Trabajo Fin de Grado
Ingeniería de las Tecnologías Industriales

Estudio de interfaz MPTL-AXI en la tarjeta ADM-XRC-KU1 y adaptación a sistema de inyección de fallos

Autor: Javier González Moreno

Tutor: Hipólito Guzmán Miranda

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

Sevilla, 2020

Trabajo Fin de Grado
Ingeniería de las Tecnologías Industriales

Estudio de interfaz MPTL-AXI en la tarjeta ADM-XRC-KU1 y adaptación a sistema de inyección de fallos

Autor:

Javier González Moreno

Tutor:

Hipólito Guzmán Miranda

Profesor Titular

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

Sevilla, 2020

Trabajo Fin de Grado: Estudio de interfaz MPTL-AXI en la tarjeta ADM-XRC-KU1 y adaptación a sistema de inyección de fallos

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 TFG (Trabajo Final de Grado), ya que ha sido su valiosa experiencia, acogida y orientación lo que ha permitido su realización. Me gustaría hacer una mención especial a Luis, cuyo conocimiento y paciencia han hecho de mi introducción a *Debian* y C++ un camino mucho menos pedregoso. A resto de componentes del equipo de investigación: María, Álvaro y Víctor, por todas las tardes de reuniones y sesiones de trabajo.

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 quiero agradecer a Eduardo su aportación, que ha hecho que este documento sea mucho más accesible a cualquiera que no esté familiarizado con los términos que se utilizan.

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.

A mis hermanos, Emma e Ignacio, quiero tenerlos presentes en este día, por lo importantes que son para mí y por todo lo que me han aportado 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. De Charlie y a Antonio, con especial cariño, ya que todas esas conversaciones no se mantienen solas.

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 Grado parte de la oportunidad de participar en el diseño de un sistema de inyección de fallos en la tarjeta ADM-XRC-KU1 de Alpha Data. Para ello, se estudia el funcionamiento de la tarjeta, los modos de comunicación y los diseños proporcionados en su *Development Kit* (Kit de desarrollo).

Concretamente, se centra en el estudio de la interfaz ADM-XRC-KU1-HSAXI, incluida en uno de los ejemplos del *Development Kit*, con el fin de adaptarla un sistema de inyección de fallos. Con ese objetivo, se comparan las dos formas de transmitir datos que la interfaz proporciona: mediante el *Direct Slave* (Mapeo de memoria) y a través de *DMA Engines* (Direct Memory Access: Acceso Directo a Memoria).

Por otro lado, se estudia la viabilidad de las memorias bRAM (Block Random Access Memory) como interfaces de entrada/salida de los sistemas de inyección de fallos.

Por último, se diseña y simula una interfaz que permite la comunicación entre la ADM-XRC-KU1-HSAXI (a través de un *DMA Engine*) y un *ftu stream* (flujo de datos del sistema de inyección de fallos a implementar).

El objetivo de este documento es recoger toda la información recabada de la tarjeta e interfaz bajo estudio, así como los resultados de las pruebas y arquitecturas diseñadas con el fin de adaptar esta a un sistema de inyección de fallos.

Abstract

This project begins when given the opportunity to work in the design of a fault injection system in the ADM-XRC-KU1 board, developed by Alpha Data. In order to achieve that, FPGA operation, configuration modes and designs included in the SDK (Software Development Kit) are studied.

Specifically, this project is based on the study of the ADM-XRC-KU1-HSAXI interface (included in the SDK). The objective is to adapt it to a fault injection system. To this purpose, both *DMA Engines* and *Direct Slave* (included in the interface) are compared.

Furthermore, the viability of bRAMs as input/output interfaces of a fault injection system are studied.

Lastly, an architecture which allows the transmission of data between a *DMA Engine* and *ftu stream* is designed and simulated.

The purpose of this document is to collect all the information obtained, as well as the results of the tests and architectures designed with the purpose of adapting the ADM-XRC-KU1-HSAXI interface to a fault injection system.

Índice Abreviado

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

DDR4 SDRAM
: Double Data Rate 4 Synchronous Dynamic RAM

DMA
: Direct Memory Access

DSP
: Digital Signal Processor

FF
: Flip-Flop o Latch

FIFO
: First In, First Out

FPGA
: Field-Programmable Gate Array

FT-UNSHADES (FTU)
: Fault Tolerance University of Sevilla Hardware

GIE
: Grupo de Ingeniería Electrónica

JTAG
: Joint Test Action Group

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

XMC
: Switched Mezzanine Card

1 Introducción

Todo se andará hijito.

PILAR VICENCE RECUERO

Históricamente, el ser humano ha mostrado un gran interés por estudiar y entender el universo. Últimamente, la exploración espacial está adquiriendo cada vez más relevancia, debido a la necesidad de encontrar nuevas fuentes de recursos que son cada vez más escasos en nuestro planeta. Es por ello que la mirada al cielo ha adquirido una perspectiva algo distinta: ya no solo la curiosidad y la búsqueda del conocimiento mueven la exploración espacial.
Aunque este tipo de investigación no es sencilla, cada vez se dispone de satélites más avanzados, potentes y robustos. No obstante, estas nuevas tecnologías plantean retos conocidos de manera diferente. Este es el caso de la radiación ionizante, que puede llegar a modificar los datos de las memorias de programa de los satélites, provocando fallos es los sistemas de control y pérdida de comunicación en el peor de los casos.
Consecuentemente, antes de embarcar cualquier dispositivo electrónico en uno de estos vehículos, se somete a pruebas muy estrictas. Entre ellas destaca el test de radiación, que se considera indispensable.

1.1 Inyección de fallos: FT-UNSHADES

Además del test de radiación, es aconsejable someter estos dispositivos a la inyección de fallos, que se define como la técnica de validación de sistemas tolerantes a fallos donde se observa el comportamiento del sistema en presencia de fallos generados mediante la inyección de fallos en el sistema[8].
Esta técnica permite estudiar la robustez y fiabilidad del sistema bajo estudio durante el proceso de diseño. Para ello, se compara la salida del sistema antes y después de generar los fallos[6].

1.1.1 FT-UNSHADES

FTUNSHADES (FTU) es un sistema de inyección de fallos híbrido (utiliza conceptos de inyección por hardware, simulación y software) desarrollado por el grupo de investigación GIE (Grupo de Ingeniería Electrónica) del departamento de Ingeniería Electrónica de la Universidad de Sevilla.
A día de hoy, la última versión vigente es la FTU2, para la SRAM-FPGA Xilinx Virtex5. Esta utiliza el puerto USB (Universal Serial Bus) para las comunicaciones[5], lo cual supone que las tasas de transmisión estén limitadas, debido a que el USB hace de cuello de botella. Es por ello, que las nuevas tendencias apuntan al uso del PCI express (Peripheral Component Interconnect express) para mejorar tanto la velocidad como la integración; lo que supondría una nueva generación de sistemas de inyección de fallos. Solo faltaría encontrar un dispositivo que lo permita.

1.2 ADM-XRC-KU1

Esta tarjeta de desarrollo, distribuida por Alpha Data, es una candidata interesante a esta proposición de integración del sistema de fallos FTU2 a través del PCIe, ya que sus características encajan con lo que se busca.
En primer lugar, Xilinx la recomienda para el prototipado de sistemas aeroespaciales, ya que se trata de una tarjeta muy completa, que permite diseñar un sistema sin restricciones y más adelante fabricar la tarjeta que se enviará al espacio únicamente con lo que ese diseño necesite. Consiguiéndose recortar gastos y facilitar el de proceso diseño.
Esa completitud también afecta a la aplicación de la técnica de inyección de fallos, porque el prototipo final no suele incluir los elementos necesarios para ello.
Además, esta tarjeta permite la reconfiguración remota sin que el ordenador tenga que ser reiniciado.
Por último, Alpha Data proporciona un *Development Kit* con varios ejemplos e interfaces, entre ellas la interfaz ADM-XRC-KU1-HSAXI. Esta interfaz es candidata para la adaptación buscada, al contar con varios *DMA Engines* (Acceso Directo a Memoria[7]) y un *Direct Slave* (Memoria Mapeada).

1.3 Alcance del Proyecto

La finalidad principal de este proyecto es estudiar la tarjeta ADM-XRC-KU1, e investigar si es posible y, en el caso de que lo sea, cómo se puede integrar el sistema de inyección de fallos en esta tarjeta.
Para ello, se proponen los siguientes objetivos:

1. Estudiar los componentes, la estructura y el funcionamiento de la tarjeta ADM-XRC-KU1 y de los ejemplos proporcionados en el *Development Kit* (kit de desarrollo) de Alpha Data.
2. Realizar un *set up* inicial para y probar esos ejemplos proporcionados en el *Development Kit*.
3. Estudiar y comparar el funcionamiento del *Direct Slave* y de los *DMA Engines*.
4. Diseñar un test en el que se escriba y lea en una memoria de la target FPGA (bRAM).
5. Investigar la viabilidad del uso de bRAMs como interfaces de entrada/salida de un sistema de inyección de fallos.
6. Diseñar una arquitectura que permita la interpretación y ejecución de comandos simples utilizando bRAMs como dispositivos de entrada/salida del sistema.
7. Adaptar la interfaz ADM-XRC-KU1-HSAXI al sistema de inyección de fallos. Para ello será recomendable la conexión entre un *DMA Engine* (protocolo AXI) y una interfaz tipo FIFO como un *ftu stream*.

2 Tarjeta ADM-XRC-KU1

Me voy a subir al palo.

MANUEL MORENO BORRÁS

La finalidad de este capítulo es estudiar y comprobar el funcionamiento de la tarjeta ADM-XRC-KU1 de Alpha Data, así como probar los ejemplos proporcionados en el *Development Kit*.
En primer lugar, es necesario comprender las características de esta tarjeta, así como de la Carrier Board y los programas y arquitecturas ejemplo que vienen en el *Development Kit*. A continuación, se expone toda la información recabada.

2.1 Características Tarjeta

La ADM-XRC-KU1 es una XMC reconfigurable de alto rendimiento basada en la familia Xilinx Kintex Ultrascale de FPGAs, concretamente cuenta con una **XCKU060** (target FPGA).
Incluye una interfaz PCIe Gen2, memoria externa, puertos de E/S de alta densidad, monitoreo del sistema y memoria flash de arranque. Además, posee una API integral multiplataforma con soporte para Microsoft Windows, Linux y VxWorks, que proporciona acceso a la funcionalidad completa de estas características de hardware.
La placa es gestionada por la combinación de una **FPGA Artix** (bridge FPGA) y un microcontrolador AVR los cuales permiten administrar la tarjeta a través del PCIe o USB.
En la figura 2.1 se pueden observar ambas FPGAs incluidas en la tarjeta: la bridge se encuentra dentro del rectángulo azul de la izquierda y la target es el rectángulo azul del medio (Kintex Ultrascale)[2].

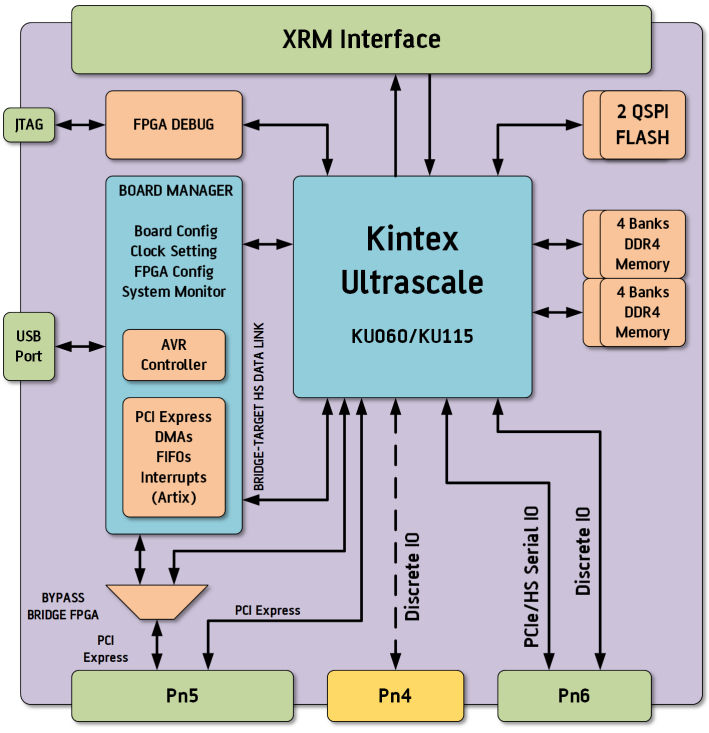

Figura 2.1 ADM-XRC-KU1[2].

2.1.1 Estructura Interna

La target FPGA (XCKU060) contiene los siguientes recursos hardware, además de 32 GTH Transceivers:

Target Device	LUTs	FFs	DSPs	bRAMs
XCKU060	221k	663k	2760	38.0MB

2.1.2 Modos de programación y comunicación

La target FPGA se puede programar de tres maneras diferentes:

- A través del conector JTAG.
- Vía puente PCIe al encender la tarjeta.
- Mediante software a través del puente PCIe.

Para el desarrollo de este proyecto se va a emplear la tercera opción, ya que Alpha Data proporciona una API (incuida en su *Development Kit*) con las funciones necesarias, que aparecen descritas en la sección 2.5, junto con otras que también serán utilizadas.
Por otro lado, es fundamental entender los múltiples modos de comunicación que proporciona la ADM-XRC-KU1, que se listan a continuación:

- Mediante el PCIe (Gen2 x4) a través de la bridge FPGA, con un enlace PCIe (Gen3 x4) opcional directo a la target FPGA.
- A través del enlace PCIe (Gen3 x8) directo a la target FPGA, cuando el puente está en modo USB.
- Con el enlace PCIe (Gen3 x8) a través de Pn6, utilizando un operador XMC compatible.

Para poder aprovechar la opción de la programación vía software, se va a utilizar el primer modo, ya que el puente MPTL (Modular Plug Terminated Link) entre la bridge FPGA (Artix) y la target FPGA (XCKU060) permite la reconfiguración de esta última sin generar errores en el PCIe. Gracias a esto y a lo descrito en el párrafo anterior, se puede reconfigurar o programar la target FPGA en remoto y sin la necesidad de reiniciar el ordenador al que esté conectada[3].

2.2 Características Carrier Board

Para poder acceder al PCIe conectado a la bridge FPGA se utiliza una *Carrier Board*. Se trata de una tarjeta auxiliar que se conecta a los Pn4, Pn5 y Pn6 (conectores XMC) de la ADM-XRC-KU1 (ver imagen 2.1).
Se encarga de adaptar los puertos XMC de la tarjeta a PCIe, facilitando la conexión al ordenador desde el cual se programará la tarjeta.
Concretamente, se va a aprovechar el adaptador ADC-PCIE-XMC [1] de Alpha Data, una tarjeta PCIe de media longitud diseñada para transportar un solo XMC. Además, puede admitir el uso de entornos de señalización PCIe Gen2 x1, x2, x4 y x8 en ranuras PCIe x8 o x16 y el conector XMC en la tarjeta admite hasta en PCIe x8 o dos enlaces PCIe x4. Cabe mencionar que el VPWR proporcionado al sitio XMC se establece en 12V.
De esta forma, se puede aprovechar la conexión del PCIe de la *Carrier Board* al conector Pn5, el cual está enlazado con la bridge FPGA, consiguiendo lo que se proponía en la subsección 2.1.2.

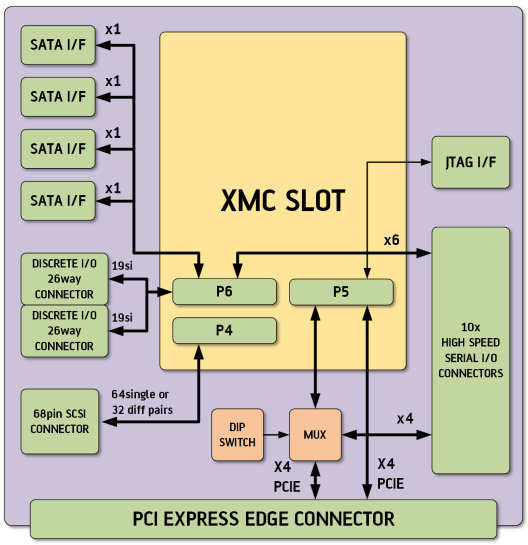

Figura 2.2 ADC-PCIE-XMC[1].

2.3 Development Kit de Alpha Data

Finalizado el estudio de las características de la tarjeta, se procede a comprobar su buen funcionamiento. Con tal fin, se utilizan los ejemplos incluidos en el *Development Kit* que proporciona Alpha Data[3]. Para cada uno de los 5 ejemplos, contiene tanto el código necesario para compilar el host program (software que se ejecuta desde el PC al que está conectado la tarjeta y que utiliza el PCIe para comunicarse con ella) como los archivos para generar el proyecto (utilizando Vivado 2016.2) con el que implementar los .bit de configuración para la target FPGA. A continuación, se describe resumidamente el funcionamiento de cada uno de estos:

1. **Standalone DDR4 Test FPGA Design**: Demuestra el uso de las SDRAM DDR4 incluidas en la placa.
2. **DMA Demonstration FPGA Design**: Demuestra el uso de los *DMA Engines* (AXI4), de la interfaz *ADM-XRC-KU1-HSAXI* de Alpha Data, para enviar datos entre la target FPGA (a una bRAM) y el host vía *PCIe to MPTL Bridge*.
3. **DMA Demonstration (PCIe) FPGA Design**: Demuestra el uso de la interfaz *ADM-XRC-KU1-P5HI* de Alpha Data, que incluye un PCIe endpoint con *DMA engines* (AXI4).
4. **Simple Demonstration FPGA Design**: Demuestra el uso de la interfaz *ADM-XRC-KU1-HSAXI*, que permite al host escribir y leer registros de la target FPGA vía *PCIe to MPTL Bridge*, utilizando el *Direct Slave*.
5. **Simple Demonstration (OCP) FPGA Design**: Este ejemplo es muy similar al anterior, pero utiliza el protocolo OCP y la interfaz *ADM-XRC-KU1-HSOCP* (también utiliza únicamente el *Direct Slave*).

2.3.1 Interfaces proporcionados por Alpha Data

Las arquitecturas de estos diseños de ejemplo utilizan distintas interfaces (desarrolladas por Alpha Data) para recoger los datos que entran a través del MPTL o PCIe.

- **ADM-XRC-KU1-HSAXI** Host Interface IP: Proporciona una interfaz de MPTL a AXI4 con dos *DMA Engines* y el *Direct Slave*. Permite a la CPU del host intercambiar datos con la target FPGA vía *PCIe to MPTL Bridge*.
- **ADM-XRC-KU1-HSOCP** Host Interface IP: Proporciona una interfaz de MPTL a OCP con el *Direct Slave*. Realiza una función parecida a la de la interfaz anterior (ADM-XRC-KU1-HSAXI), pero con el protocolo OPC en vez del AXI4.
- **ADM-XRC-KU1-P5HI** PCIe Host Interface IP: Proporciona una interfaz de PCIe a AXI4 con un número configurable de DMA Engines (AXI4), entre otras cosas, en la target FPGA. Está pensado para el desarrollo de aplicaciones que envíen datos a la target FPGA sin pasar por el *PCIe to MPTL Bridge*

2.4 Elección diseño de partida

El siguiente paso del proyecto es decidir qué ejemplo va a ser utilizado como punto de partida. Tras revisar la lista de la sección 2.3, varios son desechados por los motivos expuestos a continuación:
El número 1 es el primero que se descarta, puesto que en este ejemplo se demuestra el uso de las SDRAM DDR4 y la idea principal del proyecto es aprovechar los *DMA Engines* o el *Direct Slave* (se compararán para decidir cual es mejor para el desarrollo de proyecto) para escribir y leer en la memoria de la target FPGA.
Los números 4 y 5 tampoco interesan, ya que solo utilizan el *Direct Slave* para realizar las escrituras y lecturas en los registros de la target FPGA.
Por último, el 3 tiene un problema que inicialmente puede pasar inadvertido: no se puede programar la FPGA con esa arquitectura vía software. Esto es debido a que la bridge se encarga de gestionar este proceso y esta arquitectura utiliza todos los canales PCIe para conectarse a la target FPGA, de forma que la información no pasa en ningún momento por esta FPGA auxiliar. Se decide, por tanto, no tener en cuenta este ejemplo a priori.
El único diseño que cumple con las características buscadas es el número 2: **DMA Demonstration FPGA Design**, ya que aprovecha tanto los *DMA Engines* como el *Direct Slave* para realizar las transmisiones de datos. Además, permite la reconfiguración en remoto y tiene ya una memoria bRAM en la que se puede escribir y leer (que es lo que se busca).

2.5 Funciones API

Antes de comenzar la descripción de los test desarrollados durante la investigación, es importante explicar el funcionamiento de las funciones de la API que proporciona Alpha Data[3], que han sido utilizadas tanto en los ejemplos como en los tests desarrollados.

2.5.1 ADMXRC3_Open()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_Open(
   __in unsigned int index,
   __out ADMXRC3_HANDLE* phCard
);
```

Los parámetros de esta función son los siguientes:

index (in)
Identifica el dispositivo a abrir.

phCard (out)
Apunta a una variable de tipo ADMXRC3_HANDLE que recibirá el identificador del dispositivo abierto.

Descripción

Esta función abre un dispositivo y devuelve un identificador que se puede utilizar en las siguientes llamadas a cualquier función de la API de ADMXRC3.
El identificador permanece válido hasta que se llama a ADMXRC3_Close. Un dispositivo determinado se puede abrir varias veces, mediante el mismo proceso o diferentes procesos.
En los host programs ha sido utilizada para obtener el identificador del dispositivo, ya que es necesario para poder llamar al resto de las funciones (descritas a continuación).

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El proceso no tiene suficientes privilegios para abrir el dispositivo.
ADMXRC3_DEVICE_NOT_FOUND	El identificador apunta a un dispositivo que no existe en el sistema.
ADMXRC3_NO_MEMORY	No se pudo asignar memoria para realizar un seguimiento de un nuevo identificador de dispositivo.

2.5.2 ADMXRC3_Close()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_Close(
   __in ADMXRC3_HANDLE hDevice);
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador del dispositivo a cerrar.

Descripción

Esta función cierra el identificador de un dispositivo y lo invalida. No debe ser llamada hasta que todos los hilos hayan acabado de usar ese identificador de dispositivo.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.

2.5.3 ADMXRC3_ConfigureFromFile()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_ConfigureFromFile(
   __in ADMXRC3_HANDLE hDevice,
   __in unsigned int targetIndex,
   __in uint32_t flags,
   __in const char* pFilename);
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador del dispositivo (target FPGA) a configurar.

targetIndex (in)
El índice de la target FPGA que se va a configurar.

flags (in)
Flags que modifican el funcionamiento de la función.

pFilename (in)
Especifica el nombre del archivo .bit que se descargará en la target FPGA.

Descripción

Esta función descarga los datos en un archivo bitstream (.bit) al dispositivo especificado. Como el archivo .bit contiene un registro que especifica para qué tipo de FPGA se generó, esta función verifica que el tipo de archivo .bit coincide con el del dispositivo, y devuelve un error si hay una discrepancia.
El comportamiento predeterminado es una reconfiguración completa de la target FPGA, donde la descarga del bitstream es precedida por la desconfiguración esta.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El identificador del dispositivo se abrió con privilegios insuficientes para modificar el estado del dispositivo.
ADMXRC3_CANCELLED	Durante la operación, otro hilo llamó ADMXRC3_Cancel() con el identificador del dispositivo o este fue cerrado.
ADMXRC3_FILE_NOT_FOUND	No se pudo abrir el archivo identificado por el parámetro pFilename.
ADMXRC3_FPGA_MISMATCH	La FPGA en el encabezado del archivo .bit no coincide con la FPGA de destino en el dispositivo especificado.
ADMXRC3_HARDWARE_ERROR	La configuración de la FPGA de destino con el bitstream no se realizó correctamente debido a un error de hardware.
ADMXRC3_INVALID_BITSTREAM	El archivo identificado por el parámetro pFilename no es un archivo .bit válido.
ADMXRC3_INVALID_FLAG	El parámetro flags contiene una flag no reconocida.
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_INVALID_INDEX	El parámetro targetIndex está fuera de rango.
ADMXRC3_NOT_OWNER	El identificador de dispositivo no es el propietario de la target FPGA.
ADMXRC3_NO_MEMORY	No se pudo asignar memoria donde guardar los datos del bitstream.
ADMXRC3_NULL_POINTER	El parámetro pFilename es un puntero a NULL.

2.5.4 ADMXRC3_Lock()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_Lock(
   __in ADMXRC3_HANDLE hDevice,
   __in const void* pBuffer,
   __in size_t length,
   __out ADMXRC3_BUFFER_HANDLE* phBuffer
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador del dispositivo que va a ser el propietario del buffer bloqueado.

pBuffer (in)
Apunta al buffer que va a ser bloqueado.

length (in)
Tamaño del buffer a bloquear, en bytes.

phBuffer (out)
Apunta a una variable de tipo ADMXRC3_BUFFER_HANDLE que va a recibir el identificador del buffer bloqueado.

Descripción

Esta función bloquea un buffer (perteneciente al usuario) en la memoria para que el sistema operativo no pueda utilizarlo como memoria swap. Si tiene éxito, la función devuelve un identificador de tipo ADMXRC3_-BUFFER_HANDLE y garantiza que el buffer estará totalmente residente en la memoria física hasta que:

- La aplicación desbloquea el buffer con una llamada inversa a ADMXRC3_Unlock, o
- La aplicación cierra el identificador del dispositivo con una llamada a ADMXRC3_Close, o
- La aplicación termina sin limpiar; en este caso, el buffer se desbloquea cuando el sistema cierra automáticamente cualquier identificador de dispositivo abierto.

El uso de esta función reduce la sobrecarga incurrida al realizar transferencias a través de la DMA al eliminar la necesidad de bloquear y desbloquear el buffer para cada transferencia DMA.
El buffer puede bloquearse una vez cuando se inicie el host program, usarse en un número arbitrario de transferencias DMA y desbloquearse cuando el programa termine.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El identificador del dispositivo se abrió con privilegios insuficientes para modificar el estado del dispositivo.
ADMXRC3_INVALID_BUFFER	Los parámetros pBuffer y length representan un búfer que no es válido en el espacio de direcciones de la persona que llama.
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_NO_MEMORY	No se pudo asignar memoria suficiente para el buffer.
ADMXRC3_NULL_POINTER	El parámetro phBuffer es un puntero a NULL.
ADMXRC3_RESOURCE_LIMIT	Se ha alcanzado ya el número máximo de buffers bloqueados.

2.5.5 ADMXRC3_Unlock()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_Unlock(
   __in ADMXRC3_HANDLE hDevice,
```

```
    __in ADMXRC3_BUFFER_HANDLE hBuffer);
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador de dispositivo utilizado para bloquear el buffer.

hBuffer (in)
Identificador del buffer a desbloquear.

Descripción

Esta función desbloquea un buffer de espacio de usuario bloqueado para que el sistema operativo pueda utilizar nuevamente como memoria swap. Una aplicación debe asumir que el sistema operativo puede comenzar a cambiar el buffer tan pronto como se llame a esta función. ADMXRC3_Unlock es el inverso de ADMXRC3_-Lock.
Los valores de ADMXRC3_BUFFER_HANDLE son globales para el sistema. Sin embargo, cada identificador de buffer tiene un propietario, que es el identificador de dispositivo que se utilizó para crearlo. Solo el identificador de dispositivo utilizado para bloquear un buffer se puede usar para llamar correctamente a ADMXRC3_Unlock.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_BUFFER_INVALID_HANDLE	El parámetro hBuffer no es un identificador válido de buffer bloqueado.
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_NOT_OWNER	El buffer bloqueado fue creado con un identificador de dispositivo diferente al especifivado en hDevice.

2.5.6 ADMXRC3_WriteDMALockedEx()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_WriteDMALockedEx(
    __in ADMXRC3_HANDLE hDevice,
    __in unsigned int dmaChannel,
    __in uint32_t flags,
    __in ADMXRC3_BUFFER_HANDLE hBuffer,
    __in size_t offset,
    __in size_t length,
    __in uint64_t localAddress);
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador del dispositivo para realizar la transferencia (a través de la DMA).

dmaChannel (in)
Especifica qué canal DMA del dispositivo se utilizará para realizar la transferencia.

flags (in)
Flags que modifican el funcionamiento de la función.

hBuffer (in)
Identificador del buffer bloqueado que contiene los datos a escribir en el dispositivo.

offset (in)
Offset en el buffer bloqueado donde se encuentran los datos a escribir en el dispositivo.

length (in)
Numero de bytes a escribir en el dispositivo.

localAddress (in)
Dirección local (en bytes) del dispositivo en la que empezar a escribir los datos.

Descripción

Esta función escribe un bloque de datos desde un buffer bloqueado en el dispositivo, comenzando en la dirección local especificada. La transferencia de datos se realiza mediante el motor DMA dentro del dispositivo.
Esta función debe obtener un identificador a un buffer ya bloqueado, obtenido llamando a ADMXRC3_Lock previamente.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El identificador del dispositivo no tiene privilegios suficientes para modificar el estado del dispositivo.
ADMXRC3_CANCELLED	Durante la operación, otro hilo llamó ADMXRC3_Cancel() con el identificador del dispositivo o este fue cerrado.
ADMXRC3_HARDWARE_ERROR	Un error de hardware ha ocurrido durante la transferencia a través de la DMA.
ADMXRC3_INVALID_BUFFER_HANDLE	El parámetro hBuffer no es un identificador válido de buffer bloqueado.
ADMXRC3_INVALID_FLAG	El parámetro flags contiene una flag no reconocida
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_INVALID_INDEX	El parámetro dmaChannel especifica un canal que no existe.
ADMXRC3_INVALID_LOCAL_REGION	Los parámetros localAddress y length señalan una región del espacio de memoria del bus local que es inválido.
ADMXRC3_INVALID_REGION	Los parámetros localAddress y length señalan una región que excede los límites del buffer bloqueado.
ADMXRC3_NO_MEMORY	No se pudo asignar un bloque de control para realizar un seguimiento de la transferencia DMA.

2.5.7 ADMXRC3_ReadDMALockedEx()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_ReadDMALockedEx(
   __in ADMXRC3_HANDLE hDevice,
   __in unsigned int dmaChannel,
   __in uint32_t flags,
   __in ADMXRC3_BUFFER_HANDLE hBuffer,
   __in size_t offset,
   __in size_t length,
   __in uint64_t localAddress
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)
Identificador del dispositivo para realizar la transferencia.

dmaChannel (in)
Especifica qué canal DMA del dispositivo se utilizará para realizar la transferencia.

flags (in)
Flags que modifican el funcionamiento de la función.

hBuffer (in)
Identificador del buffer bloqueado donde escribir los datos leídos del dispositivo.

offset (in)
Offset en el buffer bloqueado donde escribir los datos leídos del dispositivo.
length (in)
Numero de bytes a leer del dispositivo.
localAddress (in)
Dirección local (en bytes) del dispositivo en la que empezar a leer datos.

Descripción

Esta función lee un bloque de datos de un dispositivo en un buffer bloqueado, comenzando en la dirección local especificada. La transferencia de datos se realiza mediante el motor DMA dentro del dispositivo.
Esta función debe obtener un identificador a un buffer ya bloqueado, obtenido llamando a ADMXRC3_Lock previamente.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El identificador del dispositivo no tiene privilegios suficientes para modificar el estado del dispositivo.
ADMXRC3_CANCELLED	Durante la operación, otro hilo llamó ADMXRC3_Cancel() con el identificador del dispositivo o este fue cerrado.
ADMXRC3_HARDWARE_ERROR	Un error de hardware ha ocurrido durante la transferencia a través de la DMA.
ADMXRC3_INVALID_BUFFER_HANDLE	El parámetro hBuffer no es un identificador válido de buffer bloqueado.
ADMXRC3_INVALID_FLAG	El parámetro flags contiene una flag no reconocida
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_INVALID_INDEX	El parámetro dmaChannel especifica un canal que no existe.
ADMXRC3_INVALID_LOCAL_REGION	Los parámetros localAddress y length señalan una región del espacio de memoria del bus local que es inválido.
ADMXRC3_INVALID_REGION	Los parámetros localAddress y length señalan una región que excede los límites del buffer bloqueado.
ADMXRC3_NO_MEMORY	No se pudo asignar un bloque de control para realizar un seguimiento de la transferencia DMA.

2.5.8 ADMXRC3_MapWindow()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_MapWindow(
   __in ADMXRC3_HANDLE hDevice,
   __in unsigned int windowIndex,
   __in uint64_t offset,
   __in uint64_t length,
   __out void** ppVirtualBase
);
```

Los parámetros de esta función son los siguientes:
hDevice (in)
Identificador del dispositivo que contiene la ventana de interés.
windowIndex (in)
Identifica la ventana de interés en el dispositivo.
offset (in)
El offset (en bytes) en la ventana de interés de la región que se va a mapear.
length (in)

Tamaño en bytes de la región a mapear.

ppVirtualBase (out)

Apunta a una variable de tipo void* que va a recibir la dirección donde se asigna la región mapeada en el espacio de direcciones del proceso que llama a la función.

Descripción

Esta función asigna una región de una ventana de memoria al espacio de direcciones del proceso que la llama. Si tiene éxito, el puntero devuelto puede ser utilizado para leer y escribir la región mapeada. La región permanece mapeada hasta que ocurre una de las siguientes situaciones:

- La región se desmapea mediante una llamada a ADMXRC3_UnmapWindow.
- El identificador del dispositivo utilizado para mapear la ventana se cierra mediante una llamada a ADMXRC3_Close.
- El proceso finaliza sin limpiar, en cuyo caso se pierde la asignación, ya que el sistema operativo cierra automáticamente el identificador del dispositivo y se destruye el proceso.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_ACCESS_DENIED	El identificador del dispositivo no tiene privilegios suficientes para modificar el estado del dispositivo.
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_INVALID_INDEX	El parámetro windowIndwx especifica una ventana que no existe.
ADMXRC3_INVALID_REGION	Los parámetros offset y length señalan una región inválida de la ventana especificada.
ADMXRC3_NO_MEMORY	No se pudo asignar memoria para realizar un seguimiento del mapeo.
ADMXRC3_NULL_POINTER	El parámetro ppVirtualBase es un puntero a NULL.
ADMXRC3_REGION_TOO_LARGE	La región especificada por los parámetros offset y length es demasiado grande para ser mapeada en una sola operación.

2.5.9 ADMXRC3_UnmapWindow()

Declaración

```
ADMXRC3_STATUS
ADMXRC3_UnmapWindow(
   __in ADMXRC3_HANDLE hDevice,
   __in void* pVirtualBase
);
```

Los parámetros de esta función son los siguientes:

hDevice (in)

Identificador del dispositivo que contiene la ventana de interés.

pVirtualBase (in)

Dirección virtual base de una región de una ventana que se mapeó mediante ADMXRC3_MapWindow.

Descripción

Esta función desmapea una región de una ventana de memoria del espacio de direcciones del proceso que la llama y es la inversa de ADMXRC3_MapWindow. El proceso debe considerar la región a la que pVirtualBase apunta como inválida tan pronto como se llame a esta función.

Valor devuelto

Un valor de ADMXRC3_SUCCESS indica que la función se ejecutó correctamente. De lo contrario, si se produce un error, se pueden devolver los siguientes valores:

Valor devuelto	Descripción
ADMXRC3_INVALID_HANDLE	El parámetro hDevice no es un identificador de dispositivo válido.
ADMXRC3_INVALID_REGION	El parámetro pVirtualBase no se reconoce como una dirección de memoria base de una región mapeada del dispositivo.
ADMXRC3_NULL_POINTER	El parámetro pVirtualBase es un puntero a NULL.

2.6 DMA Demonstration FPGA Design: Test 0

Una vez se ha decidido el diseño que se va a utilizar como punto de partida, se procede a probar el funcionamiento de este.

2.6.1 Diseño ejemplo

En este punto, es fundamental mencionar que todos los diseños constan de dos partes, el host program, que es el programa que ejecuta el ordenador en el cual está enchufada la placa y la arquitectura de la FPGA. Entre ambos dispositivos, se produce una comunicación, de forma que cada uno se comporta acorde a las órdenes que se encuentran en el host program (para el ordenador) y la arquitectura (para la FPGA). A continuación, se describe su comportamiento.

Arquitectura

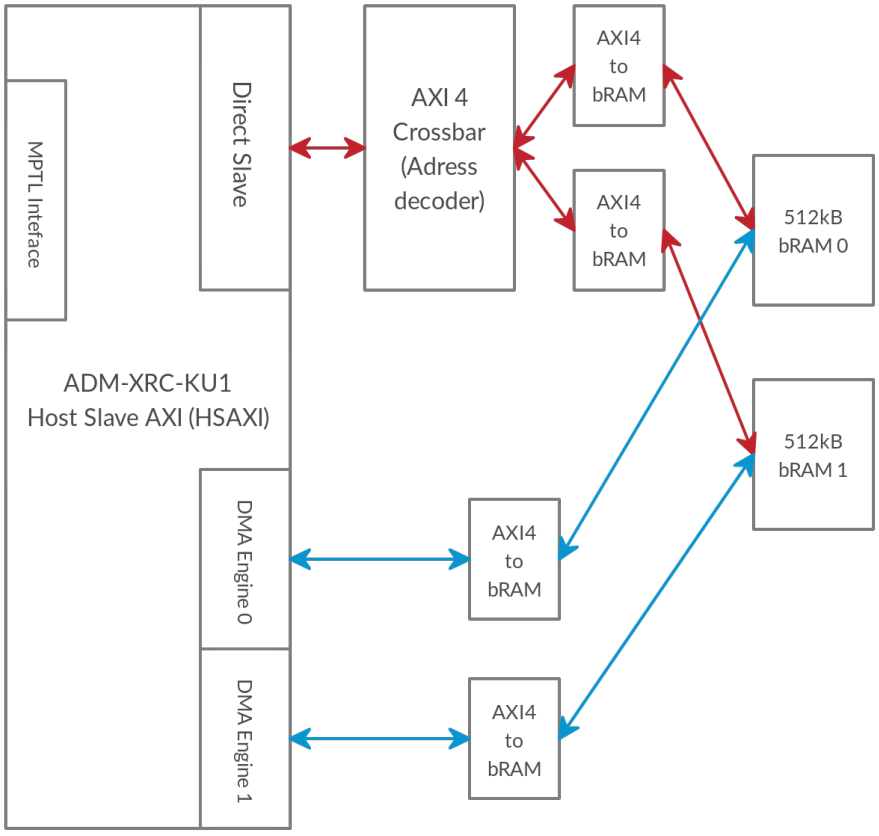

Figura 2.3 Arquitectura Test 0 (dma_demo-admxrcku1)[3].

La arquitectura de la target FPGA en este diseño contiene los siguientes elementos (figura 2.3):

- Una instanciación de la interfaz ADM-XRC-KU1-HSAXI diseñada por Alpha Data. Esta proporciona, entre otras cosas, el canal MPTL (por donde entran/salen los datos de la target FPGA), dos *DMA Engines* y el *Direct Slave*.
- Dos bRAMs de 512kB, una para cada *DMA Engine*. Estas tienen dos canales, de forma que pueden ser utilizadas de forma simultánea por el *DMA Engine* y el host (vía *Direct Slave*).
- Además por cada bRAM hay dos interfaces que convierten de AXI4 a bRAM. El primero permite al *DMA Engine* correspondiente la escritura/lectura y el otro, al *Direct Slave*.
- Por último, hay una interfaz que se encarga de decodificar las direcciones del *Direct Slave* dividiéndolas en dos canales AXI4, uno para cada bRAM.

De esta forma, el mapa de direcciones del *Direct Slave* posee dos regiones, como se puede ver en la siguiente tabla:

Address range	Size	Purpose
0x00000 - 0x0FFFF	512kB	Permite al host leer y escribir en la bRAM 0.
0x80000 - 0xFFFFF	512kB	Permite al host leer y escribir en la bRAM 1.
Others	-	Reservado.

Host Program

Este funciona de la siguiente manera:

1. Abre un dispositivo ADMXRC3 ya sea por índice o número de serie, dependiendo de los argumentos pasados por la línea de comandos.
2. Configura la FPGA de destino con el archivo .bit preconstruido.
3. Lanza un hilo para cada *DMA Engine* que se ha seleccionado para participar en la prueba (según los argumentos de la línea de comandos). Para cada hilo, una estructura, inicializada por el hilo principal, proporciona información sobre el tamaño y la dirección de transferencia de DMA, etc., también según los argumentos de la línea de comandos.
4. El hilo principal ordena a todos los hilos DMA que realicen transferencias DMA de forma continua durante un período especificado por los argumentos de la línea de comandos, con cada subproceso acumulando un recuento de bytes transferidos, y espera a que finalicen todos los subprocesos DMA.
5. El hilo principal verifica los datos transferidos por la última transferencia DMA de cada hilo DMA, informando cualquier error de verificación encontrado.
6. El hilo principal informa las estadísticas de rendimiento de transferencia de DMA, para cada *DMA Engine* y en conjunto.
7. Finalmente, el hilo principal libera la memoria asignada, destruye la sincronización y los objetos del hilo, etc. y cierra el identificador del dispositivo ADMXRC3.

2.6.2 Resultados y conclusiones

Una vez se tiene todo conectado, se procede a probar el ejemplo. Cabe notar que este test permite modificar su comportamiento en función de los argumentos de entrada. Si se ejecuta sin argumentos, este tan solo escribirá 512kB en la bRAM 0 (utilizando tanto el *Direct Slave*, como el *DMA Engine 0*) para testearla y a continuación imprimirá las estadísticas de la transmisión, tal y como se puede ver a continuación:

Código 2.1 Resultados Test 0 (sin argumentos).

```
INFO: Using 1 DMA engine(s): 0
INFO: DMA transfer size is 0x80000(524288) byte(s)
INFO: Testing BlockRAM 0 using Direct Slave channel...
INFO: No errors were detected in initial test of data transfer to and from
    BlockRAMs using Direct Slave channel.
```

```
INFO: Doing DMA performance test...
INFO: 0 data error(s) detected for DMA engine 0
INFO: DMA engine 0 wrote 1498.5 MiB to the FPGA in 2.00014 s at 749.199 MiB/s
INFO: 1 DMA engine(s) transferred 1498.5 MiB to/from the FPGA at 749.199 MiB/s
```

Sin embargo, si se ejecuta con los siguientes argumentos:

```
./dma_demo 0x03 0x02 0x7FFFF
```

Donde 0x03 hace que que se utilizen ambos *DMA Engines*, 0x02 que el primero escriba del host a la FPGA y el segundo al revés y 0x7FFFF es el tamaño de transferencias; se obtienen los siguientes resultados:

Código 2.2 Resultados Test 0 (con argumentos).

```
INFO: Using 2 DMA engine(s): 0, 1
INFO: DMA transfer size is 0x7ffff(524287) byte(s)
INFO: Testing BlockRAM 0 using Direct Slave channel...
INFO: Testing BlockRAM 1 using Direct Slave channel...
INFO: No errors were detected in initial test of data transfer to and from
    BlockRAMs using Direct Slave channel.
INFO: Doing DMA performance test...
INFO: 0 data error(s) detected for DMA engine 0
INFO: 0 data error(s) detected for DMA engine 1
INFO: DMA engine 0 wrote 1781 MiB to the FPGA in 2.00014 s at 890.435 MiB/s
INFO: DMA engine 1 read 1910 MiB from the FPGA in 2.00012 s at 954.942 MiB/s
INFO: 2 DMA engine(s) transferred 3690.99 MiB to/from the FPGA at 1845.38 MiB/
    s
```

De este test, por tanto, se sacan varias conclusiones:
La primera, y más obvia, es que la tarjeta funciona correctamente, lo cual es bastante importante para el desarrollo del proyecto.
Por otro lado, tanto la arquitectura como el *host program* proporcionados por Alpha Data funcionan correctamente y pueden ser modificados y utilizados para conseguir el primer objetivo: realizar lecturas y escrituras en una memoria de la target FPGA de la manera más sencilla posible. En consecuencia, se procede a diseñar un nuevo *host program* que permita alcanzarlo.

2.7 Conclusiones

Tras lo expuesto anteriormente, las conclusiones de este capítulo son las siguientes:
Primero, no se pueden utilizar diseños que aprovechen todos los canales del PCIe para comunicarse con la tarjeta, ya que la programación de forma remota y sin necesidad de reinicio no sería posible. Es por ello que se decide utilizar como punto de partida cualquiera de los ejemplos del *Development Kit* que cumplan esa condición. Más concretamente aquel que implementa el bloque ADM-XRC-KU1-HSAXI.
Además, la tarjeta funciona correctamente, tal y como se ha verificado mediante el Test 0.

3 DMA Engine y Direct Slave

Caos ordenado y orden caótico.

María de Luján

La finalidad de este capítulo, una vez se comprende y conoce el entorno y se ha comprobado que la tarjeta funciona correctamente, es estudiar el funcionamiento del *Direct Slave* y los *DMA Engines* (incluidos en la interfaz ADM-XRC-KU1-HSAXI) y compararlos. En primer lugar, se diseña un test (en C++) capaz de escribir y leer una memoria de la target FPGA de la manera más sencilla posible utilizando ambos métodos.

3.1 Escritura en bRAM via DMA Engine y Direct Slave I: Test 1

Este nuevo test parte del anterior (descrito en la sección 2.6) y consiste en limpiar el diseño del *host program*, con el fin de obtener el mínimo código necesario para escribir en una bRAM a través del *Direct Slave* y leerla mediante de un DMA Engine y viciversa.
Para este test se empleará la misma arquitectura (imagen 2.3), aunque el *DMA Engine 1* no sea utilizado.

3.1.1 Diseño: Host Program

Para su diseño se emplean varias funciones de la API que proporciona Alphadata (desarrolladas en la sección 2.3). Concretamente, realiza lo siguiente:

1. Abre el dispositivo utilizando la función ADMXRC3_Open() (2.5.1), programa la FPGA con la función ADMXRC3_ConfigureFromFile() (2.5.3), mapea la bRAM en la memoria del host con la función ADMXRC3_MapWindow() (2.5.8) y bloquea el buffer para las transferencias a través del *DMA Engine* con la función ADMXRC3_Lock() (2.5.4).
2. Crea un patrón aleatorio de 512kB y lo escribe en la bRAM mediante el *Direct Slave*.
3. A continuación, utiliza la función ADMXRC3_ReadDMALockedEx() (2.5.7) para leer la bRAM y compara lo obtenido con el patrón aleatorio. Imprime los resultados de esta comparación (Test A).
4. Después, realiza el test al revés (Test B): escribe mediante la función ADMXRC3_WriteDMALockedEx() (2.5.6) en la bRAM y la lee mediante el *Direct Slave*. Compara los resultados y los imprime.
5. Por último, desbloquea el buffer del *DMA Engine* con la función ADMXRC3_Unlock() (2.5.5), desmapea la bRAM de la memoria del host y cierra el dispositivo mediante la función ADMXRC3_Close() (2.5.2).

Aunque es muy parecido a lo que se hace en el Test 0, este nuevo *host program* prescinde de los hilos y los argumentos. De este modo, queda mucho más claro y ordenado su funcionamiento.

3.1.2 Resultados y conclusiones

Los resultados que devuelve este test son los siguientes:

Código 3.1 Resultados Test 1.

```
Test A succeeded
pattern: 5cf6ee792cdf05e1ba2b6325c41a5f10
baseaddr (in): 5cf6ee792cdf05e1ba2b6325c41a5f10
buffer (out): 5cf6ee792cdf05e1ba2b6325c41a5f10
Test B succeeded
pattern: 5cf6ee792cdf05e1ba2b6325c41a5f10
baseaddr (in): 5cf6ee792cdf05e1ba2b6325c41a5f10
buffer (out): 5cf6ee792cdf05e1ba2b6325c41a5f10
```

Nota: Solo se imprime una parte del patrón, ya que es suficiente para comprobar visualmente que ha ido bien (el host program lo comprueba entero).
La conclusión que se obtiene de este test es que se puede escribir y leer en memoria utilizando tanto el *Direct Slave* como los *DMA Engines*, mediante las funciones de la API de Alpha Data (sección 2.5) y de manera muy simple.
Una vez se tienen ambos bloques funcionando correctamente se procede a diseñar un nuevo test (Test 2) que permita compararlos.

3.2 Comparación DMA Engine y Direct Slave (Profiling): Test 2

Para comenzar, se modifica Test 1 de forma que utilice el *Direct Slave* para hacer 100 escrituras y lecturas en la bRAM, comprobando que han ido bien y midiendo el tiempo que tarda. Ese número de repeticiones, permite disminuir la desviación típica y por tanto el error estándar. De esta forma los resultados son más fiables.
Un vez se ha realizado el test sobre el *Direct Slave*, se procede a hacer lo mismo con el *DMA Engine*: 100 escrituras, lecturas y comprobaciones, midiendo el tiempo que tarda en hacerlo todo.
De nuevo, se utiliza la misma arquitectura para la target FPGA (descrita en la sección 2.6).

3.2.1 Diseño: Host Program

Este realiza las siguientes funciones:

1. Abre el dispositivo, programa la FPGA con el bitstream correspondiente, mapea la bRAM en la memoria del host y bloquea el buffer para las transferencias a través del *DMA Engine*.
2. Crea un patrón aleatorio de 512kB y lo escribe en la bRAM mediante el *Direct Slave*. A continuación, lee la bRAM utilizando de nuevo el *Direct Slave* y compara lo obtenido con el patrón inicial.
3. Realiza este proceso de escritura/lectura 100 veces midiendo el tiempo que tarda e imprime los resultados (Test A).
4. Posteriormente, realiza lo mismo pero con el *DMA Engine 0* (Test B) e imprime los resultados.
5. Para terminar, desbloquea el buffer del *DMA Engine*, desmapea la bRAM de la memoria del host y cierra el dispositivo.

3.2.2 Resultados y conclusiones

Los resultados obtenidos son los siguientes:

Código 3.2 Resultados Test 2.

```
Test A: mapped: 00:00:12.276743
Test B: buffer: 00:00:00.242816
```

De estos resultados, se concluye que los *DMA Engines* son mucho más rápidos que el *Direct Slave*. Por ello, a partir de ahora se decide prescindir de este último para la transmisión de datos a la target FPGA.

4 Arquitectura FPGA: Manipulación de datos

Neither ever, nor never, goodbye.

APPARAT, GOODBYE

Una vez se conoce cómo diseñar programas que permiten la comunicación con la tarjeta y se ha decidido utilizar los *DMA Engines* para estas transmisiones, se procede a estudiar más a fondo la arquitectura empleada anteriormente.
En consecuencia, la finalidad de este capítulo es familiarizarse con el entorno de programación de la target FPGA y, utilizando Vivado 2016.2, diseñar arquitecturas que manipulen los datos a la salida de la interfaz ADM-XRC-KU1-HSAXI (*DMA Engines*). Si se consigue, se investigará como adaptar la interfaz al sistema de inyección de fallos.
Es importante notar que la simulación del diseño completo no es posible, ya que el *Development Kit* no proporciona ningún modelo de simulación del MPTL, lo cual motiva a trabajar haciendo cambios sencillos y probándolos.
Por otro lado, hay que tener en cuenta que la pérdida de un único dato podría hacer que uno de los extremos de la comunicación se quede colgado esperando.

4.1 Inversión de datos a la salida del *DMA Engine*: Test 3

Para comenzar, se decide diseñar una arquitectura que a la salida del *DMA Engine*, antes de llegar a la interfaz AXI to bRAM, invierta los bytes en bloques de 16, tal y como se puede ver en la siguiente imagen (4.1):

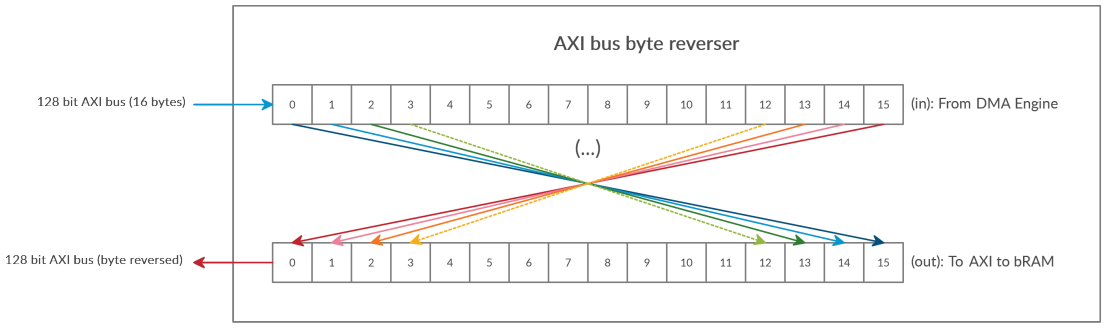

Figura 4.1 AXI Bus byte reverser: Test 3.

Arquitectura

Para su diseño, se parte de la arquitectura utilizada en los Test 0, 1 y 2 (descrita en la sección 2.6).
Primero, se elimina la interfaz AXI4 Crossbar (Adress decoder), las interfaces AXI to bRAM del *Direct Slave*, la bRAM 1 y su bloque AXI to bRAM, así como todas las señales asociadas.
Por otro lado, se añade la interfaz que invertirá los bytes del bus AXI y se conecta a la salida del *DMA Engine* y a la entrada del bloque AXI to bRAM. La arquitectura de la target FPGA queda compuesta por los siguientes elementos:

- La interfaz ADM-XRC-KU1-HSAXI diseñado por Alpha Data, de la que se aprovecha únicamente el *DMA Engine 0*.
- Una bRAMs de 512kB en la que se escriben y de la que se leen los datos.
- Una interfaz que invierte el bus AXI por bytes.
- Por último, una interfaz AXI4 a bRAM que permite la comunicación entre el *DMA Engine* y la bRAM.

En la siguiente figura (4.2) se aprecia la arquitectura resultante, así como las conexiones entre las interfaces descritas en la lista anterior:

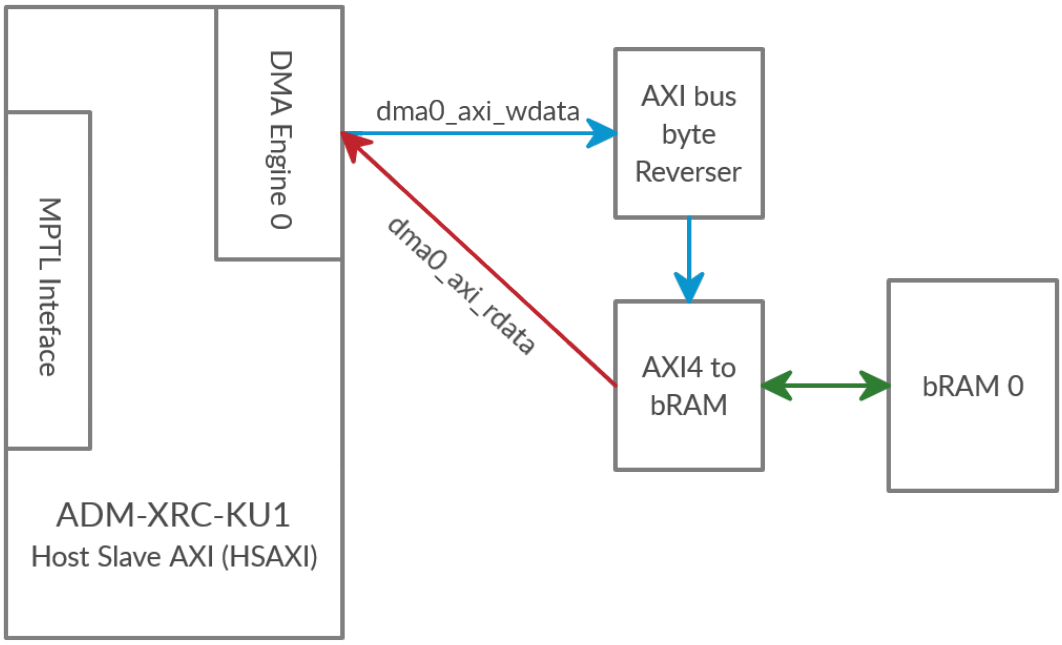

Figura 4.2 Arquitectura Test 3.

Host Program

Realiza lo siguiente:

1. Abre el dispositivo, programa la FPGA con el bitstream correspondiente y bloquea el buffer para las transferencias a través del *DMA Engine 0*.
2. Crea un patrón aleatorio y escribe en la bRAM 0 a través del *DMA Engine*.
3. Lee la bRAM 0 a través del *DMA Engine* e imprime los últimos 16 bytes para comprobar visualmente el resultado.
4. A continuación, imprime el tiempo que ha tardado en hacer la transferencia.
5. Por último, desbloquea el buffer del *DMA Engine* y cierra el dispositivo.

4.1.1 Resultados y conclusiones

El test devuelve lo siguiente:

Código 4.1 Resultados Test 3.

```
Test result: data read
pattern       : 5cf6ee792cdf05e1ba2b6325c41a5f10
DMA read      : 105f1ac425632bbae105df2c79eef65c
DMA duration  : 00:00:00.002411
```

El resultado obtenido verifica que la inversión ha funcionado correctamente, consiguiendo el primer objetivo del capítulo: conocer el entorno de programación de arquitecturas para la target FPGA (Vivado 2016.2).
En este punto, se procede a investigar cómo adaptar la interfaz bajo estudio (ADM-XRC-KU1-HSAXI) al sistema de inyección de fallos.
Estos sistemas necesitan un par de memorias, una a la entrada para los comandos y datos y otra a la salida donde depositar los resultados. Concretamente, utilizan *ftu streams* (buses de datos basados en una memoria FIFO), pero se propone aprovechar la presencia de las bRAMs en las arquitecturas empleadas hasta el momento y se decide estudiar su viabilidad como interfaces de entrada y salida.

4.2 Escritura y lectura de bRAMs conectadas: Test 4

Con el fin de estudiar la viabilidad de las bRAMs como interfaces de entrada/salida en sistemas de inyección de fallos se realiza un test que disponga de dos memorias bRAMs conectadas (para mantener el test lo más simple posible) y que escriba una y lea la otra.
Este consiste en la escritura de la bRAM 0 a través del *DMA Engine 0* y la posterior lectura de la bRAM 1 a través del *DMA Engine 1*, esperando obtener los mismos datos enviados a la bRAM0.
Con este objetivo, se diseña una interfaz en VHDL (Very High Speed Integrated Circuit Hardware Description Language: lenguaje de programación para circuitos hardware) que conecte las memorias y transfiera los datos de una a otra.

Arquitectura

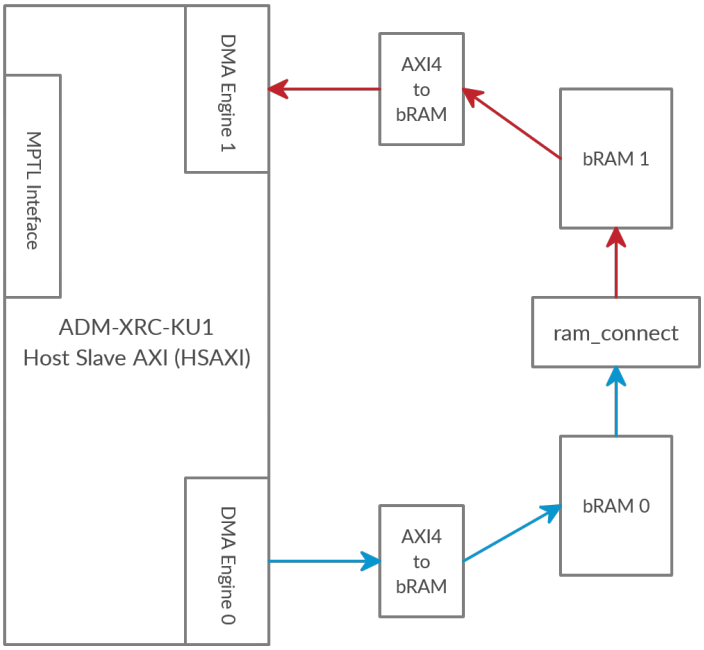

Figura 4.3 Arquitectura Test 4.

Esta arquitectura parte de la descrita en la sección 2.6. De nuevo, se elimina todo lo referente al *Direct Slave* y se dejan ambos *DMA Engines*, las dos bRAMs y sus interfaces AXI to bRAM.
Quedan los siguientes elementos:

- La interfaz ADM-XRC-KU1-HSAXI diseñada por Alpha Data, de la que se aprovechan los dos *DMA Engines*.
- Dos bRAMs de 512kB, una para cada *DMA Engine*, ya que cuentan con dos canales, de forma que pueden ser utilizadas por el *DMA Engine* y la interfaz que copia los datos de una a otra simultáneamente.
- Una interfaz ram_connect que transfiere los datos de una bRAM a la otra.
- Para terminar, por cada bRAM hay dos interfaces que convierten de AXI4 a bRAM. La primera permite al *DMA Engine* correspondiente la escritura/lectura y la segunda al *Direct Slave*.

En la imagen (4.3) se puede ver la distribución de los elementos de la lista anterior y conexiones entre ellos.

Host Program

De nuevo, el diseño del host program se hace a partir de los anteriores y utilizando las funciones de la API (seccion 2.5). Concretamente, realiza lo siguiente:

1. Abre el dispositivo, programa la FPGA con el bitstream correspondiente y bloquea el buffer para las transferencias a través de los *DMA Engines*.
2. Llena la bRAM 1 de ceros.
3. Llena la bRAM 0 de unos.
4. Lee continuamente el dato de la última dirección de la bRAM 1, hasta que este pase de ser un cero a ser un uno.
5. A continuación, lee la bRAM 1 entera comprobando que la transmisión ha ido bien (la bRAM 1 contiene todo unos) e imprime el tiempo que ha tardado el test.
6. Por último, desbloquea el buffer de los *DMA Engines* y cierra el dispositivo.

4.2.1 Resultados y conclusiones

Código 4.2 Resultados Test 4.

```
Test result:
DMA 0 write    : 01010101010101010101010101010101
DMA 1 read     : 01010101010101010101010101010101
```

5 Adaptación a sistema de inyección de fallos

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 investigar cómo adaptar la interfaz bajo estudio al sistema de inyección de fallos.
Considerando que en el capítulo anterior se consiguió manejar dos bRAMs como interfaces de entrada y salida, se propone comenzar diseñando una arquitectura que permita la interpretación de datos y comandos basada en esta tecnología.

5.1 Intérprete de comandos: Test 5

Este test consiste en diseñar una interfaz capaz de leer e interpretar instrucciones y datos almacenados en la bRAM 0, ejecutándolos y depositando los resultados en la otra (bRAM 1).
Para mantener el test lo más sencillo posible, se decide comenzar implementando los siguientes comandos:

- **version**: Devuelve (escribe en la bRAM 1) la versión actual del sistema de inyección de fallos. Ejemplo: "v1.0.0".
- **echo <arg>**: Devuelve lo mismo (<arg>).

Para ello se parte de la arquitectura del test 4 (sección 4.2), sustituyendo la interfaz que copia los datos de una bRAM a otra por una que implementa una máquina de estados capaz de gestionar y ejecutar esas instrucciones.

5.1.1 Estructura comandos

El sistema de inyección de fallos cuenta con una hoja de instrucciones que determina los comandos que es capaz de manejar y la estructura de cada uno de ellos.
Concretamente, se utiliza un identificador único (o código) para cada orden y, en el caso de que sea necesario añadir argumentos, se especifica antes el tamaño de estos en bytes.

Nombre	Código	Argumentos	Devuelve
echo	0x02	i32 size, i8[size] data	i32 size, i8[size] data
version	0x03		i32 size, i8[size] version

Ejemplos

Si se ejecuta el comando "version" y estamos en la versión "v14.23.8", el intérprete escribe lo siguiente en la bRAM 1: "8 v14.23.8". Donde "8" indica el tamaño del dato que viene a continuación (versión en la que se

encuentra el sistema de inyección de fallos).
Por otro lado, si se ejecuta el comando "echo 4 test", la interfaz escribe exactamente lo mismo en la bRAM de salida: "4 test"

5.1.2 Conclusiones

Una vez se conoce la estructura de los comandos, se concluye que su implementación utilizando bRAMs como dispositivos de entrada/salida no es trivial. Esto se debe a que las memorias bRAM, a diferencia de las FIFO, funcionan mediante direcciones, lo cual se traduce en la necesidad de rellenar los huecos libres (en el caso de que los datos no ocupen todo el espacio asociado a la dirección en la que se encuentran). Por tanto, la gestión de esas direcciones así como de los rellenos resulta bastante compleja. Debido a esto, se decide buscar otro método más sencillo.
Como se mencionaba en los resultados y conclusiones del test 3 (sección 4.1.1), los sistemas de inyección de fallos trabajan con *ftu streams* (ver sección 5.3.1). En consecuencia, se propone adaptar los datos que salen de los *DMA Engines* a este bus.
Hay que tener en cuenta que si se consigue, se evita tener que diseñar una nueva interfaz que interprete los comandos, ya que el sistema de inyección de fallos cuenta con una capaz de trabajar con el *ftu stream*.
Por todo lo anterior, se decide afrontar el problema desde esta nueva perspectiva e investigar si es posible diseñar una interfaz que adapte los datos a la salida de los *DMA Engines* (buses AXI) al *ftu stream*.

5.2 DMA Engine: Protocolo AXI4

Para comprobar si es posible la conexión entre el *ftu stream* y los *DMA Engines* es indispensable estudiar a fondo el protocolo AXI4, ya que es utilizado por el bus de salida de estos últimos.
El protocolo AXI4 forma parte de los buses para microcontroladores ARM AMBA. Permite la comunicación a alta frecuencia, mediante *bursts* (ráfagas) de hasta 256 bits con tan solo una dirección.
Las especificaciones indican que la comunicación se produce entre un *master* y un *slave*. Esto implica que el *master* se encarga de la gestión del bus, ya que siempre es el que envía o pide datos al *slave*[4].

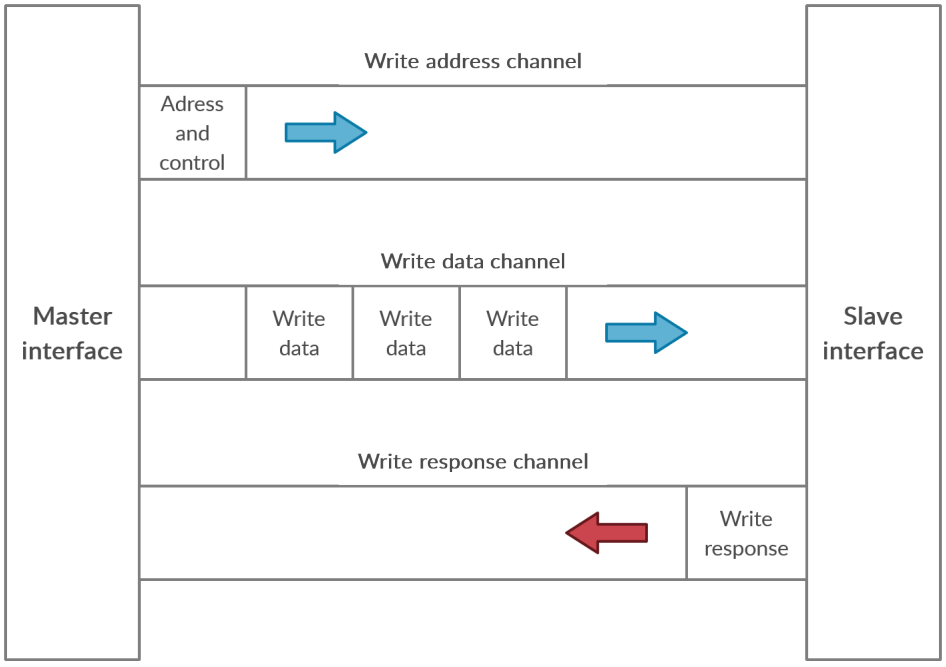

Figura 5.1 Canales Escritura AXI4[4].

En la interfaz bajo estudio (ADM-XRC-KU1-HSAXI), el *DMA Engine* es el *master* del bus AXI, por lo que el *ftu stream* tendrá que actuar como *slave*. Esto último no supone ningún problema, ya que como se verá en

la sección 5.3.1, el *ftu stream* funciona perfectamente como un *slave*.
Hay que destacar que las interfaces AXI4 tienen 5 canales diferentes (tal y como se puede apreciar en las imagenes 5.1 y 5.2), 3 asociadas a la escritura de datos y 2 a la lectura:

- **Read Address Channel** (prefijo "ar")
- **Write Address Channel** (prefijo "aw")
- **Read Data Channel** (prefijo "r")
- **Write Data Channel** (prefijo "w")
- **Write Response Channel** (prefijo "b")

Esta separación de canales permite la escritura y lectura de forma simultánea, lo cual se tendrá en cuenta en el diseño de la interfaz que gestione las transferencias entre el bus AXI4 (del *DMA Engine*) y el *ftu stream*.

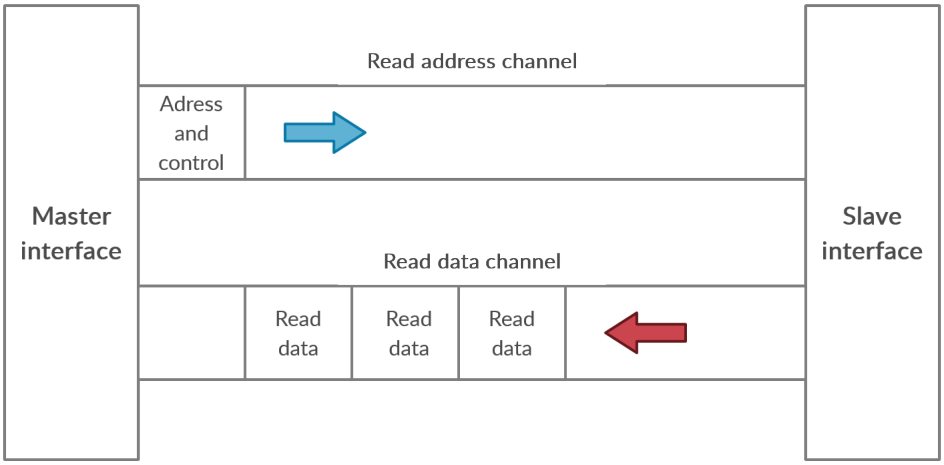

Figura 5.2 Canales Lectura AXI4[4].

5.2.1 Señales

Las señales más importantes que utiliza este protocolo son las siguientes:

Tabla 5.1 Señales canales de escritura[9].

Señal	Origen	Descripción
awaddr	Master	Indica la dirección de la primera transferencia del primer burst (ráfaga)
awburst	Master	Indica el tipo de burst
awsize	Master	Tamaño de cada transferencia (tabla)
awlen	Master	Número de transferencias por burst
awvalid	Master	Indica que hay datos válidos en el canal write address
awready	Slave	Indica que el slave está listo para recibir datos del canal write address
wdata	Master	Datos a escribir
wstrb	Master	Indica qué bytes de wdata son datos válidos
wlast	Master	Indica la última transferencia de un burst
wvalid	Master	Indica que hay datos válidos en el canal write data
wready	Slave	Indica que el slave está listo para recibir datos del canal write data
bresp	Slave	Indica el resultado de la transmisión
bvalid	Slave	Indica que hay datos válidos en el canal write response
bready	Master	Indica que el master está listo para recibir datos del canal write response

Tabla 5.2 Señales canales de lectura[9].

Señal	Origen	Descripción
araddr	Master	Indica la dirección de la primera transferencia del primer burst (ráfaga)
arburst	Master	Indica el tipo de burst
arsize	Master	Tamaño de cada transferencia (tabla)
arlen	Master	Número de transferencias por burst
arvalid	Master	Indica que hay datos válidos enel canal read address
arready	Slave	Indica que el slave está listo para recibir datos del canal read address
rdata	Slave	Datos leidos
rresp	Slave	Indica el resultado de la transmisión
rlast	Slave	Indica la última transferencia de un burst
rvalid	Slave	Indica que hay datos válidos en el canal read data
rready	Slave	Indica que el slave está listo para recibir datos del canal read data

5.2.2 Ejemplo: diagrama de tiempo escritura

A continuación, se presenta un ejemplo de *burst* de escritura acorde al protocolo AXI4, mostrando la evolución de las señales:

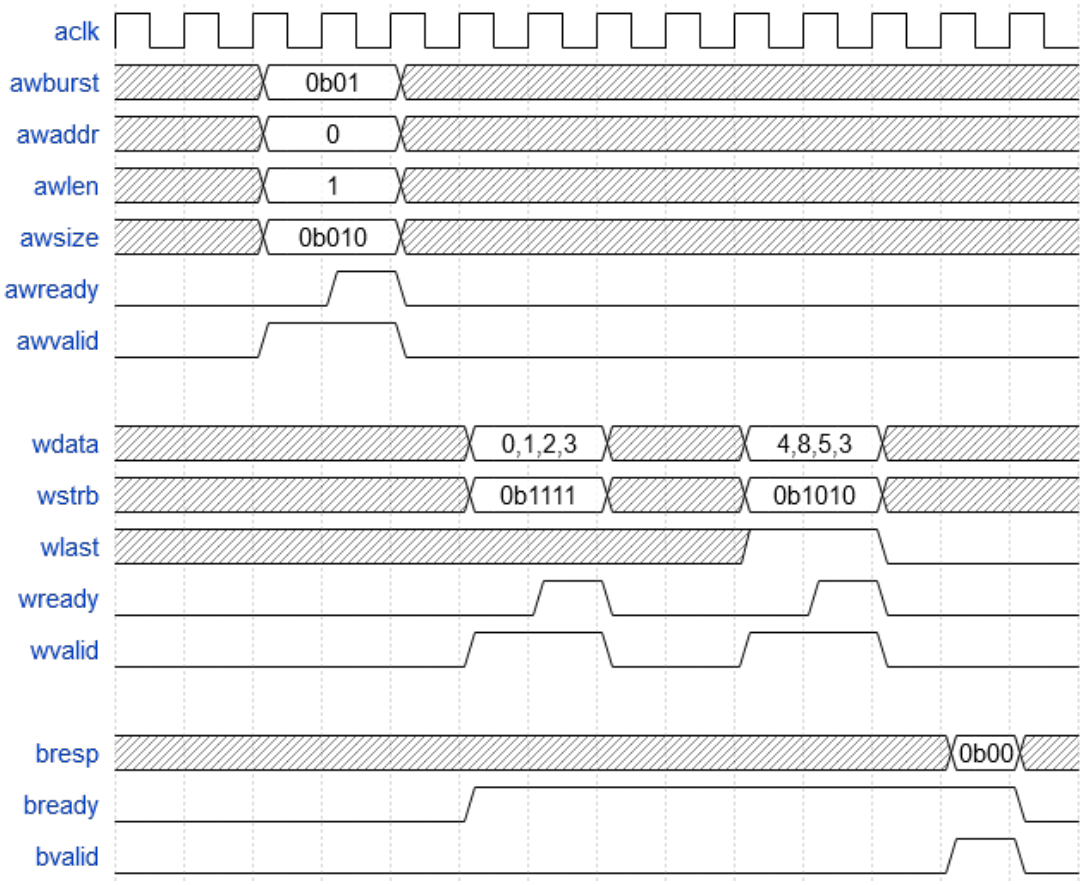

Figura 5.3 Diagrama de tiempo escritura (Protocolo AXI4).

5.2.3 Ejemplo: diagrama de tiempo lectura

En la siguiente figura (5.4) aparece el comportamiento de las señales en un *burst* de lectura de ejemplo:

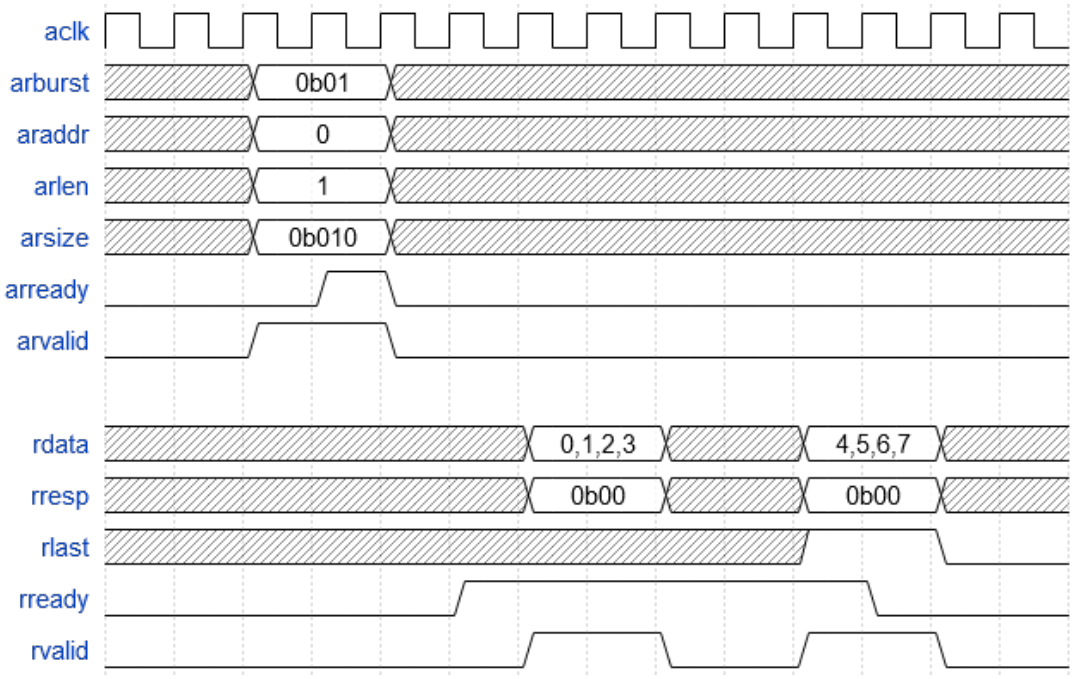

Figura 5.4 Diagrama de tiempo lectura (Protocolo AXI4).

5.3 Ftu stream

El sistema de inyección de fallos cuenta con un *ftu stream* como interfaz de entrada/salida, basado en una memoria FIFO, en la cual se puede leer y escribir de manera simple. A continuación se describe su funcionamiento y estructura.

5.3.1 Escritura en ftu stream

El lado de escritura cuenta con las siguientes señales:

- **wr_ready**: std_logic (out).
- **wr_op**: integer range 0 to 8 (in).
- **wr_data**: std_logic_vector(63 downto 0) (in);

Si **wr_ready** = "1" significa que se pueden guardar datos en el stream. Para ello se tiene que activar **wr_op** indicando el número de bytes a introducir.
Si **wr_op** = 0, no se escribe nada. Si **wr_op = 1**, se escribe un byte (lo que haya en **wr_data(7 downto 0)**), si **wr_op** = 2, se escriben 2 bytes (lo que haya en **wr_data(15 downto 0)**), y así se pueden meter tantos bytes como se necesiten, hasta 8 de golpe (con **wr_op** = 8 se almacenarían los 64 bits que haya en **wr_data**).

5.3.2 Lectura del ftu stream

El otro lado, de lectura, utiliza también tres señales:

- **rd_ready**: std_logic (out);
- **rd_op**: integer range 0 to 8 (in);
- **rd_data**: std_logic_vector(63 downto 0) (out);

Si **rd_ready** = "1" implica el stream está listo para que se le pidan datos. Si se pone **rd_op** a un valor distinto de 0, leerá ese número de bytes (hasta 8 de una vez: si se le pide 1 byte cogerá del 7 al 0, si se le pide

2 cogerá del 15 al 0, etc). Como puede tardar varios ciclos en componer el dato si éste es grande, avisa mediante la señal **rd_ready**: cuando esta se pone a "1" de nuevo que ya está el dato en **rd_data** (en los bits correspondientes, en función del **rd_op** que se le haya dado, es decir, dependiendo de cuántos bytes se hayan pedido).

5.3.3 Otras consideraciones importantes

Es importante tener en cuenta que, aunque el stream esté listo para que se le pidan datos (**rd_ready** = "1"), eso no significa que realmente contenga datos. Si se pide 1 byte y no tiene, o si se piden 4 bytes y sólo tiene 1, se quedará manteniendo **rd_ready** a "0" hasta que le lleguen datos por el lado de escritura y pueda preparar el dato para darlo.
Igualmente si se le da un dato para que lo escriba y la FIFO está llena, lo almacena internamente y se queda manteniendo **wr_ready** = "0" hasta que haya hueco en la FIFO y lo pueda almacenar.
De esa forma se hace el control de flujo en la última plataforma de inyección de fallos: se sabe cuántos datos se envían con cada comando y cuántos se espera recibir, y mientras los datos no estén, las máquinas de estados estarán paradas esperando a los streams.

5.4 AXI to ftu stream: Test 6

El objetivo de este test es crear una interfaz que convierta el protocolo AXI4 (a la salida del *DMA Engine*) a *ftu stream* y comprobar su funcionamiento mediante simulaciones[10]. Para ello, se diseñará una máquina de estados que maneje las transmisiones de datos de uno a otro y viceversa, así como un *test bench* que simule las transmisiones entre buses.

5.4.1 Arquitectura

La arquitectura queda de la siguiente forma:

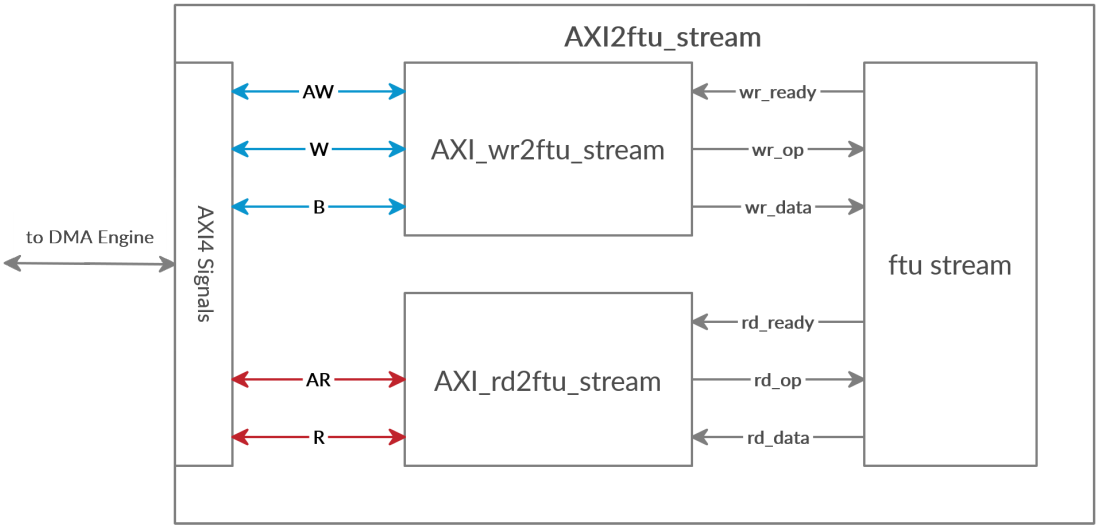

Figura 5.5 Arquitectura Test 6.

5.4.2 Máquinas de estados

Con el fin de gestionar todas las señales descritas en la sección 5.2 (tablas 5.1 y 5.2) y adaptarlas al *ftu stream*, se decide utilizar dos máquinas de estados (figura 5.5). Una de estas máquinas de estado (axi_wr2ftu_stream) se encargará de gestionar los canales de escritura y la otra (axi_rd2ftu_stream) los de lectura. De esta forma el máster (*DMA Engine*) podrá escribir y leer simultáneamente el *ftu stream*.

5.4.3 Resultados simulación

Una vez se ha diseñado la interfaz, se procede a simular transferencias AXI para comprobar su funcionamiento.

Escritura: axi_wr2ftu_stream

Con el objetivo de comprobar el buen funcionamiento de esta máquina de estados, se decide simular la escritura de 16 bytes en el *ftu stream*, es decir, todos los bytes de **wdata** (del bus AXI).
Para ello es necesario configurar las señales del canal AW de la siguiente forma en el test bench:

Señal	Valor	Descripción
awburst	01	Tipo Incremental
awsize	100	$2^4 = 16$ bytes (128 bit) por transferencia
awlen	0	1 Transferencia

De esta forma, en la figura 5.6, se pueden ver los resultados obtenidos. A continuación, se procede a realizar un test algo más complicado. Es este caso, se harán 2 transmisiones de datos (en total 18 bytes) y se utilizará **wstrb** para determinar qué datos son válidos y en qué posiciones se encuentran.
Los resultados de esta segunda simulación se pueden observar en la imagen 5.7.

Lectura: axi_rd2ftu_stream

Una vez se que ha comprobado el bloque que se encarga de gestionar la escritura (acorde al protocolo AXI) no presenta problemas, se procede a comprobar la máquina de estados responsable de la lectura.
Para ello se supondrá que el *ftu stream* tiene almacenado el mismo dato que se escribió en la primera simulación: "FFFFFFFF00000000000000000FFFFFFFF" (ver imagen 5.6), es decir, la FIFO dentro del *ftu stream* contendría los siguientes datos:

FIFO
FFFFFFFF00000000
00000000FFFFFFFF

Por tanto, se simula que el *DMA Engine* pide (mediante el bus AXI4) una transmisión de datos (16 bytes). La señales que se configuran de canal AR en el test bench son las siguientes:

Señal	Valor	Descripción
arsize	100	$2^4 = 16$ bytes (128 bit) por transferencia
arlen	0	1 Transferencia

Para ello, se pide al *ftu stream* 16 bytes (en dos lecturas de 8 bytes). Los resultados de este test se pueden ver en la figura 5.8.

Diseño completo: axi2ftu_stream

Para terminar, se simula una lectura y escritura a través de un *DMA Engine* (protocolo AXI4) de 16 bytes. Los resultados de esta simulación se pueden ver en la imágenes 5.9 y 5.10.

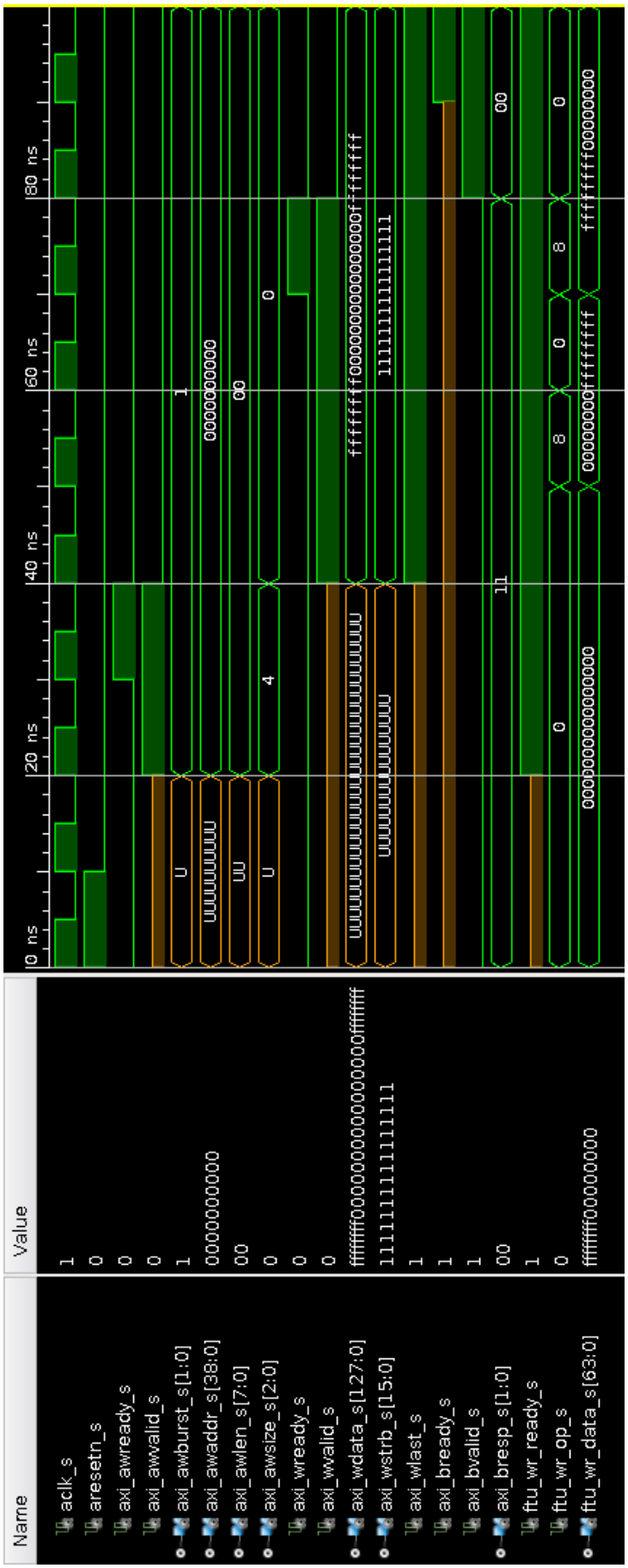

Figura 5.6 Simulación escritura en ftu stream (16 bytes).

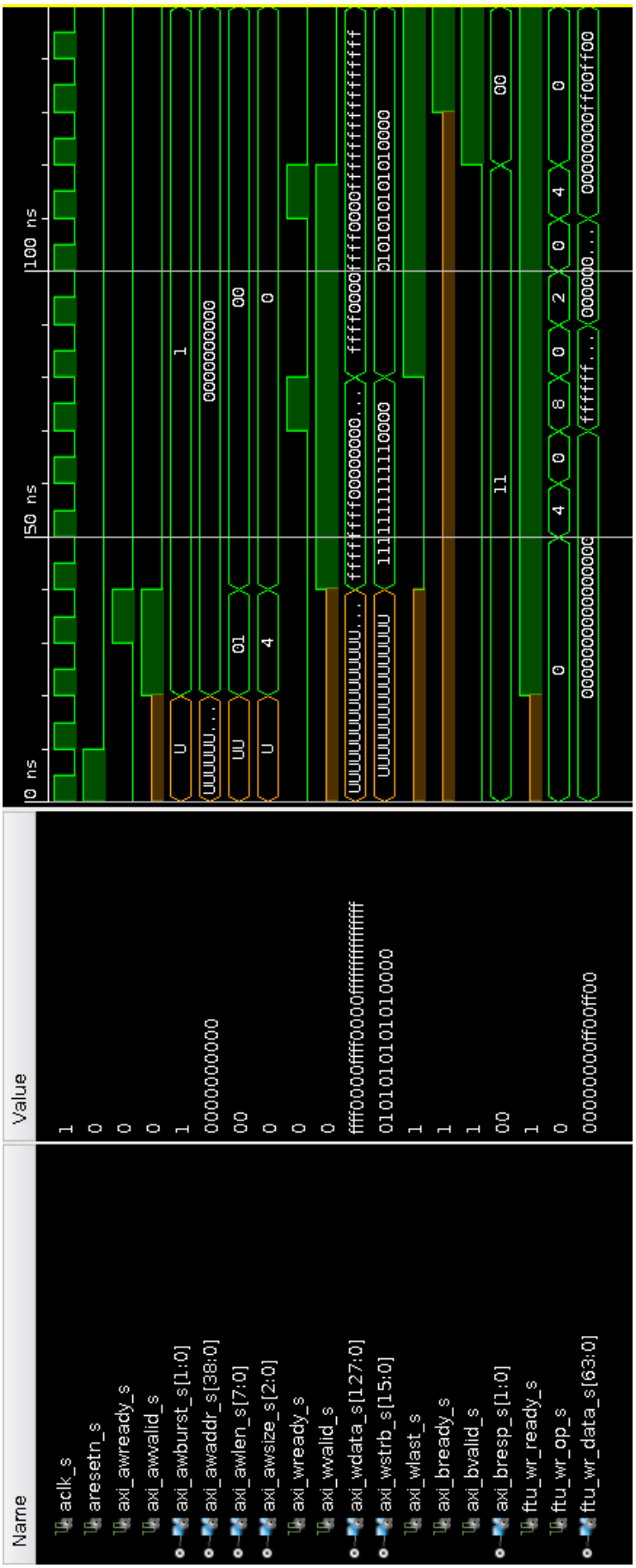

Figura 5.7 Simulación escritura en ftu stream (18 bytes).

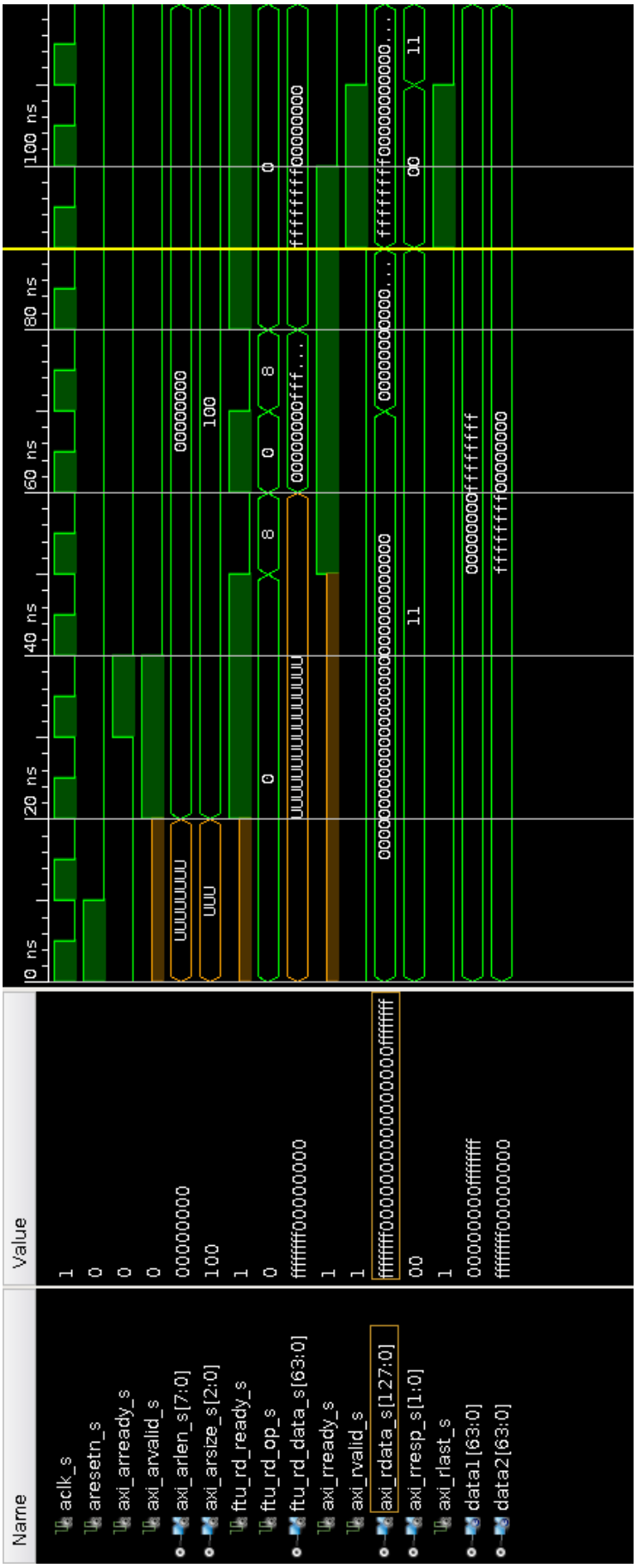

Figura 5.8 Simulación lectura en ftu stream (16 bytes).

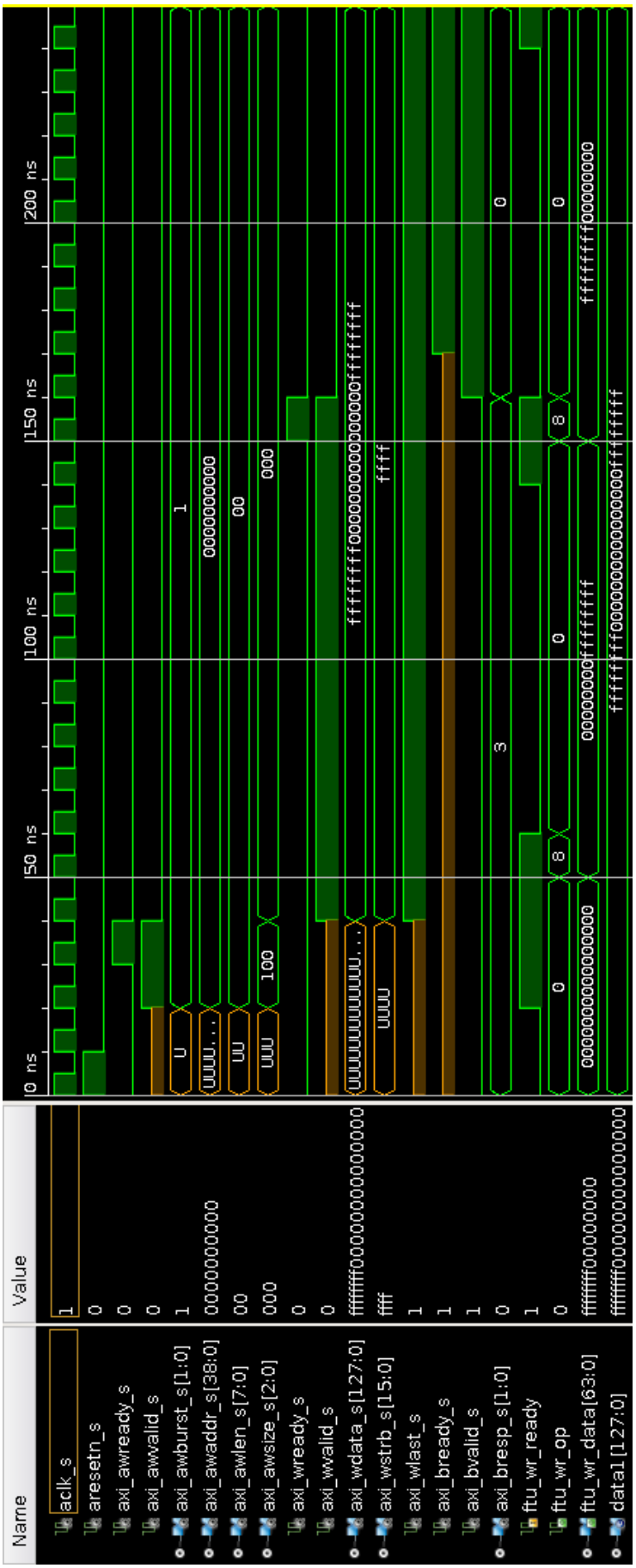

Figura 5.9 Simulación diseño completo (Escritura).

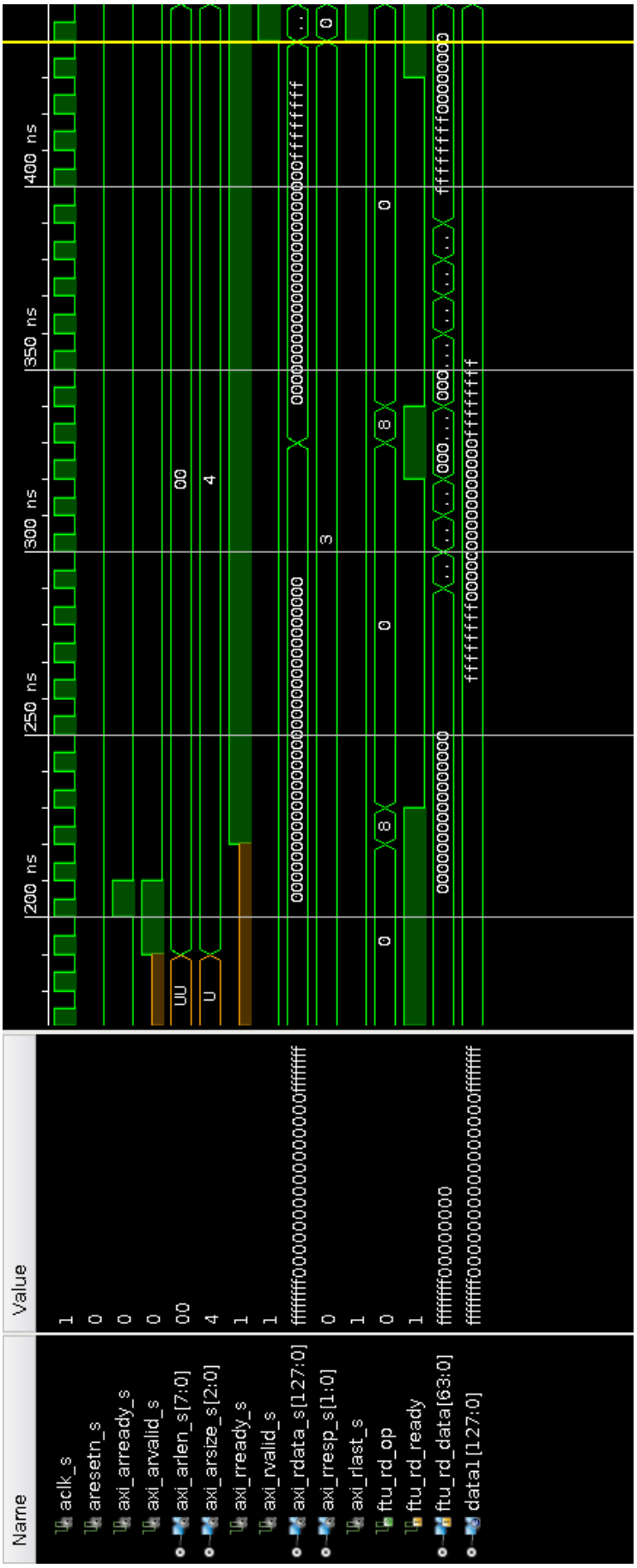

Figura 5.10 Simulación diseño completo (Lectura).

5.4.4 Arquitectura Completa: ADM-XRC-KU1 y FTU

Por último, se propone la arquitectura resultante de implementar la interfaz diseñada en esta sección junto con el sistema de inyección de fallos (FTU) y la interfaz bajo estudio (ADM-XRC-KU1-HSAXI):

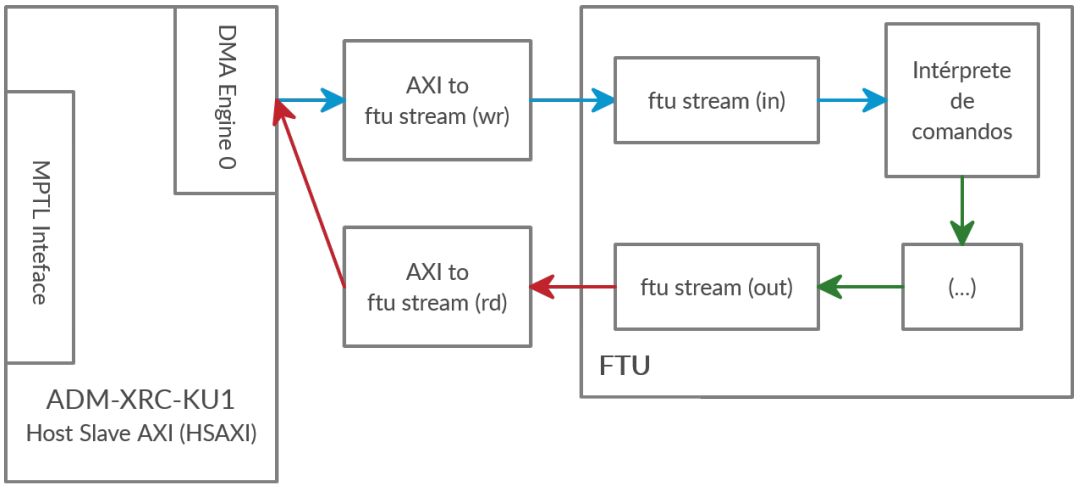

Figura 5.11 Arquitectura Test 6.

5.4.5 Conclusiones

Los resultados de esta prueba son muy relevantes para el objetivo del proyecto, ya que la simulación de la comunicación entre el *DMA Engine* y el *ftu stream* es exitosa. Aún así, sería interesante en un futuro implementarlo en el diseño propuesto en la sección anterior (5.4).

6 Conclusiones y trabajos futuros

No vuelvo a dejar nada para el último día.

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 se pueden llevar a cabo en base a los conocimientos adquiridos y las arquitecturas diseñadas sobre el tema en cuestión.

6.1 Conclusiones

A continuación, se exponen las principales conclusiones obtenidas:
En primer lugar, se concluyó que no se pueden utilizar diseños que aprovechen todos los canales del PCIe para comunicarse con la tarjeta, debido a que la programación de forma remota y sin necesidad de reinicio no sería posible. Por tanto, si se quieren aprovechar estas ventajas se ha de utilizar la interfaz ADM-XRC-KU1-HSAXI.
En segundo lugar, se determinó (mediante comparación) que los *DMA Engines* son mucho más rápidos que el *Direct Slave* a la hora de realizar lecturas/escrituras en una memoria de la target FPGA. A partir de entonces únicamente se utilizaron estos para las transmisiones de datos.
Por otro lado, se ha constatado que las bRAMs pueden ser utilizadas como interfaces de entrada/salida en un sistema de inyección de fallos. No obstante, debido a que presentan varios problemas complejos (solventables) se decidió buscar otros métodos (capítulo 4).
Por último, se ha determinado que es posible adaptar el bus AXI4 de los *DMA Engines* a la interfaz de entrada/salida del sistema de inyección de fallos (*ftu stream*) y se ha diseñado y comprobado mediante simulación una interfaz capaz de realizar dicha adaptación.

6.1.1 Conclusión personal

Trabajar en este proyecto me ha permitido adquirir los conocimientos generales descritos en este documento (sobre la tarjeta, los sistemas de inyección de fallos...), así como las competencias transversales necesarias para su desarrollo. Por ejemplo, desde el principio he necesitado invertir mucho tiempo de investigación, estudio y familiarización con el sistema (ADM-XRC-KU1), al tratarse de un dispositivo muy complejo. Pero esto no ha sido en vano, me siento orgulloso de haber sido capaz de aprender a leer e interpretar la documentación y de diseñar tests que me han permitido poner a prueba esos conocimientos.
Además, este trabajo me ha permitido conocer y manejar el sistema operativo *Debian*, no indispensable, pero sí muy recomendable a la hora de trabajar con estos dispositivos. También me ha permitido aprender utilizar git (sistema de control de versiones) y adquirir unas nociones básicas de C++.
Para terminar, de este proyecto me llevo la oportunidad de formar parte y trabajar en un equipo de investigación (con todo el aprendizaje personal y profesional que esto conlleva), de conocer su estructura y funcionamiento y de aportar mi granito de arena a un proyecto mucho mayor.

6.2 Trabajos futuros

Se propone probar la interfaz diseñada y simulada en el Test 6 (sección 5.4), diseñando una arquitectura que la conecte a un *DMA Engine* de la interfaz ADM-XRC-KU1-HSAXI, tal y como se plantea en la sección 5.4.4. Además, se tendrá que diseñar un test que compruebe el correcto funcionamiento de este diseño.
Por otro lado, se propone continuar el desarrollo de una interfaz que permita la interpretación y ejecución de comandos, así como la gestión de datos, basada en la tecnología bRAM.
Para finalizar, sería interesante la integración con el firmware desarrollado para la versión FTU-VEGAS, que incluye un intérprete de comandos que lee comandos de un ftu_stream y devuelve las respuestas a otro ftu_stream.

Índice de Figuras

Índice de Códigos

Bibliografía

[1] Alfa Data, *ADC-PCIE-XMC Datasheet, Revision 1.0*, https://www.alpha-data.com/pdfs/adc-pcie-xmc_-v1.0.pdf, Online. Accedido el 13 de diciembre de 2019.

[2] _____, *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.

[3] _____, *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.

[4] ARM, *AMBA AXI and ACE Protocol Specification, AXI3, AXI4, and AXI4-Lite, ACE and ACE-Lite*, https://developer.arm.com/documentation/ihi0022/d, Online. Accedido el 9 de mayo de 2020.

[5] J. M. Mogollon, H. Guzmán-Miranda, J. Nápoles, J. Barrientos, and M. A. Aguirre, *Ftunshades2: A novel platform for early evaluation of robustness against see*, 2011 12th European Conference on Radiation and Its Effects on Components and Systems, 2011, pp. 169–174.

[6] María Muñoz Quijada, *Desarrollo de inyección de fallo en memorias de bloque y distribuidas en Virtex 5*, Master's thesis, Escuela Técnica Superior de Ingeniería, Universidad de Sevilla, 2018, pp. 17–22.

[7] Y. J. M. Shirur, K. M. Sharma, and A. A, *Design and implementation of Efficient Direct Memory Access (DMA) Controller in Multiprocessor SoC*, 2018 International Conference on Networking, Embedded and Wireless Systems (ICNEWS), 2018, pp. 1–3.

[8] 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.

[9] Xilinx, *Vivado Design Suite : AXI Reference Guide (UG1037), Version 4.0*, 2017.

[10] Xilinx, *Vivado Design Suite User Guide: Logic Simulation (UG900), Version 2018.3*, 2018.