



# Proyecto estimador de canal OFDM en VHDL

Miguel Nogales González-Regueral

Escuela Técnica Superior de Ingeniería  
**Universidad de Sevilla**  
Sevilla, España

February 10, 2022

# 1 INTRODUCCIÓN

Esta memoria servirá para detallar la funcionalidad de los códigos usados para la parte sobre programación en VHDL y verificación de la asignatura "Electrónica Digital para Comunicaciones" de 1º de MUIT y será presentada junto a otra memoria complementaria sobre códigos Matlab.

Casi la totalidad de los *test benches* que se presentarán han sido realizados con la herramienta de *software* libre *CoCoTb* (COroutine based COsimulation TestBench) mientras que los pocos restantes se han realizado en VHDL. Se ha elegido CoCoTb para hacer el proyecto debido a su flexibilidad y su potencial a la hora de realizar simulaciones de código, mucho más profunda que los que se pueden realizar en VHDL, añadiendo a las simulaciones la potencia de las infinitas librerías que ofrece python para realizar casi cualquier tipo de función.

El proyecto de la asignatura trata de realizar un estimador de canal enteramente en VHDL y opcionalmente complementarlo con la equalización, además de realizar verificación del funcionamiento de los códigos aportados. Esta memoria se estructurará de forma que se mostrarán los códigos de los bloques principales, con comentarios para explicar su funcionalidad en el propio fichero, y se darán breves explicaciones en la memoria de las elecciones de diseño más relevantes. Primero una sección hablando del diseño del sistema. Posteriormente se analizarán los bloques funcionales con pequeños trozos de códigos especialmente llamativos, para acabar con un resumen de hitos y mejoras. Se añade un anexo con los códigos tanto de VHDL como de Python para verificar.

## 2 ESTIMADOR DE CANAL, DISEÑO DE LA ESTRUCTURA

En la asignatura se ha propuesto una posible implementación del sistema, pero en este trabajo se ha optado por diseñar otra desde cero, y dentro de lo posible mejorarla y añadirle ventajas que la hagan útil para determinado caso.

De esta manera, se ha desarrollado un esquema en el cual se elimina, con respecto a la propuesta original, cualquier uso de memorias DRAM y máquinas de estados, haciendo el sistema más simple y, al no necesitar estados, más rápido. Esto ha facilitado bastante la programación en VHDL, pero a costa de dificultad en el diseño, teniendo que salvar varias trabas tomando enfoques diferentes. De esta forma, se ha logrado tener el sistema funcionando bajo una arquitectura en *pipeline*, haciéndolo una implementación mucho más eficiente, efectiva y requiriendo menos dispositivos. En la Figura 2 se muestra el diagrama de bloques de esta implementación.

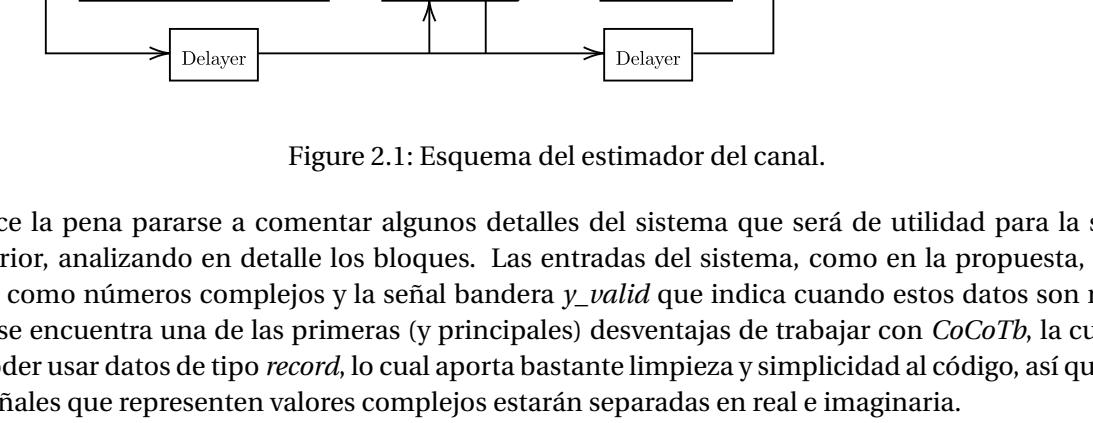


Figure 2.1: Esquema del estimador del canal.

Merece la pena pararse a comentar algunos detalles del sistema que será de utilidad para la sección posterior, analizando en detalle los bloques. Las entradas del sistema, como en la propuesta, son los datos como números complejos y la señal bandera *y\_valid* que indica cuando estos datos son nuevos. Aquí se encuentra una de las primeras (y principales) desventajas de trabajar con *CoCoTb*, la cual es el no poder usar datos de tipo *record*, lo cual aporta bastante limpieza y simplicidad al código, así que todas las señales que representen valores complejos estarán separadas en real e imaginaria.

Sobre el funcionamiento del sistema en conjunto, para empezar se asumirá, como en Matlab, que el primer piloto es la primera muestra del símbolo. La secuencia de datos comenzará a mover el sistema, haciendo que el contador tome el valor inicial 0, al estar cargado con el valor de *reset* 11, de manera que piloto y valor del contador 0 vayan al unísono. Los circuitos combinacionales controlarán la señal de activación para los bloques FSM y PRBS, donde la segunda irá un ciclo de reloj por delante, para que el PRBS tenga tiempo a calcular el siguiente signo. En el bloque FSM se calculará la *h estimada* según  $h_{estimada} = \pm Y$  dependiendo del valor del signo y por un proceso se asignarán los valores de los pilotos correspondientes. Para acabar, el bloque interpolador recibirá las señales *h* del bloque FSM en conjunto con el dato *valid*, el cual avisará de cuando debe funcionar el bloque. Las salidas pues, serán estimación real e imaginaria del canal en ese ciclo y una bandera indicando si es un dato nuevo.

Los bloques *Delayer* que se encuentran dispersos por la arquitectura del estimador sirven para compensar los retardos ocasionados por los circuitos no combinacionales, a parte de servir para asegurar en un caso de implementación real que los retardos en las puertas lógicas no afecten a nuestro circuito.

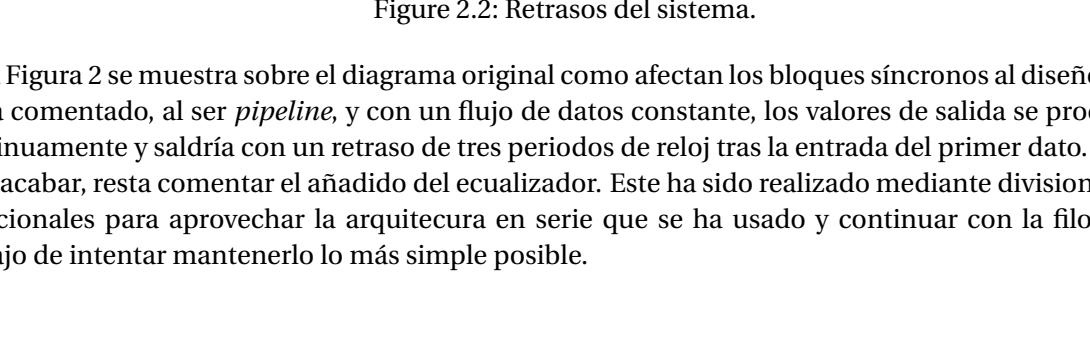


Figure 2.2: Retrasos del sistema.

En la Figura 2 se muestra sobre el diagrama original como afectan los bloques síncronos al diseño. Como se ha comentado, al ser *pipeline*, y con un flujo de datos constante, los valores de salida se procesarían continuamente y saldría con un retraso de tres períodos de reloj tras la entrada del primer dato.

Para acabar, resta comentar el añadido del ecualizador. Este ha sido realizado mediante divisiones combinacionales para aprovechar la arquitectura en serie que se ha usado y continuar con la filosofía de trabajo de intentar mantenerlo lo más simple posible.

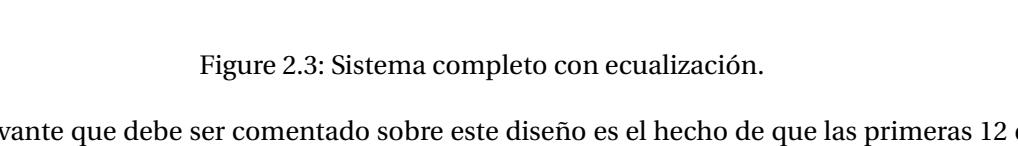


Figure 2.3: Sistema completo con ecualización.

Lo más relevante que debe ser comentado sobre este diseño es el hecho de que las primeras 12 estimaciones del estimador de canal serán inútiles, ya que corresponden a una interpolación entre el primer piloto y el valor de *reset*, por lo que los datos deben entrar en el ecualizador retrasados 12 ciclos debido a este hecho más los 3 que tarda en realizar sus procesos.

## 3.3 ANÁLISIS DE LOS CÓDIGOS

En esta sección se verán en detalle los códigos VHDL usados y los *test bench* para verificarlos uno a uno, dando énfasis en las partes más importantes. Para una mayor claridad, los códigos de VHDL estarán resaltados en azul y los de Python en negro.

### 3.1 Contador

El contador es un bloque muy simple, el cual contará de 0 a 11 para señalar cuando viene un piloto. Lo más interesante es el genérico para controlar el cuál valor de inicio, usado para poder en fase el contador con los pilotos. Es habilitado por la señal de dato válido.

```
entity counter_12 is
    generic (
        -- Generic defines beginning value for counter, in case it is needed on
        -- unclocked start
        generic (
            start : integer := 11
        );
    port (
        rst : in std_logic;
        clk : in std_logic;
        ena : in std_logic;
        out : out unsigned(3 downto 0)
    );
end counter_12;
```

En cuanto a su verificación, ha sido realizada una muy simple para comprobar su correcto funcionamiento. Así así, es interesante echarle un ojo para comentar algunas bases de CoCoTb que se usarán en muchos *testbench*.

```
import cocotb
from cocotb.triggers import Timer
from cocotb.triggers import RisingEdge
import matplotlib.pyplot as plt
import numpy as np
# import cocotb2py
```

Un ejemplo de las librerías que se usarán, donde destaca cocotb, que es la principal. Por otro lado está matplotlib.pyplot para gráficas, numpy para operar con vectores y oct2py para mover códigos matlab.

```
maxCycles = 120
currentCycle = 0
async def generate_clock(dut):
    """Generate clock pulses"""
    global maxCycles,currentCycle
    for cycle in range(maxCycles):
        dut.clk.value = 0
        await cocotb.start(generate_clock(dut))
        dut.clk.value = 1
        await Timer(1, units="ns")
        dut.clk.value = 0
        await Timer(1, units="ns")
        currentCycle += 1
        currentCycle = currentCycle % maxCycles
```

Este trozo de código define variables globales para el número máximo de ciclos y el ciclo actual. Mientras tanto en la función se define el ciclo de reloj y se aumenta este valor de la variable para seguirlo por todo el programa.

```
@cocotb.test
def test1(dut):
    global maxCycles,currentCycle
    values = []
    await cocotb.start(generate_clock(dut))
    dut.rst.value = 1
    await RisingEdge(dut.clk)
    dut.ena.value = 1
    dut.rst.value = 0
    while currentCycle < maxCycles-1:
        await RisingEdge(dut.clk)
        values.append(int(dut.cuenta))
        print(values)
```

Para scacular el test en si, primero se definen las variables, se le da el reset y se le da a aumentar el clk con la función `Await.RisingEdge`. Este será el bucle principal del *testbench*, donde se tocará la mayor parte de las cosas.

### 3.2 Delayer

Este bloque también es muy sencillo pero merece comentarse aún así. Ha sido verificado con gtkwave en conjunto con el estimador porque lo que no se muestra su *test bench* aquí. Su funcionamiento se basa en generar puestas para todos los bits de los datos y dárseles de entrada al bloque FSM.

### 3.3 PRBS

Bloque para la generación del código pseudo aleatorio para los pilotos. Es bastante directo hacer este bloque barriendo del contador. Algunos detalles interesantes son: el bloque principal, donde hubo problemas de compilación al no especificar a ghdl mediante el makefile de CoCoTb la versión del VHDL deseada.

```
process (reg)
begin
    -- not working titill GHDL_ARGS ?= -std=08 is used in GHDL compiler from
    -- cocotb (thanks to $hostotzguman)
    reg <= (11 downto 2 => reg(10 downto 1), 1 => reg(11) xor reg(9));
    -- workaround
    -- reg(11)'0 to 2 => reg(10 downto 1);
    -- reg(11)'1 to reg(11) xor reg(9);
end process;
```

También es interesante el valor de inicio del PRBS, el cual no coincide con el dicho en el estándar, que es todo 0. Esta inicialización es así debido a que es necesario retrasar los signos del PRBS un ciclo, y así se consigue sin modificar la secuencia futura, consiguiendo que sea totalmente compatible con un sistema DVBT.

```
process (clk,rst)
begin
    if rst = '1' then
        if ena = '1' then
            reg <= 0;
        elsif rising_edge(clk) then
            if ena = '1' then
                reg <= pre;
            else
                reg <= reg;
            end if;
        end if;
    end process;
```

En un principio se tuvo en cuenta que el PRBS solo se activaba cada doce ciclos, cuando no es así. Esto fue fácilmente modificado para respetar el retraso combinacional de habilitación del PRBS, dependiendo directamente del dato válido y no del contador.

Respecto al *testbench*, se han realizado dos, uno muy básico para ver los signos, y otro más interesante en el cual se usa la librería oct2py para comparar con el código Matlab.

```
from oct2py import octave
goldenSignos = octave.PRBS(1,1705)
goldenSignos = (goldenSignos[0,:]*1)/2
# dut._log.info(goldenSignos.shape)
# print(goldenSignos.shape)

plt.subplot(211)
plt.stem(signos)
plt.subplot(212)
plt.stem(goldenSignos)
plt.show()
```

Llama la atención como se accede fácilmente a la función previamente definida PRBS.m, como se tratan los datos y los plots. Además se usa `dut._log.info` para escribir en el logger con el nivel de altera "info".

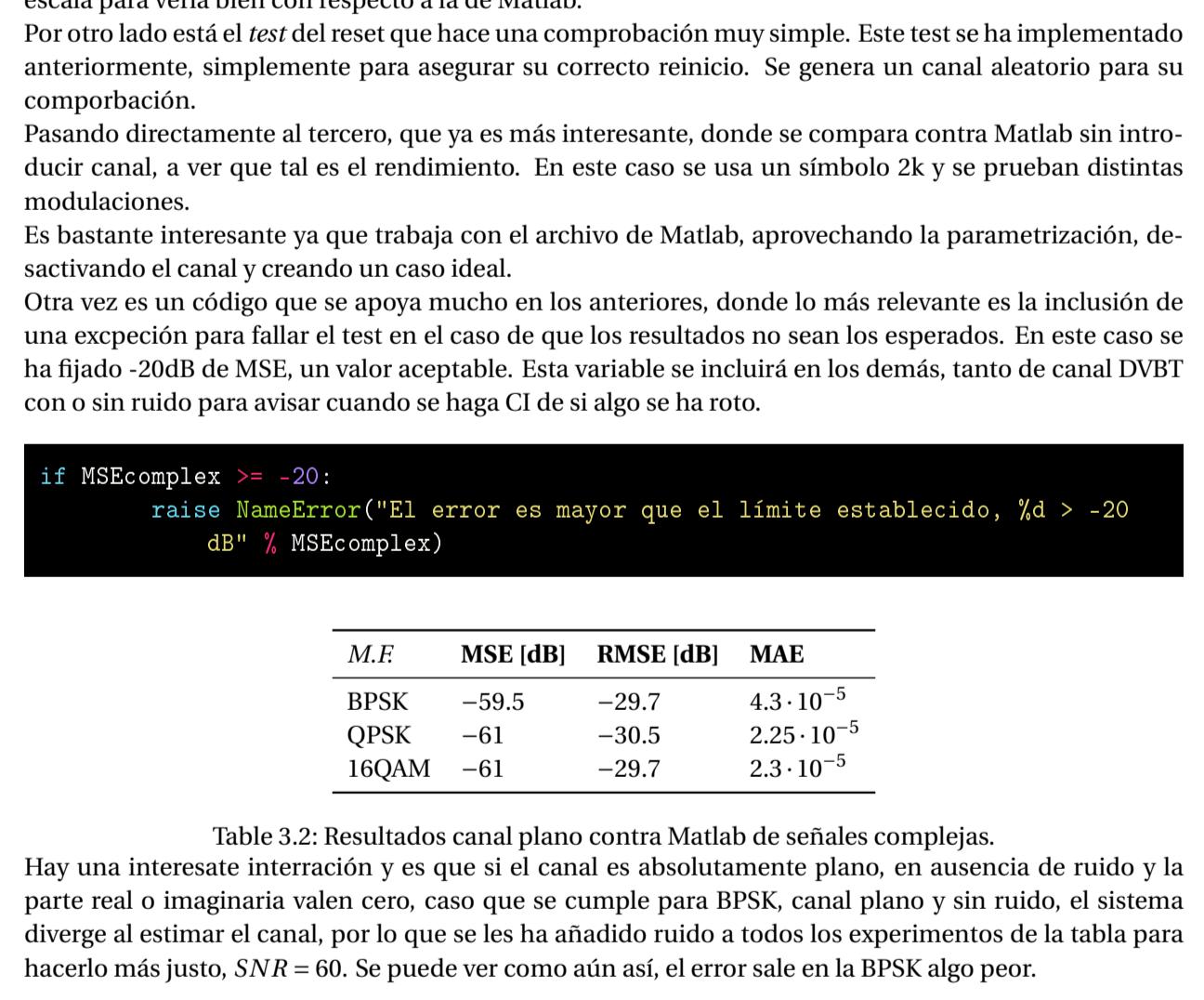


Figura 3.1: Pilotos Matlab vs VHDL.

### 3.4 FSM

El bloque FSM se llama así porque en las primeras etapas del diseño del funcionamiento del sistema iba a ser una máquina de estados, pero al final se consiguió simplificar tanto que se convirtió en un circuito sincrónico y conservó ese nombre como bloques. Su función es la de recibir los pilotos y el signo del generador PRBS para así estimar el valor de h. Para esta función deberá, cada vez que venga un piloto, cambiar el piloto servido como superior al interpolador por el inferior y añadir el nuevo que llega como inferior.

Básicamente es un circuito sincrónico que se encarga de cuando le llega la señal `wireEnable` cambiar los valores de los estímulos superior e inferior, cambiándoles el signo si es necesario según el valor del signo arrojado por el bloque PRBS.

```
-- changing sign depending on PRBS
if PRBS <= '1' then
    p_h_sup.re <- piloto_y_re;
    p_h_sup.im <- piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
else
    p_h_sup.re <- -piloto_y_re;
    p_h_sup.im <- -piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
end if;
```

Por dentro tiene una lógica, color de destacar el if decidido como cambia el signo de los pilotos según el valor de inicio. El resto del código es un circuito sincrónico común.

En verificación se ha estudiado si realmente obtiene los pilotos en el momento correcto y no en cualquier otro. Para esto se ha pasado el valor de entrada 69 en todos los instantes incorrectos, mientras que en los slots de los pilotos se pasa 10 + i \* 5 para la parte real y 10 + i \* 10 para la imaginaria para comprobar que correctamente aumentan los valores, es decir, cambian cada doce ciclos.

Si por alguna razón toma valores incorrectos, es decir, 69, salta una excepción.

```
# try avoids int(wire) or nt(wire) case
# sync'd with cocotb's
# resolve u, which would change u bit values to
# one at a time, which we would lose some sight of
# what is happening.
try:
    int(dut.inf_re.value)
    int(dut.inf_im.value)
    int(dut.sup_re.value)
    int(dut.sup_im.value)
except:
    pass
else:
    # if pilots values are correctly casted into int, check if they got
    # wrong type. If so, raise an error
    if dut.inf_re.value == 69 or dut.inf_im.value == 69 or dut.sup_re.value
        == 69 or dut.sup_im.value == 69:
        raise "El valor incorrecto ha sido propagado!"
```

Aquí ya van varios temas interesantes. Para empezar la estructura `try..except..else`, sobre todo para realizar posteriormente el "try trick" que se explicará luego. Si el valor de los pilotos es incorrecto, salta la excepción.

Eso es de los primeros testbenches donde todavía estaba aprendiendo a manejarlo, un tema importante es que si quieras pasar las variables de vhdl a python y en vhdl valen U o X habrá problemas graves. CoCoTb puede resolver automáticamente estos valores a 0 o 1, pero eso le restaría algo de fiabilidad a las verificaciones que ademas ignoraría.

### 3.5 Interpolador

Este bloque es el original que fue aportado para el trabajo de la asignatura pero no es el usado finalmente para la implementación, pero se proporciona también al haber realizado una verificación de este.

La verificación es bastante interesante ya que ha sido realizado un *test bench* que contrasta contra datos generados aleatoriamente usando una arquitectura *self-checking*. Estos datos posteriormente se grafican para ser comparados.

```
inf_im = int(np.round((np.random.rand())*2+9))
inf_im = int(np.round((np.random.rand())*2+9))
sup_im = int(np.round((np.random.rand())*2+9))
sup_im = int(np.round((np.random.rand())*2+9))
```

Así se generan fácilmente valores aleatorios como entradas. Para comparar si interpola bien en este caso se comparó con un predictor hecho en Python y no en matlab para mostrar la amplia gama de opciones a la hora de comprobar el funcionamiento del circuito, aunque aquí no se explora demasiadas hay muchas más.



Figura 3.2: Interpolation Python vs VHDL.

Se puede ver como las formas de onda aleatorias son exactamente iguales pero ligeramente escaladas una respecto a otra.

### 3.6 InterMig

Este es el interpolador que se usó finalmente para el proyecto. Se modificó para que cumpliera los requisitos fundamentales, uno de ellos que tardara menos en interpolar, ya que el proporcionado tardaba trece por cada doce bits, lo cual arruinaba la arquitectura *pipeLine*, mientras que el otro requiere que el que sea validu tuviera que estar validando 1 en cada ciclo para que la interpolación se realice. Esto último es debido a que si no se validara deje de emitir datos, todo el sistema debe pararse a la vez, si la interpolación sigue emitiendo hará que se detenga el resto del sistema.

En verificación se ha estudiado si realmente obtiene los pilotos en el momento correcto y no en cualquier otro. Para esto se ha pasado el valor de entrada 69 en todos los instantes incorrectos, mientras que en los slots de los pilotos se pasa 10 + i \* 5 para la parte real y 10 + i \* 10 para la imaginaria para comprobar que correctamente aumentan los valores, es decir, cambian cada doce ciclos.

Si por alguna razón toma valores incorrectos, es decir, 69, salta una excepción.

```
# try avoids int(wire) or nt(wire) case
# sync'd with cocotb's
# resolve u, which would change u bit values to
# one at a time, which we would lose some sight of
# what is happening.
try:
    int(dut.inf_re.value)
    int(dut.inf_im.value)
    int(dut.sup_re.value)
    int(dut.sup_im.value)
except:
    pass
else:
    # if pilots values are correctly casted into int, check if they got
    # wrong type. If so, raise an error
    if dut.inf_re.value == 69 or dut.inf_im.value == 69 or dut.sup_re.value
        == 69 or dut.sup_im.value == 69:
        raise "El valor incorrecto ha sido propagado!"
```

Aquí ya van varios temas interesantes. Para empezar la estructura `try..except..else`, sobre todo para realizar posteriormente el "try trick" que se explicará luego. Si el valor de los pilotos es incorrecto, salta la excepción.

Eso es de los primeros testbenches donde todavía estaba aprendiendo a manejarlo, un tema importante es que si quieras pasar las variables de vhdl a python y en vhdl valen U o X habrá problemas graves. CoCoTb puede resolver automáticamente estos valores a 0 o 1, pero eso le restaría algo de fiabilidad a las verificaciones que ademas ignoraría.

### 3.7 AllButInterpolator

Este código implementa todos los bloques menos el interpolador para comprobar que en conjunto todo va bien. Hubo un momento en el que los resultados fallaban y se comprobó el funcionamiento de los subbloques poco a poco para asegurar que estaban bien.

Las interacciones entre los bloques eran bastante sutiles, por lo que se tuvo que hacer un bucle constante para ver si las señales se sincronizaban bien. Para esto se realizó un *test bench* que verifica que las señales de salida estén sincronizadas entre sí.

Respecto al *testbench*, se han realizado dos, uno muy básico para ver los signos, y otro más interesante en el cual se usa la librería oct2py para comparar con el código Matlab.

```
sync'd with cocotb's
# sync'd with cocotb's
# resolve u, which would change u bit values to
# one at a time, which we would lose some sight of
# what is happening.
try:
    int(dut.inf_re.value)
    int(dut.inf_im.value)
    int(dut.sup_re.value)
    int(dut.sup_im.value)
except:
    pass
else:
    # if pilots values are correctly casted into int, check if they got
    # wrong type. If so, raise an error
    if dut.inf_re.value == 69 or dut.inf_im.value == 69 or dut.sup_re.value
        == 69 or dut.sup_im.value == 69:
        raise "El valor incorrecto ha sido propagado!"
```

Aquí ya van varios temas interesantes. Para empezar la estructura `try..except..else`, sobre todo para realizar posteriormente el "try trick" que se explicará luego. Si el valor de los pilotos es incorrecto, salta la excepción.

Eso es de los primeros testbenches donde todavía estaba aprendiendo a manejarlo, un tema importante es que si quieras pasar las variables de vhdl a python y en vhdl valen U o X habrá problemas graves. CoCoTb puede resolver automáticamente estos valores a 0 o 1, pero eso le restaría algo de fiabilidad a las verificaciones que ademas ignoraría.

### 3.8 TopLevel (Estimador)

Este unidad se encarga de conectar los anteriores bloques de la forma descrita en el primer apartado para lograr su correcto funcionamiento. Se define también como un proceso sincrónico el retraso de la señal de dato válido para ser utilizado en otros bloques. Aquí se ven las sentencias que sintetizarán los bloques de habilitación de FSM y PRBS.

```
-- adding "and y_valid_delayed" because if we stop and cuenta_value is 11 we get
-- many prbs
if PRBS <= '1' then
    p_h_sup.re <- piloto_y_re;
    p_h_sup.im <- piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
else
    p_h_sup.re <- -piloto_y_re;
    p_h_sup.im <- -piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
end if;
```

Por dentro tiene una lógica, color de destacar el if decidido como cambia el signo de los pilotos según el valor de inicio. El resto del código es un circuito sincrónico común.

En verificación se ha estudiado si realmente obtiene los pilotos en el momento correcto y no en cualquier otro. Para esto se ha pasado el valor de entrada 69 en todos los instantes incorrectos, mientras que en los slots de los pilotos se pasa 10 + i \* 5 para la parte real y 10 + i \* 10 para la imaginaria para comprobar que correctamente aumentan los valores, es decir, cambian cada doce ciclos.

Si por alguna razón toma valores incorrectos, es decir, 69, salta una excepción.

```
# try avoids int(wire) or nt(wire) case
# sync'd with cocotb's
# resolve u, which would change u bit values to
# one at a time, which we would lose some sight of
# what is happening.
try:
    int(dut.inf_re.value)
    int(dut.inf_im.value)
    int(dut.sup_re.value)
    int(dut.sup_im.value)
except:
    pass
else:
    # if pilots values are correctly casted into int, check if they got
    # wrong type. If so, raise an error
    if dut.inf_re.value == 69 or dut.inf_im.value == 69 or dut.sup_re.value
        == 69 or dut.sup_im.value == 69:
        raise "El valor incorrecto ha sido propagado!"
```

Aquí ya van varios temas interesantes. Para empezar la estructura `try..except..else`, sobre todo para realizar posteriormente el "try trick" que se explicará luego. Si el valor de los pilotos es incorrecto, salta la excepción.

Eso es de los primeros testbenches donde todavía estaba aprendiendo a manejarlo, un tema importante es que si quieras pasar las variables de vhdl a python y en vhdl valen U o X habrá problemas graves. CoCoTb puede resolver automáticamente estos valores a 0 o 1, pero eso le restaría algo de fiabilidad a las verificaciones que ademas ignoraría.

### 3.9 Ecuilizador

El bloque final es el ecualizador o canal, el cual se encarga de dividir las muestras y entre el sistema iba a ser una máquina de estados, pero al final se consiguió simplificar tanto que se convirtió en un circuito sincrónico y conservó ese nombre como bloques. Su función es la de recibir la señal de los pilotos y el signo del generador PRBS para así estimar el valor de h. Para esta función deberá, cada vez que venga un piloto, cambiar el piloto servido como superior al interpolador por el inferior y añadir el nuevo que llega como inferior.

Básicamente es un circuito sincrónico que se encarga de cuando le llega la señal `wireEnable` cambiar los valores de los estímulos superior e inferior, cambiándoles el signo si es necesario según el valor del signo arrojado por el bloque PRBS.

```
-- changing sign depending on PRBS
if PRBS <= '1' then
    p_h_sup.re <- piloto_y_re;
    p_h_sup.im <- piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
else
    p_h_sup.re <- -piloto_y_re;
    p_h_sup.im <- -piloto_y_im;
    p_h_inf.re <- h_sup.re;
    p_h_inf.im <- h_sup.im;
end if;
```

Por dentro tiene una lógica, color de destacar el if decidido como cambia el signo de los pilotos según el valor de inicio. El resto del código es un circuito sincrónico común.

En verificación se ha estudiado si realmente obtiene los pilotos en el momento correcto y no en cualquier otro. Para esto se ha pasado el valor de entrada 69 en todos los instantes incorrectos, mientras que en los slots de los pilotos se pasa 10 + i \* 5 para la parte real y 10 + i \* 10 para la imaginaria para comprobar que correctamente aumentan los valores, es decir, cambian cada doce ciclos.

Si por alguna razón toma valores incorrectos, es decir, 69, salta una excepción.

```
# try avoids int(wire) or nt(wire) case
# sync'd with cocotb's
# resolve u, which would change u bit values to
# one at a time, which we would lose some sight of
# what is happening.
try:
    int(dut.inf_re.value)
    int(dut.inf_im.value)
    int(dut.sup_re.value)
    int(dut.sup_im.value)
except:
    pass
else:
    # if pilots values are correctly casted into int, check if they got
    # wrong type. If so, raise an error
    if dut.inf_re.value == 69 or dut.inf_im.value == 69 or dut.sup_re.value
        == 69 or dut.sup_im.value == 69:
        raise "El valor incorrecto ha sido propagado!"
```

Aquí ya van varios

## 4 RESUMEN HITOS REALIZADOS

Recogiendo todo lo mencionado anteriormente, se han realizado las siguientes tareas:

- Diseño de estimador de canal OFDM desde cero con arquitectura *pipeline* y retraso de solo tres ciclos de reloj.
- Depuración de códigos mediante la herramienta GtkWave y gráficas con pyplot.
- Verificación de todos los bloques usados y algunos extra que finalmente no se emplearon.
- Uso de CoCoTb para la mayor parte de la verificación VHDL.
- Aprovechamiento de la potencia de los *Test Benches* en Python para hacer uso de herramientas como *Matplotlib* u *Oct2Py*.
- Uso de Integración Continua (CI) mediante GitLab soportado por Docker con extracción de información variada.
- GitLab cuidado y memorias en formato README.md para leerse desde el propio GitLab.

### 4.1 Tests realizados

- Test contador
  - Test básico
  - Test contra Matlab
- Test FSM con alerta
- Test Interpolador con datos aleatorios contra Python
- Test Interpoladro Mig doble con reset
- Test AllButInterpolator
- TopLevel Estimador
  - Test pilotos incial
  - Test reset
  - Test interpolación símbolos 2k
  - Test interpolaciín símbolos 8k
- Test ecualizador
- Test Registro
- Sistema completo
  - Test canal plano entradas random
  - Test reset
  - Test matlab canal plano ruidoso 2k (varias modulaciones)
  - Test matlab canal DVBT 2k (varias modulaciones)
  - Test matlab canal DVBT 8k (varias modulaciones)
  - Test matlab canal DVBT 2/8k ruidoso (varias modulaciones)