir de un sistema diseñado por el tutor del TFM (Hipólito Guzmán Miranda) para *bitstreams* de la familia Virtex-5 (V5). Este se ha modificado, adaptado y mejorado para que funcione con los *bitstreams* de la familia KU.

7.1.1 Versión inicial

La versión inicial del desensamblador, el cual está escrito en *python*, hace lo siguiente:

1. Lee una palabra (4 bytes) del *bitstream* a desensamblar.
2. La compara con el diccionario propio de la familia a la que pertenece el *bitstream*, obteniendo una pequeña descripción de lo que hace.
3. Imprime por el terminal y escribe en un archivo de texto la palabra con su descripción al lado.
4. Vuelve al paso 1, repitiendo el proceso hasta leer y analizar todos los bytes del *bitstream*.

7.1.2 Modificaciones y mejoras implementadas

Para adaptar el sistema a la familia KU, se realizan las siguientes modificaciones:

- Se añade un diccionario con los comandos propios de la familia KU.
- Se desarrolla una función que busca el *idcode* (palabra que indica la familia a la que pertenece el *bitstream*) y selecciona el diccionario de comandos a utilizar (el de KU o V5).
- Se incluyen dos *bitstreams* parciales de la familia KU (resultado de la sección anterior: 6.2) para poder probar el sistema.

7.2 Comandos

La descripción de los distintos comandos obtenidos tras desensamblar un *bitstream* de la familia KU se recogen en el manual UltraScale Architecture Configuration de Xilinx [14].

Estos se dividen en dos tipos: comandos de Tipo 1 (*Type 1 Packet Registers*) y comandos de tipo 2 (*Type 2 Packet Registers*).

7.2.1 Comandos Tipo 1

Los paquetes de tipo 1 se utilizan para la lectura y escritura de registros. Sólo se utilizan cinco de los 14 bits de dirección de registro y la sección de cabecera es siempre una palabra de 32 bits.

Tabla 7.1 Comandos Tipo 1: Familia Kintex Ultraescale.

Nombre	Lectura/Escritura	Dirección	Descripción
CRC	Lectura/Escritura	00000	Registro CRC
FAR	Lectura/Escritura	00001	Registro de la dirección de la trama
FDRI	Escritura	00010	Registro de datos de trama, registro de entrada (escribir datos de configuración)
FDRO	Lectura	00011	Registro de datos de la trama, registro de salida (leer datos de configuración)
CMD	Lectura/Escritura	00100	Registro de comandos
CTL0	Lectura/Escritura	00101	Registro de control 0
MASK	Lectura/Escritura	00110	Registro de enmascaramiento para CTL0 y CTL1
STAT	Lectura	00111	Registro de estado
LOUT	Escritura	01000	Registro hereado para daisy chain
COR0	Lectura/Escritura	01001	Registro de opciones de configuración 0
MFWR	Escritura	01010	Registro de escritura de múltiples tramas
CBC	Escritura	01011	Registro del valor inicial de CBC
IDCODE	Lectura/Escritura	01100	Registro de identificación del dispositivo
AXSS	Lectura/Escritura	01101	Registro de acceso de usuarios
COR1	Lectura/Escritura	01110	Registro de opciones de configuración 1
WBSTAR	Lectura/Escritura	10000	Registro de dirección de arranque en caliente
TIMER	Lectura/Escritura	10001	Registro del watchdog
BOOTSTS	Lectura	10110	Registro de estado del historial de arranque
CTL1	Lectura/Escritura	11000	Registro de control 1
BSPI	Lectura/Escritura	11111	Registro de opciones de configuración BPI/SPI

7.2.2 Comandos Tipo 2

Los paquetes de Tipo 2, que deben seguir a un paquete de Tipo 1, se utilizan para escribir bloques largos.

No se presenta ninguna dirección porque utilizan la misma dirección que el paquete Tipo 1 anterior y la sección de cabecera es siempre una palabra de 32 bits.

7.3 Resultado

El resultado de desensamblar *counter_high.bit*, obtenido en el Test 11 (6.2) se presenta a continuación (código 7.1). Por confidencialidad, no se muestra el desensamblado del bitstream completo, sino únicamente las primeras instrucciones.

Código 7.1 Ejemplo resultado desensamblar *counter_high.bit.*.

```
ffffffff   Dummy pad word
[...]
ffffffff   Dummy pad word
000000bb   Bus width auto detect, word 1
11220044   Bus width auto detect, word 2
ffffffff   Dummy pad word
ffffffff   Dummy pad word
aa995566   Sync word
20000000   NOOP
[...]
20000000   NOOP
30008001   Type 1 Write ['CMD', '(Command register)'] register, WORD_COUNT = 1
00000007    command: RCRC (Resets the CRC register)
20000000   NOOP
20000000   NOOP
30018001   Type 1 Write ['IDCODE', '(Device ID register)'] register, WORD_COUNT = 1
03919093    FPGA IDCODE: ['KU', 'KU060']
30012001   Type 1 Write ['COR0', '(Configuration option register 0)'] register,
           WORD_COUNT = 1
38003fe5     ['COR0', '(Configuration option register 0)'] data word 1
30008001   Type 1 Write ['CMD', '(Command register)'] register, WORD_COUNT = 1
0000000b    command: SHUTDOWN (Begin shutdown sequence: Initiates the shutdown
            sequence, disabling the device when finished. Shutdown activates on the
            next successful CRC check or RCRC instruction (typically an RCRC
            instruction)
20000000   NOOP
[...]
30000001   Type 1 Write ['CRC', '(Cyclic Redundancy Check)'] register, WORD_COUNT = 1
24fb49a2     ['CRC', '(Cyclic Redundancy Check)'] data word 1
20000000   NOOP
[...]
```

7.4 Conclusión

Tras observar los resultados obtenidos y compararlos con la guía de configuración de la familia Ultrascale [15], se concluye que el desensamblador funciona correctamente y, por tanto, se ha cumplido el objetivo del capítulo.

8 Conclusiones y trabajos futuros

Sin ser apenas consciente, perdido en su caótica mente, fue tejiendo poco a poco una canción. Esta era nueva. No tenía ni idea de donde había salido.

JGM

En este capítulo se desarrollan las conclusiones a las que se ha llegado tras la realización del proyecto, así como los posibles trabajos futuros que podrían llevarse a cabo para continuar con el desarrollo de una plataforma de inyección de fallos basada en Kintex Ultrascale.

8.1 Resultados y conclusiones

A continuación, se exponen los principales resultados y conclusiones:

Se ha obtenido un sistema de comunicaciones que permite enviar y recibir datos desde una interfaz de entrada y salida de un sistema de inyección de fallos (*ftu_stream*). Este sistema está compuesto por un *host program* y una arquitectura que se encuentra dentro de la FPGA. El primero envía las instrucciones a la FPGA a través del PCIe y del Modular Plug Terminated Link (MPTL), gracias a la interfaz ADM-XRC-KU1-HSAXI y a las funciones de la API proporcionadas por *Alphadata*). La segunda mueve los datos desde el MPTL hasta las interfaces de entrada y salida del sistema de inyección de fallos.

Una vez resueltas las comunicaciones entre el *host* y la FPGA, se ha mejorado la infraestructura encargada de gestionar y preparar las instrucciones (*host program*), de forma que se pueda enviar y recibir cualquier cantidad de datos sin problema y se lleve un registro exhaustivo de todo el proceso.

A continuación, se ha desarrollado una interfaz capaz de interpretar comandos y datos, la cual se conecta a los *ftu_streams* de entrada y salida del sistema. De esta forma, se ha conseguido emular un sistema de inyección de fallos capaz de ejecutar varios comandos sencillos.

Por otro lado, se ha creado un proyecto que permite reconfigurar parcialmente la FPGA, obteniéndose en el proceso *bitstreams* parciales funcionales, necesarios para la inyección de fallos y se ha demostrado que funcionan correctamente.

Por último, con este trabajo, se ha reducido al mínimo la incertidumbre a la hora de desarrollar la plataforma completa. Se han eliminado muchos bloqueos que han surgido durante el desarrollo y que han sido de difícil depuración, debido a la nula o casi nula observabilidad y capacidad de depuración cuando se tienen problemas al utilizar módulos de terceros de los cuales no se disponen los códigos fuente, como pueden ser la interfaz PCIe, la interfaz MTPL, la API *software* de Alphadata y el interfaz de reconfiguración de la FPGA Kintex Ultrascale.

Debido a que estos proyectos suelen realizarse bajo estrictas condiciones temporales y presupuestarias, la reducción del riesgo previa resulta crítica a la hora de planificar y abordar el desarrollo de la plataforma completa.

8.1.1 Conclusión personal

Trabajar en este proyecto me ha permitido adquirir nuevos conocimientos acerca de los sistemas de inyección de fallos, así como desarrollar las competencias transversales necesarias para trabajar con ellos.

Además, he aprendido a diseñar y trabajar con sistemas de comunicaciones complejos, así como a depurar y solucionar diversos problemas enfrentados durante el desarrollo.

Por otro lado, me ha permitido ampliar mis conocimientos sobre el sistema operativo *Debian GNU/Linux*, git (sistema de control de versiones), C++ y (sobre todo) VHDL.

Para terminar, me llevo la oportunidad de investigar junto a un gran profesional (con todo el aprendizaje que esto conlleva) y sentar las bases necesarias para el desarrollo de un proyecto mucho mayor.

8.2 Trabajos futuros

Se propone estudiar y comparar los *bitstreams* parciales de la KU060 obtenidos con los de una FPGA de la familia V5, con el objetivo de aprender cómo enviar tramas a través del *SelecMap* que permitan cambiar *bits* de configuración concretos, sin modificar nada más.

Por otro lado, se propone investigar como leer tramas de la FPGA, de forma que se pueda hacer *readback*: obtener el estado de cualquier interfaz (o trama) del sistema, modificarla acorde al fallo que se quiera inyectar y enviar la trama modificada a la FPGA a través del *SelectMap*, realizando la inyección de fallos.

Por último, se propone desarrollar la plataforma de inyección de fallos completa, a partir de toda la infraestructura desarrollada (sistema de comunicaciones, *host program*, intérprete de comandos y desensamblador de *bitstreams*).

Índice de Figuras

Índice de Códigos

Bibliografía

[1] *Oneweb*, (2022), Online. Accedido el 10 de junio de 2022.

[2] M. A. Aguirre, J. Barrientos, H. Guzmán-Miranda, F. Márquez, F. Muñoz, and L. Sanz, https://ftu.us.es/, Online. Accedido el 10 de junio de 2022.

[3] Alfa Data, *ADM-XRC-KU1 Datasheet, Revision 1.1*, https://www.alpha-data.com/pdfs/adm-xrc-ku1_-v1.1.pdf, Online. Accedido el 13 de diciembre de 2019.

[4] _____, *ADM-XRC-KU1 SDK, Revision 1.0.0*, https://support.alpha-data.com/pub/admxrcg3/linux/archive/admxrcku1_sdk-1.0.0.tar.gz, Online. Accedido el 13 de diciembre de 2019.

[5] Javier González Moreno, *Estudio de interfaz MPTL-AXI en la tarjeta ADM-XRC-KU1 y adaptación a sistema de inyección de fallos*, Universidad de Sevilla Escuela Técnica Superior de Ingeniería – Trabajo Fin de Grado. Director: Hipólito Guzmán Miranda, 2020.

[6] H. Guzmán-Miranda, L. Sanz, M. Muñoz-Quijada, F. Muñoz, F. Márquez, and D. Vela-Calderón, *Ft-unshades2 ccn3: Milestone 1 presentation. contrato esa nº: 22981/09/nl/jk*.

[7] Maria Muñoz-Quijada, Luis Sanz, and Hipolito Guzman-Miranda, *A virtual device for simulation-based fault injection*, Electronics **9** (2020), no. 12.

[8] Python, *C-style parser for command line options*, https://docs.python.org/3/library/getopt.html, Online. Accedido el 17 de septiembre de 2021.

[9] _____, *Logging facility for python*, https://docs.python.org/3/library/logging.html, Online. Accedido el 14 de septiembre de 2021.

[10] S. Rezgui, R. Velazco, R. Ecoffet, S. Rodriguez, and J.R. Mingo, *Estimating error rates in processor-based architectures*, IEEE Transactions on Nuclear Science **48** (2001), no. 5, 1680–1687.

[11] SpaceX, https://www.starlink.com/, 2022, Online. Accedido el 10 de junio de 2022.

[12] R. M. Tawfeek, M. G. Egila, Y. Alkabani, and I. M. Hafez, *Fault injection for fpga applications in the space*, 2017 12th International Conference on Computer Engineering and Systems (ICCES), 2017, pp. 390–395.

[13] Raoul Velazco and Fabien Faure, *Error rate prediction of digital architectures: Test methodology and tools*, pp. 233–258, Springer Netherlands, Dordrecht, 2007.

[14] Xilinx, *ug570 ultrascale configuration*, 2016, Online. Accedido el 02 de julio de 2022.

[15] _____, *ug908 vivado programming debugging*, 2016, Online. Accedido el 27 de junio de 2022.

[16] _____, *ug909 vivado partial reconfiguration user guide*, 2016, Online. Accedido el 10 de mayo de 2021.

[17] _____, *ug947 vivado partial reconfiguration tutorial*, 2016, Online. Accedido el 10 de mayo de 2021.