

Trabajo final

Eléctronica Digital
Para Comunicaciones

Antonio José Aragón Molina

Índice

Índice	ii
Índice de Figuras	1
1. Explicación general	2
2. Implementación en Matlab	4
<i>2.1. Transmisor</i>	4
<i>2.2. Canal</i>	9
<i>2.3. Receptor</i>	12
<i>2.3.1. Estimador</i>	12
<i>2.3.2. Ecualizador</i>	14
<i>2.3.3. Traducción a bits</i>	15
3. Implementación en VHDL	17
<i>3.1. Estimador</i>	17
<i>3.1.1. Contador</i>	19
<i>3.1.2. PRBS</i>	20
<i>3.1.3. Interpolador</i>	21
<i>3.1.4. FSM</i>	22
<i>3.2. Ecualizador</i>	24
4. Verificación	27
<i>4.1. Lectura de datos de entrada y salida</i>	27
<i>4.2. Comparación de resultados</i>	28
<i>4.3. Ficheros de test</i>	29
<i>4.4. Otros métodos de verificación</i>	29

ÍNDICE DE FIGURAS

Figura 2.1 – Constelación creada, 16QAM en este caso.....	5
Figura 2.2 – Constelación con codificación grey del estándar DVB-T para la 16QAM.	6
Figura 2.3 – Registro de generación de la secuencia pseudoaleatoria.....	7
Figura 2.4 – Todas las portadoras para el modo 2K. Representación del módulo.	8
Figura 2.5 – Representación temporal de la señal transmitida.	9
Figura 2.6 – Ecuación para simulación del canal real.....	10
Figura 2.7 – Módulo del canal real (en dB) dependiente de la frecuencia (en Hz)...	11
Figura 2.8 – Comparación de la gráfica del canal real con el canal estimado.....	13
Figura 2.9 – Módulo del símbolo estimado en la recepción.	14
Figura 2.10 – Constelación 64QAM recibida.....	15
Figura 3.1 – Estimador implementado.	18
Figura 3.2 – Bloque contador.	19
Figura 3.3 – Bloque PRBS.....	20
Figura 3.4 – Bloque Interpolador.	21
Figura 3.5 – Bloque FSM.	22
Figura 3.6 – Diagrama de estados.....	23
Figura 3.7 – Ecualizador implementado.	25
Figura 4.1 – Comparación del canal real, junto a la estimación de Matlab y a la de VHDL para 8K y una 64QAM.....	28
Figura 4.2 – Resultados del code coverage.....	30

1. EXPLICACIÓN GENERAL

Este trabajo ha consistido en la implementación en VHDL de un receptor básico para una comunicación siguiendo el estándar DVB-T.

Consta de dos partes principales, el estimador de canal y el ecualizador. El objetivo conjunto es compensar el efecto del canal en los símbolos recibidos, para poder recuperar los símbolos transmitidos.

Antes de comenzar la implementación de bloques en VHDL, ha sido creado un archivo en Matlab que contempla la transmisión y recepción completa de uno o más símbolos, simulando los efectos que el canal produce en la transmisión a través del medio. La finalidad de este código es por un lado simular los efectos que produce un canal dispersivo en un símbolo transmitido, y, por otro lado, servir de apoyo para la correcta verificación del código creado en VHDL.

Una vez entendido el funcionamiento del receptor completo, y del estándar DVB-T, se ha procedido a la implementación en VHDL. La entrada que recibirá el receptor VHDL es el conjunto de portadoras complejas recibidas, llegando una a una en serie cada ciclo de reloj. No es competencia de este trabajo separar el prefijo cíclico, ni traducir los símbolos transmitidos a bits, entre otros, como tampoco se ha realizado la implementación en una FPGA real.

Por último, los archivos de Matlab han sido adaptados a Octave para poder ser ejecutados simultáneamente al código de VHDL desde un archivo superior, creado sobre Python. De este modo ha sido sencilla la detección de error y verificación del correcto funcionamiento de los bloques. Al ser la verificación una parte importante de la asignatura, se le ha dedicado un capítulo en esta memoria, donde se

detallarán todos los test creados.

Como comentario adicional, durante el desarrollo de este trabajo, todos los códigos han ido siendo subidos a un repositorio público en Gitlab¹. Aunque no se ha explorado en profundidad la utilidad de esta herramienta, ya que solo un usuario tenía acceso al repositorio, ha sido muy interesante como introducción a este complejo mundo. En lo personal me parece una herramienta muy útil que cualquier desarrollador de software debería conocer. Se explicarán algunos detalles adicionales explorador sobre este tema en el último capítulo.

¹ Enlace al repositorio: [edcmit / 2021-2022 / antaramol · GitLab](https://edcmit/2021-2022/antaramol · GitLab)

2. IMPLEMENTACIÓN EN MATLAB

Como se ha comentado en la introducción, primero ha sido simulada una transmisión completa en Matlab, desde la generación de bits (que será aleatoria, no tendrá información), hasta la recepción y comparación de la tasa de error de bit.

Esta comunicación completa se encuentra en el archivo ‘Matlab/trabajo.m’, dentro del directorio adjuntado. Aunque este archivo tiene comentarios que con detalle explican el funcionamiento, vamos a explicar los puntos del código.

2.1. Transmisor

El transmisor es representado por la primera parte del código, y su función es generar los símbolos que serán enviados al receptor. En las primeras líneas, nos encontramos con algunos parámetros de configuración. Estos parámetros son:

- NUM_SYMB: número de símbolos a transmitir.
- SEED: semilla para la generación de la misma secuencia de bits entre simulaciones.
- CONSTEL: constelación, a elegir entre QPSK, 16QAM y 64QAM.
- MODO: modo de transmisión, a elegir entre 2K y 8K.
- SNR: relación señal a ruido expresada en dB.
- CP: prefijo cíclico, a elegir entre 1/4, 1/8, 1/16 o 1/32.

Los parámetros pueden ser editados de forma sencilla, sin tener que preocuparse por cambiar algún otro valor dentro del código.

Una vez configurada, se asignarán valores a algunas variables dependientes de la configuración, como el número de portadoras que depende del modo. Posteriormente, se creará la constelación elegida.

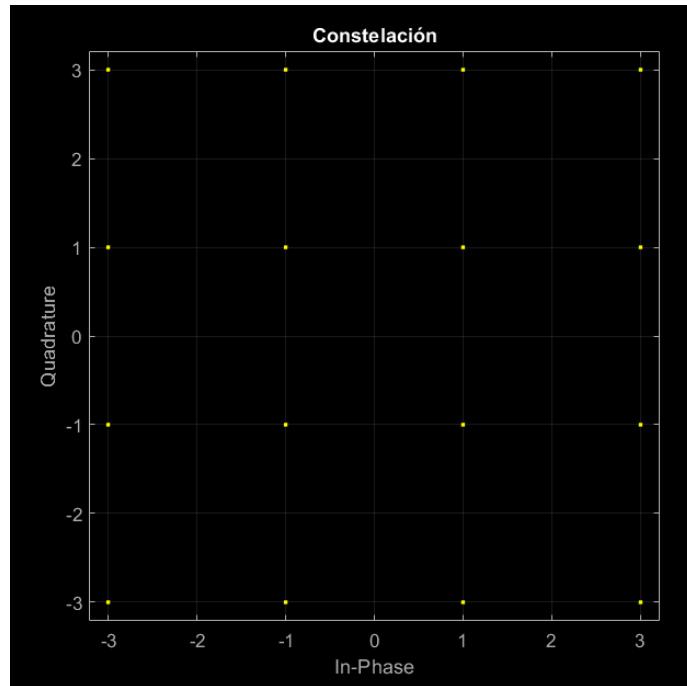


Figura 2.1 – Constelación creada, 16QAM en este caso.

Estos símbolos complejos han sido escritos en una matriz fila, de 16 filas en este caso. El orden que han seguido es el número en decimal que representan, y se ha realizado mirando la constelación grey del estándar DVB-T.

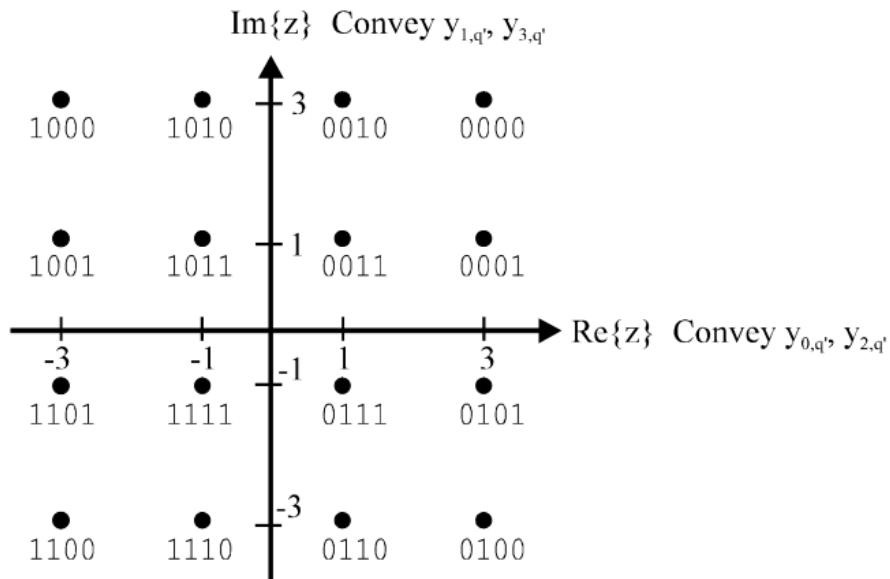


Figura 2.2 – Constelación con codificación grey del estándar DVB-T para la 16QAM.

El siguiente paso es normalizar la constelación antes de transmitirla, dividiendo entre el módulo del símbolo más energético.

Ahora vamos a generar los pilotos, en nuestro caso estarán colocados siempre cada 12 portadoras de información. En un receptor real la implementación de pilotos es mucho más compleja ya que en cada símbolo van cambiando de posición, pero la decisión se debe a simplificar el diseño.

Los pilotos tienen todos amplitud $4/3$ en valor absoluto, pero cada uno tiene un signo que le ha sido asignado mediante la generación de una secuencia pseudoaleatoria. Para la prueba de la generación de este registro se ha dejado además el archivo 'Matlab/registros.m' en la carpeta.

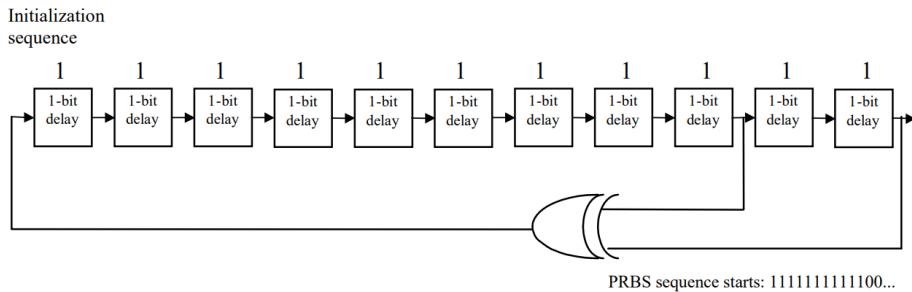


Figura 2.3 – Registro de generación de la secuencia pseudoaleatoria.

Fuente: Estándar DVB-T.

Aunque solo se aprovechará para nuestro caso 1 de cada 12 signos generados, se van a generar tantos símbolos como portadoras de información tengamos según el modo, ya que así es como dicta el estándar que debe ser. Si generásemos signos solo para los pilotos y el receptor supiese cómo interpretarlo, sería igualmente funcional e incluso más eficiente a priori ya que no generaríamos signos de más. Sin embargo, se ha decidido seguir el estándar.

Una vez hemos convertido nuestra secuencia de bits aleatorios en una serie de símbolos, hemos normalizado la constelación y hemos insertado los pilotos, hacemos la transformación al dominio del tiempo. El paso al dominio del tiempo es necesario para hacer la convolución con el canal, aunque luego en el receptor volvamos a pasar a frecuencia la señal recibida.

Para hacer más eficientes estos cambios de dominio temporal a frecuencial y viceversa, vamos a llenar nuestro símbolo OFDM con ceros a ambos lados, hasta llegar a un número potencia de 2. Por ejemplo, para el modo 2K, tenemos 1705 portadoras, pero vamos a llenar con ceros al principio y al final hasta que el símbolo contenga 2048 portadoras.

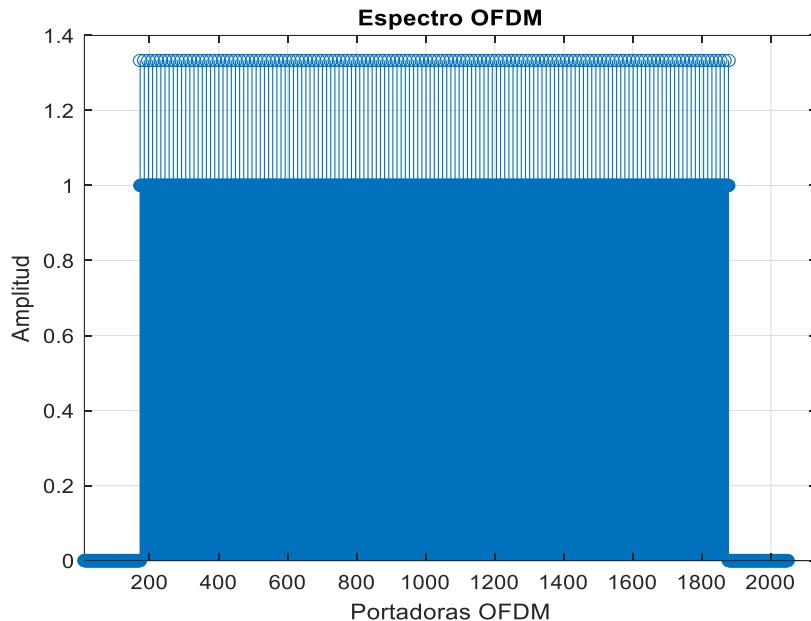


Figura 2.4 – Todas las portadoras para el modo 2K. Representación del módulo.

Convertimos el símbolo al dominio temporal y añadimos el prefijo cíclico como última medida para prevenir errores. La señal estaría lista para pasar a través del canal.

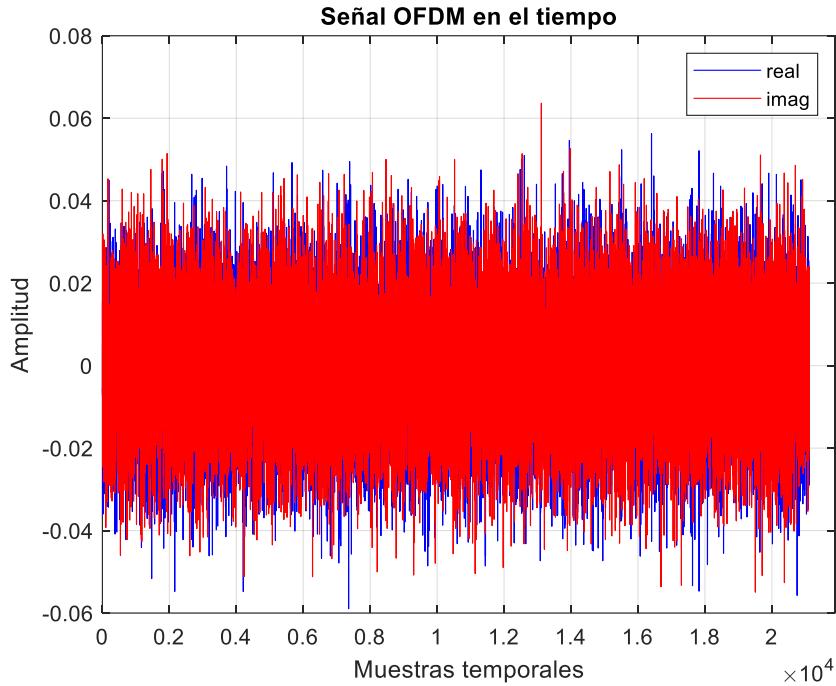


Figura 2.5 – Representación temporal de la señal transmitida.

2.2. Canal

Vamos a simular el efecto que produciría un canal real sobre los símbolos transmitidos. El canal real en Matlab será una matriz de 2048 portadoras para el modo 2K y 8192 para el modo 8K, y vendrá dada por el numerador de la ecuación de la figura 2.6.

$$y(t) = \frac{\rho_0 x(t) + \sum_{i=1}^N \rho_i e^{-j\theta_i} x(t - \tau_i)}{\sqrt{\sum_{i=0}^N \rho_i^2}}$$

Figura 2.6 – Ecuación para simulación del canal real.

Los valores de los parámetros han sido obtenidos de una tabla en el anexo DVB-T. Este sumatorio ha sido implementado mediante multiplicaciones matriciales como puede observarse en el código. En un principio se implementó mediante un bucle for, pero fue reemplazado para ganar eficiencia en el código.

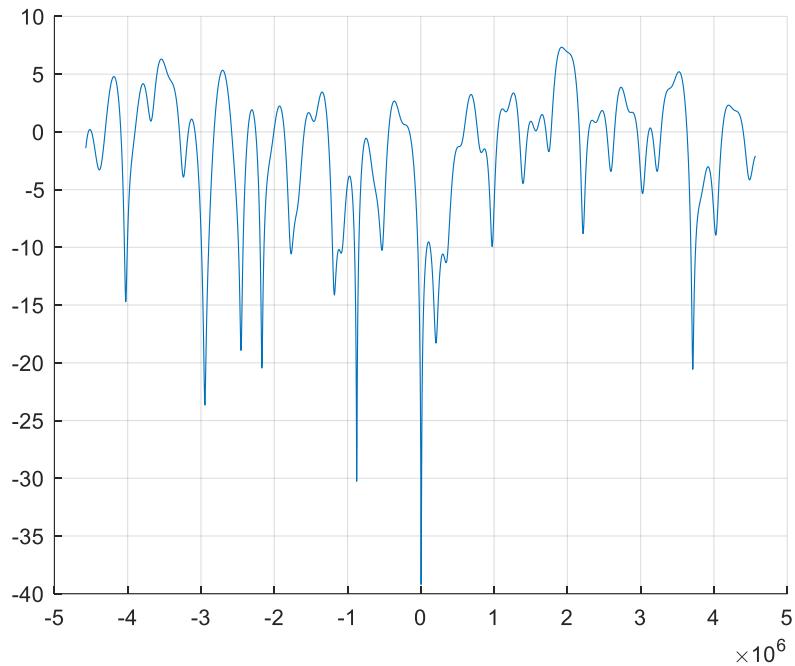


Figura 2.7 – Módulo del canal real (en dB) dependiente de la frecuencia (en Hz).

Convertimos el canal al dominio del tiempo y realizamos la convolución con la matriz de símbolos transmitidos para simular el efecto. Además, añadimos ruido AWGN, en proporción a la SNR determinada al comienzo.

Ya tendríamos el símbolo que llegaría al receptor.

2.3. Receptor

Una vez ha llegado un símbolo, el receptor debe intentar recuperar la información transmitida. Primero debe desprenderse de la redundancia y quedarse solo con la información, eliminando el prefijo cíclico y las portadoras que habíamos añadido inicialmente con valor nulo. Estas operaciones son inversamente equivalentes a las realizadas en el transmisor.

Nos quedaríamos con un vector de tamaño 1705 o 6817, dependiendo del modo. En este punto el receptor va a intentar estimar qué aspecto tiene el canal en frecuencia, para después intentar compensar estos cambios ecualizando la señal recibida.

2.3.1. Estimador

Tal y como se ha comentado, pero siendo un poco más técnicos, este apartado del código busca obtener una matriz lo más parecida posible a la matriz que simulaba el canal real en la sección anterior.

Para lograrlo, entran en juego los pilotos, y es que al ser una detección coherente, desde el receptor vamos a saber qué pilotos se han transmitido, cómo han cambiado en su paso por el canal y qué provoca ese cambio.

Nos quedamos por tanto con los pilotos y los normalizamos dividiendo por los pilotos transmitidos. Tenemos el canal muestreado, vamos a unir los puntos mediante una interpolación lineal, que es la más sencilla de todas, de nuevo buscando la sencillez en el diseño.

Esta interpolación se implementó en primer lugar mediante un bucle for, pero después fue descubierta la función 'interp1' de Matlab. Esta función realiza la interpolación lineal de forma matricial, que, comprobando mediante el comando 'tic-toc' mejora la eficiencia del bucle for. Además mediante esta función es sencillo cambiar a otro tipo de interpolación, de modo que podríamos cambiar a otra más compleja a modo de prueba con facilidad. De todos modos, se ha dejado comentado el bucle original por si fuera necesario utilizarlo en algún momento.

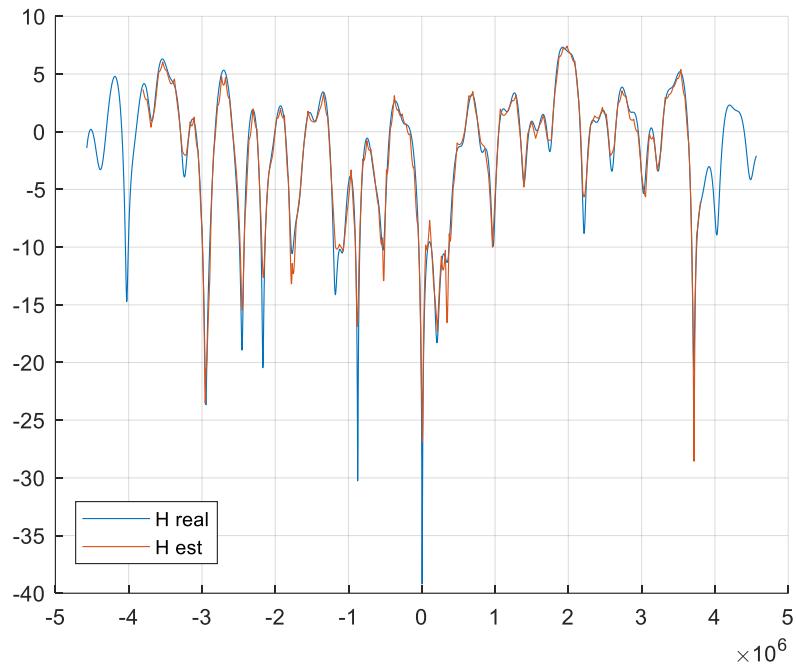


Figura 2.8 – Comparación de la gráfica del canal real con el canal estimado.

Se representa el valor absoluto, pero se ha probado la comparación de la parte real y la imaginaria por separado. El cambio entre representaciones es obvio y además está comentado en el código.

Como comentario, se observa una ondulación entre los lóbulos superiores, debido a la interpolación. No ha sido posible desacernos de estas.

Para el modo 8K, la estimación es más precisa ya que tiene más puntos.

2.3.2. Ecualizador

Una vez tengamos la estimación del canal, debemos compensar sus efectos en los símbolos recibidos. Este proceso de compensación de frecuencias es denominado ecualización.

Para eliminar el efecto del canal, de forma teórica dividimos el símbolo recibido directamente por el canal estimado. Si hubiésemos estimado idealmente el canal en todos sus puntos, el resultado sería el símbolo transmitido originalmente. Sin embargo esto no es así en la realidad, lo que provocará que nuestra BER final no sea cero, además del ruido que hemos añadido.

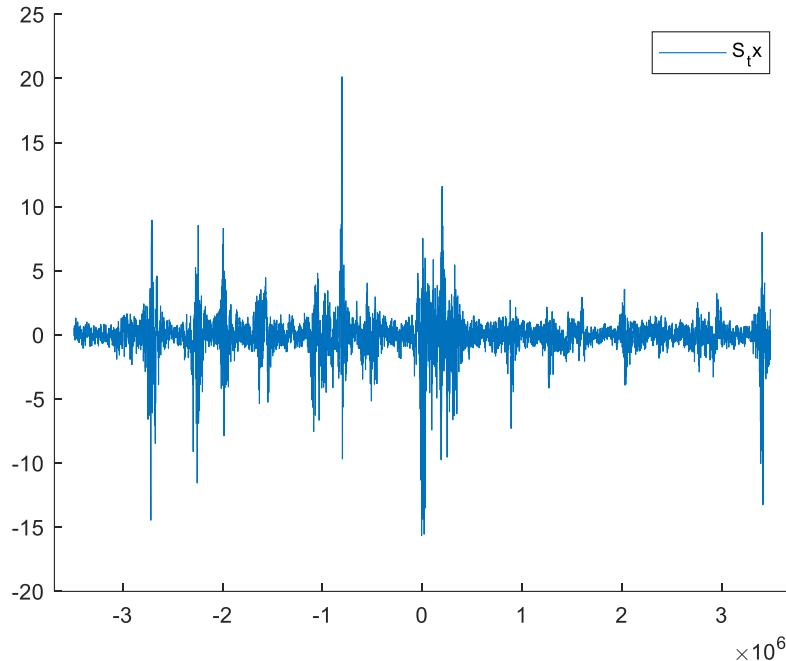


Figura 2.9 – Módulo del símbolo estimado en la recepción.

2.3.3. Traducción a bits

El ultimo paso en recepción es recuperar los bits transmitidos a partir de los símbolos estimados. Este paso se realiza colocando cada símbolo complejo en el plano IQ, y, comparando con la constelación que ha elegido el transmisor, asignar un número en binario según al que más se parezca. De esta función se encarga el Demapper.

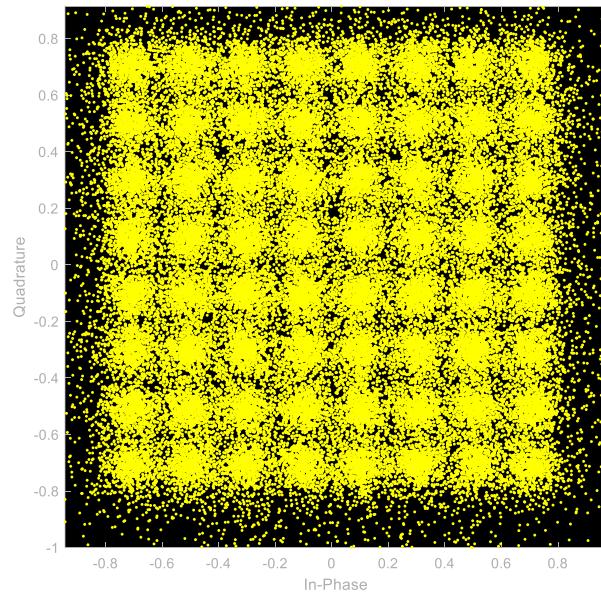


Figura 2.10 – Constelación 64QAM recibida.

Aunque podemos compensar en cierta medida el efecto del canal, nunca podremos hacerlo de forma ideal. Es por eso que siempre tendremos cierto error, que podremos medir mediante la BER. Este resultado se muestra al final de la ejecución del código, es un dato interesante para comprobar qué tan bueno es nuestro receptor, y además sirve para comprobar unas configuraciones con otras.

Como era de esperar, las constelaciones con un mayor número de símbolos registran más errores. El modo 8K reduce la tasa de error con respecto al modo 2K. La redundancia del prefijo cíclico también ayuda a reducirla.

NOTA: En el código se han preparado una serie de figuras para ser mostradas al ejecutarlo, pero pueden ser comentadas si solo se desea obtener la BER final, haciendo de este modo mucho más rápida su ejecución. Del mismo modo pero al contrario, hay un par de figuras comentadas ya que no aportan mucha información. Están comentadas y pueden ser descomentadas si se quisieran visualizar.

3. IMPLEMENTACIÓN EN VHDL

La implementación en VHDL comenzó después de acabar la parte de Matlab. Este orden se debe a que las salidas del código de Matlab nos servirán para verificar que nuestros bloques de VHDL están funcionando correctamente. Además, para el símbolo dado como entrada del receptor en VHDL, también utilizaremos los datos de Matlab, justo después de eliminar la redundancia en el receptor.

3.1. Estimador

Este bloque tiene como objetivo obtener la respuesta del canal en frecuencia de la forma más fidedigna posible. Después en el ecualizador, se realizará la compensación del efecto del canal y se recuperarán los símbolos transmitidos.

Para la implementación del estimador, se proponía el uso de una memoria ram de doble lectura/escritura, donde se escribiría el signo entero una vez llegara al receptor. Un bloque posterior que implementase una máquina de estados, se encargaría de ir leyendo la memoria una vez estuviesen disponibles las primeras 12 portadoras.

Sin embargo, este diseño implicaría el uso de una memoria de tamaño considerable (11 bits para 2K y 13 bits para 8K). Además es difícilmente escalable, ya que de un modo de transmisión a otro el tamaño la memoria requería ser multiplicada por cuatro.

También se probó a utilizar una memoria de tamaño más reducido, de solo 12 posiciones, para que la máquina de estados tuviera siempre disponible dos pilotos para el siguiente bloque, el interpolador. El problema era que el interpolador tardaba más de 12 ciclos en completar su tarea, en concreto 14 ciclos. Entonces al

cabo de un tiempo, la recepción perdía su sincronía y los datos se estimaban de forma incorrecta.

La solución fue hacer el interpolador más eficaz², tardando en esta nueva versión solo 12 ciclos, los mínimos necesarios para no cortar el flujo de entrada. Es sencillo ver que si se cumple esto, la memoria es totalmente inútil. Cuando el interpolador termina su tarea y está listo para recibir un nuevo piloto de entrada, ese piloto está llegando, ya que llegan cada 12 ciclos. Esta solución es más compleja que utilizar una memoria y leer de ella, pero es la práctica ocupa menos recursos y

Otra posible solución que no ha sido implementada en ningún momento sería emplear una memoria un poco más pequeña que la original, de tamaño el número de pilotos. Ocuparía menos recursos que la primera memoria planteada, pero seguiría siendo escalable, y además requeriría diseñar otro bloque que controlada la escritura cada 12 portadoras.

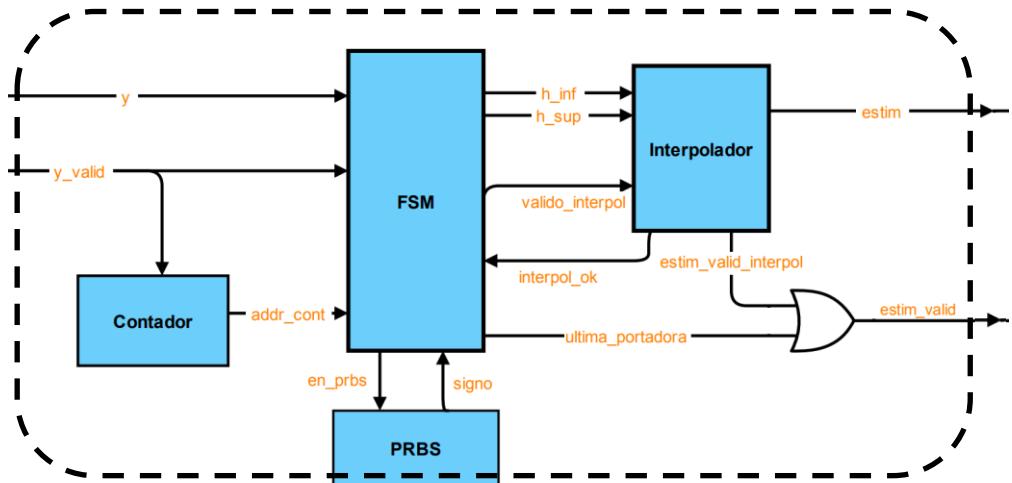


Figura 3.1 – Estimador implementado.

² Los bloques son explicados en detalle más adelante.

Las señales tienen en la figura 3.1 los mismos nombres que en el archivo top. La entrada 'y' del estimador se trata de un vector de 20 bits, donde los 10 primeros bits corresponden a la parte real de las portadoras recibidas y los últimos bits a la parte imaginaria. Cada ciclo de reloj 'y' cambia su valor a la siguiente portadora. Esta entrada va acompañada de la señal 'y_valid', que indica que el dato que se está escribiendo es válido y puede ser leído.

Van a ser presentados los bloques del estimador, representando en color verde las entradas del bloque y en rojo las salidas. De todos modos, el código correspondiente con la implementación de estos bloques se encuentra en el directorio 'VHDL' dentro de la carpeta adjunta.

3.1.1. Contador

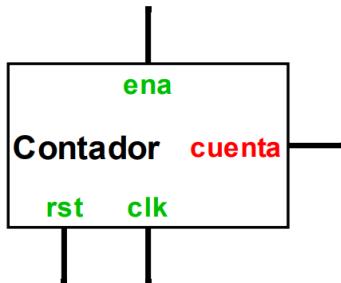


Figura 3.2 – Bloque contador.

Este bloque consiste en un sencillo contador de 4 bits cuya única finalidad es determinar a la máquina de estados que ya han pasado 12 ciclos. Esto será útil al principio del símbolo, ya que debemos esperar a que lleguen los dos primeros pilotos para realizar la primera interpolación. Luego, la sincronía entre la entrada y el tiempo que tarda el interpolador en realizar su tarea harán que no haga falta

contar más veces.

3.1.2. PRBS

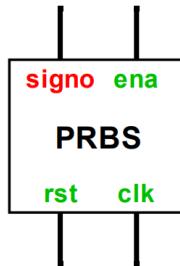


Figura 3.3 – Bloque PRBS.

Este bloque se encarga de generar unos y ceros que serán interpretados por la máquina de estados como un signo negativo o positivo, respectivamente. La generación de esta secuencia pseudoaleatoria se basa en el registro que se define en el estándar DVB-T, exactamente igual que en la parte de Matlab.

El funcionamiento de este bloque se basa en devolver una salida justo en el instante en el que recibe una señal de enable, para mantener la sincronía de todos los bloques. El registro debe tener preparado el bit de salida en el valor correcto antes de que llegue la señal de enable, de modo que pueda ser leído inmediatamente. Esto se consigue adelantando un ciclo la salida, inicializando el registro con el siguiente valor al inicial, es decir, un cero en la posición más alejada de la salida en vez de todo unos.

3.1.3. Interpolador

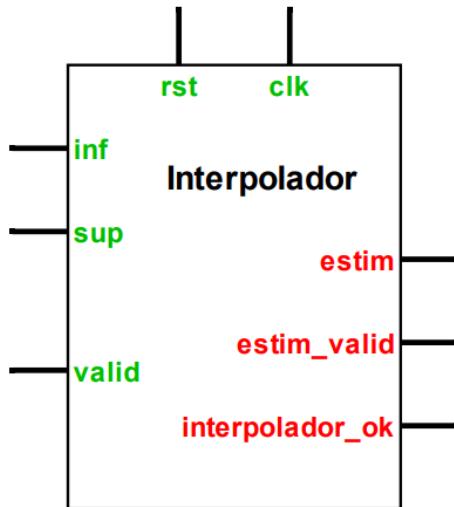


Figura 3.4 – Bloque Interpolador.

El interpolador recibe los valores complejos de dos pilotos, el inferior y el superior, y devuelve de forma secuencial los valores de 12 portadoras, incluyendo el piloto inferior.

En realizar esta tarea tarda justo 12 ciclos, necesario para la sincronización de todos los bloques, como se ha explicado antes. Para lograr esta eficiencia, han sido necesarios dos ajustes:

- Piloto inferior preparado: justo cuando llega el flanco de subida de 'valid', debemos escribir el valor del primer piloto en 'estim' y marcarlo como válido para el siguiente bloque. Esto es posible debido a que conocemos el piloto inferior por el ciclo anterior, entonces podemos asignarlo

directamente.

- Aviso a la FSM de que voy a terminar: el interpolador debe indicar a la máquina de estados que ya está escribiendo el último piloto. Es necesario este previo aviso para que la máquina de estados pueda preparar el siguiente estado sin retraso, y se implementa gracias a la señal ‘interpolador_ok’, que se activa durante el último ciclo de escritura del interpolador.

3.1.4. FSM

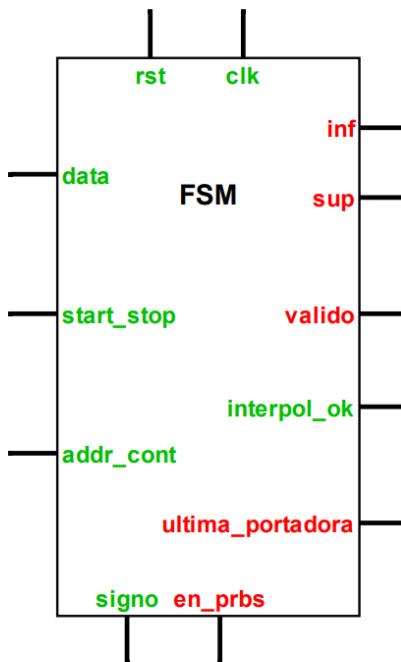


Figura 3.5 – Bloque FSM.

Este bloque es el más grande y complejo de nuestro diseño del estimador, por eso se ha dejado en la última posición en esta memoria. Se encarga de controlar las entradas y el resto de bloques.

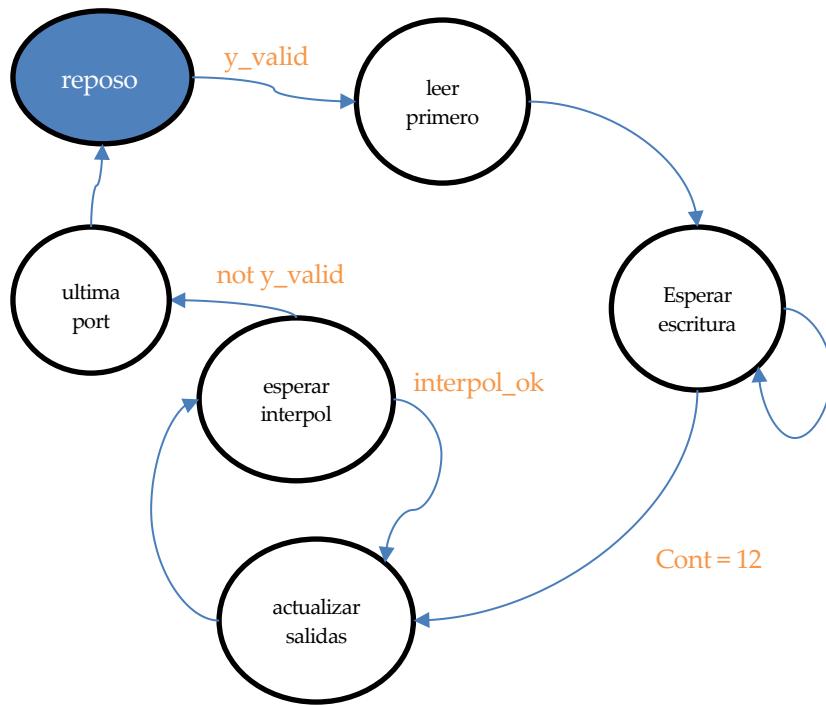


Figura 3.6 – Diagrama de estados.

En la figura se representan los estados de la FSM y las transiciones, pero no las salidas.

La FSM actualiza cada 12 ciclos de reloj los valores de los pilotos inferior y superior, para que el interpolador pueda utilizarlos. El estado ‘ultima_portadora’ se encarga de que a la salida se lea un último piloto que queda descolgado. Esto se debe a que el interpolador escribe a la salida el piloto inferior y las siguientes 12 portadoras, no escribiendo el actual piloto superior ya que en la siguiente iteración este será escrito al ser convertido en piloto inferior. Al ser impares, el último piloto queda descolgado, sirviendo como solución una puerta OR que indique al siguiente bloque que puede leer un ciclo más de ‘estim’.

En medio del proceso, la máquina de estado leerá la dirección del contador para saber cuándo iniciar la interpolación, y estará activando el bloque PRBS para poder asignar el signo a los pilotos. Igual que en Matlab, se genera un signo para cada portadora, sea o no piloto, aunque solo nos serán útiles los de los pilotos.

3.2. Ecualizador

Este bloque implementa la misma funcionalidad que el ecualizador de Matlab, compensar el efecto del canal en los símbolos recibidos.

La implementación en VHDL no ha sido la más limpia posible, teniendo dos bloques divisores combinacionales que por sí solos consumirían la mayor parte de los recursos de nuestra FPGA. Es la principal línea de mejora de este diseño.

Se ha tomado la decisión de dejar estos bloques, debido a que el divisor del que disponíamos iba a requerir el uso de una memoria de gran tamaño, donde pudiésemos leer y escribir las portadoras. Con todas las decisiones tomadas hasta el momento para evitar el uso de la memoria, ponerla ahora hubiese significado una complejidad excesiva en la parte anterior a cambio de nada, así que como primera prueba de implementación es suficiente con los divisores combinacionales.

Se comentó la idea de implementar un divisor pipeline que sacara los datos de forma secuencia con N ciclos de retraso, lo cual sería una buena solución y que encajaría con nuestro diseño. Sin embargo finalmente el tiempo no fue suficiente.

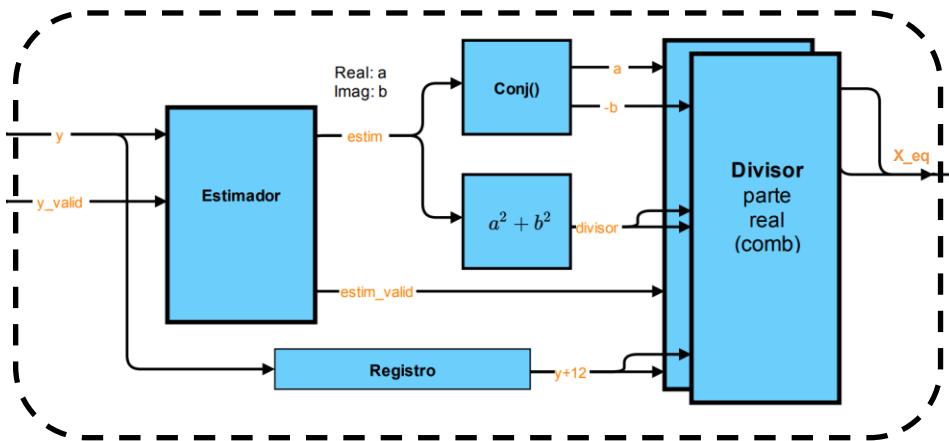


Figura 3.7 – Ecualizador implementado.

Como comentario, con respecto a la implementación prouesta, hemos tenido que adelantar la multiplicación por el factor de escala antes de la división, debido a que si dividíamos números de 10 bits perdíamos toda nuestra sensibilidad. Esta multiplicación se basa en añadir siete ceros a la derecha en los numeradores de la división.

El registro se encarga de retrasar la entrada 12 ciclos de reloj para sincronizarla con la salida del estimador. 12 ciclos es el retraso que tiene nuestro sistema con respecto a la entrada, ya que, tal y como está implementado, el ecualizador es puramente combinacional.

4. VERIFICACIÓN

La verificación es una parte importante de la asignatura, y por ello se le ha dedicado un capítulo en concreto.

Este capítulo tiene como objetivo justificar el correcto funcionamiento de nuestra implementación de un receptor para DVB-T en VHDL, mediante la explicación de pruebas realizadas.

Basta con introducir en una línea de comandos (desde el directorio donde se encuentra esta memoria) ‘python3 trabajo/general/run.py’ para realizar una transmisión y recepción completa.

Todo lo que se comenta a continuación puede comprobarse ejecutando el código de la carpeta ‘trabajo/general’, ahí están los códigos ordenados y comentados. En la carpeta ‘trabajo/dos_simbolos’ se incluyen los mismos archivos, pero con diferencias en los tests, ya que se ha preparado para comprobar el correcto funcionamiento de la transmisión de dos símbolos consecutivos, que en este caso se implementa repitiendo el símbolo.

4.1. Lectura de datos de entrada y salida

Como se ha comentado antes, la entrada del estimador implementado en VHDL será una secuencia de portadoras generada en código Matlab. La decisión ha sido escribir la matriz de portadoras en un documento con formato csv, para después poder leerlo desde un test bench en VHDL y generar una señal de entrada.

El problema con el que nos topamos se trata de que los valores de Matlab están escritos en formato double, de modo que teníamos que transformarlo para poder leerlo desde VHDL, es decir, multiplicar los valores por un factor de escala para quedarnos con un valor entero que pudiésemos interpretar. Como nuestra precisión es de 10 bits y las portadoras tienen signo, nuestro rango máximo va desde -512 hasta 511, ambos inclusive. Realizando pruebas, la potencia de dos

mayor por la que podemos multiplicar las portadoras sin que desborde es 2 elevado a 7.

Una vez los datos escritos en un fichero, desde un test bench utilizando las librerías de lectura y escritura proporcionadas por vunit, podemos introducir los datos leídos como entradas del sistema. Del mismo modo, podemos leer las salidas desde el test bench y escribirlas en un archivo con formato csv.

4.2. Comparación de resultados

En un principio, diversas pruebas fueron realizadas comparando la salida del estimador en VHDL con el estimador en Matlab. Este proceso era bastante tedioso, ya que involucraba dos sistemas operativos diferentes y, a pesar de contar con el repositorio común de gitlab, era muy tedioso comprobar resultados.

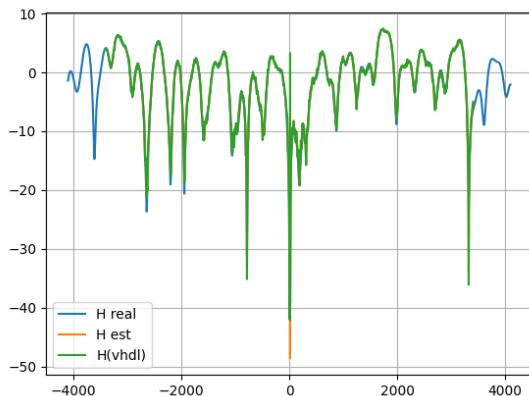


Figura 4.1 – Comparación del canal real, junto a la estimación de Matlab y a la de VHDL para 8K y una 64QAM.

Finalmente se optó por adaptar los códigos de Matlab a Oracle, para que pudieran ser ejecutados desde el archivo ‘run.py’ escrito en Python, sincronizándolos con la ejecución de los códigos de VHDL. Estos ficheros llamados ‘escribir_portadoras.m’ y ‘resultados.m’, se encargan de generar los datos de entrada para el receptor VHDL y de leer los resultados, respectivamente.

El archivo ‘run.py’ muestra por pantalla una comparación de las gráficas resultantes de Matlab y las de VHDL, además de la BER entre otros resultados.

4.3. Ficheros de test

- Run.py: archivo de simulación general. Los parámetros de la transmisión pueden ser editados igual que se explicó para el fichero de Matlab. Ejecuta el archivo de generación de Oracle, realiza los tests de VHDL, lee los resultados e imprime las gráficas y la BER. Implementa code coverage en el test general.
- Ficheros .m: se encuentran dentro de la capeta ‘trabajo/general/Matlab’. Son llamados desde ‘run.py’ y realizan las funciones del código de Matlab que le faltan a los códigos VHDL escritos.
- Test benches: Se han creado ficheros de test para la prueba individual de los bloques de contador, prbs, interpolador, fsm, estimador (por separado) y top (para todos los bloques conjuntos). Tienen nombres intuitivos y están comentados, algunos tienen más de un test. Pueden ser probados en teros, donde se pueden visualizar las formas de onda.
- Carpeta ‘dos_simbolos’: en esta carpeta se incluyen los mismos archivos que en la carpeta ‘general’, pero transmitiendo dos símbolos para comprobar el correcto funcionamiento de la señal de reset.

4.4. Otros métodos de verificación

Como método de verificación adicional, ha sido implementado code coverage

mediante la llamada a ‘gcovr’ en el fichero ‘run.py’. Esta orden solo ha sido implementada en el directorio general, y no en dos_símbolos.

Esta prueba aporta información adicional y se han detectado un par de puntos por los que la ejecución no ha pasado, lo que nos indica que quizás deberíamos realizar un test diferente, o quizás esas líneas son inútiles.

Debido al avanzado punto del proyecto, code coverage solo se ha utilizado como método de aprendizaje y no como mejora del proyecto.

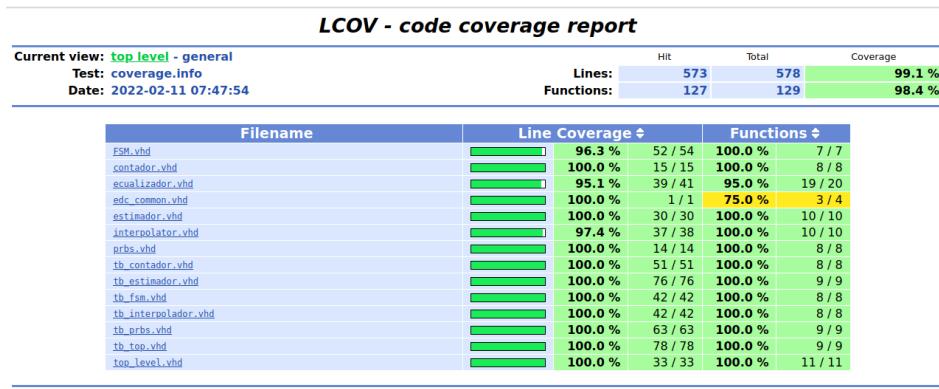


Figura 4.2 – Resultados del code coverage.

Nota: la ejecución de gcovr ha generado múltiples warnings, en principio no nos preocupan porque están relacionados con cambios de nombre de archivos.

También se ha implementado integración continua en gitlab. Ha sido creado un archivo ‘.yml’ que se encarga de cargar el entorno con las dependencias y ejecutar ‘run.py’ para el test general y el test de dos símbolos cada vez que se hace push

hacia el repositorio remoto. Además, se ha implementado la prueba de code coverage y generación de archivos de coverage para el directorio general

Esta prueba es muy interesante ya que nos enseña una potente herramienta para usar dentro de gitlab que nos permite verificar un diseño de forma rápida.