UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

DISEÑO E IMPLEMENTACIÓN DE UN JAMMER CONFIGURABLE EN FGPA

Autor: Alberto Martínez Cuesta

Tutor: Rafael Gadea Gironés

Tutor de Empresa: José Antonio Armero Torres

Trabajo Fin de Máster presentado en el Departamento de Ingeniería Electrónica de la Universitat Politècnica de València para la obtención del Título de Máster Universitario en Ingeniería de Sistemas Electrónicos

Curso 2019-20

Valencia, Enero de 2020

Resumen

Ante la proliferación actual de vehículos aéreos no tripulados, más conocidos como drones, han convertido a las infraestructuras críticas en vulnerables a los ataques desde el aire. Con el objetivo de conseguir comunicaciones más robustas y a prueba de interferencias, se ha extendido el uso de la modulación por salto en frecuencia en sistemas de comunicaciones militares y sistemas de radiocontrol de vehículos aéreos no tripulados. Esta situación deriva en la necesidad de crear un sistema capaz de interferir estas comunicaciones de manera eficiente con el objetivo de proteger las infraestructuras críticas de posibles ataques externos.

En este trabajo se desarrolla un *jammer* configurable basado en FPGA capaz de interferir las comunicaciones anteriormente expuestas. Para ello, se implementan una serie de módulos capaces de generar una señal de ruido con un ancho de banda limitado y que permiten centrar esta banda en la frecuencia deseada dentro del ancho de banda permitido por la FPGA. De esta manera, se consigue seguir los saltos en frecuencia de la comunicación que se desea interferir.

Finalmente, se muestra la implementación realizada en una FPGA Zynq UltraScale+, donde se presentan los resultados obtenidos, y la comunicación existente entre el diseño realizado en VHDL y el microprocesador presente en la FPGA que permite configurar el diseño.

Resum

Davant la proliferació actual de vehicles aeris no tripulats, més coneguts com a drons, han convertit a les infraestructures crítiques en vulnerables als atacs des de l'aire. Amb l'objectiu d'aconseguir comunicacions més robustes i a prova d'interferències, s'ha estés l'ús de la modulació per salt en freqüència en sistemes de comunicacions militars i sistemes de radiocontrol de vehicles aeris no tripulats. Aquesta situació deriva en la necessitat de crear un sistema capaç d'interferir aquestes comunicacions de manera eficient amb l'objectiu de protegir les infraestructures crítiques de possibles atacs externs.

En aquest treball es desenvolupa un *jammer* configurable basat en FPGA capaç d'interferir les comunicacions anteriorment exposades. Per a això, s'implementen una sèrie de mòduls capaços de generar un senyal de soroll amb una amplada de banda limitada i que permeten centrar aquesta banda en la freqüència desitjada dins de l'amplada de banda permesa per la FPGA. D'aquesta manera, s'aconsegueix seguir els salts en freqüència de la comunicació que es desitja interferir.

Finalment, es mostra la implementació realitzada en una FPGA Zynq UltraScale+, on es presenten els resultats obtinguts, i la comunicació existent entre el disseny realitzat en VHDL i el microprocessador present en la FPGA que permet configurar el disseny.

Abstract

Given the current proliferation of unmanned aerial vehicles, better known as drones, they have made critical infrastructures vulnerables to attacks from the air. In order to achieve more robust and interference-proof communications, the use of frequency hopping modulation has been extended in military communications systems and unmanned aerial vehicle radio control systems. This si-

tuation leads to the need to create a system capable of interfering these communications efficiently in order to protect critical infrastructures from possible external attacks.

This work develops a configurable *jammer* based on FPGA capable of interfering the communications described above. For that, a series of modules are implemented with VHDL. This modules have the ability of generating a noise signal with a limited bandwidth. Also, it is allowed to center this band on the desired frequency within the bandwidth allowed by the FPGA. In this way, it is possible to follow the frequency hops of the communication desired to interfere.

Finally, the design implemented on a Zynq UltraScale+ FPGA is shown, where the results obtained are presented. Additionally, is shown the communication between the design made in VHDL and the microprocessor present in the FPGA that allows the design to be configured.

Índice general

Índice de figuras

Índice de tablas

Parte I

Memoria

Capítulo 1

Introducción

1.1. Motivación

Hoy en día la protección de infraestructuras críticas energéticas, tales como gas, petróleo y electricidad, además de las de transporte y operaciones logísticas, como pueden ser carreteras, ferrocarriles, transporte aéreo y los puertos, se está convirtiendo en una prioridad en todo el mundo en el contexto de inestabilidad socioeconómica actual. Esta situación ha derivado en nuevas legislaciones por parte de la Unión Europea que están orientadas a la protección de infraestructuras críticas y, por tanto, a la prevención, detección y reducción al mínimo de los efectos ante los posibles ataques físicos y cibernéticos a dichas infraestructuras.

En el caso de los ataques físicos, debido a los avances tecnológicos y la proliferación de los vehículos aéreos no tripulados (UAVs por sus siglas en inglés), comúnmente conocidos como drones, han convertido a las infraestructuras críticas en particularmente vulnerables a los ataques desde el aire. El rápido progreso en la tecnología de vehículos aéreos no tripulados, que permite mayor autonomía, mayor altitud, mayor rango de operación, mayor velocidad y aviones no tripulados de tamaño mayor, puede tener un grave impacto desde la perspectiva de la seguridad. El paradigma de las operaciones de aviación ha cambiado drásticamente, ya que la diversidad de las aeronaves, las altitudes de vuelo y la velocidad y sus aplicaciones ha cambiado drásticamente, por lo que su detección y control operacional cada vez más difícil, empujando los límites de la tecnología.

Esta coyuntura ha propiciado que para conseguir comunicaciones más robustas y a prueba de interferencias, se haya implantado la modulación por salto en frecuencia (*Frequency Hopping Spread Spectrum* o FHSS). Mediante esta técnica, se varía la frecuencia central del canal entre un número entero de frecuencias (dentro de un ancho de banda total), reduciendo así las interferencias instantáneas. Este tipo de modulación se ha extendido en sistemas de comunicaciones militares y sistemas de radiocontrol de UAV. Dada esta nueva tesitura, donde al usar la modulación FHSS la frecuencia de central de transmisión varia muy rápido, no es posible hacer uso de los sistemas interferidores tradicionales, o jammers, de banda estrecha como medida de protección. Esto se debe a que por términos de eficiencia y potencia resulta descabellado intentar interferir en el ancho de banda total de salto. Por lo tanto, surge la necesidad de crear un sistema con la capacidad de interferir este tipo de comunicaciones.

1.2. Objetivo

Como consecuencia de la situación descrita en la introducción, surge la necesidad dentro de la empresa DAS Photonics S.L. de la creación de un prototipo de un sistema jammer de RF, capaz de interferir este tipo de comunicaciones, presentando la posibilidad de modificar automáticamente sus parámetros para ajustarse dinámicamente a cada situación. El desarrollo de este sistema se orienta para ser integrado en un sistema de protección de infraestructuras críticas, reduciendo el coste y aumentando las prestaciones respecto a las soluciones tradicionales existentes en el mercado.

Este trabajo fin de máster forma parte de este proyecto, teniendo como objetivo el diseño e implementación de un jammer configurable basado en FPGA. Para ello, se debe generar una interferencia donde se permita configurar sus características para adaptarse a cada estado y así ser efectiva ante la modulación FHSS.

Capítulo 2

Fundamentos teóricos

Durante este capítulo se va a realizar una presentación de la teoría que hay tras este proyecto. Con esto se pretende situar al lector en el contexto en el que se desarrolla este trabajo y dar explicación a los diseños realizados posteriormente. Para ello, en primer lugar, se va a realizar una breve introducción a la guerra electrónica. A continuación, se expondrá que es el *jamming* y distintas técnicas que hay de este, para acabar finalizando con el caso de aplicación de este proyecto, el *follower jammer*.

2.1. Guerra electrónica

La guerra electrónica (EW, del inglés *electronic warfare*) es el nombre que se le aplica a las actividades cuyo objetivo consiste la interceptación o negación de las comunicaciones llevadas a cabo por parte del adversario y, por tanto, de la reducción o impedimento del uso del espectro electromagnético de forma hostil. Se puede dividir en tres campos principales:

- Ataque electrónico (EA, del inglés *electronic attack*)
- Soporte electrónico (ES, del inglés *electronic support*)
- Protección electrónica (EP, del inglés *electronic protect*)

EA, también llamada contramedidas electrónicas (ECM, del inglés *electronic countermeasures*), consiste en el uso de señales activas para evitar que un sistema de comunicación pueda llevar a cabo de manera efectiva su tarea. De forma general, se considera que EA está formado de tres actividades principales: interferencias (*jamming*), engaño y la aplicación de energía dirigida.

De los tres principios principales de la información, relevancia, precisión y puntualidad, el *jamming* se enfoca principalmente en el último de estos. Si la información es transmitida con éxito, hay poco que el *jamming* pueda hacer para afectar de forma directa a la relevancia y precisión de esta. Sin embargo, con el *jamming* se puede atacar a la puntualidad del intercambio de información, al menos, ralentizando esta comunicación. Igualmente, también puede afectar a la relevancia de esta información, porque si llega al destino más tarde de lo que debería esta información se vuelve irrelevante.

En cuanto al engaño, se centra en el segundo de los principios. Su intención es la de engañar al oponente mediante la manipulación de las señales de comunicación que este recibe.

Por último, la aplicación directa de energía es similar al *jamming*, a excepción de que su objetivo es dañar o destruir de forma permanente los equipos de comunicación. Esto requiere mayores cantidades de energía que el *jamming*.

En lo relativo a la parte de soporte electrónico, cumple la función de apoyo para la parte de EA. Se utiliza para medir parámetros del espectro electromagnético y poder detectar la presencia de señales de comunicación y sus características. Sería una perdida de tiempo y de energía si se aplicase *jamming* sobre señales inexistentes.

Para finalizar, EP consiste en los esfuerzos realizados para evitar que algún adversario utilice técnicas de EA y ES en contra de tus comunicaciones [1].

2.2. Comunicaciones contra interferencias

Ya sea la intención interceptar las comunicaciones o simplemente negarlas, en una relación de confrontación siempre hay un interés obvio en impedir el éxito del contrario. Las técnicas de comunicación anti-interferencias –o *anti-jamming*(AJ)– fueron desarrolladas para facilitar las comunicaciones cuando un adversario tiene interés en negar la capacidad de un transmisor de comunicarse. De igual manera, estas técnicas también sirven para evitar la interceptación de las comunicaciones.

Estrictamente hablando, la tecnología de comunicación AJ se refiere a la capacidad de hacer frente a un ataque de *jamming* a un sistema de comunicación. Estar totalmente libre de los efectos del *jamming* de radiofrecuencia en un entorno de comunicación inalámbrica es un objetivo poco realista, dadas las circunstancias adecuadas todos los sistemas de radiofrecuencia pueden ser interferidos. Son técnicas comunes para la implementación del AJ ocultar la señal para que el interceptor no sepa que se encuentra ahí, mover la señal por las distintas frecuencias del espectro de forma rápida para que los interceptores tradicionales de banda estrecha no vean la señal, y tener codificación redundante de las señales digitales. Esta última fue desarrollada de manera inicial para contrarrestar los efectos de ruido en las señales digitales. Además, si la señal de *jamming* se asemeja al ruido térmico, estas técnicas también son efectivas contra él.

2.2.1. Tecnología de espectro ensanchado

Inicialmente, la tecnología de comunicaciones de espectro ensanchado (SS, del inglés *spread spectrum*) fue desarrollada dentro del ámbito militar con el objetivo de impedir la detección, la explotación y los ataques a las transmisiones de información por parte de los adversarios. Aunque en sus primeras fases su aplicación se centraba exclusivamente en este ámbito, rápidamente, su uso se amplió también a aplicaciones comerciales. Un ejemplo sería la técnica de acceso múltiple por división de código (CDMA, del inglés *code division multiple access*) que se basa en la tecnología de espectro ensanchado. Estas aplicaciones comerciales de la tecnología SS harán que su uso se prolongue durante los próximos años.

Gracias al desarrollo de esta tecnología que proporciona cierto grado de protección frente a los ataques electrónicos, como es el *jamming*, los *jammers* tradicionales de una única banda tienen poco efecto en el rendimiento de los sistemas que usan esta tecnología. Esto ha llevado a adoptar

distintas formas de ataque por parte de los *jammers*, provocando que como poco se tengan que preocupar de abarcar un rango de frecuencia mucho más amplio. Aunque hay más tipos de comunicaciones AJ, las dos técnicas que están más extendidas son espectro ensanchado por secuencia directa (DSSS, del inglés *direct sequence spread spectrum*) y espectro ensanchado por salto de frecuencia (FHSS). A continuación, se van a detallar ambas técnicas centrándose especialmente en FHSS por ser la técnica de interés para este proyecto.

2.2.1.1. DSSS

Los sistemas DSSS difunden la señal digital portadora de información a través de un amplio ancho de banda ocupándose todo ese ancho de banda de forma instantánea, es decir, la señal se extiende por todo el ancho de banda al mismo tiempo, véase la figura 2.1. Llevar a cabo esto supone difundir una señal de datos con una cierta energía a través de un ancho de banda muy amplio, provocando que la energía presente en cada frecuencia en particular o en cada banda de frecuencia sea muy pequeña. A menudo, esta energía es tan pequeña como para estar por debajo incluso de la del ruido térmico presente en esas frecuencias. Los receptores que simplemente analicen el espectro a la frecuencia de operación de estos sistemas de comunicación pueden confundir la señal con ruido y, por tanto, fallar en su detección. Es requerido cierto procesado de señal para poder extraer la señal.

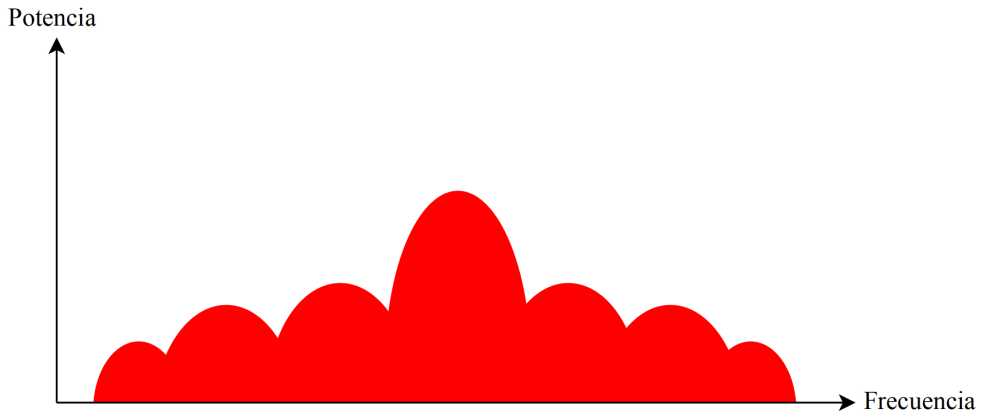


Figura 2.1: Espectro frecuencial de una señal DSSS

2.2.1.2. FHSS

A diferencia de DSSS, en los sistemas FHSS la señal de datos siempre es una banda estrecha que ocupa un solo canal en cada instante de tiempo. En esta técnica de modulación la señal se emite sobre una secuencia de frecuencias pseudoaleatorias entre las que va saltando de forma indefinida. En la figura 2.2 se puede ver un esquema temporal de una señal con modulación FHSS. Este tipo de transmisiones tienen como principal ventaja la resistencia al ruido a las interferencias, además, de ser difíciles de interceptar.

Dentro de los sistemas FHSS, se puede hacer dos divisiones más detalladas. La primera sería el espectro ensanchado por salto de frecuencia rápido (FFHSS, del inglés *fast frequency hopping spread spectrum*) y, la segunda, el espectro ensanchado por salto de frecuencia lento (SFHSS, del

inglés *slow frequency hopping spread spectrum*). Esta distinción normalmente se basa en el número de bits de datos enviados en un salto en particular. Si hay varios bits de datos en un salto, entonces se llama SFHSS, mientras que si hay saltos múltiples para cada bit de datos, entonces se llama FFHSS.

Volviendo al FHSS en general, tiene como virtud la diversidad de frecuencias que usa. Transmitiendo la misma información en diferentes frecuencias incrementa la probabilidad de que la información llegue a su receptor de forma correcta, dado que las múltiples rutas que se pueden usar entre el transmisor y el receptor pueden provocar un desvanecimiento o atenuación de la señal a lo largo del trayecto. Esta atenuación es dependiente de la frecuencia y, si el transmisor o el receptor se están moviendo, esta atenuación cambia con tal movimiento.

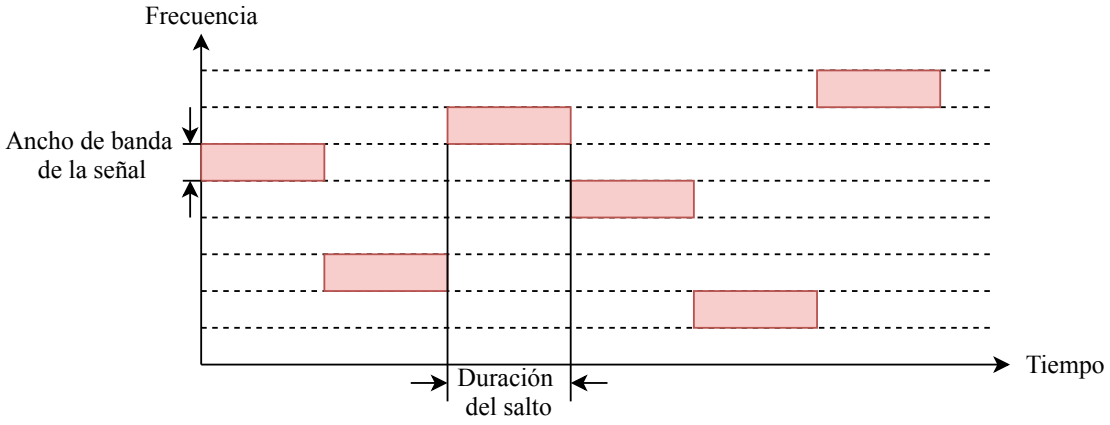


Figura 2.2: Esquema temporal de una señal FHSS

2.3. *Jamming*

Tras exponer en el apartado anterior en qué consiste la guerra electrónica y situar el *jamming* dentro de ella, se puede pasar a dar una explicación más profunda sobre este método y las distintas formas en que se puede aplicar en función del tipo de señal objetivo.

En este punto se debe hacer una pequeña diferenciación. Ya se ha comentado que el *jamming* no son más que interferencias, pero estas vienen motivadas por el objetivo de interrumpir o evitar, por parte de un atacante, la comunicación de un sistema de comunicación inalámbrica. Por otro lado, estarían las interferencias involuntarias que puede ser causadas por otro tipo de comunicaciones existentes emitiendo en el mismo espectro donde ya hay otras comunicaciones presentes. Igualmente, hay que destacar que aunque el *jamming* se usa de forma general como un método de ataque, también existe la posibilidad de que este sirva para contrarrestar ataques de espionaje [2].

2.3.1. Definición de *jamming*

Jamming puede definirse como el uso deliberado de señales de radiofrecuencia con el objetivo de crear interferencias para interrumpir o perturbar otra comunicación presente por radiofrecuencia. Con estas interferencias intencionadas se consigue una disminución de la relación señal-ruido de la

señal de comunicación que llega al receptor. Por tanto, estos ataques se producen en la capa física manteniendo el medio de transmisión ocupado.

2.3.2. Técnicas de *jamming*

En este subapartado se van a presentar las distintas estrategias que un *jammer* puede usar contra objetivos que uses comunicaciones AJ. Un jammer puede usar varias estrategias posibles teniendo cada una sus propias ventajas y desventajas y pudiendo ser la mejor estrategia contra un conjunto particular de objetivos.

Hay dos formas de onda que son típicamente usadas contra los sistemas que usan comunicaciones AJ. Una señal portadora centrada en la frecuencia de transmisión, modulada por una o más señales de tono, o modulada por una señal de ruido, donde ancho de banda del ruido puede ser variado. Cuando la portadora no es modulada, la señal de *jamming* es un simple tono, cuando es modulada por más de un tono, los múltiples son emitidos por el *jammer*. Usualmente, la colocación de estos tonos está basada en el conocimiento previo de algunos parámetros o características de los objetivos que van a ser interferidos. En cuanto al ruido, este es usado para aumentar el ruido de fondo presente en la banda frecuencial donde el sistema objetivo está emitiendo [3].

La clasificación de las distintas estrategias de *jamming* consisten en cómo se distribuye la potencia disponible en el *jammer* por el espectro frecuencial, el tipo de modulación transmitida, como reparte el tiempo el *jammer* entre varios objetivos.

2.3.2.1. *Noise jamming*

Para este tipo de *jamming*, la señal portadora es modulada por ruido aleatorio. La intención es la de interrumpir la comunicación AJ insertando ruido en el receptor. El ancho de banda de la señal puede ser tan amplio como la totalidad del espectro utilizado por el sistema AJ o mucho más estrecho, ocupando solo un canal. Los efectos son diferentes dependiendo de los detalles de la implementación.

Por lo general, se supone que el ruido es gaussiano y, por lo tanto, este es el tipo usado en los generadores de ruido incluidos en los *jammers*. De forma teórica, el ruido gaussiano tiene ancho frecuencial infinito. El ruido gaussiano coloreado es el ruido gaussiano que ha sido sometido a un filtrado y es el tipo apropiado para usar en situaciones donde los efectos de filtrado son importantes.

Dentro de esta categoría se puede hacer el siguiente desglose:

- *Noise jamming* de banda ancha: La energía se reparte por todo el ancho de banda que usa el sistema de comunicación objetivo. Este tipo de *jamming* es útil contra todos los tipos de comunicaciones AJ. En general, también es útil para cubrir un área contra ataque de con propósitos de detección.

  La principal limitación de este tipo de *jamming* es que el nivel de potencia por frecuencia va a ser bajo debido a que esta se reparte en un ancho de banda muy grande. En esencia, lo que se consigue es elevar el nivel de ruido de fondo en el receptor, creando un ambiente de mayor ruido para el sistema AJ; dificultando de esta manera el funcionamiento del sistema de comunicación. Como mínimo, se disminuye el rango sobre el cual el sistema de comunicación es efectivo.

- *Noise jamming* de banda parcial: En este caso, el ruido se sitúa en múltiples canales del espectro, pero sin llegar a ser todos los que usan los sistemas objetivos. Además, estos canales pueden ser o no contiguos.
- *Noise jamming* de banda estrecha: Para este tipo, se coloca toda la energía de la interferencia en un solo canal. El ancho de banda de esta señal de ruido podría ser todo el ancho del canal o podría ser solo el ancho de la señal de datos.

2.3.2.2. *Tone jamming*

En el *tone jamming*, uno o más tonos son estratégicamente situados en el espectro, tanto la situación como el número de tonos afectan directamente al rendimiento de esta estrategia de interferencia. Dentro de esta categoría se puede diferenciar entre dos tipos:

- Tono único: Consiste en un tono situado en una frecuencia concreta. Este tipo de interferencia es totalmente ineficaz ante la modulación FHSS, pero si que puede ser útil contra sistemas que no cambien su frecuencia de operación.
- Múltiples tonos: En este caso, se emite más de un tono que pueden estar colocados aleatoriamente o colocados en frecuencias específicas. Si se supiera que el objetivo es particularmente vulnerable a unos tonos en concreto, es cuando se deben colocar de forma específica para ser más efectivo.

2.3.2.3. *Swept jamming*

Un concepto similar al del *noise jamming* de banda ancha y de banda parcial es el *swept jamming*. Esto es cuando una señal de banda relativamente estrecha y que puede ser tan estrecha como un tono, aunque es más a menudo como una señal de banda parcial, es barrida en el tiempo a través de la banda frecuencial de interés. En un instante de tiempo dado, el *jammer* se centra en una frecuencia y la única región del espectro que es interferida es la región que se encuentra alrededor de esta frecuencia. Sin embargo, dado que la señal va haciendo un barrido en frecuencia, un amplio número de frecuencias pueden ser interferidas en un corto período de tiempo. Por ejemplo, cuando se implementa digitalmente, el *jammer* puede pasar 100 μs por frecuencia antes de pasar a la siguiente. Normalmente, estas frecuencias suelen ser consecutivas, pero esto no tiene por qué ser así, también se pueden seleccionar al azar con sintetizadores digitales.

El efecto total de esta estrategia de interferencia es similar al que se obtiene si se interfieren todas las frecuencias a la vez, excepto que la potencia total del *jammer* se emplea en la frecuencia que se esté interfiriendo en ese instante. También es posible configurar esta estrategia para evitar ciertas bandas de frecuencia que podrían estar en uso por fuerzas amigas. Todo esto solo funciona cuando el tiempo que se permanece en cada frecuencia se adapta al objetivo, de modo que la señal de interferencia esté presente en el receptor el tiempo justo y necesario. Por tanto, se deben tener en cuenta las características de los objetivos a interferir para que esta estrategia sea efectiva.

Como ya se ha mencionado antes, el tiempo es uno de los parámetros más importantes para en un *jammer* de este tipo. El barrido realizado debe ser lo suficientemente rápido como para garantizar que toda la banda a interferir esté cubierta en un periodo corto de tiempo o los cambios de frecuencia se producirán cuando la señal a interferir ya no esté presente.Por otro lado, tampoco

puede ser muy rápido ya que el tiempo que estaría interfiriendo la señal sería muy corto y este no tendría el efecto deseado.

2.3.2.4. *Smart jamming*

En esta categoría de técnicas de *jamming* se intenta interrumpir solo ciertas porciones de señales digitales, seleccionando solo aquellas que son necesarias para negar las comunicaciones. Algunos tipos de sistemas de comunicación deben estar sincronizados para funcionar correctamente, para realizar esta sincronización utilizan un canal concreto. Atacando solo ese canal se podría degradar el proceso de sincronización.

Del mismo modo, algunas comunicaciones AJ requieren de un rastreo del tiempo y la fase de la señal transmitida. Para ello, a veces se utilizan canales separados, los cuales no hacen uso de la tecnología de espectro ensanchado, para este propósito. Algunos sistemas de FHSS usan solo unas pocas frecuencias conocidas para la adquisición de los datos, por lo que los nuevos miembros que quieran unirse a esta comunicación deberán tener alguna manera de conocer ese patrón. Esto hace posible, por lo tanto, un ataque a la adquisición por parte del receptor de la temporización de la señal digital entrante y la información de trama.

Dentro de este apartado se puede realizar otra clasificación, el llamado *deception jamming*. En este caso se enviarían mensajes falsos al receptor como, por ejemplo, ordenes de movimientos. Por último, si se lleva al extremo la definición de *smart jamming* tendríamos lo que se llama *brilliant jamming*. En esta categoría, se intenta cambiar patrones de bits específicos dentro de un mensaje digital para que el receptor reciba un mensaje que es incorrecto pero válido. Para estos dos últimos casos, las mayores limitaciones son que se requiere precisión en la temporización de la comunicación muy elevada, así como un conocimiento previo y total del tipo de señal y de la comunicación que usa el objetivo.

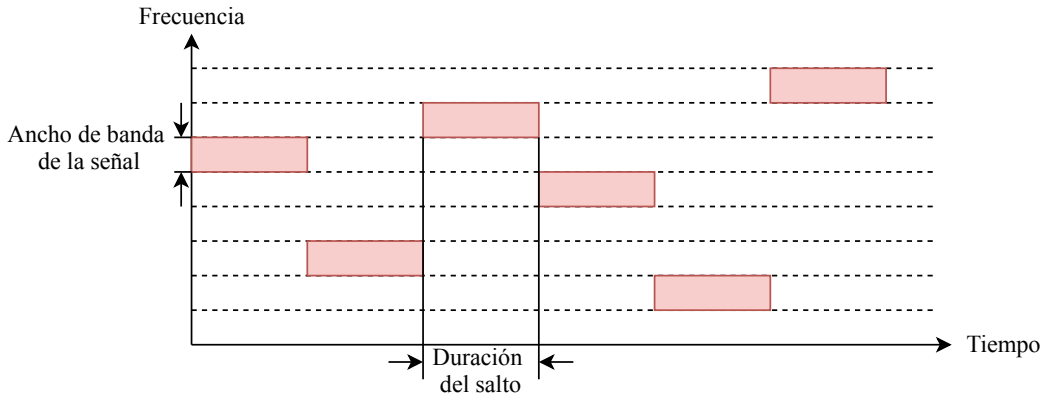
2.3.2.5. *Follower jamming*

Por último, se va a exponer el caso del *follower jammer*. Esta estrategia es la de mayor interés para el presente documento, ya que es la que ha sido seleccionada para implementar.

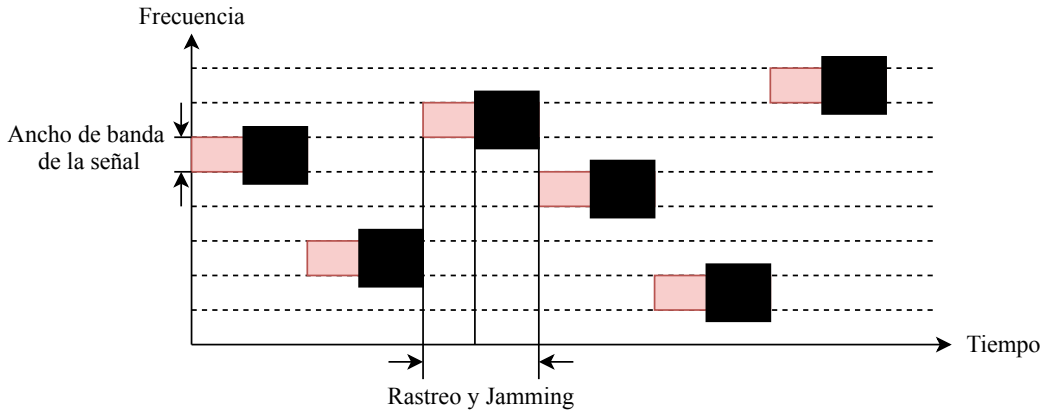
En un *follower jammer* se intenta localizar la frecuencia a la que el transmisor de la señal a interferir está emitiendo, se identifica que la señal es la de interés y se lleva a cabo la interferencia, este proceso se repite en cada salto de frecuencia que haga el transmisor. Para poder detectar la nueva frecuencia a la que ha saltado la comunicación, el *jammer* debe escanear todo el ancho de banda usado por el sistema de comunicación. La forma de onda usada en este *jammer* suele ser una señal de un tono o, también, ese tono modulado por una señal de ruido con un ancho de banda estrecho. La principal ventaja que tiene esta estrategia es el ahorro de energía que se produce al solo tener que interferir un solo canal en cada instante de tiempo. Estas características hacen que esta estrategia sea particularmente efectiva contra los sistemas de comunicación que usan la técnica de modulación FHSS.

Las señales generadas por parte del *follower jammer* van a tener unas características muy similares a la señal que se va a interferir. En la figura 2.3 se muestran los diagramas conceptuales de una señal FHSS y del funcionamiento de esta técnica contra ella, donde se observa el cambio de frecuencias en el tiempo. En la subfigura 5.14d se ejemplifica el procedimiento que se sigue,

analizando la señal objetivo tras cada salto producido y la posterior interferencia.



(a) Diagrama de una comunicación FHSS



(b) Diagrama conceptual del *follower jammer*

Figura 2.3: Efecto del *follower jammer* sobre una comunicación FHSS

Dentro de este *jammer* se tienen dos partes claramente diferenciadas, la primera sería en la que se determina a que frecuencia se ha movido la señal FHSS, esta parte se englobaría dentro del campo de soporte electrónico (ES) que ya se explicó en el apartado 2.1. Por otro lado, se encuentra la parte donde se genera la señal de *jamming* y se coloca en la frecuencia correspondiente, esta parte pertenecería al campo de ataque electrónico (EA). Como ya se ha introducido antes, la determinación de la frecuencia a la que se está produciendo la comunicación se realiza midiendo en todo el espectro donde se encuentra la señal FHSS. En estas mediciones se observa si hay pérdidas o ganancias de energía en alguna o varias frecuencias. Una pérdida de energía implica que el objetivo se ha movido a una nueva frecuencia, mientras que una ganancia de energía en un canal implica que hay una nueva señal presente, que puede o no ser el objetivo.

Del mismo modo que en las técnicas anteriores, esta no está exenta de ciertas limitaciones. Una de las limitaciones más básica es la posición del *jammer* en el espacio respecto al transmisor, lo cual va a imponer unas restricciones de tiempo a la hora de hacer la detección de la señal objetivo y el posterior ataque. Estas restricciones temporales que se deben cumplir vienen marcadas por el tiempo de propagación de las señales y por el tiempo de procesamiento de las señales. El resultado es que el *jammer* solo va a ser efectivo en una cierta región con forma de elipse, en la figura 2.4 se

muestra esta situación [1] [4].

La ecuación que nos marca las restricciones de tiempo a cumplir es la siguiente:

$$\frac{d_2 + d_3}{\nu} + T_j \leq \frac{d_1}{\nu} + \eta T_h \tag{2.1}$$

donde las distancias se corresponden con las mostradas en la figura 2.4, T_j es el tiempo de procesamiento empleado en el *jammer*, T_h es el tiempo que se mantiene la señal FHSS antes de cambiar de frecuencia, η es la fracción del tiempo T_h que debe ser interferida la señal para que sea efectivo, y ν es la velocidad de una onda electromagnética. Reescribiendo la ecuación 2.1, se obtiene la ecuación que determina la elipse con el transmisor y el receptor como focos:

$$\frac{4(x - d_1)^2}{(d_1 + \nu\eta T_h)^2} + \frac{4y^2}{(d_1 + \nu\eta T_h)^2 - d_1^2} = 1 \tag{2.2}$$

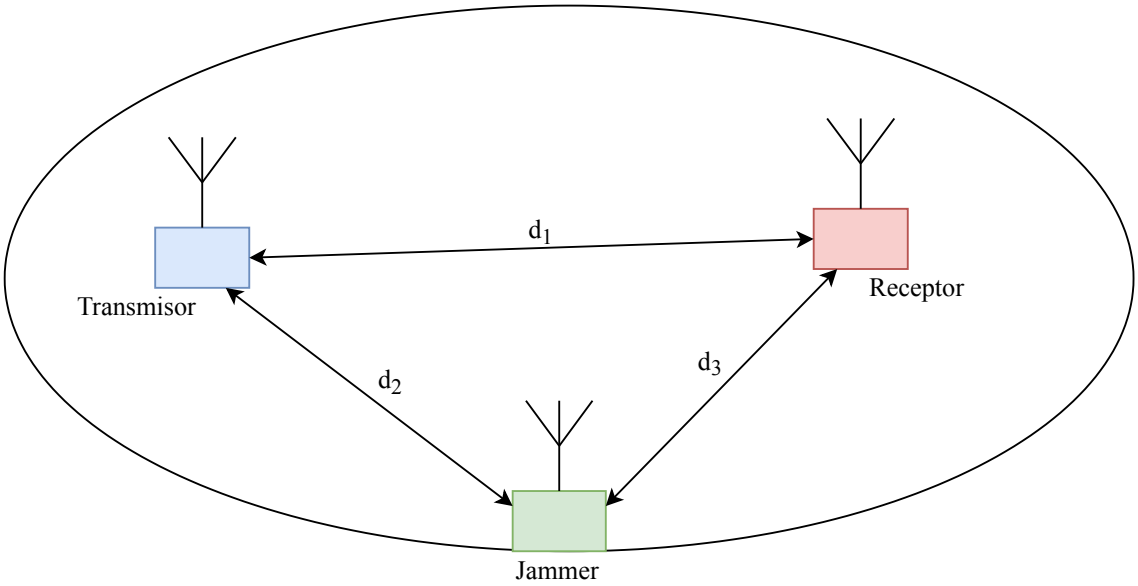


Figura 2.4: Escenario de aplicación del *follower jammer*

El diagrama temporal que seguirían las señales se muestra en la figura 2.5. T_δ representa la diferencia del tiempo de propagación entre la señal que va del transmisor hacia el receptor y la que va del transmisor al *jammer*. Este tiempo podría ser también negativo en el caso de que el *jammer* estuviera más cerca del transmisor que el receptor. El tiempo de procesamiento del *jammer* está marcado por el parámetro T_{proc}. T_{prop} marca el tiempo que tarda la señal del *jammer* en llegar al receptor. Aunque para la mayoría de casos, estos tiempos de propagación son insignificantes comparados con los otros tiempos involucrados en el proceso. Finalmente, el producto ηT_h marca la fracción de tiempo en la que la señal del *jammer* será efectiva sobre el receptor. Este tiempo vendrá marcado, tanto por la diferencia de longitud entre la ruta entre el transmisor y el receptor con la del transmisor y el *jammer* como por la cantidad de tiempo de procesamiento requerido para adecuar la señal de interferencia.

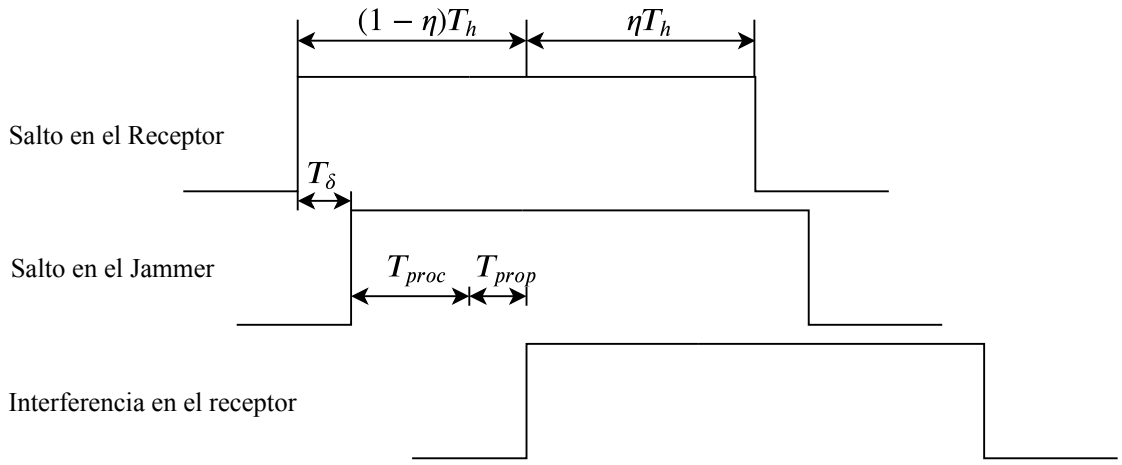


Figura 2.5: Diagrama temporal del FHSS *follower jammer*

Capítulo 3

Metodología y herramientas

En este capítulo se va a exponer la metodología de trabajo empleada, la cual ha sido aprendida en el seno del departamento de FPGA de DAS Photonics. Además, se van a mencionar todas las herramientas empleadas a lo largo del desarrollo, dando una explicación más detallada de una nueva herramienta de testeo empleada en este trabajo, llamada cocotb.

3.1. Metodología

El objetivo principal de esta metodología de trabajo es la de agilizar el desarrollo mediante la automatización de ciertos procesos y crear un estándar de trabajo para facilitar el trabajo en equipo y aumentar la productividad. El flujo de trabajo seguido se muestra en la figura 3.1, en ella se han marcado los distintos puntos que hay dentro del desarrollo y que se van a analizar más adelante. Además, se ven dos partes claramente diferenciadas. En la parte de arriba del dibujo se muestran los servidores disponibles donde se tienen todos los procesos automatizados, mientras que en la parte de abajo se define el trabajo que se realiza en el ordenador personal del desarrollador y que por norma general suelen ser procesos más manuales.

De una forma resumida, el flujo de trabajo de trabajo empezaría con la creación de una *issue* por parte del cliente (1), esto no es más que la definición de un problema o tarea a resolver con una serie de especificaciones a cumplir. Tras esto, el desarrollador recibe esta información y comienza su desarrollo (2). Para este desarrollo, se hace uso de un IDE para la creación de forma más ágil del código necesario. También, hay una parte de testeo que se hace de forma manual, estos tests permiten simular nuestro código para verificar el correcto funcionamiento de este y, además, mediante la herramienta de simulación se obtiene una cobertura de código que nos indica las partes que han sido testeadas de nuestro código. En el siguiente paso, tanto el código como los tests creados se suben al servidor (3) donde, de forma automática, se volverán a ejecutar los tests, pero en este caso serán mas exhaustivos al disponer de una mayor capacidad de cómputo en los servidores. Finalmente, el servidor recibirá los distintos resultados generados y el compilado final poniéndolos a disposición del desarrollador (4 y 5).

A continuación, se van a ir desglosando cada una de las etapas anteriormente mencionadas:

1. Como ya se ha mencionado, el punto de partida sería la creación de una nueva *issue*. Esta tarea llega al desarrollador mediante un servidor donde se dispone de la herramienta *GitLab*.

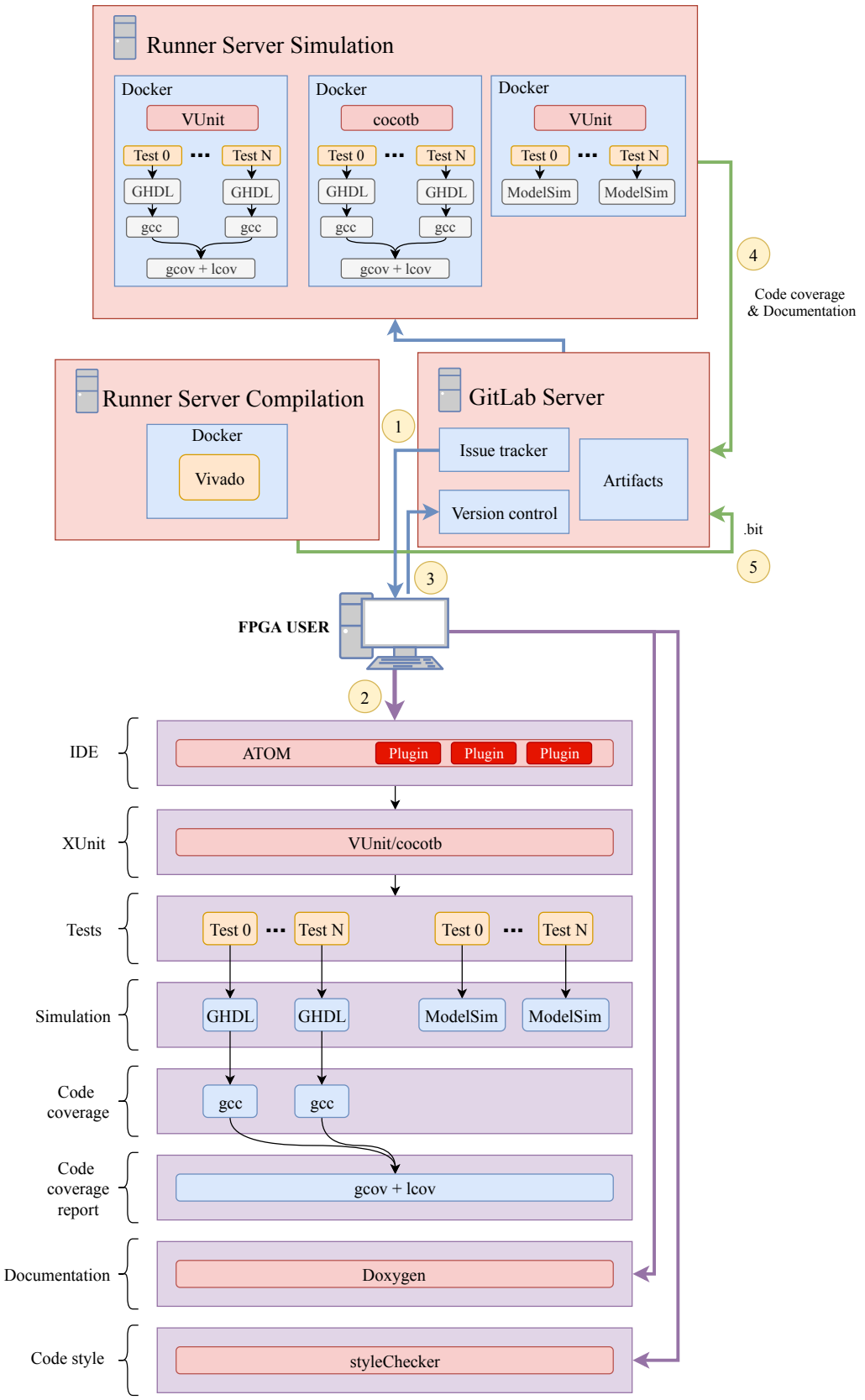


Figura 3.1: Flujo de trabajo empleado

Las fases de desarrollo a seguir se muestran en la figura 3.2. En primer lugar, está la definición de la tarea, en segundo lugar, se realiza el diseño tanto del test como del módulo, el siguiente paso sería el desarrollo de estos y, por último, la aprobación del diseño implementado por parte de una o varias personas del equipo de desarrollo.

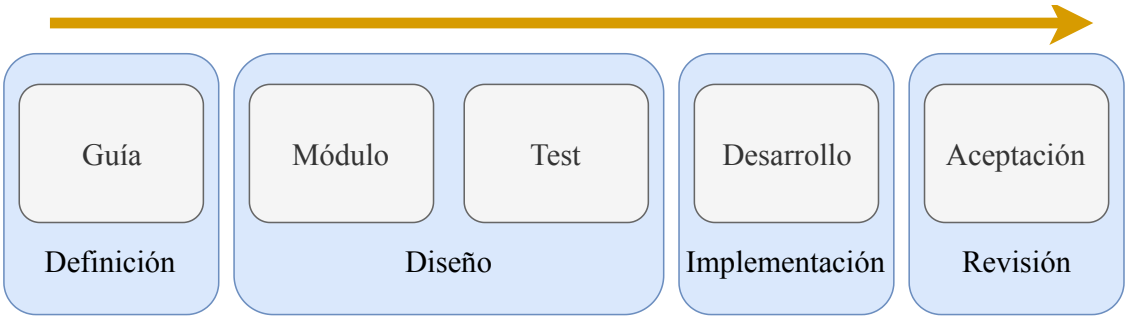


Figura 3.2: Fases de desarrollo

2. En este punto ya empieza el desarrollo del diseño como tal. El IDE (en inglés *Integrated Development Environment*) usado es el Atom, el cual permite añadir numerosos *plugins* que permiten implementar distintas funcionalidades como:

   - Formateo de código
   - Generación de documentación
   - Revisión de sintaxis
   - Automatización de código
   - Abstracción de herramientas
   - Librerías estándar

   Por otra parte, para el desarrollo de todos los tests se utiliza el *framework VUnit*, realizando, en primer lugar, los tests unitarios de cada módulo y escalando posteriormente con los tests de integración del sistema, de forma que se pueda prevenir cualquier tipo de *bug* en el desarrollo. Para este TFM, se ha introducido la novedad de que los test se han realizado con la herramienta cocotb. El objetivo es el aprendizaje y desarrollo de esta herramienta la cual permite agilizar esta parte del desarrollo. Finalmente, para la ejecución de los tests realizados se disponen de dos simuladores, ModelSim y GHDL. En este último se tiene implementada mediante las herramientas *gcov* y *lcov* la cobertura del código que se testea.

3. A partir de este punto ya se entra en la parte de los servidores. Conforme el desarrollador va realizando sus diseños y verificándolos mediante los tests correspondientes estos son subidos al servidor de *GitLab*, donde se tiene un control de versiones basado en Git. Dentro de este servidor se organizan los módulos diseñados en distintos repositorios, y en cada repositorio se mantiene la misma estructura diferenciando entre los archivos de test, los fuentes del módulo y archivos de documentación.

4. Tras la configuración previa, cada vez que se sube código al servidor se ejecutan de forma automática los test definidos para ese módulo. Para ello, el servidor manda el test al *docker* correspondiente, identificando cada docker con una etiqueta en función de su configuración. Tras la ejecución de el o los tests, se envía un informe de vuelta al servidor con los resultados

obtenidos, permitiendo de una forma muy sencilla y visual evaluar si hay algún problema o no. Esta parte sería la llamada integración continua, cuyo objetivo principal es evaluar el código lo más a menudo posible para así poder detectar fallos cuanto antes. En la figura 3.3 se muestra un ejemplo de como se ve la integración continua [5].

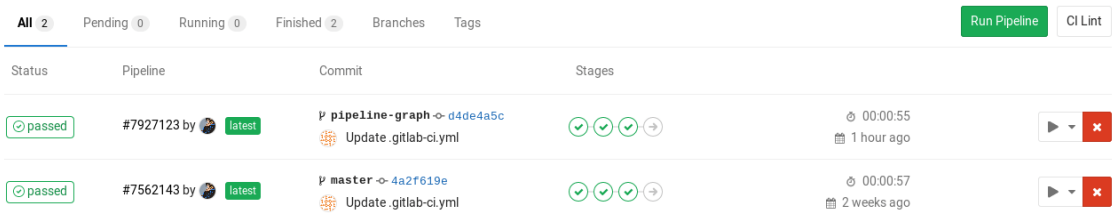


Figura 3.3: *Pipeline* de integración continua en GitLab

5. Finalmente, cuando todos los tests se pasan de forma satisfactoria, el código se envía al *docker* que tiene el software Vivado donde se genera el archivo bitstream.

3.2. Herramientas

3.2.1. Atom

El IDE utilizado es el Atom. La principal ventaja de este IDE radica en la facilidad de personalizarlo y añadir nuevos *plugins* que permiten ampliar las funcionalidades de este. Además, incluye el control de versiones Git integrado en el propio editor. A continuación, se va a mostrar el principal *plugin* utilizado, TerosHDL.

3.2.1.1. TerosHDL

Este *plugin* desarrollado para el Atom tiene como objetivo facilitar el desarrollo de código de lenguajes HDL. Para ello, automatiza ciertos procesos que son repetitivos dentro de un desarrollo y proporciona un entorno visual más agradable. En la figura 3.4 se muestra una captura de como quedaría el IDE con este plugin.

En la barra de herramientas se encuentran las funcionalidades principales, estando cada una de ellas representadas por un icono. En la figura 3.5 se muestra la barra herramientas, la cual sería una representación del flujo de trabajo de izquierda a derecha. Siguiendo este orden, las funcionalidades son:

1. Crear una estructura de repositorio
2. Crear un archivo de *testbench*
3. Crear un archivo de VUnit
4. Ejecutar el test
5. Generar documentación del diseño

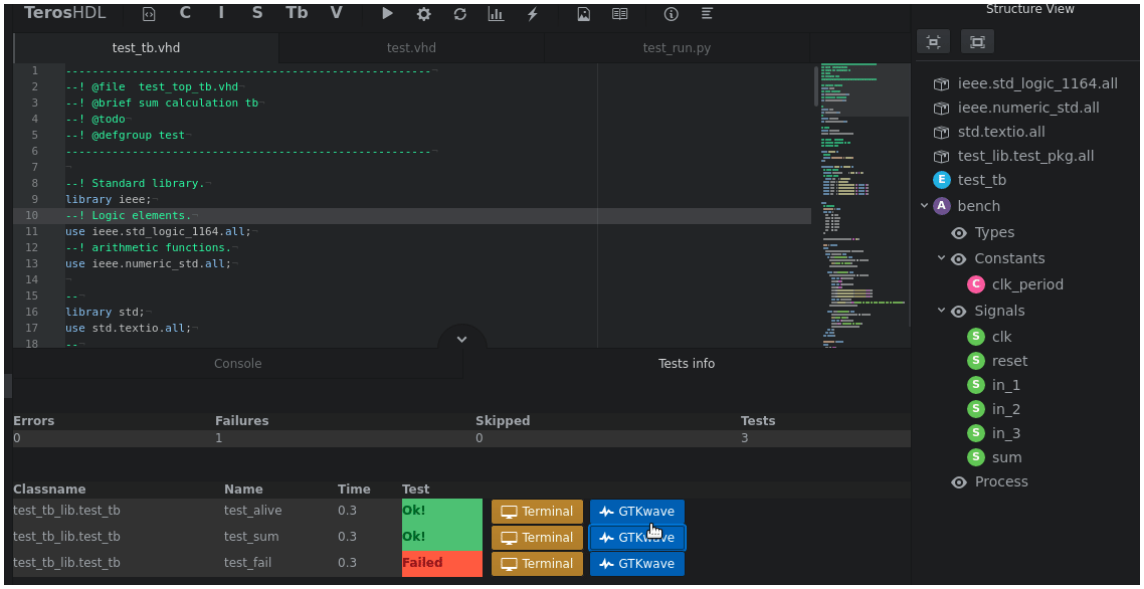


Figura 3.4: Vista general del IDE con TerosHDL

Figura 3.5: Barra de herramientas de TerosHDL

3.2.2. Simuladores

- GHDL: GHDL es una abreviatura de G *Hardware Design Language*. Es un analizador, compilador y simulador de VHDL que puede ejecutar cualquier programa VHDL. Hay que mencionar que esta herramienta es de software libre, trabajando bajo la licencia GPLv2. Además, funciona en GNU/Linux, Windows y macOS.

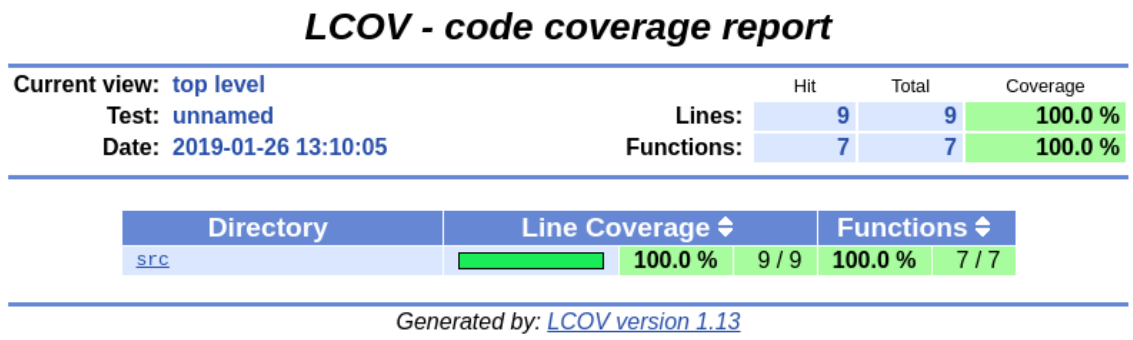
  La versión actual de GHDL no contiene ningún visor gráfico incorporado para poder ver las formas de onda de las señales, pero permite generar archivos GHW, VCD o FST que se pueden ser visualizados con un visor de ondas, como *GtkWave*.

  GHDL tiene como objetivo implementar VHDL según lo definido por la norma IEEE 1076. Admite las versiones de 1987, 1993 y 2002 y, parcialmente, la última, 2008. También, se admiten varios proyectos de terceros como VUnit, UVVM, OSVVM y cocotb [6].

  En conjunto con este simulador, se usan las herramientas *gcov* y *lcov* para poder obtener la cobertura del código que se está testeando. A continuación, se expone cada una de ellas:

  - *gcov*: Es un programa de cobertura de código. Funciona junto con *GCC* para analizar distintos códigos y descubrir partes no probadas de estos. Como resultados se obtiene con qué frecuencia se ejecuta cada línea de código y cuáles son las líneas de código que se han ejecutado. Con estas medidas se tiene un mayor conocimiento de como funciona el código creado, permitiendo realizar distintas optimizaciones si fueran necesarias.
  - *lcov*: Es una interfaz gráfica para la herramienta *gcov*. Recopila los datos generados por *gcov* para múltiples archivos fuente y crea páginas HTML que contienen el código fuente con la información de cobertura. También agrega páginas de información general para facilitar la navegación dentro de la estructura de archivos [7]. En la figura

3.6 se muestra un ejemplo de los resultados proporcionados por esta herramienta. En la vista general se muestran los archivos fuentes y la cobertura de cada uno, haciendo click en cada archivo se puede ver en detalle las veces que se ha ejecutado cada línea de código para el test realizado.



(a) Vista general de la interfaz

```
        :
     54 :   process (clk)
        :   begin
    324 :     if (rising_edge(clk)) then
     90 :       if reset='1' then
     36 :         sum <= (OTHERS => '0');
        :       else
    108 :         sum <= "000" & std_logic_vector(unsigned(ir
        :       end if;
        :     end if;
        :   end process;
        :
        : end rtl;
```

(b) Código fuente marcado

Figura 3.6: Resultados de la herramienta *lcov*

- ModelSim: El otro simulador del que se dispone es ModelSim, el cual soporta tanto VHDL como Verilog, además de añadir otras funcionalidades adicionales que no están disponibles en GHDL. En este caso, es un simlador comercial.

3.2.3. Cocotb

Cocotb es un entorno para verificar código VHDL o Verilog mediante el uso de *Python*. Su nombre viene de *COroutine based COsimulation TestBench* y es que esta es una herramienta de cosimulación, donde el diseño y el test son simulados de forma independiente. Destacar que esta herramienta es completamente gratuita, de código abierto y se encuentra alojada en *GitHub* [8]. Para su funcionamiento requiere de un simulador, los simuladores soportados hasta el momento por cocotb son, en linux, Icarus Verilog, GHDL, Aldec Riviera-PRO, Synopsys VCS, Cadence, ModelSim (DE Y SE) y Verilator. Mientras que en Windows solo Icarus Verilog, Aldec Riviera-PRO y ModelSim.

Esta herramienta nace con el objetivo de agilizar la tarea de testeo y no tener que hacer uso de los lenguajes HDL que son más complicados y que están creados para diseñar hardware y no para testear. El diseño y la verificación de hardware son distintos problemas, es por eso que hay que atacarlos con distintos enfoques. En el caso de los test de verificación se está hablando de software, y es por eso que usar lenguajes de alto nivel para esta tarea hace que sea más fácil llevarla a cabo. La filosofía que se sigue es la misma que en UVM donde se reutiliza diseño y se aleatorizan los test, con la diferencia de que está implementado en Python en vez de SystemVerilog.

3.2.3.1. Funcionamiento

Un *testbecnh* típico de cocotb no requiere código RTL adicional. El diseño bajo test (DUT, en inglés *device under test*) se instancia como el nivel superior en el simulador sin necesidad de ningún código *wrapper*. Cocotb inserta el estímulo a las entradas del DUT y monitoriza las salidas directamente desde Python. En la figura 3.8 se muestra como es la comunicación entre cocotb y el simulador. Para ello hace uso de *Verilog Procedural Interface (VPI)* o *VHDL Procedural Interface (VHPI)*.

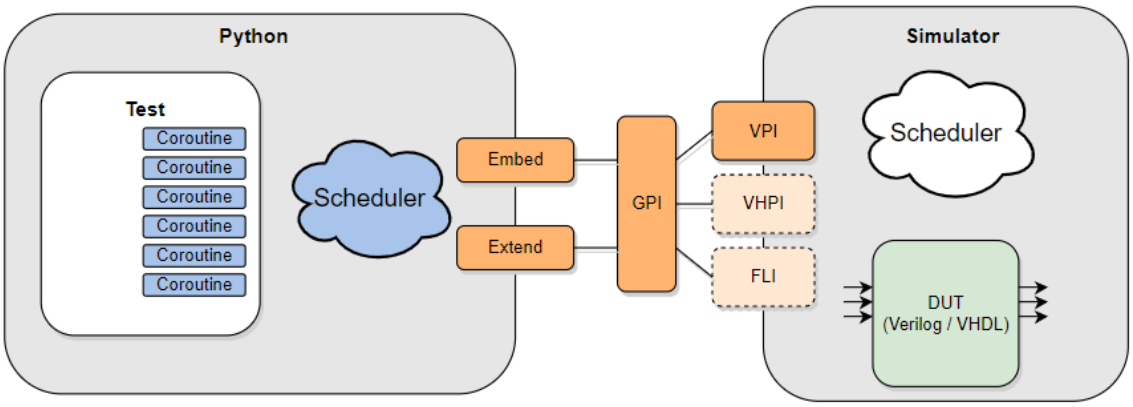


Figura 3.7: Interfaz de comunicación entre el simulador y Python [9]

Un test consiste simplemente en una función de Python. En cualquier momento dado, o se está ejecutando el simulador o el código de Python. La palabra clave *yield* se usa para indicar cuándo pasar el control de ejecución al simulador. Dentro de un test puede haber múltiples corutinas, lo que permite tener distintas rutinas de ejecución independientes.

3.2.3.2. Ejemplo de uso

Para realizar la verificación de nuestro módulo se requieren dos ficheros, un archivo *makefile* y el propio archivo de Python donde se encuentra el test. El primero de ellos sirve para especificar los ficheros que se van a incluir en la simulación, cuál será el *top level*, qué simulador se va a usar y otros tantos parámetros que permiten configurar la simulación que se va a realizar.

Para comprender todo esto mejor se va a mostrar un ejemplo de verificación de un módulo que realiza la suma. El código VHDL se muestra en el código 3.1.

```
library ieee ;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity adder is
generic(
    DATA_WIDTH : positive := 4);
port(
    A : in unsigned(DATA_WIDTH-1 downto 0);
    B : in unsigned(DATA_WIDTH-1 downto 0);
    X : out unsigned(DATA_WIDTH downto 0)
    );
end adder;

architecture RTL of adder is
begin

    process(A, B)
    begin
     X <= resize(A, X'length) + B;
    end process;

end RTL;
```

Código 3.1: adder.vhd

Para realizar la verificación se han usado los códigos 3.2 y 3.3. En el *makefile* mostrado se puede ver la configuración utilizada. El lenguaje va a ser VHDL, el fichero a verificar es el *adder.vhd* el cual va a ser el top level, el archivo que contiene el test se llama test_adder y, por último, el simulador es GHDL, al que se le pasa un argumento extra para que genere un archivo de ondas que se podrá visualizar posteriormente con *GtkWave*.

```
TOPLEVEL_LANG = vhdl
VHDL_SOURCES = $(PWD)/../hdl/adder.vhd
TOPLEVEL=adder
MODULE=test_adder
SIM=ghdl
SIM_ARGS=--wave=salida.ghw
include $(shell cocotb-config --makefiles)/Makefile.inc
include $(shell cocotb-config --makefiles)/Makefile.sim
```

Código 3.2: Makefile

En cuanto al fichero utilizado para la verificación hay un par de puntos claves que merece la pena explicar. En este código se han creado dos tests distintos. Para declarar una función como test se ha de utilizar el decorador *@cocotb.test()*. Otro punto es que la variable *dut* indica la jerarquía de nuestro módulo, gracias a ella se pueden acceder a las distintas señales y leer y escribir sus valores. Con la instrucción *yield Timer(2, units='ns')* se espera a que el simulador avance 2 ns en su simulación. Finalmente, se hace una comprobación de los resultados obtenidos, para ello se lee el valor de la variable de salida mediante *dut.X* y se compara con un modelo creado en Python. En

caso de ser erróneo el resultado, se lanza un mensaje de error y se finaliza la ejecución del test con la instrucción *raise TestFailure*.

```
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure
from adder_model import adder_model
import random

@cocotb.test()
def adder_basic_test(dut):
    """Test for 5 + 10"""
    yield Timer(2, units='ns')
    A = 5
    B = 10

    dut.A = A
    dut.B = B
    yield Timer(2, units='ns')

    if int(dut.X) != adder_model(A, B):
        raise TestFailure(
            "Adder result is incorrect: %s != 15" % str(dut.X))
    else: # these last two lines are not strictly necessary
        dut._log.info("Ok!")

@cocotb.test()
def adder_randomised_test(dut):
    """Test for adding 2 random numbers multiple times"""
    yield Timer(2, units='ns')

    for i in range(10):
        A = random.randint(0, 15)
        B = random.randint(0, 15)

        dut.A = A
        dut.B = B

        yield Timer(2, units='ns')

        if int(dut.X) != adder_model(A, B):
            raise TestFailure(
                "Randomised test failed with: %s + %s = %s" %
                (int(dut.A), int(dut.B), int(dut.X)))
        else: # these last two lines are not strictly necessary
            dut._log.info("Ok!")
```

Código 3.3: test_adder.py

Para finalizar este ejemplo se realiza la ejecución del test. Simplemente basta con tener el archivo *makefile* en el mismo directorio que el archivo de test y escribir el comando *make*. En la

siguiente figura se muestran los resultados obtenidos, este test ha sido ejecutado usando el sistema operativo Linux.

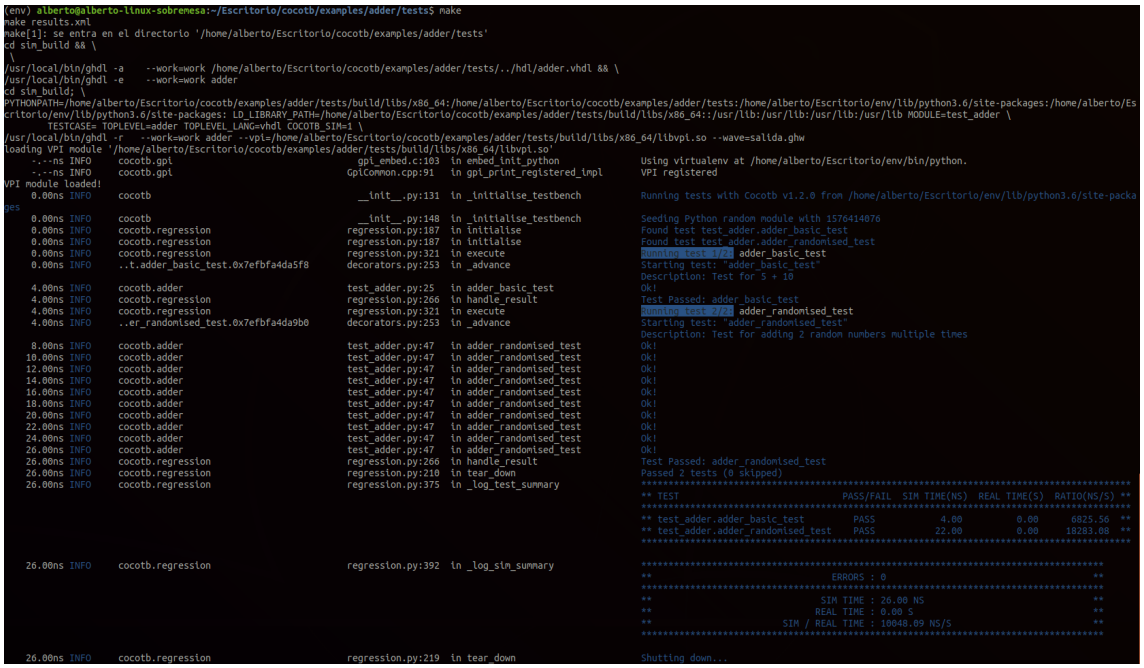


Figura 3.8: Resultados del test con cocotb

Capítulo 4

Diseño implementado

4.1. Introducción

Como ya se mencionó en el subapartado 2.3.2.5, el objetivo final es hacer un sistema *follower jammer*, para ello se plantea el diseño mostrado en la figura 4.1. Dentro de este diseño hay dos partes claramente diferenciadas, la primera sería la parte de detección de la señal objetivo y que entraría dentro del campo de ES; la segunda parte, consiste en interferir esa señal detectada con los parámetros obtenidos en la primera. Esta parte entraría dentro del campo de EA.

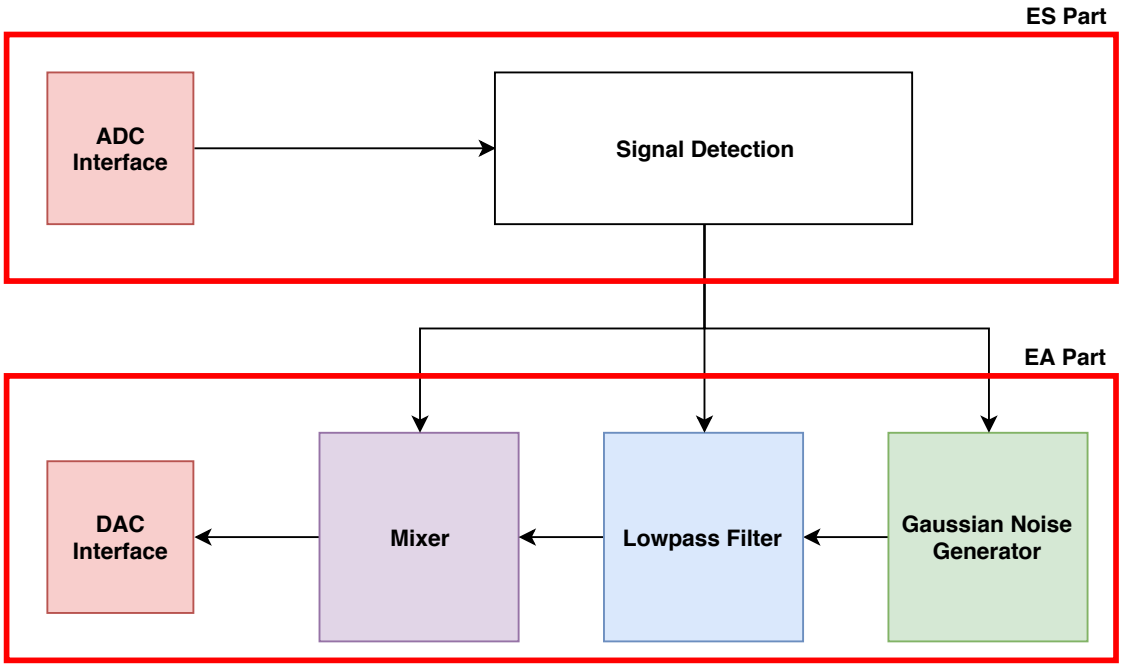


Figura 4.1: Diagrama de bloques del *follower jammer* para la FPGA

El desarrollo de este TFM se ha centrado en la segunda parte, realizando la implementación de un *jammer* configurable que pueda adaptarse a la situación detectada en la primera parte del

sistema. A lo largo de este capítulo se van a desglosar los módulos diseñados para la consecución de esta parte, formada por un generador de ruido, un filtro paso bajo y un módulo encargado de desplazar la banda filtrada a una frecuencia intermedia, a este módulo se le ha denominado mixer.

4.2. Generador de ruido

4.2.1. Descripción del módulo

En este módulo se ha implementado la funcionalidad de un generador de ruido gaussiano. En concreto, el ruido que se va a generar es ruido gaussiano blanco. El ruido que afecta al sistema objetivo puede provenir tanto del entorno natural como del fuentes humanas. El ruido natural se genera por el movimiento aleatorio de electrones libres en un medio conductor. Las fuentes de ruido aleatorio en aplicaciones de comunicación inalámbrica incluyen ruido atmosférico recogido por la antena del sistema y componentes eléctricamente conductores dentro del propio receptor. Todos estos procesos aleatorios siguen una distribución normal o gaussiana. La señal electromagnética resultante, en teoría, tiene una potencia aproximadamente igual en todas las frecuencias, un fenómeno conocido como ruido blanco gaussiano. El uso de ruido gaussiano blanco es por tanto el más adecuado para la práctica del *jamming* ya que imita la distribución de ruido ya presente en el propio receptor del sistema [10].

La estrategia seguida para la generación del ruido gaussiano se basa en el uso de *Linear Feedback Shift Registers* (LFSRs) y la aplicación del teorema del límite central. Mediante el uso de los LFSRs se busca generar números pseudo-aleatorios distribuidos de forma uniforme, este método es bastante sencillo de implementar en una FPGA. Un LFSR es un generador lineal recurrente, este consiste en un registro de desplazamiento formado por *flip-flops* o registros donde cada registro transfiere su valor de salida al siguiente registro en cada ciclo de reloj, mientras que el último registro produce la salida. Por otro lado, el valor del primer registro viene marcado por una realimentación que viene determinada por un polinomio característico. La implementación de esta realimentación se suele hacer con la operación lógica XOR de los registros que define el polinomio característico. En función del número de registros, y por tanto de bits, usados en el LFSR el polinomio característico será distinto. A este polinomio se le denomina primitivo cuando el período de repetición de los números pseudo-aletorios es el máximo posible. En la ecuación 4.1 se muestra el período máximo (T) en función del número de bits usados (n) [11].

$$T = 2^n - 1 \tag{4.1}$$

Tras la obtención de los números aleatorios distribuidos uniformemente hay que realizar una transformación para conseguir una distribución normal o gaussiana, que es la de objetivo. Con el teorema del límite central se consigue llevar a cabo esta transformación. Este teorema establece que las medias de las variables aleatorias convergen en un variable aleatoria con distribución gaussiana [12]. Para implementar la central teorema de límite junto el algoritmo del LFSR mencionado anteriormente, se generan varios conjuntos de datos distribuidos uniformemente y se hace el promedio de todos ellos obteniendo así un conjunto de datos con distribución normal,

4.2.2. Implementación

4.2.2.1. LFSR

El diagrama de bloques del diseño implementado se muestra en la figura 4.2, a esta implementación se le de denomina Fibonacci LFSR.

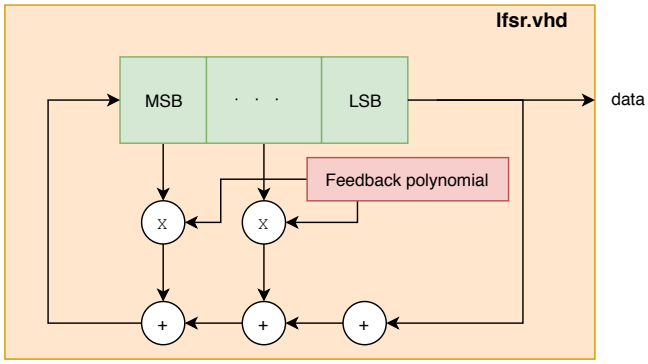


Figura 4.2: Diagrama de bloques del módulo LFSR

En este módulo se genera una señal de un bit pseudo-aleatoria. Como ya se ha comentado en el subapartado anterior, el diseño consta de un registro de desplazamiento donde la entrada es un bit generado mediante la aplicación de una transformación lineal a un estado anterior, esta transformación consiste en sumar unas determinadas posiciones del registro. Para implementar esta transformación, primero, se carga desde un fichero los coeficientes del polinomio característico para el tamaño del registro seleccionado. Cabe mencionar aquí que este módulo está totalmente parametrizado, pudiendo elegir un tamaño del registro de desplazamiento desde 2 bits hasta 64. A continuación, se multiplican todos los bits del registro de desplazamiento con los coeficientes del polinomio característico y, por último, se hace la operación XOR de todas las posiciones del vector resultante para realizar la suma y obtener el bit de realimentación. Además, para poder obtener datos válidos se debe cargar un valor inicial distinto de cero, llamado semilla, en el registro de desplazamiento.

4.2.2.2. *Gaussian Noise generator*

El diagrama de bloques del diseño implementado se muestra en la figura 4.3. Como se puede observar en el diagrama de bloques, en la implementación hecha se instancian 64 módulos LFSR parametrizados con el mismo números de bits y con distinta semilla cada uno. De cada modulo LFSR se obtiene un bit de salida, siendo en total 64 bits con los que se forman cuatro palabras de 16 bits. De este modo, se obtienen cuatro señales aleatorias distribuidas uniformemente e independientes entre sí. Posteriormente, se realiza el promedio de las cuatro señales. Para ello, se suman las señales entre sí y se realiza un desplazamiento hacia la derecha de dos posiciones con el objetivo de generar la división por 4. Con estas dos operaciones se consigue aplicar el teorema del límite central, consiguiendo así una única señal de 16 bits de números aleatorios distribuidos de forma normal o gaussiana.

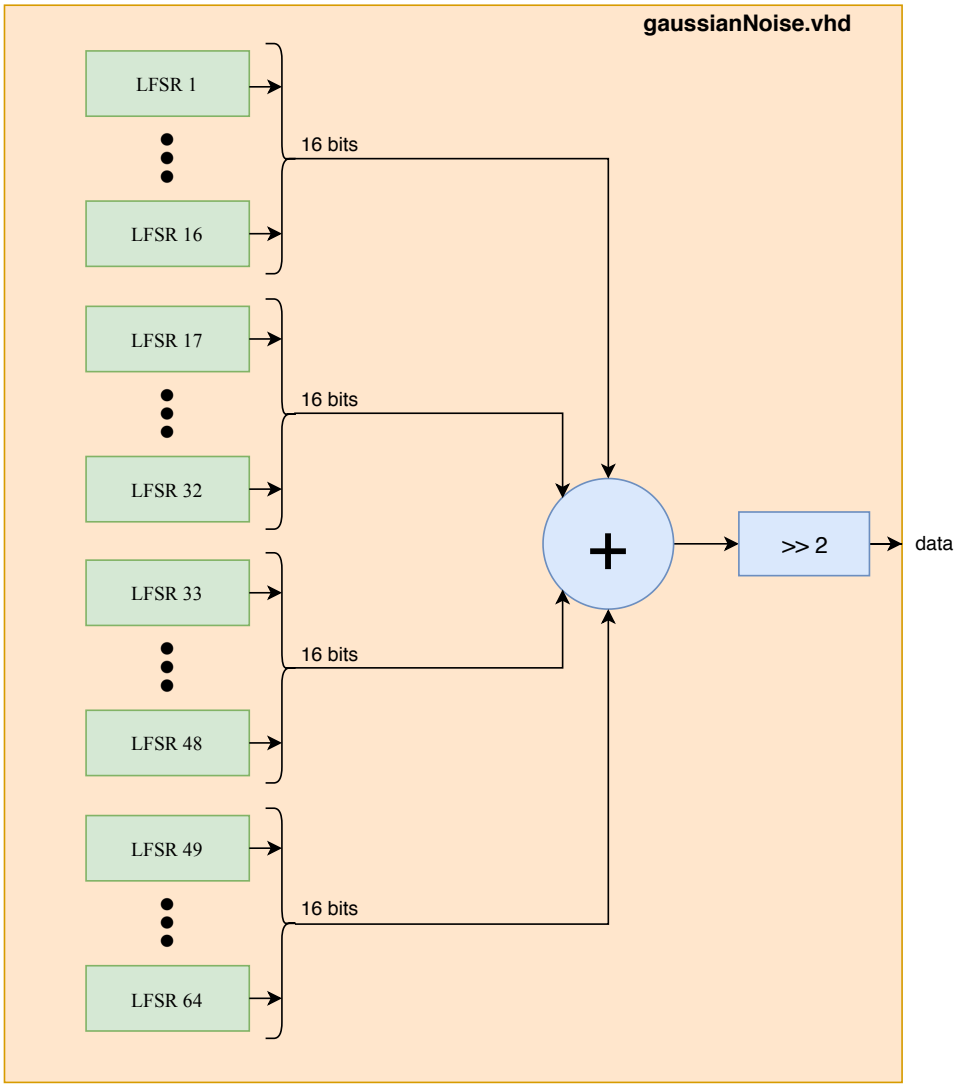


Figura 4.3: Diagrama de bloques del módulo gaussianNoise

4.2.3. Verificación

Para la verificación de este módulo se han realizado dos tests. Uno donde se comprueba el correcto funcionamiento del LFSR, y otro donde se verifica el modulo *gaussianNoise* que es el top level del diseño. Tanto para estos dos tests como para los que se han realizado para el resto de módulos, la estructura empleada ha sido la misma. En la figura 4.4 se muestra la estructura de los tests empleados en este trabajo. Esta estructura sigue la metodología marcada por el estándar UVM, en ella se pueden diferenciar varios bloques:

- *Driver*: Este módulo se encarga de la generación de los datos de entrada o estímulos que van a entrar al DUT.
- *Monitor*: Este módulo monitoriza los datos seleccionados en su entrada.
- *Reference model*: Modelo de referencia con la funcionalidad que se desea implementar en

nuestro módulo.

- *Scoreboard*: Módulo donde se realiza la comparación de los datos obtenidos del DUT con los del modelo de referencia.
- DUT: *Device Under Test*, módulo donde se encuentra el diseño realizado que se va a verificar.

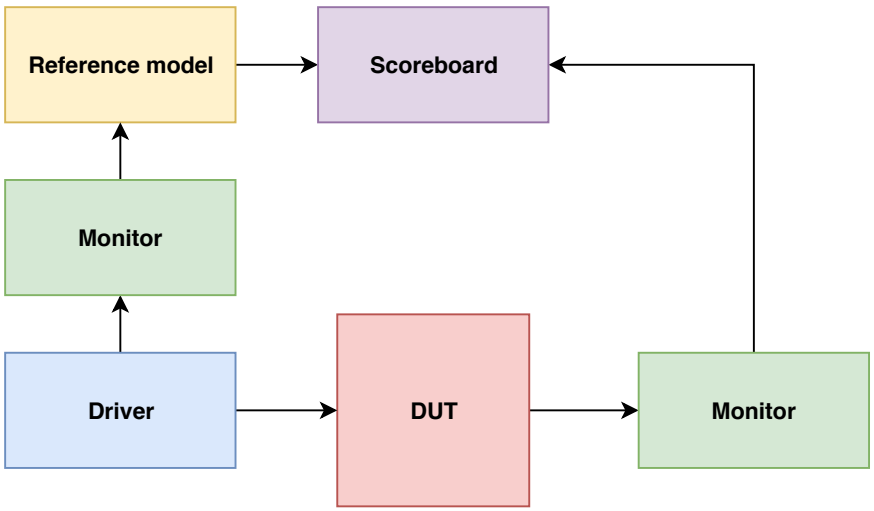


Figura 4.4: Estructura de los tests

De estos bloques, el DUT se encuentra en el simulador utilizado para la verificación. El *Driver*, el *Monitor* y el *Scoreboard* forman parte del test realizado con cocotb, siendo cada uno de ellos una clase de Python proporcionada por la propia herramienta. Por último, el modelo de referencia es un fichero independiente escrito en Python.

4.2.3.1. LFSR

En este apartado, se muestran los resultados obtenidos del LFSR. En concreto, se va a mostrar la distribución obtenida tras la formación de una palabra de 16 bits con la concatenación de la salida de 16 módulos LFSR. Todos los LFSR tienen un tamaño interno de 16 bits, siendo su período de repetición de $2^{16} - 1$. En la figura 4.5 se muestra la distribución que siguen estos datos, verificando así que se corresponde con la distribución normal esperada.

4.2.3.2. *Gaussian Noise generator*

A continuación,se muestran los resultados obtenidos del módulo gaussianNoise. En este caso, la distribución de los datos obtenidos es una normal. En la figura 4.6 se puede verificar que se ha obtenido lo deseado. Además, si se realiza la FFT de estos datos (veáse figura 4.7) se ve como la potencia se reparte a lo largo de todo el espectro, que en este caso esta limitado entre 0 y 50 MHz por haber utilizado un reloj de 100 MHz en la simulación.

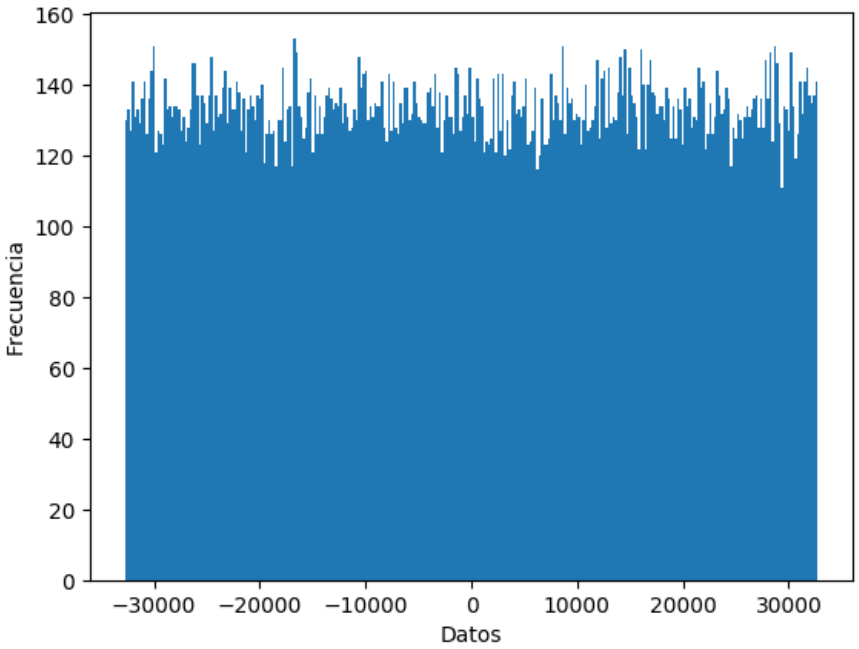


Figura 4.5: Distribución de los datos de 16 LFSR concatenados

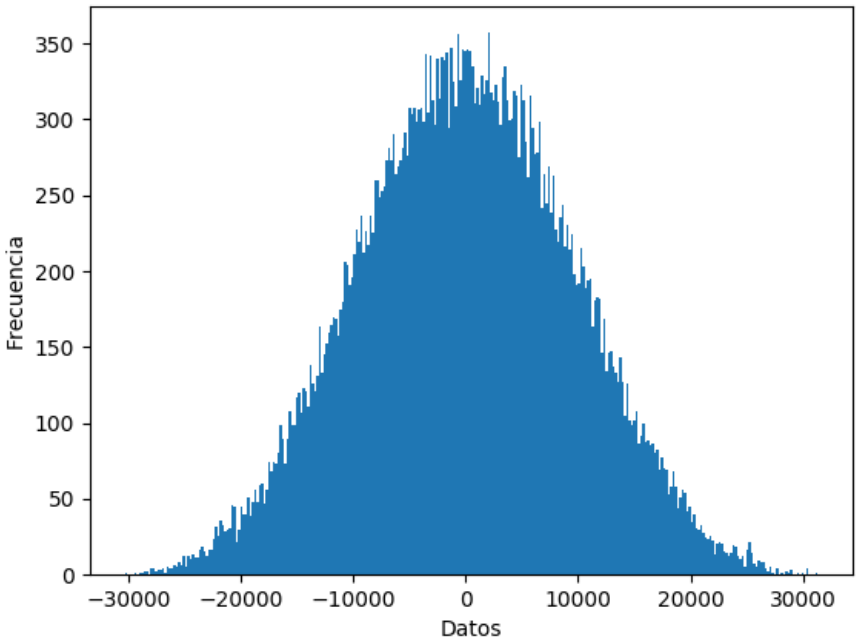


Figura 4.6: Distribución de los datos del módulo gaussianNoise

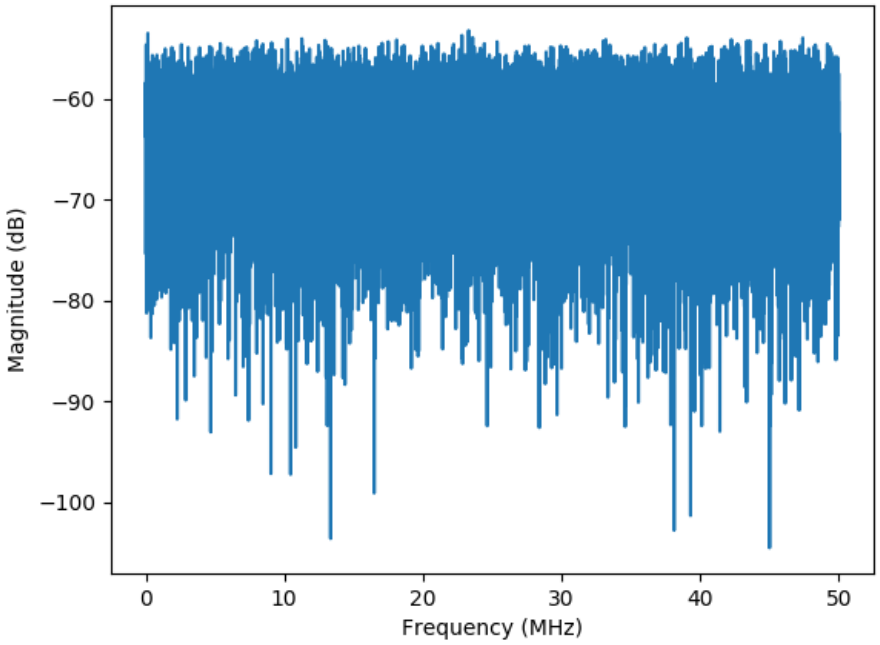


Figura 4.7: FFT de los datos del módulo gaussianNoise

4.3. Filtro paso-bajo

4.3.1. Descripción del módulo

Como ya se expuso en el subapartado 2.3.2.5, para realizar un *follower jammer* se debe generar una señal de ruido con un ancho de banda determinado, en concreto, con el ancho de banda calculado en la parte ES del sistema. Es en este módulo donde se procede a filtrar paso-bajo el ruido gaussiano generado en el módulo anterior para obtener la banda de ruido deseada. Para la implementación del filtro digital hay dos posibilidades, un filtro FIR (en inglés, *finite impulse response*) o un filtro IIR (en inglés, *infinite impulse response*). En cuanto a los filtros FIR, pueden tener fase lineal a lo largo de todo el rango frecuencial y son siempre estables con coeficientes cuantificados, en cambio, necesitan un mayor número de coeficientes que un filtro IIR para las mismas especificaciones. Por su parte, los filtros IIR tienen fase no lineal y la cuantificación de sus coeficientes puede afectar a la estabilidad del filtro y a la respuesta en frecuencia de este en mayor medida que en los filtros IIR. Sin embargo, como ya se ha dicho, estos filtros requieren menos coeficientes que un filtro FIR para las mismas especificaciones. Finalmente, se el filtro escogido ha sido un FIR debido a su mayor sencillez a la hora de implementarlo y a su estabilidad inherente, ya que en caso de que el filtro se hiciera inestable su salida tendería a infinito y acabaría saturando el DAC (en inglés, *Digital-to-analog converter*.

4.3.2. Implementación

Al ser este proyecto un prototipo inicial y no haber unas necesidades reales de cumplir ciertas especificaciones por parte de la empresa, se ha optado por hacer un diseño parametrizable del filtro y así poder configurarlo cuando surja esta necesidad y, además, ser reutilizado en futuros proyectos donde se requiera este tipo de filtro.

A pesar de esta indeterminación, para el diseño del filtro se plantearon unas especificaciones cercanas a una situación real con el objetivo de poder realizar un ejemplo donde se demuestre el correcto funcionamiento del sistema. Como herramienta de diseño se ha usado el software *GNU Radio* el cual dispone de una aplicación para diseño y análisis de filtros denominada *filter design tool*. Esta herramienta es muy similar a la ya conocida *fdatool* que integra Matlab, con la diferencia de ser gratuita al ser un programa de software libre. El método de diseño escogido ha sido el método Parks-McClellan, también llamado Equiripple en la herramienta. Se ha escogido este método por ser el más flexible de todos, pudiendo controlar la frecuencia final de la banda de paso (f_{pass}), la frecuencia a la que comienza la banda eliminada (f_{stop}) y el rizado en la banda de paso (A_{pass}) y la atenuación en la banda eliminada (A_{stop}) [13]. En la tabla 4.1 se muestran las especificaciones seleccionadas y en la figura 4.8 se muestra la herramienta de diseño del filtro.

Frecuencia de muestreo (f_s) (MHz)	100
f_{pass} (MHz)	1
f_{stop} (MHz)	5
A_{pass} (dB)	1
A_{stop} (dB)	41

Tabla 4.1: Especificaciones del filtro paso-bajo

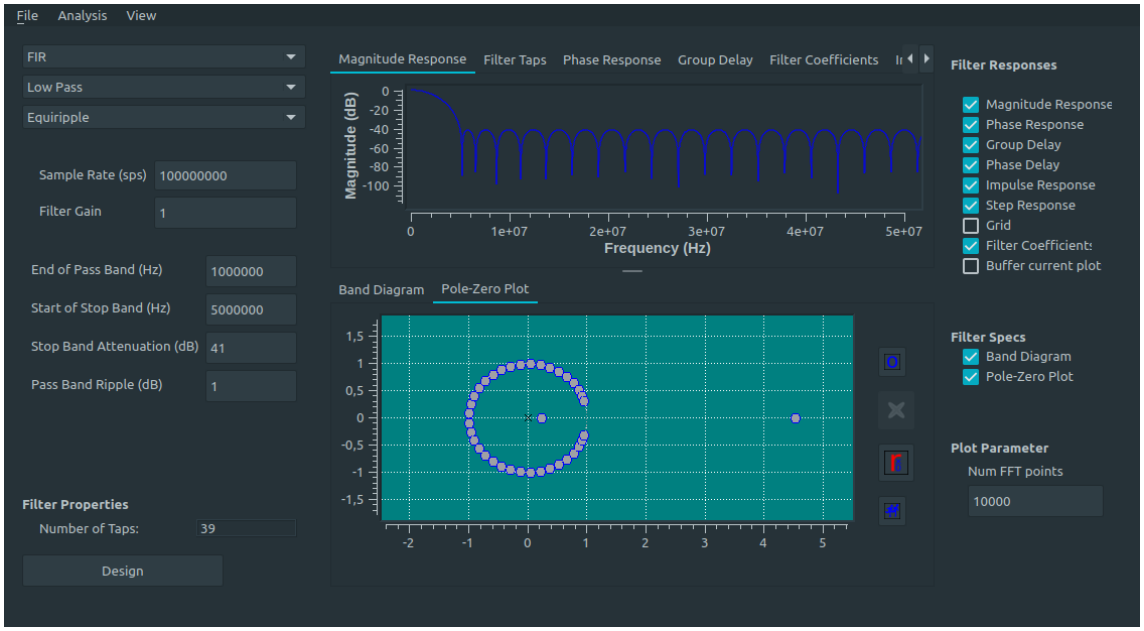


Figura 4.8: Herramienta *filter design tool* usada para el diseño del filtro

Con esas especificaciones la respuesta en frecuencia del filtro es como se muestra en la figura

4.9, en este caso con los coeficientes sin cuantificar. El número de coeficientes calculados para el filtro son 39. Para realizar la implementación del filtro en la FPGA se requiere una cuantificación de estos coeficientes, en concreto, se ha usado una cuantificación con 15 bits con el formato [15,14] con signo. Tras realizar la cuantificación de los coeficientes la respuesta en frecuencia del filtro es prácticamente igual, quedando como se representa en la figura 4.10.

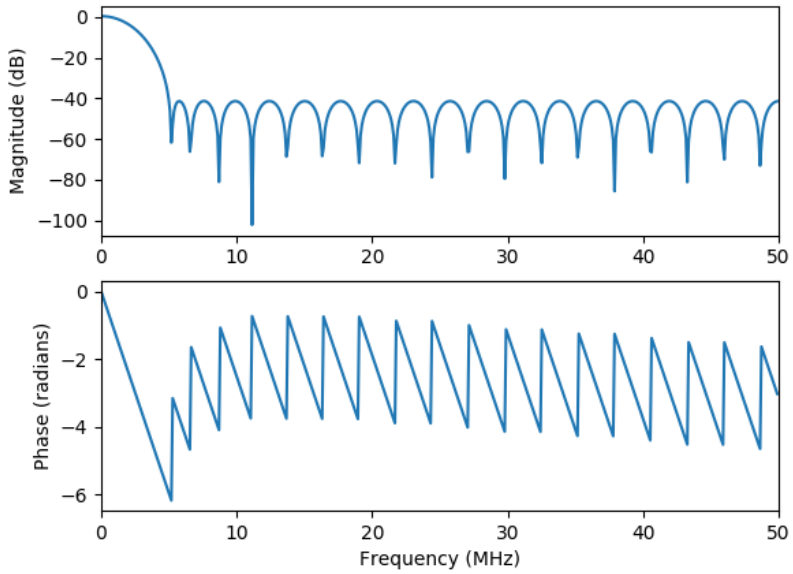


Figura 4.9: Respuesta en frecuencia del filtro sin cuantificar

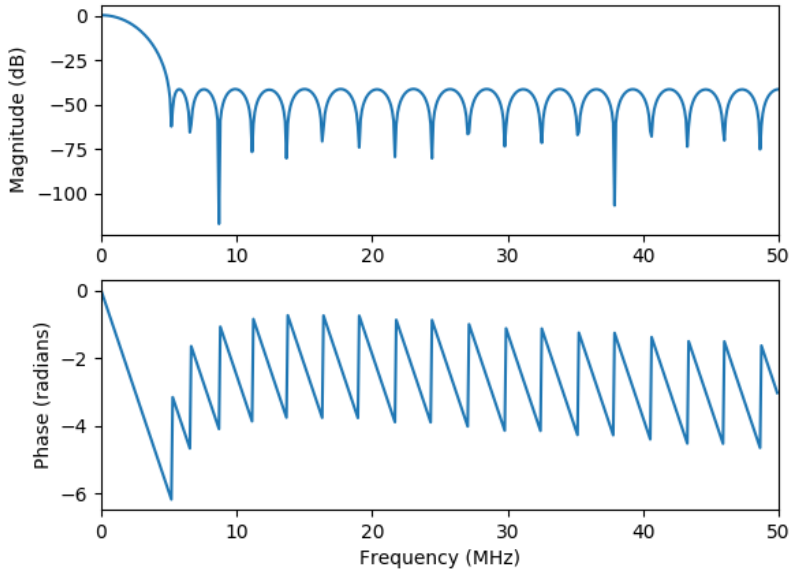


Figura 4.10: Respuesta en frecuencia del filtro cuantificado

Finalmente, se hace la elección de la estructura del filtro FIR a implementar. La elección de esta estructura viene determinada principalmente por la frecuencia a la que se desea trabajar, en este caso se desea que el filtro trabaje en tiempo real, es decir, que la frecuencia de muestreo sea igual que la frecuencia de reloj. Por otro lado, el filtro diseñado tiene coeficientes simétricos, lo que permite implementar una estructura simétrica donde se optimizan los recursos utilizados. Con el objetivo de hacer el filtro totalmente configurable se han diseñado dos módulos, uno para cuando el número de coeficientes son pares y otro para cuando son impares, ya que la estructura es ligeramente distinta. En las figuras 4.11 y 4.12 se muestran las estructuras implementadas para coeficientes pares e impares respectivamente [14].

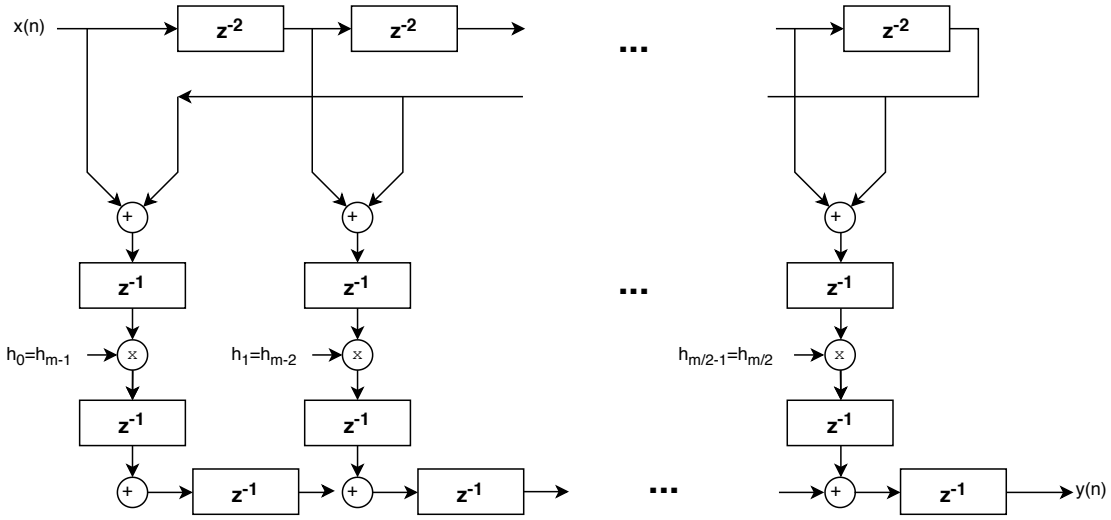


Figura 4.11: Estructura del filtro FIR para coeficientes pares

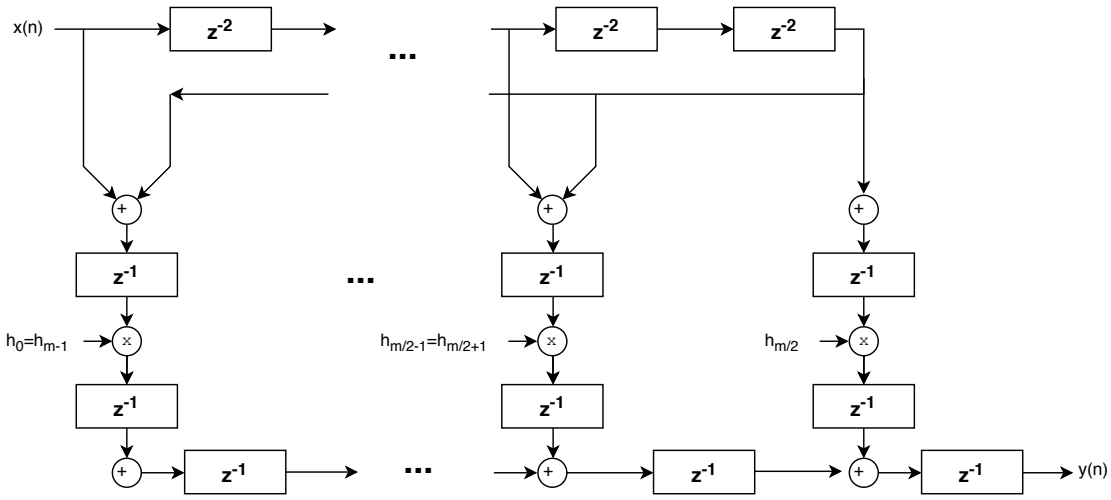


Figura 4.12: Estructura del filtro FIR para coeficientes impares

En ambas imagenes se puede ver que se repite siempre la misma estructura. El diseño de esta estructura se ha hecho con el objetivo de cumplir con la estructura de los bloques DSP que incluyen las FPGAs Ultrascale y Ultrascale+ de Xilinx, ya que la FPGA donde se va a implementar este

diseño es de esa familia. En la figura 4.13 se muestra la estructura del bloque DSP48E2 presente en estas FPGAs.

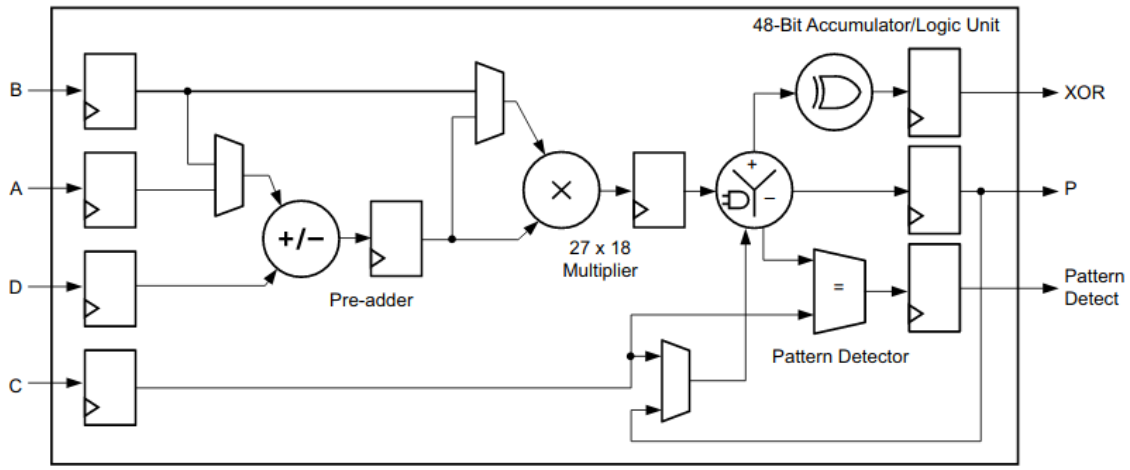


Figura 4.13: Bloque DSP presente en las FPGAs Ultrascale y Ultrascale+ de Xilinx [15]

Una vez diseñado el filtro y definida su estructura se realiza el diseño del módulo lowPassFilter. El diagrama de bloques del módulo se muestra en la figura 4.14, en ella se observan dos estructuras claramente diferenciadas, una es la memoria RAM donde se almacenan los coeficientes y otra es la estructura del filtro FIR mostrada anteriormente. La memoria RAM se ha incluido con el objetivo de hacer el módulo reconfigurable. En ella se almacenan los coeficientes del filtro, pudiendo cargar otros coeficientes distintos cuando se desee y así modificar la respuesta en frecuencia del filtro. Hay que destacar que una vez sintetizado el diseño para un número de coeficientes determinado las posteriores reconfiguraciones se deben hacer respetando este número de coeficientes.

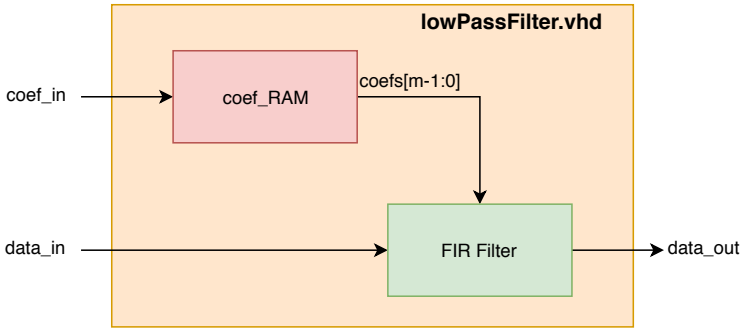
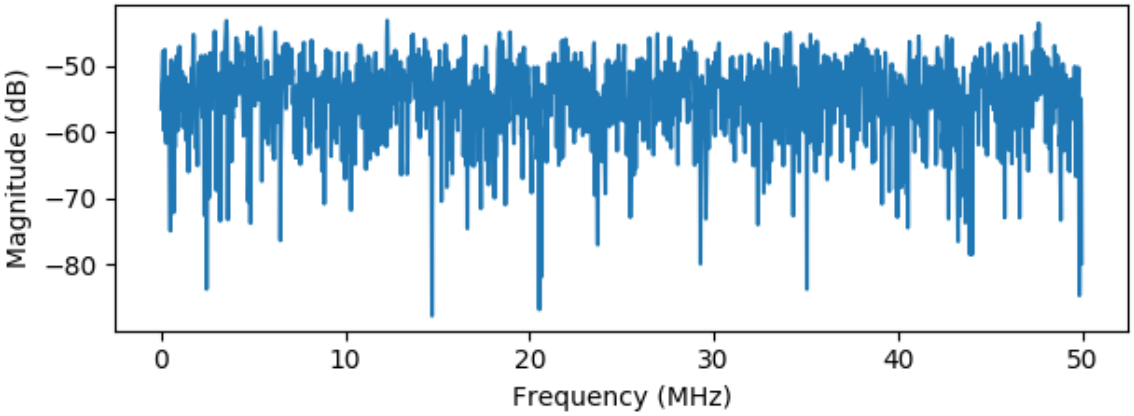


Figura 4.14: Diagrama de bloques del módulo lowPassFilter

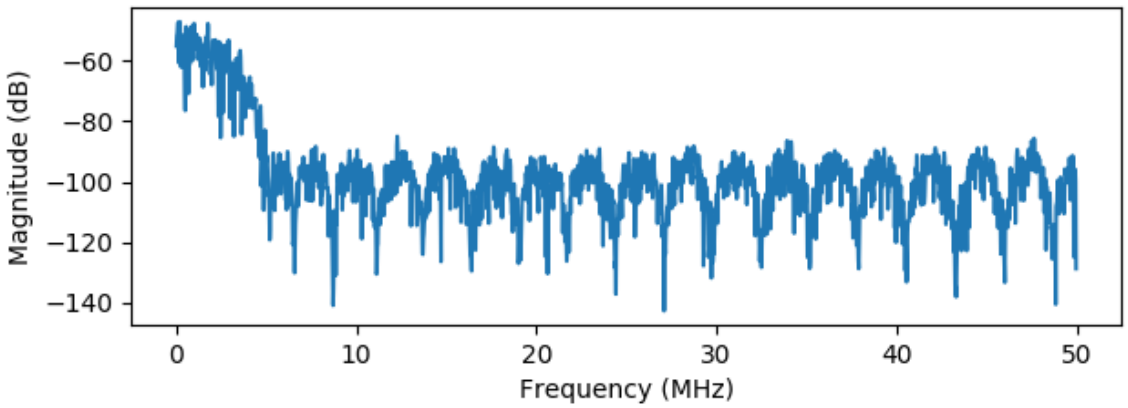
4.3.3. Verificación

La metodología empleada para realizar la verificación es la misma que la que se expuso en el subapartado 4.2.3. El testeo de las dos estructuras del filtro diseñadas se ha realizado de la misma manera, mostrando en este apartado el resultado de solo una de ellas. En el test diseñado se ha utilizado ruido gaussiano como datos de entrada al módulo. En la figura 4.15 se comparan las FFTs de los datos de entrada con los datos de salida del módulo, observando así el correcto

funcionamiento de este. Para la simulación se ha utilizado un reloj de 100 MHz y la configuración del filtro mostrada anteriormente, haciendo uso de la estructura para coeficientes impares al tener 39 de ellos.



(a) FFT del ruido de entrada al módulo lowPassFilter



(b) FFT de los datos filtrados en el módulo lowPassFilter

Figura 4.15: Comparación de las FFTs entre los datos de entrada y salida del módulo lowPassFilter

4.4. Mixer

4.4.1. Descripción del módulo

Como elemento final de la cadena de procesado se encuentra este módulo, denominado mixer. Tras el filtrado del ruido generado con el ancho de banda deseado falta por colocar la banda de ruido en la frecuencia central adecuada. Con este propósito se diseña este módulo. El proceso mediante el cual se va a desplazar la banda de ruido colocada en banda base a una frecuencia central determinada se denomina mezclado digital. Para llevarlo a cabo se ha de generar una señal de un tono en la frecuencia deseada, esto se consigue mediante la inclusión de un DDS (en inglés, *Direct digital synthesis*) que genere un seno de tal frecuencia. Posteriormente, el desplazamiento de la banda se realiza multiplicando la señal de ruido con el seno generado, finalizando así el proceso del mezclado digital.

Para el caso particular de este proyecto, se ha optado por este procedimiento para poder adaptarse rápidamente a la situación de la señal a interferir, la cual al seguir una modulación FHSS va a estar cambiando su frecuencia central de transmisión continuamente, como ya se explicó en el apartado teórico 2.2.1.2. Dicha señal irá saltando dentro de una banda determinada en el espectro, la cual puede encontrarse en frecuencias de hasta 6 GHz. Con este módulo se van imitando los saltos de la señal dentro de esa banda modificando la frecuencia del DDS, con esto se tiene la señal en un frecuencia intermedia. Para finalizar, el DAC estará sintonizado a la frecuencia central de la banda donde se mueve la señal FHSS, que puede llegar hasta 6 GHz. Es en este DAC donde se hará la subida en frecuencia de la señal de ruido generada.

4.4.2. Implementación

4.4.2.1. DDS

En primer lugar, se ha realizado el diseño de un DDS para conseguir así el seno necesario. En concreto, la implementación se ha realizado haciendo uso del algoritmo CORDIC. Este algoritmo permite realizar el cálculo del seno y el coseno de un ángulo θ mediante sumas, restas y desplazamientos, siendo muy sencilla su implementación en dispositivos tales como una FPGA. En la imagen 4.17 se muestra la estructura que se ha implementado para la realización de este algoritmo. El parámetro n vale para este caso 16 y son las etapas empleadas para el cálculo del coseno y el seno, este valor determina la precisión con la que se hace el cálculo.

El diagrama de bloques del módulo diseñado para cálculo del seno y el coseno se muestra en la figura 4.16. En el bloque llamado adapter input se hace una adaptación del ángulo de entrada. Esto viene motivado porque se ha realizado el diseño para que el módulo reciba ángulos desde 0 hasta 2π en radianes, pero el algoritmo del CORDIC solo realiza el cálculo de ángulos comprendidos entre los valores $\pi/2$ y $-\pi/2$. Para ello, se pasan todos los ángulos de entrada a estos cuadrantes y se almacena en una variable el cambio de cuadrante hecho. Finalmente, tras pasar por el CORDIC, se aplica a los resultados obtenidos un cambio de signo en función del cuadrante al que pertenecía el ángulo de entrada, consiguiendo así el cálculo del seno y el coseno de ángulos entre 0 y 2π.

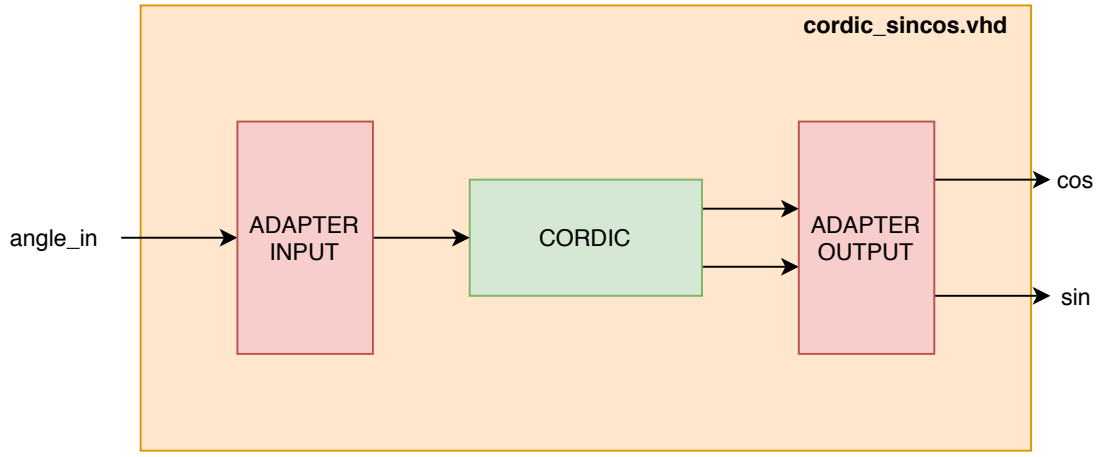


Figura 4.16: Diagrama de bloques del módulo cordic_sincos

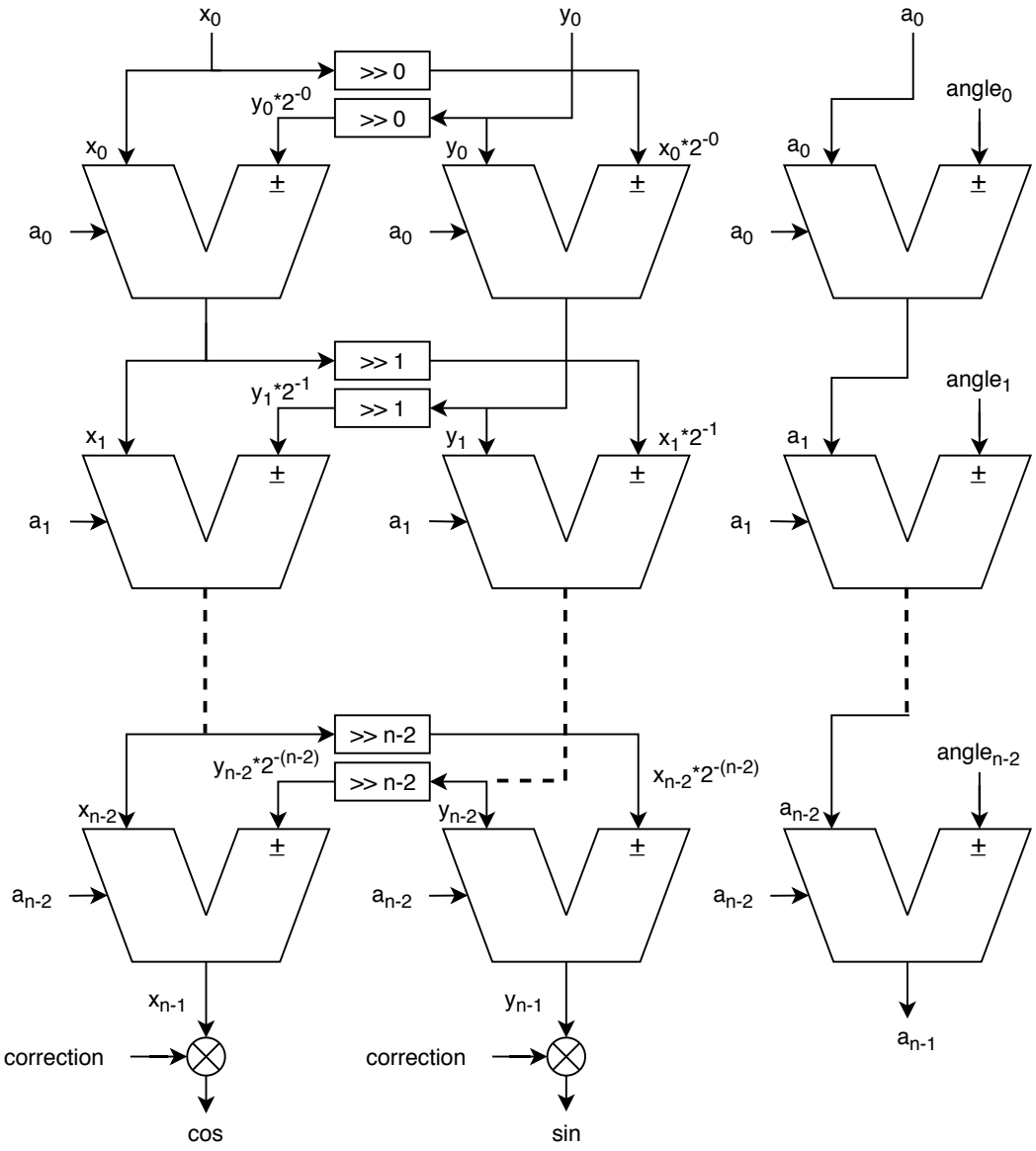


Figura 4.17: Estructura del algoritmo CORDIC implementado

El siguiente paso para el diseño del DDS es la realización de un contador que genere los ángulos de entrada del CORDIC. El diagrama de bloques de este módulo se muestra en la figura 4.18. Este módulo consiste en un contador con paso configurable y que reinicia la cuenta cuando se llega al valor de 2π. Con esto se consigue generar los ángulos necesarios desde el 0 hasta 2π. Si el paso seleccionado no es múltiplo del valor 2π no se llegará de forma exacta a este valor, para solucionar esto, y mantener una generación fluida del seno y el coseno, se añade la diferencia entre la cuenta actual del contador y el valor de 2π al inicio del siguiente ciclo del contador. Con esto, se van generando todos los ángulos con la misma diferencia de paso entre sí, evitando meter saltos al reiniciar el contador.

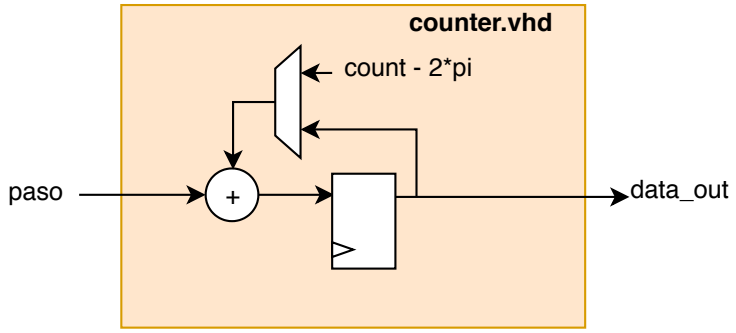


Figura 4.18: Diagrama de bloques del módulo counter

Por último, el DDS queda finalmente diseñado como se muestra en la figura 4.19. En este módulo se genera un seno y un coseno a la frecuencia que se le indique. Esto se consigue modificando el paso del contador representado en la señal de entrada *dds_freq*. Destacar que el diseño del DDS se ha realizado para obtener una muestra por cada ciclo de reloj, siendo la frecuencia de muestro la de reloj de la FPGA. La siguiente fórmula marca como se tiene que calcular el paso del contador en función de la frecuencia deseada (f), de la frecuencia de reloj usada (f_{clk}) y el número de bits usados:

$$paso = \frac{f \cdot 2\pi \cdot 2^{bits}}{f_{clk}} \tag{4.2}$$

Y la resolución que tiene el DDS:

$$f = \frac{f_{clk}}{2\pi \cdot 2^{bits}} \tag{4.3}$$

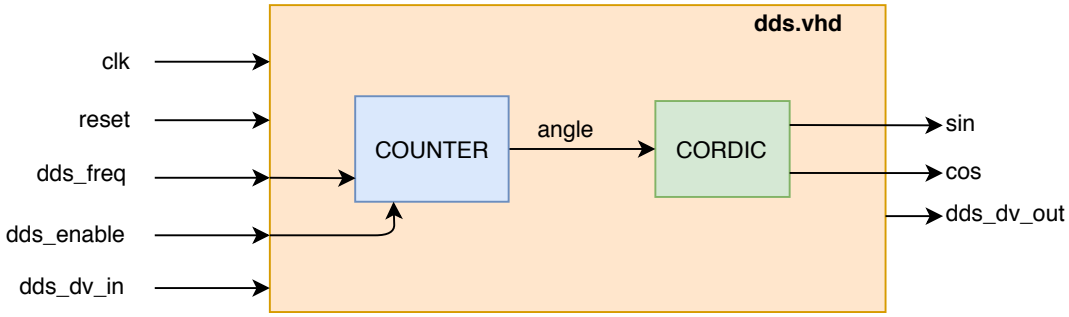


Figura 4.19: Diagrama de bloques del módulo dds

4.4.2.2. Mixer

En la implementación realizada del mixer, consiste simplemente en multiplicar el seno de salida del DDS con la señal de entrada del módulo, que viene del filtro paso-bajo. Al ser la frecuencia de muestreo igual que la de reloj de la FPGA para ambas señales no se hace necesaria la inclusión de ningún filtro interpolador ni antialiasing en el diseño. Por tanto, el ancho de banda disponible para desplazar la banda de ruido va estar determinado por la frecuencia de reloj usada en la FPGA. En la siguiente figura se muestra el diagrama de bloques de este módulo.

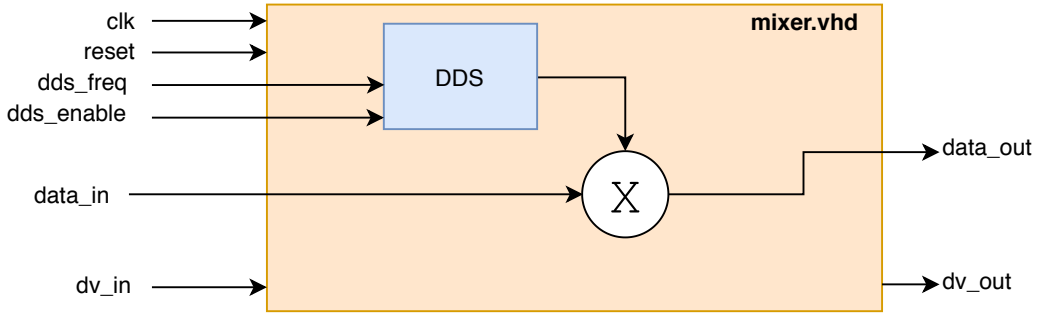


Figura 4.20: Diagrama de bloques del módulo mixer

4.4.3. Verificación

Al igual que en los casos anterior, esta verificación se ha realizado como se indicó en el subapartado 4.2.3.

4.4.3.1. DDS

Para la verificación del DDS, se han realizado primero los tests unitarios de los módulos que lo componen, el contador y el cordic.

En primer lugar, en el test realizado del cordic se ha introducido números aleatorios entre 0 y 2π como ángulo de entrada. Posteriormente, se ha comprobado el valor del seno y el coseno calculado con el valor real obtenido de las funciones sin y cos de Python. Se ha verificado que este cálculo se realiza bien con una precisión igual o superior a 0,001. En la siguiente imagen se puede ver una captura de la simulación realizada, en ella se observa que la latencia de este bloque es de 19 ciclos de reloj y que se consigue que el bloque trabaje en tiempo real, obteniendo un nuevo dato en cada ciclo de reloj. En cuanto a la latencia obtenida, 16 ciclos de reloj vienen determinados por las etapas usadas en el cálculo del CORDIC, 2 ciclos se corresponden con el adapter input y adapter output respectivamente; el último ciclo restante pertenece a un registro realizado de todas las entradas del módulo.

Figura 4.21: Simulación del módulo cordic_sincos

En segundo lugar, se muestra la verificación del contador. Al ser un módulo tan sencillo su verificación es bastante trivial. En este documento simplemente se muestra una captura (véase figura 4.22) de la simulación realizada para el contador donde se puede observar el offset que añade al inicio del siguiente ciclo del contador para un paso que no es múltiplo de 2π. Mencionar que el valor límite del contador es $2\pi \cdot 2^{16}(411774)$, es decir, que este valor se trata con un formato [19,16] sin signo.

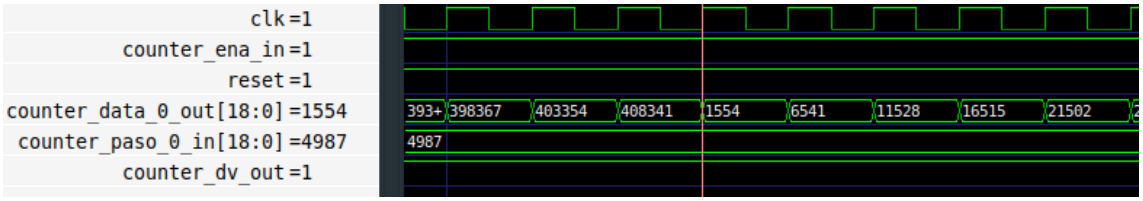


Figura 4.22: Simulación del módulo counter

Por último, se realiza la comprobación del módulo DDS. En la figura 4.23 se observan el seno y el coseno generados en la simulación de este módulo. Ambas señales tienen una frecuencia de 5 MHz y el reloj usado ha sido de 100 MHz. Realizando la FFT de una de estas señales se puede verificar además que la frecuencia se ha seleccionado de forma correcta (véase figura 4.24).

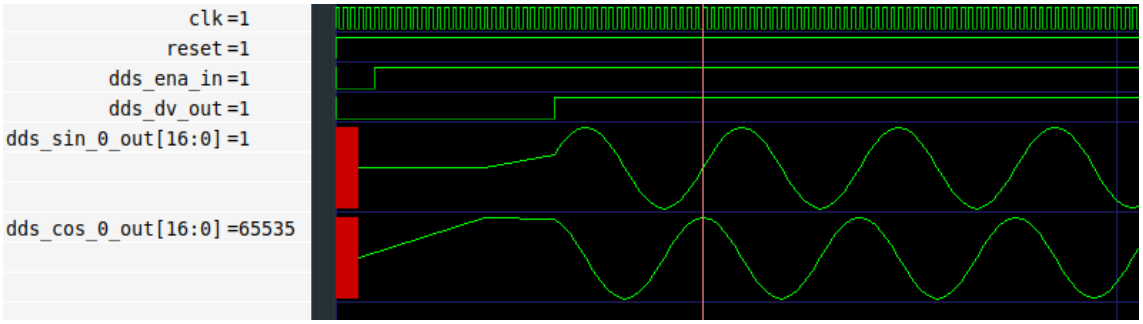


Figura 4.23: Simulación del módulo DDS

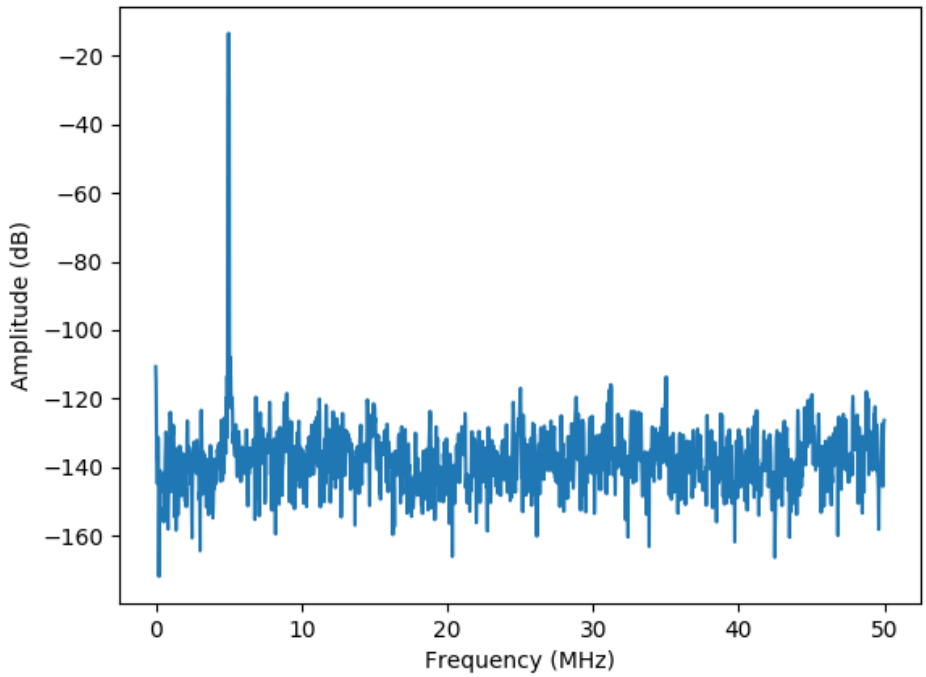
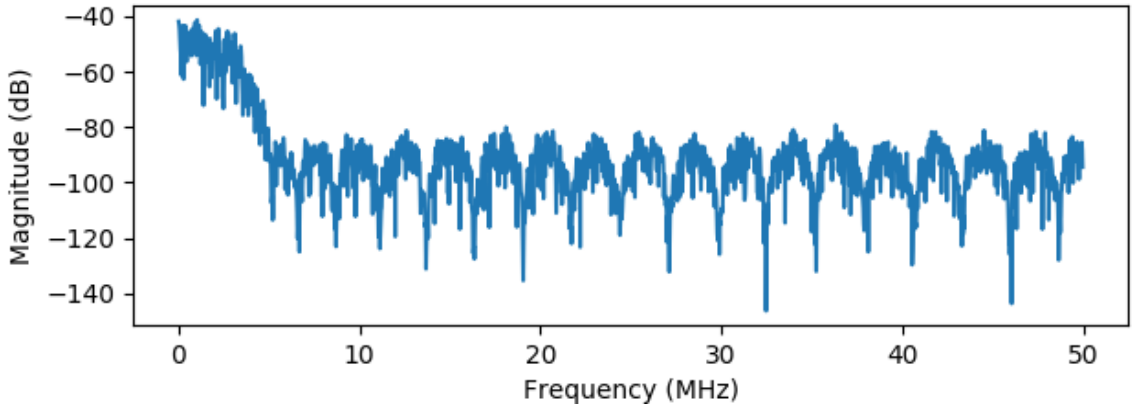


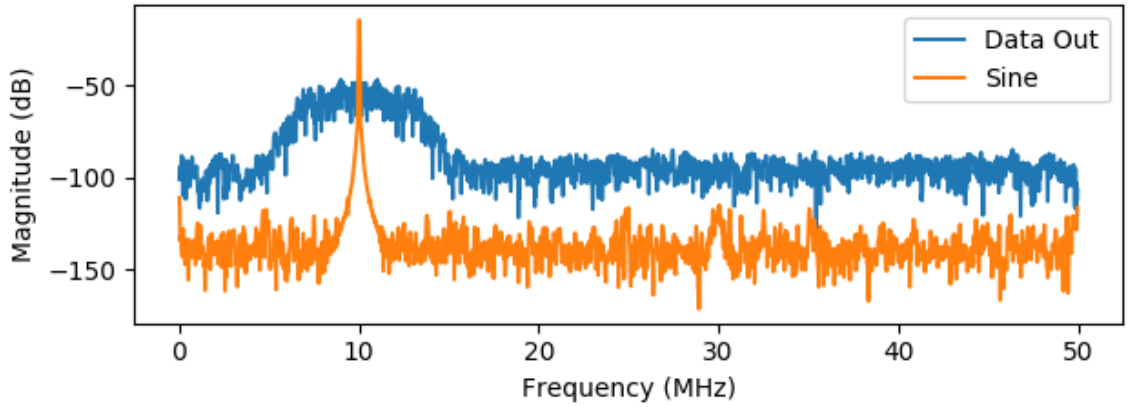
Figura 4.24: FFT del seno de salida del DDS

4.4.3.2. Mixer

Finalmente, para la verificación del módulo mixer se han introducido datos del mismo tipo que los de salida del filtro paso-bajo. En la figura 4.25 se expone el resultado obtenido a la salida de este módulo. Para esta imagen se ha generado un seno de 10 MHz desplazando la banda filtrada a esta frecuencia central. De nuevo, la frecuencia de reloj usada ha sido de 100 MHz.



(a) FFT de la entrada del mixer



(b) FFT de la salida del mixer

Figura 4.25: Comparación de las FFTs entre los datos de entrada y salida del módulo mixer

Capítulo 5

Resultados

Tras el diseño de todos los módulos necesarios para la realización del *jammer* configurable, se procede a mostrar los resultados obtenidos de la integración de todos estos módulos. Primeramente, se expondrá como se ha realizado la integración de todos ellos y una comprobación a nivel de simulación. Se finalizará con la implementación del diseño en la FPGA mostrando resultados reales de esta haciendo uso de un ILA (en inglés, *Integrated Logic Analyzer*).

5.1. Integración

Con el objetivo de facilitar la inclusión de este diseño junto con la parte ES del sistema, se ha optado por realizar un módulo que los albergue a todos ellos realizando las conexiones necesarias para el correcto funcionamiento. De esta manera, se realiza la integración de todo el diseño realizado en un módulo denominado jammer. El diagrama de bloques de este módulo se presenta en la figura 5.1. Además de integrar el generador de ruido, el filtro paso-bajo y el mixer, se puede ver que se ha incluido otro módulo, el AXI Crossbar.

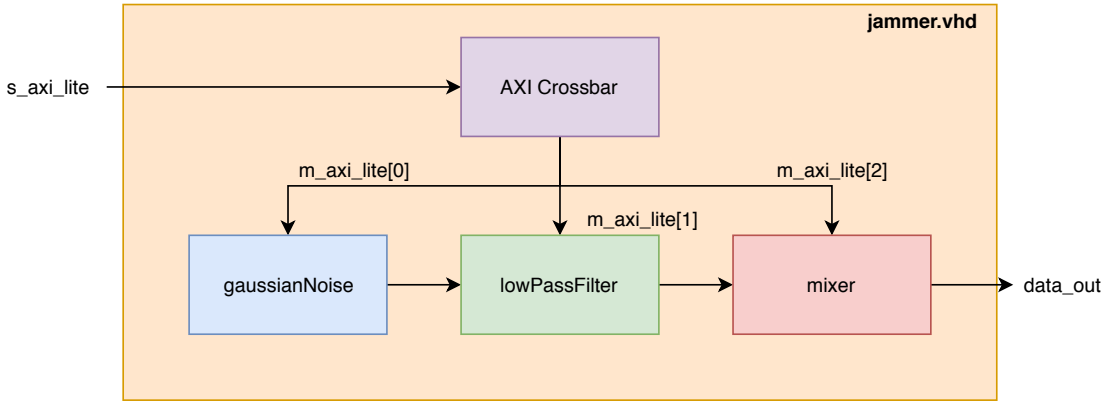


Figura 5.1: Diagrama de bloques del módulo jammer

Para poder hacer el diseño totalmente configurable se ha añadido el protocolo *Advanced eXtensible Interface* (AXI) a cada uno de los módulos presentes. Este protocolo forma parte de la especificación *Advanced Microcontroller Bus Architecture* (AMBA) que se usa como bus on-chip

para microprocesadores ARM. Actualmente, el protocolo AXI se encuentra en su segunda versión, denominada AXI4. Hay tres tipos de interfaces AXI4 [16]:

- AXI4: Para una comunicación mapeada en memoria de alto rendimiento.
- AXI4-Lite: Para una comunicación simple y de bajo rendimiento mapeada en memoria (por ejemplo, entre registros de control y estado).
- AXI4-Stream: Para transmisión de datos a alta velocidad.

En nuestro caso, la interfaz usada en todos los módulos ha sido AXI4-Lite al no tener requerimientos muy altos. Al haber implementado esta interfaz en cada módulo se hace necesaria la inclusión del módulo AXI Crossbar para poder realizar la integración de todos ellos en un único módulo. Su función es la de gestionar de una manera más eficiente la comunicación entre cada módulo y el micro, teniendo una única interfaz externa de comunicación con el micro que, posteriormente, divide en 3, una por cada módulo. Esto simplifica la comunicación final ya que el microprocesador solo tendrá mapeado en memoria un único módulo. Este módulo fue diseñado por parte de la empresa DAS Photonics, habiendo realizado unas pequeñas modificaciones para adaptarlo a los requerimientos de este proyecto. Los parámetros que se pueden configurar desde el microprocesador son los siguientes:

- gaussianNoise: Se puede activar y desactivar la generación de ruido.
- lowPassFilter: Se pueden cargar distintos coeficientes del filtro, siempre y cuando el número de coeficientes coincida con los ya presentes.
- mixer: Se puede configurar la frecuencia del DDS y así poder mover en el espectro la banda obtenida en el filtro paso-bajo.

5.1.1. Verificación

En el test realizado para el sistema completo se ha realizado primero la configuración a través de AXI-Lite de los módulos del filtro y del mixer, posteriormente, se ha activado el generador de ruido. En la figura 5.2 se puede observar este proceso y el funcionamiento del AXI Crossbar, donde el elemento 0 es el generador de ruido, el 1 es el filtro y el 2 es el mixer.

Tras la correcta configuración de todos los módulos el sistema comienza a funcionar. Se muestra una captura (véase figura 5.3) donde se ve el flujo de los datos y la latencia existente desde que se genera el primer dato en el generador de ruido hasta que se obtiene el primer dato válido a la salida del módulo jammer. Además, los datos de salida se obtienen con una tasa de un dato por cada ciclo de reloj.

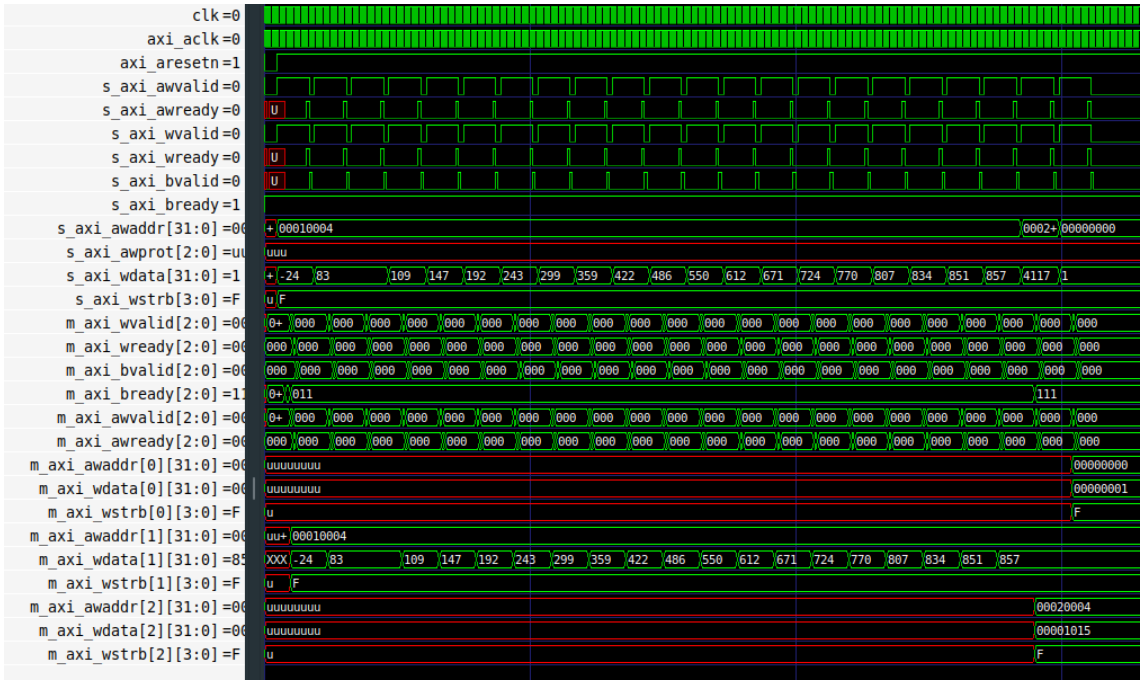


Figura 5.2: Funcionamiento del AXI Crossbar en la simulación del módulo jammer

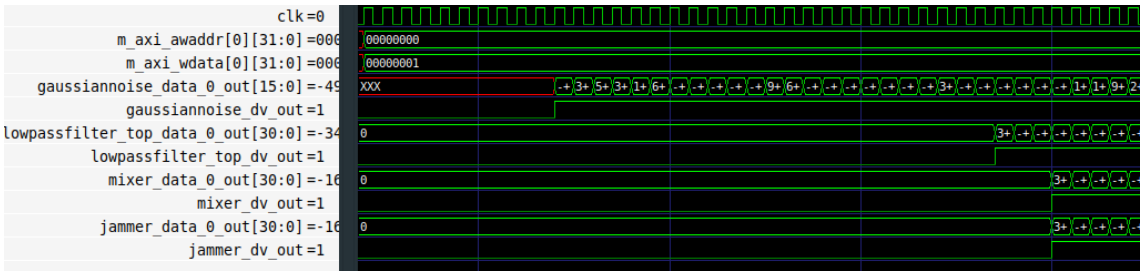
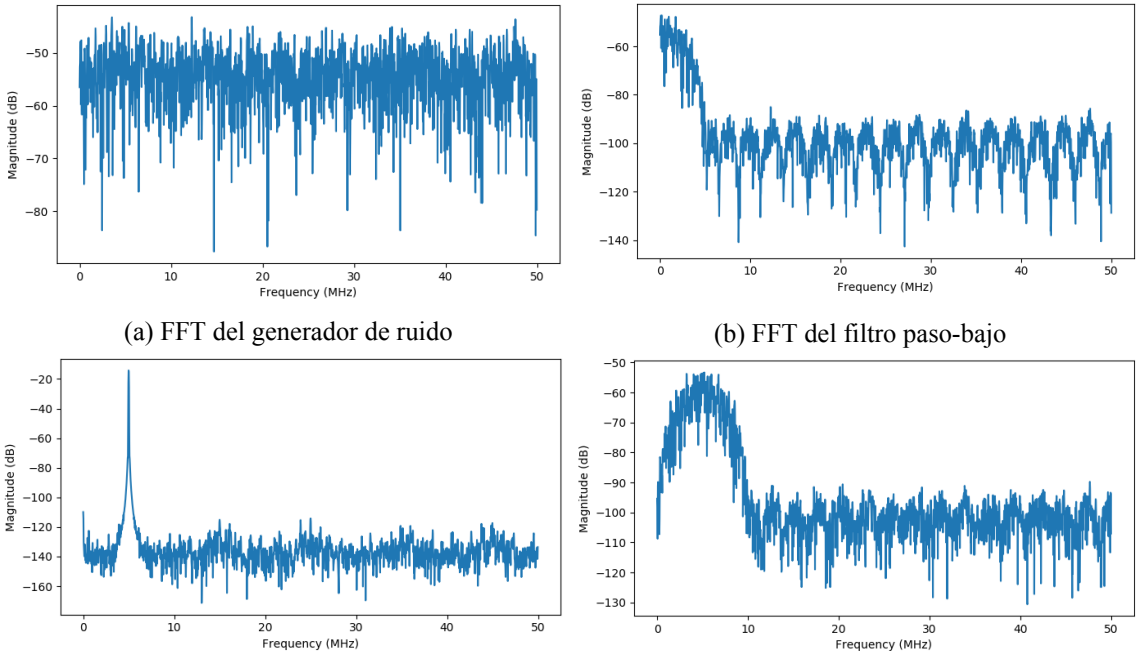


Figura 5.3: Flujo de datos en la simulación del módulo jammer

Por último, se han analizado los datos de salida realizando la FFT sobre estos. En las figuras 5.4, 5.5, 5.6 y 5.7 se muestran cuatro ejemplos para cuatro frecuencias de centrales distintas donde se han representado todas las etapas que se recorren dentro del módulo. Dentro de cada figura se representa en primer lugar la salida del generador de ruido, en segundo lugar, el resultado tras el filtrado paso-bajo, en tercer lugar, el tono a la frecuencia seleccionada en el DDS y, por último, los datos de salida del módulo tras el desplazamiento realizado de la banda. Con esto se da por verificado el correcto funcionamiento del módulo jammer a nivel de simulación.



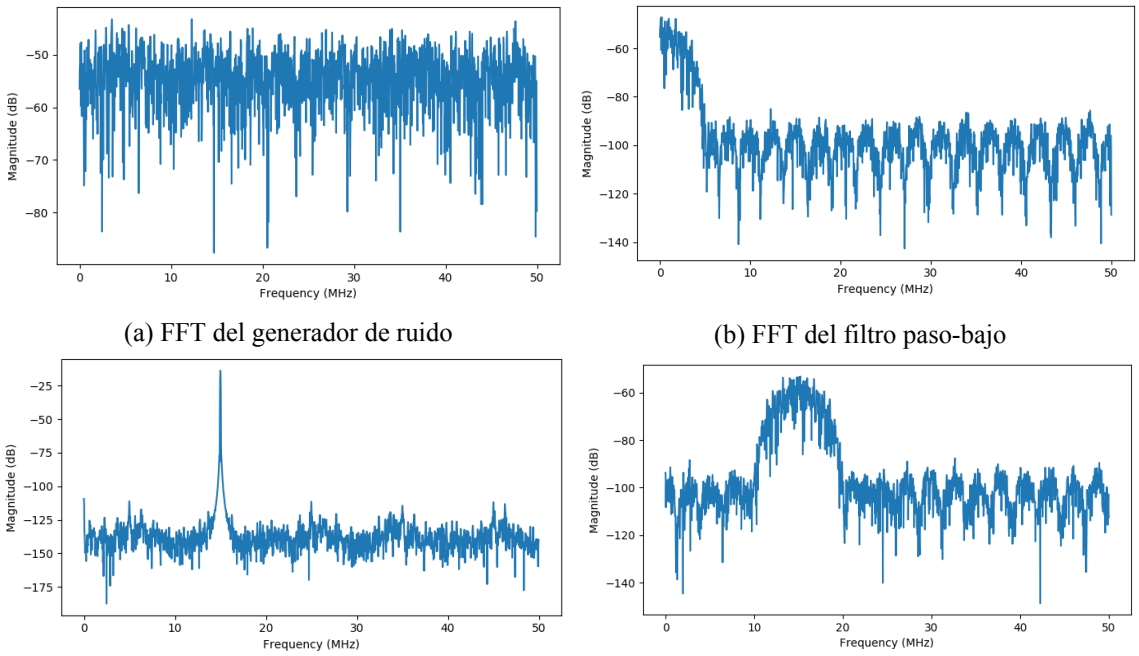
(a) FFT del generador de ruido

(b) FFT del filtro paso-bajo

(c) FFT del seno del DDS

(d) FFT de la salida del mixer

Figura 5.4: FFT de los datos del módulo jammer para una frecuencia del DDS de 5 MHz



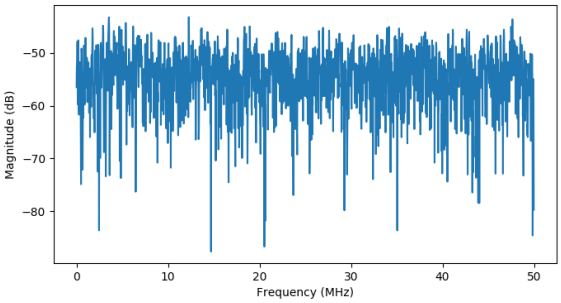
(a) FFT del generador de ruido

(b) FFT del filtro paso-bajo

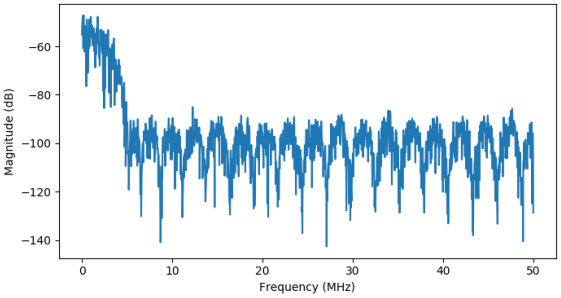
(c) FFT del seno del DDS

(d) FFT de la salida del mixer

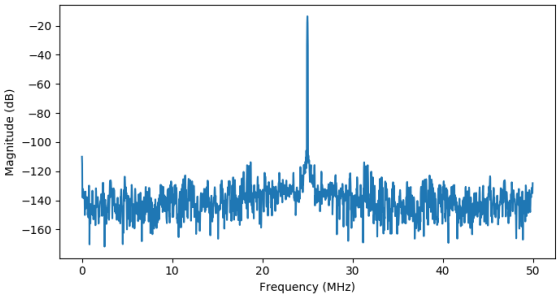
Figura 5.5: FFT de los datos del módulo jammer para una frecuencia del DDS de 15 MHz



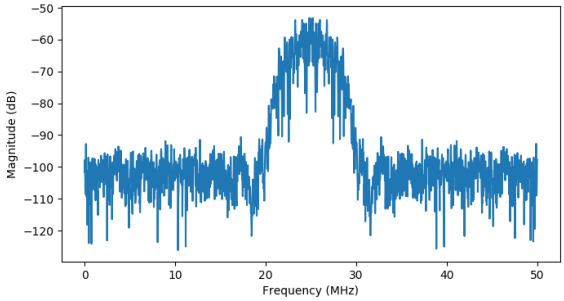
(a) FFT del generador de ruido



(b) FFT del filtro paso-bajo

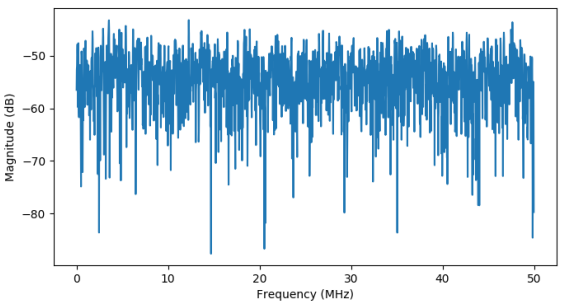


(c) FFT del seno del DDS

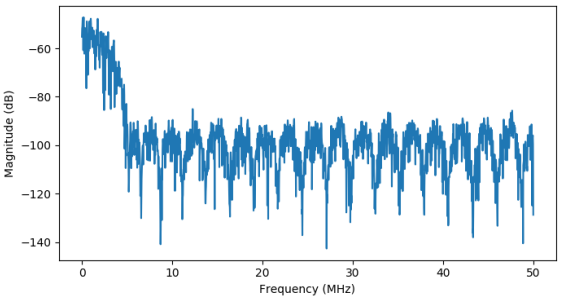


(d) FFT de la salida del mixer

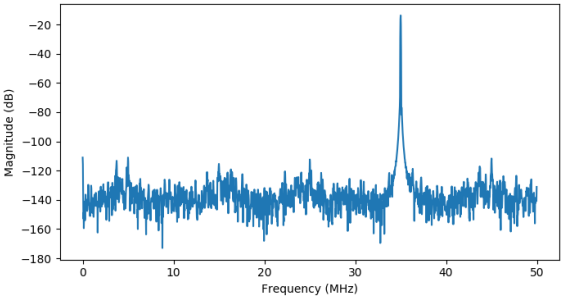
Figura 5.6: FFT de los datos del módulo jammer para una frecuencia del DDS de 25 MHz



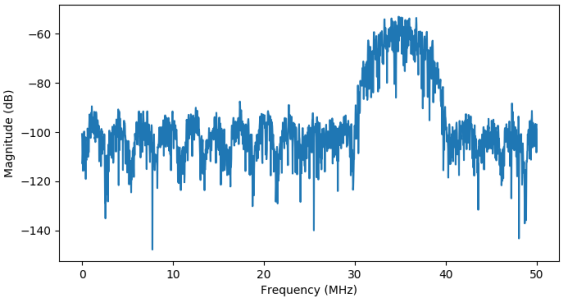
(a) FFT del generador de ruido



(b) FFT del filtro paso-bajo



(c) FFT del seno del DDS



(d) FFT de la salida del mixer

Figura 5.7: FFT de los datos del módulo jammer para una frecuencia del DDS de 35 MHz

5.2. Implementación en la FPGA

Como etapa final en el desarrollo estaría la implementación del diseño realizado en una FPGA. Para este proyecto se ha utilizado la placa de evaluación ZCU106 de Xilinx (véase figura 5.8), la cual incluye como FPGA una Zynq Ultrascale+ [17].

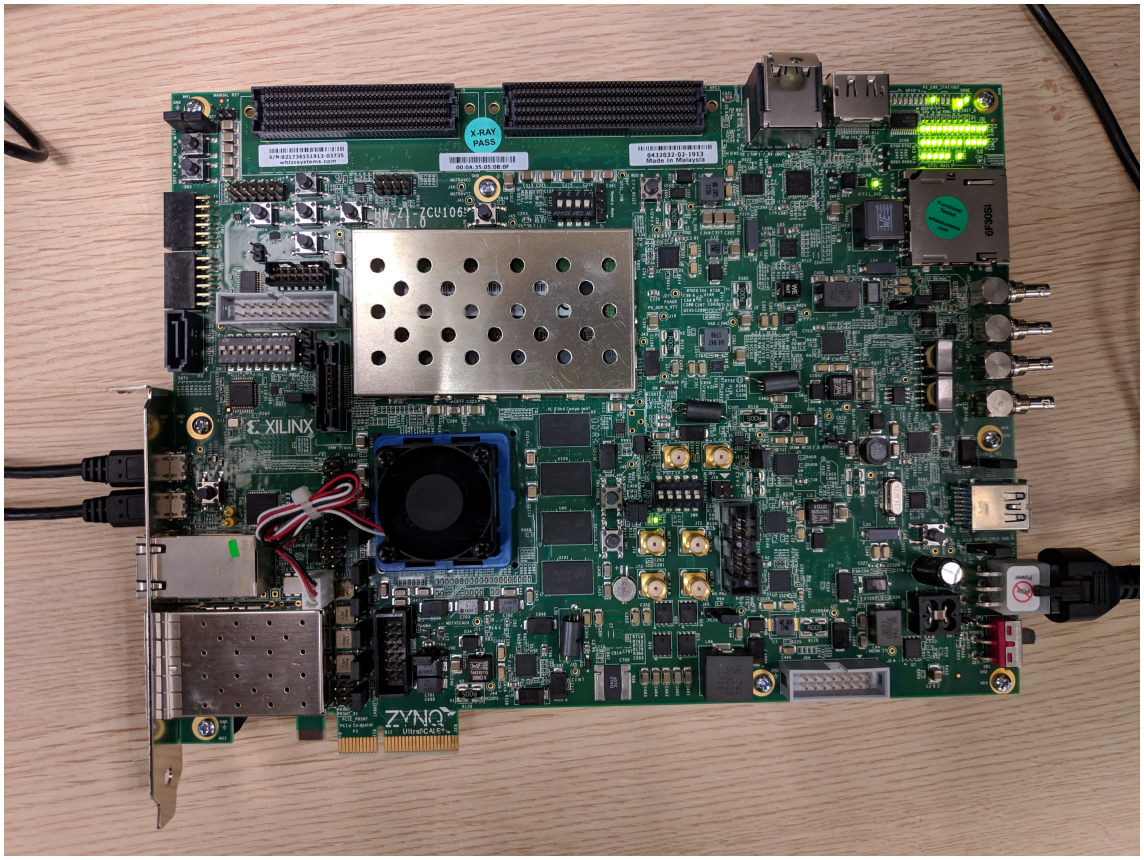


Figura 5.8: Placa de evaluación ZCU106 utilizada en el proyecto

Para poder realizar la implementación física de nuestro diseño se ha hecho uso de la herramienta Vivado de Xilinx. En ella, se ha construido un diagrama de bloques donde se ha instanciado el módulo jammer junto con otros tres ip cores de Xilinx. En la figura 5.9 se muestra la conexión realizada. El módulo Zynq UltraScale MPSoc se corresponde con el microprocesador ARM que incorpora esta FPGA, mientras que el Processor System Reset y el AXI Interconnect sirven para realizar la gestión del reset y de la interfaz de comunicación AXI entre el microprocesador y nuestro módulo. Por otro lado, el reloj utilizado para nuestro diseño se ha generado desde el microprocesador a una frecuencia de 100 MHz. Por último, se ha asignado una dirección de memoria al módulo jammer para que sea posible realizar la escritura de los registros correspondientes por parte del microprocesador. Esto se realiza mediante la interfaz AXI-Lite ya mencionada. El mapa de memoria se muestra en la figura 5.10.

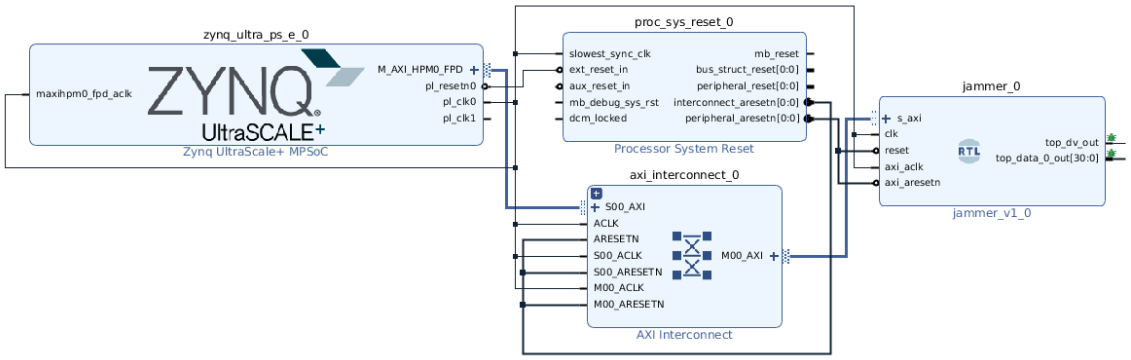


Figura 5.9: Diagrama de bloques del proyecto

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
zynq_ultra_ps_e_0					
Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G])					
jammer_0	s_axi	reg0	0x04_0000_0000	1M	0x04_000F_FFFF

Figura 5.10: Mapa de memoria con el módulo jammer

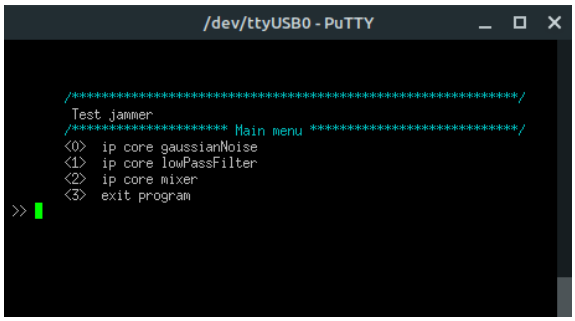
Tras realizar la compilación del diseño se obtienen los resultados de los recursos utilizados. En la figura 5.11 se detallan los recursos utilizados en el módulo jammer y en cada uno de los bloques que lo componen. Destacar que en el filtro paso-bajo aparecen los 20 bloques DSP esperados, ya que al ser simétrico se reducen los recursos del filtro de 39 bloques a estos 20.

Name	CLB Registers (460800)	CLB LUTs (230400)	LUT as Memory (101760)	LUT as Logic (230400)	DSPs (1728)	GLOBAL CLOCK BUFFERs (544)
jammer	3156	1524	98	1426	22	2
mixer_top_i (mixer_top)		1249	15	1234	2	0
lowPassFilter_top_i (lo...		47	32	15	20	0
gaussianNoise_top_i (...		193	41	152	0	0
axi_crossbar_i (axi_cr...		35	10	25	0	0

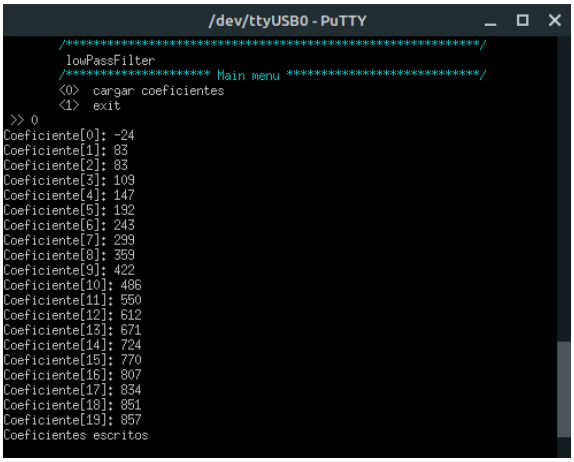
Figura 5.11: Recursos utilizados por el módulo jammer

5.2.1. Control del módulo

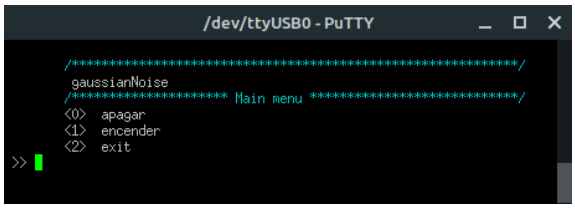
Para poder realizar la configuración y el control del módulo, se ha realizado un pequeño programa en lenguaje C que corre en el microprocesador. Los parámetros que se pueden controlar se detallaron en el apartado 5.1. La comunicación con el microprocesador de la FPGA se ha realizado a través del puerto serie y mediante la aplicación PuTTY. En la imagen 5.12 se puede ver un pequeño menú diseñado para interactuar con cada módulo.



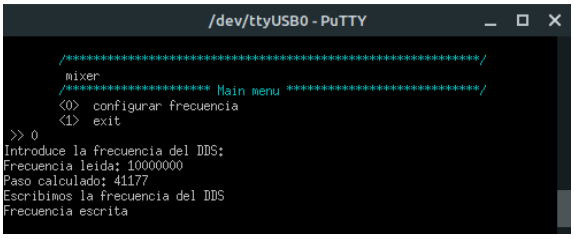
(a) Menú principal



(b) Menú del módulo lowPassFilter



(c) Menú del módulo gaussianNoise



(d) Menú del módulo mixer

Figura 5.12: Programa para el control del módulo jammer a través del microprocesador

5.2.2. ILA

Finalmente, mediante el uso de un ILA se realiza la comprobación de las señales de salida del módulo. Un ILA es un analizador lógico integrado que proporciona la herramienta de Vivado. Este elemento es totalmente personalizable y se puede utilizar para monitorizar las señales internas de un diseño.

Una vez cargado el bitstream en la FPGA se puede hacer uso del ILA. En la figura 5.13 se puede ver la interfaz del ILA en Vivado. Además, se muestra la forma de onda de salida del módulo jammer. Como era de esperar, la señal representada es simplemente ruido.

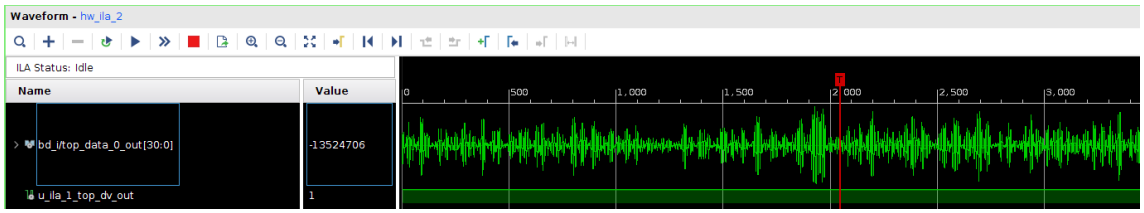
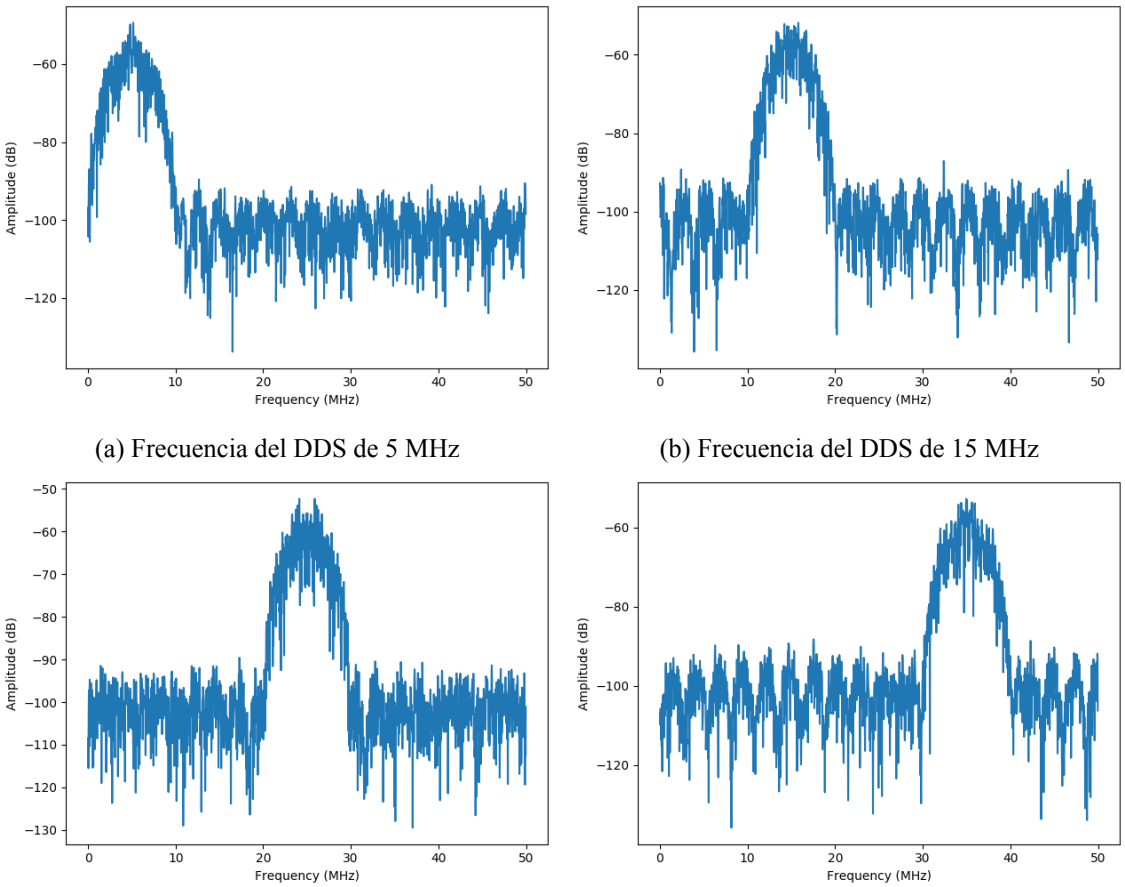


Figura 5.13: Señal de salida obtenida en el ILA

Para poder comprobar que los datos generados se corresponden con los mostrados en la parte de simulación, se han extraído los datos del ILA para distintas frecuencias del DDS. A estos datos se les ha realizado la FFT obteniendo los resultados mostrados en la figura 5.14. En esta figura se muestran las mismas frecuencias que se expusieron en la parte de simulación.



(a) Frecuencia del DDS de 5 MHz

(b) Frecuencia del DDS de 15 MHz

(c) Frecuencia del DDS de 25 MHz

(d) Frecuencia del DDS de 35 MHz

Figura 5.14: FFTs de los datos de salida del módulo jammer obtenidos con el ILA para diversas frecuencias del DDS

Capítulo 6

Conclusiones y trabajos futuros

El objetivo de este proyecto era el de realizar un jammer configurable basado en FPGA que sea adecuado para interferir comunicaciones FHSS. A lo largo del presente documento se ha expuesto el desarrollo llevado a cabo para la consecución de dicho objetivo. En primer lugar, se ha generado ruido gaussiano aplicando el teorema central del límite a números uniformemente distribuidos, los cuales se han producido mediante LFSRs. Posteriormente, se ha filtrado el ruido para obtener un ancho de banda similar al que usa la señal que se pretende interferir. Para esta parte se ha usado un filtro FIR paso bajo de orden 39. Por último, tras la obtención del ruido en banda base y con un ancho de banda limitado, se ha utilizado un mezclador digital que permite desplazar la banda de ruido a distintas frecuencias intermedias. Esto posibilita seguir los saltos que va dando la señal a interferir.

Todo este diseño ha sido implementado en una FPGA Xilinx Zynq Ultrascale+, la cual incorpora un microprocesador de ARM que permite realizar la configuración del diseño realizado. Siendo los parámetros programables los coeficientes del filtro, la frecuencia central a la que se desplaza la banda de ruido y la activación o desactivación del generador de ruido. Para la comunicación entre el microprocesador y el hardware diseñado se ha hecho uso de la interfaz AXI-Lite añadiéndola al módulo creado. Se puede concluir, por tanto, que se han cumplido los objetivos planteados para este proyecto.

En cuanto a los trabajos futuros, como ya se expuso en la sección 4.1, faltan por realizar más partes del proyecto final. En primer lugar, se hace necesario un módulo para adaptar los datos de salida obtenidos en este proyecto al formato de entrada de los módulos que realizarán la comunicación con el DAC. En concreto, habrá que recortar la señal a 14 bits y utilizar la interfaz de comunicación AXI-Stream. Para la comunicación con el DAC será necesario hacer uso del protocolo JESD204B al ser un transmisor de alta velocidad.

Seguidamente, habrá que realizar la implementación de la parte ES del sistema donde se detecte la señal a interferir. En esta parte, de nuevo, habrá que hacer uso del protocolo JESD204B para la comunicación con el ADC y la adaptación posterior de los datos recibidos.

Por último, para la conclusión del proyecto final será necesaria la integración de todos los elementos presentes en el diseño, tanto de los módulos diseñados para la FPGA como de la comunicación entre la FPGA y los ADCs y DACs necesarios. Además, con el prototipo ya funcional se realizarán pruebas finales midiendo la latencia presente en el diseño y se optimizará en caso de ser necesario para cumplir las especificaciones por parte del cliente.

Bibliografía

[1] Richard Poisel. *Modern communications jamming principles and techniques*. Artech House, 2011.

[2] Dina Katabi y Shyamnath Gollakota. "iJam: Jamming oneself for secure wireless communication". En: (2010).

[3] Kanika Grover, Alvin Lim y Qing Yang. "Jamming and anti-jamming techniques in wireless networks: a survey". En: *International Journal of Ad Hoc and Ubiquitous Computing* 17.4 (2014), págs. 197-215.

[4] Yan Yunbin, Quan Houde y Cui Peizhang. "Research on Follower jamming of FH Communication". En: *2011 International Conference on Communication and Eletronics Information*. Vol. 2. 2011, págs. 244-247.

[5] *Creating and using CI/CD pipelines | GitLab*. `https://docs.gitlab.com/ee/ci/pipelines.html`. (Accedido el 12/14/2019).

[6] *News — GHDL 0.37-dev documentation*. `https://ghdl.readthedocs.io/en/latest/index.html`. (Accedido el 12/14/2019).

[7] *Linux Test Project - Coverage » lcov*. `http://ltp.sourceforge.net/coverage/lcov.php`. (Accedido el 12/14/2019).

[8] *cocotb/cocotb: Coroutine Co-simulation Test Bench*. `https://github.com/cocotb/cocotb`. (Accedido el 12/14/2019).

[9] *Welcome to Cocotb's documentation! — cocotb 1.2.0 documentation*. `https://cocotb.readthedocs.io/en/latest/index.html`. (Accedido en 12/14/2019).

[10] Karen L Fitch, Kathryn Gillis y Abby Harrison. "Digital Programmable Gaussian Noise Generator". Tesis doct. Worcester Polytechnic Institute, 2015.

[11] Andreas Klein. "Linear feedback shift registers". En: *Stream Ciphers*. Springer, 2013, págs. 17-58.

[12] Scott Miller y Donald Childers. *Probability and random processes: With applications to signal processing and communications*. Academic Press, 2012.

[13] Vicente Torres y M^a José Canet. *Apuntes asignatura Procesado de Señal en Sistemas Electrónicos*. MUISE. Universidad Politécnica de Valencia. 2019.

[14] Javier Valls y Asunción Pérez. *Apuntes asignatura Procesado Digital de la Señal en FPGA*. MUISE. Universidad Politécnica de Valencia. 2019.

[15] *UltraScale Architecture DSP Slice User Guide*. UG579 (v1.9). Xilinx.

[16] *AXI Reference Guide*. UG761 (v13.1). Xilinx.

[17] *ZCU106 Evaluation Board User Guide*. UG1244 (v1.4). Xilinx.