



## **Diseño Digital Avanzado – Trabajo final**

*Ecualizador Fraccionalmente Espaciado en el dominio del  
tiempo con algoritmo LMS adaptativo*

---

### **ALUMNO**

Ing. FERREYRA, Ramiro Tomás

### **PROFESOR**

Dr. Ing. POLA, Ariel Luis

### **AUXILIAR**

Ing. CAYUELA, Pablo Oscar

Córdoba, Argentina

Julio de 2025

## Resumen

Se parte de un sistema de comunicaciones digitales, con componentes y procesos que permiten transmitir y recibir información codificada. Este sistema cuenta con una fuente de información, un transmisor, un canal de comunicación, un receptor y un destino.

El canal de comunicación es el medio a través del cual se transmite la señal, e introduce distorsiones que pueden llevar a errores en la detección de símbolos por parte del receptor. Es en este punto que es necesario el uso de ecualizadores, que permiten corregir deficiencias en la respuesta del sistema.

En este caso, el objetivo es estudiar, simular e implementar un *Ecualizador Fraccionalmente Espaciado*, en conjunto con la aplicación del algoritmo *Least Mean Square*, capaz de adaptar los coeficientes del ecualizador de forma que se aproxime a la respuesta esperada.

El desarrollo del proyecto cuenta con una etapa inicial de estudio del sistema y profundización teórica de los conceptos. Con esta información se realiza un primer diagrama de alto nivel, acompañado de pequeñas simulaciones que corroboren el funcionamiento general en punto flotante. Luego se va complejizando hasta acercarse a una representación más realista del sistema. El último paso de la simulación de alto nivel es pasar esta representación a punto fijo, de manera que nos sirva de guía de cómo debería comportarse exactamente el sistema al implementarlo en RTL, junto con medidas y gráficos que lo respalden. Lo que sigue es describirlo en hardware, hacer pruebas con el Testbench, corregir los errores y realizar verificaciones de funcionalidad hasta que se

comporte como se espera, tomando de guía las simulaciones a alto nivel. Con esto correcto, se corre la síntesis e implementación y posteriormente se evalúan los resultados de tiempo, área y potencia, debiendo hacer las modificaciones necesarias del sistema para que sea implementable. Finalmente, una vez que cumple con los requerimientos, se sube lo corrido a la FPGA y de ahí se extraen los resultados finales y se contrastan con las etapas anteriores.

# Índice

<b>Resumen</b>	<b>1</b>
<b>Índice</b>	<b>3</b>
<b>Introducción</b>	<b>4</b>
<b>Simulaciones en alto nivel</b>	<b>11</b>
Ecualizador convencional con algoritmo LMS y señal senoidal	11
FSE con LMS con modulación QPSK sin pulse shaping ni decisión	14
Simulación realista del FSE con algoritmo LMS en punto flotante	17
1) Generación de la señal de entrada	18
2) Ecualizador y adaptación de coeficientes	22
Slicer	22
Constante de aprendizaje	22
Cantidad de coeficientes	23
Coeficientes iniciales	23
3) Pruebas y resultados a la salida	24
A- Variación de taps iniciales el ecualizador	24
B- Variación de constante de aprendizaje	27
C- Variación de cantidad de coeficientes del ecualizador	27
Simulación realista del FSE con algoritmo LMS en punto fijo	29
1) Esquemático	29
2) Cuantización	30
3) Tasa de error de bits	33
<b>Implementación en hardware</b>	<b>35</b>
Descripción de bloques en Verilog y Análisis RTL	35
1) LMS con ecualizador convencional	36
2) LMS con FSE	37
3) Vector Matching: simulación vs. RTL	38
4) Adición de otros bloques para validar funcionamiento en placa	39
Síntesis e Implementación	42
1) Primeras implementaciones de prueba	44
2) Implementación de diseño completo	47
3) Corrección de diseño para requisitos de tiempo	48
Generación de bitstream e implementación en FPGA	52
<b>Bibliografía</b>	<b>55</b>

# Introducción

El sistema de comunicaciones presente está compuesto, en grandes rasgos, por un transmisor, un canal y un receptor, con una modulación *Quadrature Phase Shift Keying* (QPSK), que transmite los datos alterando la fase de una portadora y contiene dos canales, uno en fase (I) y otro en cuadratura (Q), es decir, se encuentran a  $90^\circ$  entre sí.

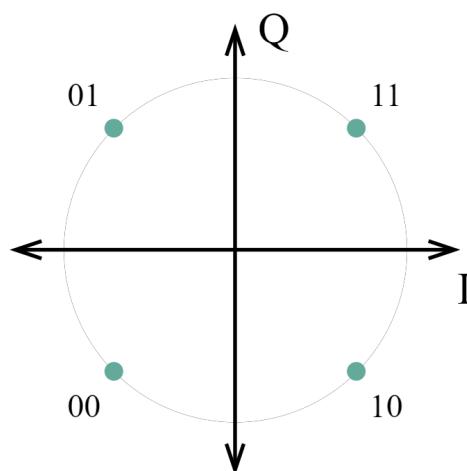


Figura 1 - Diagrama de constelación

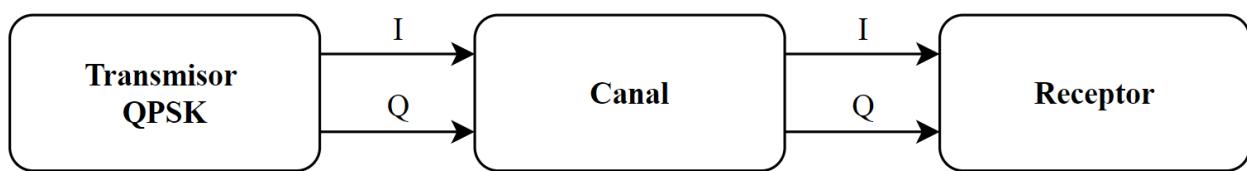


Figura 2 - Diagrama en bloques general del sistema de comunicaciones

El transmisor, mediante un bloque *Pseudo-Random Binary Sequence* (PRBS), genera una secuencia de bits que, luego de aplicar sobremuestreo y un *filtro de coseno realzado*

(RRC), es enviada a través del canal, donde se aplica una rotación de la constelación y se añade *interferencia intersímbolo* (ISI) y *ruido blanco gaussiano* (AWGN).

La interferencia entre símbolos es un problema común en sistemas de comunicación digital, donde la extensión de un pulso va más allá del intervalo de tiempo asignado y por esta razón interfiere con los pulsos vecinos. La causa de la ISI es el canal de banda limitada, que elimina y atenúa componentes de la señal.

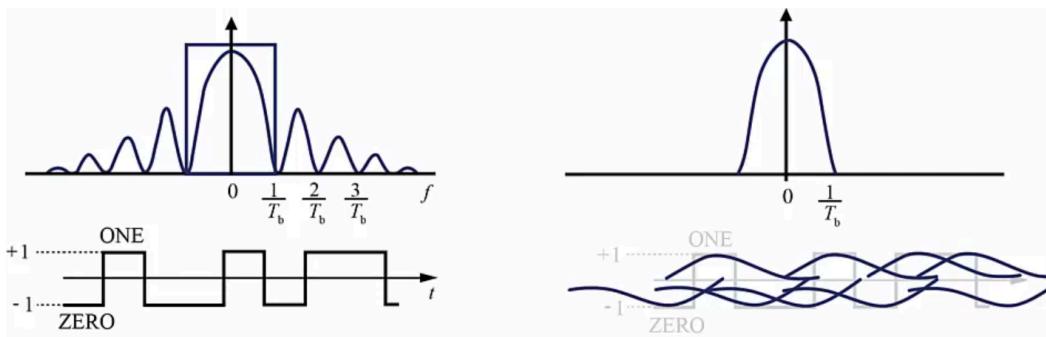


Figura 3 - Interferencia Intesimbólica

Por otro lado, el ruido blanco gaussiano es un tipo de ruido aleatorio con algunas características clave: sus muestras son estadísticamente independientes entre sí, es decir, el valor de una muestra no tiene influencia sobre el valor de las muestras adyacentes; sigue una distribución gaussiana o normal, o sea, la mayoría de las muestras se concentran cerca de un valor central y la probabilidad de obtener valores más alejados disminuye a medida que se aleja; tiene densidad espectral de potencia constante, que significa que la potencia es igual para todas las frecuencias; el valor o amplitud media es cero y no tiene sesgo hacia el positivo o negativo; su varianza es constante, es decir, la dispersión de valores se mantiene igual a lo largo del tiempo.

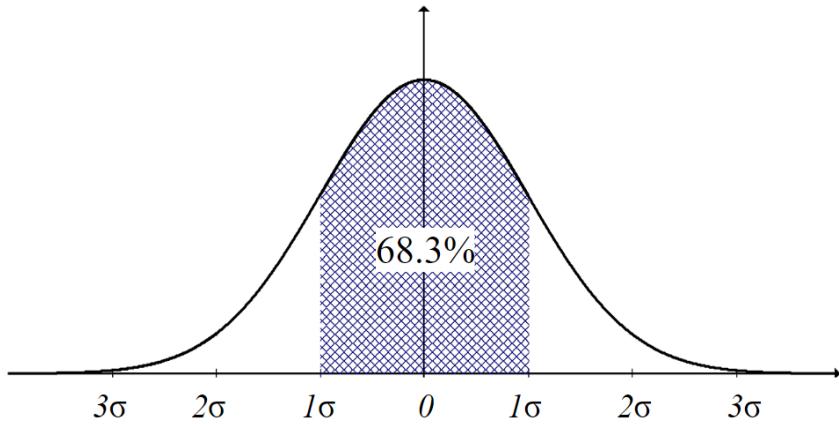


Figura 4 - Distribución gaussiana de ruido

En base a esto se puede concluir que el resultado tiene tres componentes, la señal útil, la interferencia entre símbolos y el ruido blanco gaussiano. Si la señal se muestrea en los instantes ideales  $t=nT$ , la muestra es:

$$r(nT) = a_n g(0) + \sum_{k=-\infty}^{\infty} a_k g((n - k)T) + n(nT)$$

- Señal útil:  $a_n g(0)$
- ISI:  $\sum_{k=-\infty}^{\infty} a_k g((n - k)T)$
- AWGN:  $n(nT)$

Se puede apreciar que la ISI es el segundo término y que la condición necesaria para evitarla es que el impulso recibido cumpla:

$$\text{si } n = 0: g(nT) = 1$$

$$\text{si } n \neq 0: g(nT) = 0$$

Esto garantiza que cada muestra  $r(nT)$  depende solo del símbolo actual  $a_n$  y no de los demás.

Como se dijo, la interferencia entre símbolos puede causar errores cuando se intenta recuperar los datos enviados. En ese caso es necesario diseñar un sistema correctivo que, en cascada con el canal, produzca una salida que corrija la distorsión causada por dicho canal, y que nos de una réplica de la señal transmitida deseada. En este contexto, estos sistemas correctivos son llamados **ecualizadores**, que en teoría de sistemas lineales se conocen como sistemas inversos, ya que son filtros que tienen una respuesta en frecuencia que es la recíproca de la respuesta del sistema que causó la distorsión.

Entonces, si el sistema distorsivo genera una salida  $y(n)$  que es la *convolución* entre la entrada  $x(n)$  y la respuesta al impulso  $h(n)$ , el sistema inverso toma  $y(n)$  y produce  $x(n)$  mediante *deconvolución*.

Si las características del sistema distorsivo son desconocidas, es necesario excitarlo con una señal conocida, observar la salida y, de alguna manera, intentar determinar estas características.

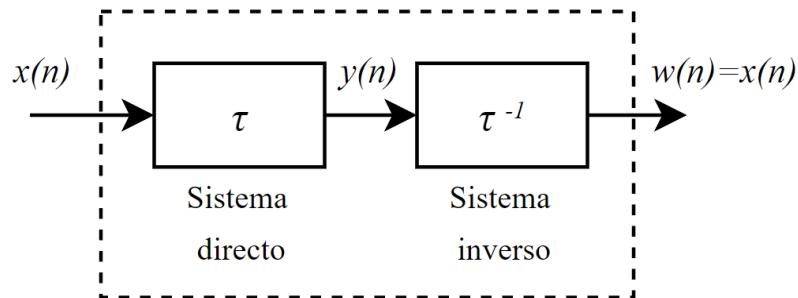


Figura 5 - Sistema  $\tau$  en cascada con su inverso  $\tau^{-1}$

Si el sistema  $\tau$  tiene una respuesta al impulso  $h(n)$  y  $\tau$  tiene una respuesta al impulso  $h_1(n)$ , su convolución es:

$$w(n) = h_1(n) * h(n) * x(n) = x(n)$$

$$h_1(n) * h(n) = \delta(n)$$

$$H(z)H_1(z) = 1$$

En muchos casos, la única manera de compensar las distintas distorsiones es disponer de un ecualizador que sea ajustable y que se vaya optimizando para minimizarlas. Estos son llamados *ecualizadores adaptativos*, y permiten ajustar sus parámetros a lo largo del tiempo, siendo muy usados en receptores de comunicación digitales para identificar el canal.

Este tipo de ecualizadores puede ser modelado tanto por *filtros de respuesta infinita* (IIR) como por *filtros de respuesta finita* (FIR), pero estos últimos son los más comúnmente utilizados. Esto se debe a que sólo contienen ceros ajustables (no polos como el IIR) y no presenta algunos problemas de estabilidad que el IIR sí. El ajuste se realiza mediante los coeficientes del filtro, y existen distintas técnicas o algoritmos para hacerlo. Una de ellas es el algoritmo *Least Mean Square* (LMS), que busca los coeficientes que permiten el valor esperado mínimo del cuadrado de la señal de error, que es la diferencia entre la señal deseada y la señal producida a la salida del filtro. En este caso, la señal deseada es la decisión tomada por el *Slicer* (1 o 0) en base a la salida. Este ajuste está dado por la siguiente expresión, donde  $w_{n+1}$  son los coeficientes de la interacción futura,  $w_n$  los de la

actual,  $\mu$  es el paso de aprendizaje o tamaño del ajuste,  $e_n$  es el error calculado y  $x_n$  son las muestras de la señal de entrada:

$$w_{n+1} = w_n + \mu \cdot e_n \cdot x_n$$

En este proyecto se busca justamente aplicar el algoritmo LMS para optimizar, en cada iteración, los coeficientes del filtro adaptativo. Pero no cualquier filtro adaptativo, sino que se ha optado por utilizar un *Ecualizador Fraccionalmente Espaciado* (FSE). El FSE se diferencia de los ecualizadores convencionales ya que muestrea la señal a una tasa mayor que la tasa de símbolos, lo que le permite capturar mejor la forma de la señal distorsionada y aplicar una compensación más precisa para reducir la ISI. Lo más normal es encontrar ecualizadores fraccionales del doble de tasa de símbolos, es decir, si un ecualizador convencional es T-espaciado, el FSE tendrá un espaciado de muestras de  $T/2$ , y tiene una longitud temporal de la mitad del primero, con el mismo número de coeficientes, haciéndolo muy superior cuando presenta distorsiones severas en los bordes de la banda.

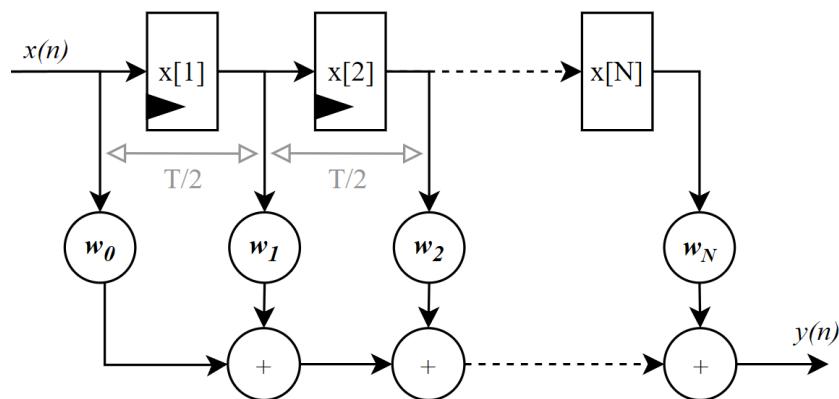


Figura 6 - Ecualizador Fraccionalmente Espaciado

Entonces, a grandes rasgos, el sistema tratado en este proyecto podría resumirse de la siguiente manera:

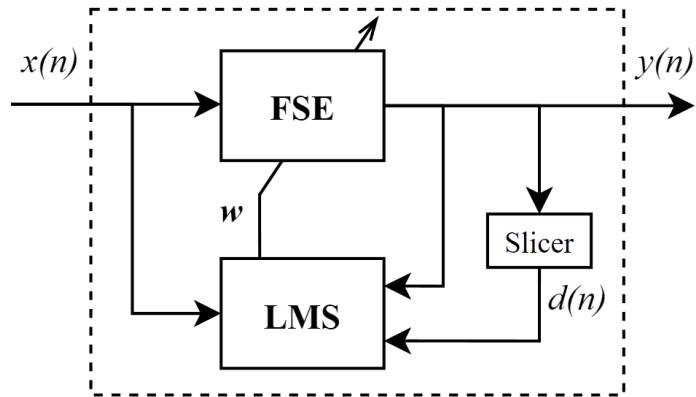


Figura 7 - Diagrama en bloques del sistema a implementar

## Simulaciones en alto nivel

Luego de ahondar en la parte teórica y definir los límites del sistema a diseñar, ya es posible arrancar con las primeras pruebas y simulaciones. Se decide utilizar *Python* como herramienta, debido a su popularidad y versatilidad, con una sintaxis clara y una gran comunidad activa que brinda un abanico de librerías necesarias para las distintas etapas del proyecto. La plataforma para escribir código es Visual Studio Code, ya que proporciona una variedad de opciones de personalización, interfaz clara y extensiones que ayudan a facilitar la escritura, tanto de la simulación como más tarde en la descripción de hardware.

Se comienza con la descripción de los bloques y se realizan pruebas simples que permitan darnos un entendimiento mayor del algoritmo, con la posibilidad de variar algunos parámetros básicos y ver el comportamiento en distintas condiciones, pero careciendo aún de una representación realista del procedimiento.

## Ecuilizador convencional con algoritmo LMS y señal senoidal

Partimos de un ecualizador convencional (no fraccional), con  $n$  taps y un LMS con constante de aprendizaje  $\mu$ . La primera señal a evaluar es una simple senoidal, que será nuestra señal deseada  $d(n)$ , a la que se le añade ruido y la utilizamos para generar nuestra entrada  $x(n)$ . A continuación se muestran algunos fragmentos importantes del código:

```
# Parámetros del algoritmo
mu = 0.001 # Tasa de aprendizaje
n_taps = 9 # Número de taps
```

```

n_iterations = 40000 # Número de iteraciones
np.random.seed(0) # Para reproducibilidad

# Algoritmo LMS
for n in range(n_taps, n_iterations):
    x_n = señal_ruidosa[n:n-n_taps:-1] # Ventana de muestras de entrada
    y[n] = np.dot(w, x_n) # Salida del filtro
    e[n] = senoidal_pura[n] - y[n] # Cálculo del error
    w = w + 2 * mu * e[n] * x_n # Actualización de coeficientes

```

Variando  $n$  y  $\mu$  en el código, tenemos los siguientes resultados:

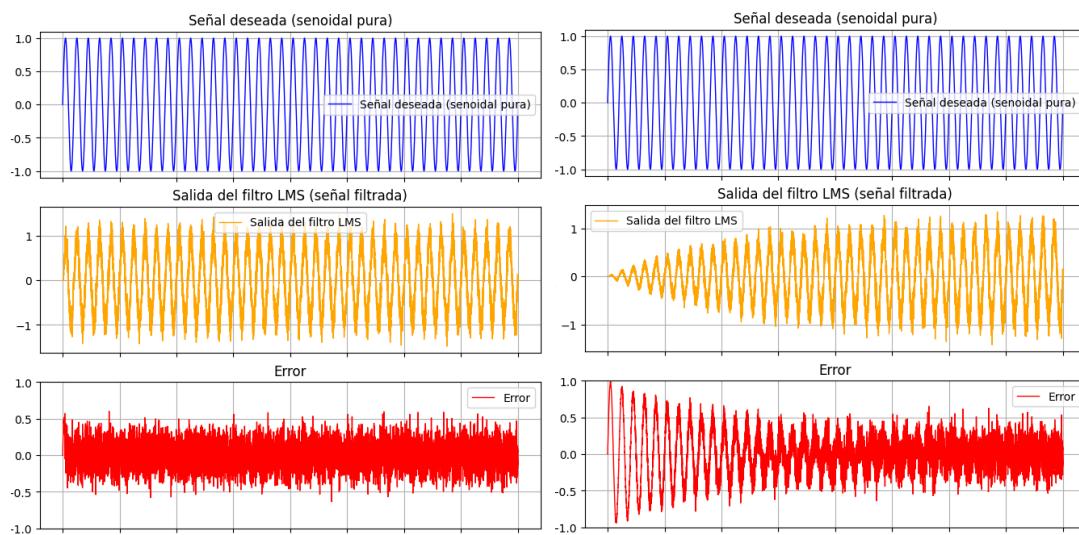


Figura 8 - Simulación LMS para  $n=9$ , con  $\mu=0.001$  (izq.) y  $\mu=0.00001$  (der.)

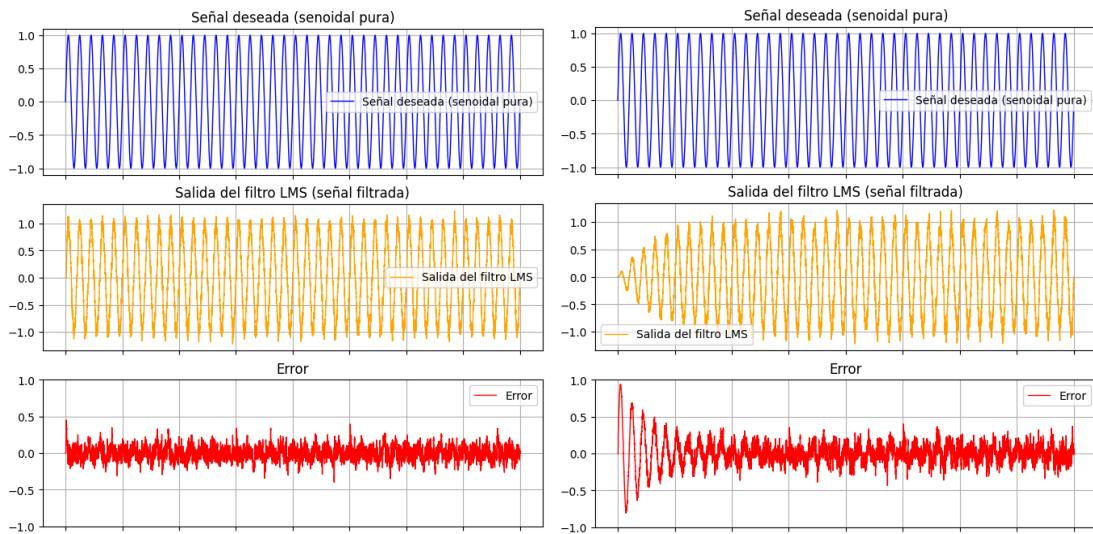


Figura 9 - Simulación LMS para  $n=32$ , con  $\mu=0.001$  (izq.) y  $\mu=0.00001$  (der.)

Con esta información se puede apreciar que al bajar la tasa de aprendizaje de 0.001 a 0.00001, el algoritmo converge más lentamente debido a la disminución del paso, lo que nos servirá en un futuro para asegurar la estabilidad del sistema y evitar oscilaciones. Debemos buscar un  $\mu$  que nos de un equilibrio entre precisión y rapidez, según los requisitos.

Por otra parte, vemos que al aumentar la cantidad de taps o coeficientes de 9 a 32, se logra un filtrado más preciso y la señal de salida es más “ limpia”, siendo más similar a la deseada y, por lo tanto, disminuyendo el error.

## FSE con LMS con modulación QPSK sin *pulse shaping* ni decisión

Ahora aumentamos la complejidad tomando una modulación QPSK, con una señal en fase y otra en cuadratura (ver Figura 1). Se genera una secuencia aleatoria donde cada símbolo representa dos bits (I y Q).

Como se vió antes, el FSE funciona tomando más de una muestra por símbolo, es decir, con un *oversampling*. Para poder aplicar este sobremuestreo, es necesario que la señal de entrada esté interpolada para que su representación tenga más muestras para tomar. Esta interpolación es generada tomando los valores de los símbolos 1 y 0 y llevándolos a niveles de +1 y -1, y posteriormente añadiendo ceros entre símbolos, dependiendo el grado de sobremuestreo. Después de esto iría un filtro de coseno realizado que actúe como conformador de pulsos, o sea, que moldee la forma de los pulsos con el objetivo de optimizar el ancho de banda de la señal al ser enviada a través del canal. En este apartado se toma un caso poco realista, ya que no se aplica el RCC, sino que entran los pulsos puros, para apreciar el funcionamiento básico del sistema.

Suponemos un modelo de canal con  $\text{ISI} = [0.9; 0.5; 0.2]$ , donde el pulso actual es afectado por los pulsos pasado y futuro, con el añadido de AWGN. Es posible variar la constante de aprendizaje, la cantidad de taps del filtro, el *oversampling* (os) y la *relación señal ruido* (SNR) para luego analizar el comportamiento.

Por último, se utilizará como señal deseada la señal original transmitida antes del canal, lo cual sería imposible en la realidad debido a que sí está el canal de por medio y, si

hubiera manera de traerla limpia hasta el receptor, no tendría sentido usar un ecualizador, pero en este caso nos sirve para partir de un sistema ideal.

```
# Parámetros de la simulación
mu = 0.01                      # Tasa de aprendizaje LMS
N = 9                            # Número de coeficientes del filtro FSE
oversampling = 2                  # Factor de sobremuestreo (T/2)
SNR_dB = 40                       # Relación señal a ruido (SNR) en dB
```

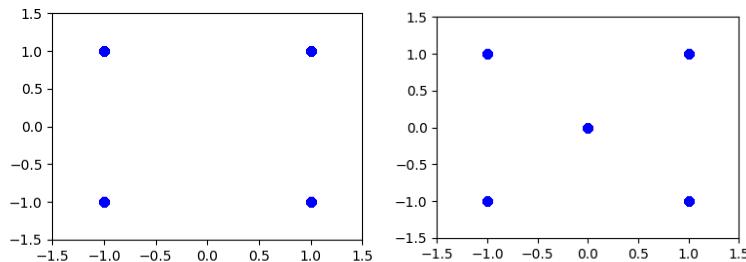


Figura 10 - QPSK original sin oversampling (izquierda) y con oversampling (derecha)

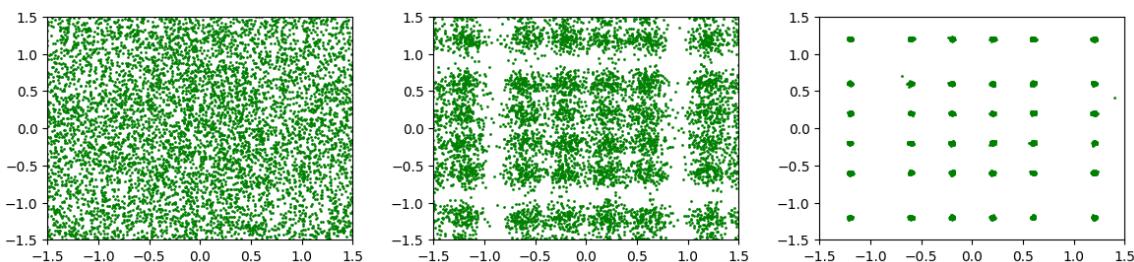


Figura 11 - QPSK con ISI y distintas SNR: 10dB (izq.), 20dB(cen.) y 40dB(der.)

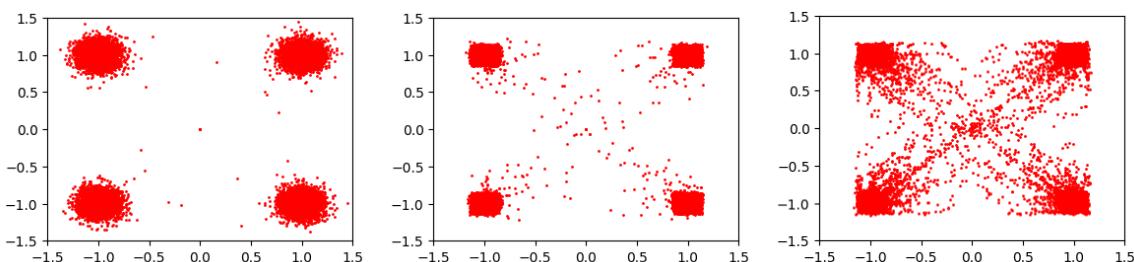


Figura 12 - Salida del FSE para  $n=4$ , con  $\mu=0.1$  (izq.),  $\mu=0.01$  (cen.) y  $\mu=0.001$  (der.)

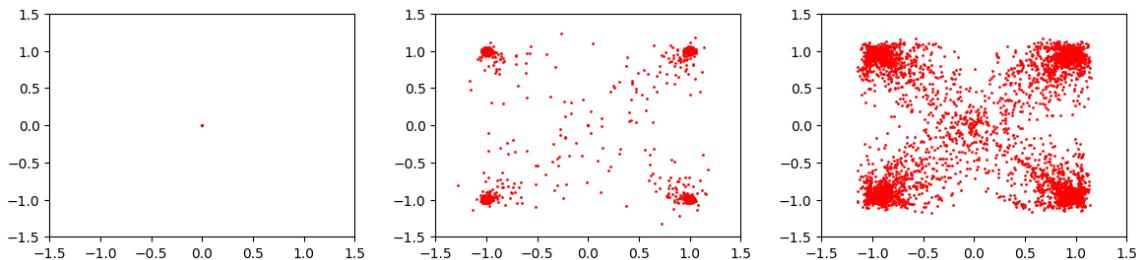


Figura 13 - Salida del FSE para  $n=17$ , con  $\mu=0.1$  (izq.),  $\mu=0.01$  (cen.) y  $\mu=0.001$  (der.)

Entonces, se ve que el oversampling hace que aparezca un punto en el centro debido al agregado de ceros ( $I=0$  y  $Q=0$ ) de la interpolación, pero es solo a modo de visualización, ya que este valor no se transmite.

En cuanto a la constante de aprendizaje, volvemos a ver las diferencias a medida que la aumentamos o disminuimos, incluso se puede apreciar que para algunos valores convergerá dependiendo de la cantidad de coeficientes. Por ejemplo, con  $\mu=0.1$  y  $n=4$ , el algoritmo, aunque no muy precisamente, converge, pero para  $n=17$  no lo hace. Ahora, para el resto de valores de  $\mu$  vemos que el filtrado es más preciso con  $n=17$  que con  $n=4$ . Estas simulaciones son útiles para tener en cuenta que los cambios no son tan lineales y obvios, y hay que ajustar los distintos parámetros a conciencia para aproximarse al punto más óptimo, donde las condiciones permitan la respuesta buscada.

## Simulación realista del FSE con algoritmo LMS en punto flotante

En este punto ya comenzamos a hacer simulaciones del sistema más cercanas a la realidad, es decir, más cercanas a lo que se implementará mediante *Register Transfer Level* (RTL) en el chip, ya sea ASIC o, en este caso, FPGA. Como en nuestro caso no contamos con una FPGA muy potente (*Arty A7-100T: Artix-7*), tenemos restricciones que nos llevarán a algunas limitaciones de área. Es por esto que debemos achicar el diseño lo máximo posible.

Una de las maneras de disminuir el tamaño del diseño es aprovechar la propia modulación. Como en QPSK tenemos un canal I y un canal Q, y a ambos se les realizan las mismas operaciones en paralelo, es posible desarrollar solo uno de los canales, ya que para el otro será exactamente lo mismo, solo que con una secuencia distinta. Entonces, a partir de ahora se tomará una constelación tipo BPSK, donde cada símbolo tiene solo un bit, en vez de dos, y nuestra entrada será una única señal con valores 1 (+1) o 0 (-1).

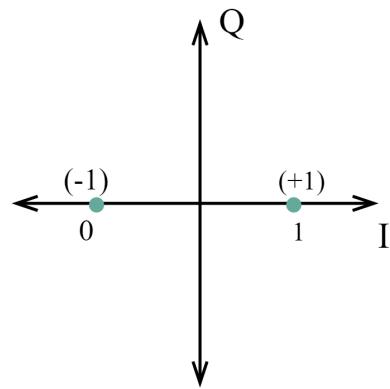


Figura 14 - BPSK

### 1) Generación de la señal de entrada

Una manera de tener un mayor control sobre los distintos parámetros de las señales en el sistema es generar o simular desde cero la señal recibida, con su secuencia, símbolos, filtros, distorsiones y demás, tal como si fuese generada por el transmisor y pasada por el canal.

En primer lugar se puede simular la aleatoriedad de los símbolos mediante una secuencia binaria pseudo-aleatoria (PRBS). En este caso, este bloque es de nueve bits por lo que es una PRBS9, y consta de una “semilla” inicial distinta de cero (no deben estar los nueve bits en cero), a partir de la cual se generan los distintos valores. Nuestro bit transmitido será el bit [0] de cada iteración, y el bloque se irá desplazando lógicamente hacia la derecha, pero en vez de llenar con cero los valores de la izquierda que van quedando vacíos, se llenan con el resultado de aplicar una *XOR* a los bits [0] y [5].

Entonces, tendremos  $2^9 - 1 = 511$  valores, no 512 debido a que la combinación 000000000 no puede darse.

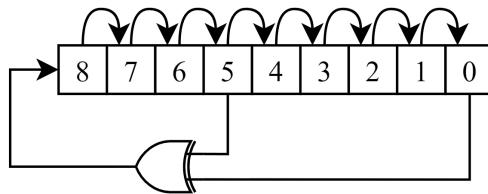


Figura 15 - PRBS9

Una vez generados los 1's y 0's, ya podemos pasarlos a +1 y -1 y aplicar *oversampling*, que para el desarrollo del proyecto se decidió que será de dos, o sea, el muestreo será a

tasa  $T/2$  siendo  $T$  la tasa de símbolos. Esto nos lleva a tener un cero interpolado por cada muestra.

A esto se le aplica un filtro de coseno realzado capaz de moldear los pulsos y acotar el ancho de banda. Este filtro RCC tiene un *factor de caída* o *factor de roll-off* ( $\beta$ ), que nos indica el exceso de ancho de banda y la forma del filtro y su caída. Este factor se encuentra entre 0 y 1. Un factor de roll-off más alto significa una transición más rápida y un mayor ancho de banda ocupado, mientras que un factor más bajo resulta en una transición más gradual y un ancho de banda más estrecho. Se elige  $\beta=0.8$ .

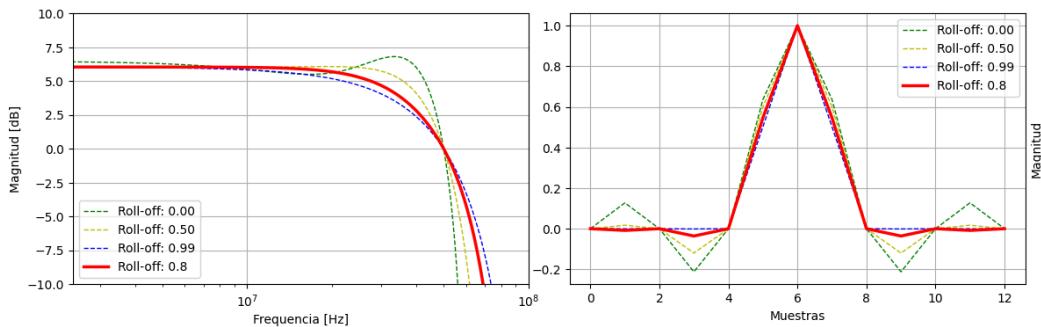


Figura 16 - Respuesta en frecuencia (izq.) y tiempo (der.) para distintos factores de caída

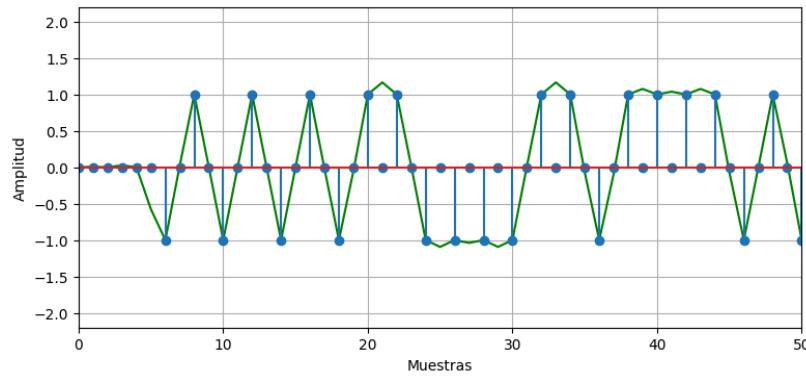


Figura 17 - Señal transmitida: pulsos con RCC

La señal de la Figura 17 es la que se envía a través del canal. A ella se le añadirá ISI y AWGN, de la misma manera que se hizo en la simulación anterior. En cuanto al AWGN, se tomará una buena SNR de 40 dB para la mayoría de las simulaciones, ya que el objetivo principal es mitigar la ISI, y se hará variar para lograr algunas medidas, como la BER.

Al no tener requisitos iniciales de ISI a mitigar, esta misma se varía a lo largo del desarrollo para apreciar el comportamiento del sistema y validarla para distintos niveles de distorsión.

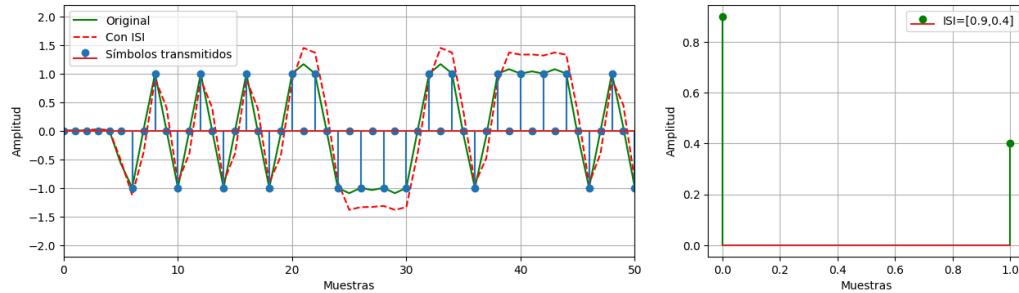


Figura 18 - Señal con y sin ISI (izq.) con valores de ISI leve (der.)

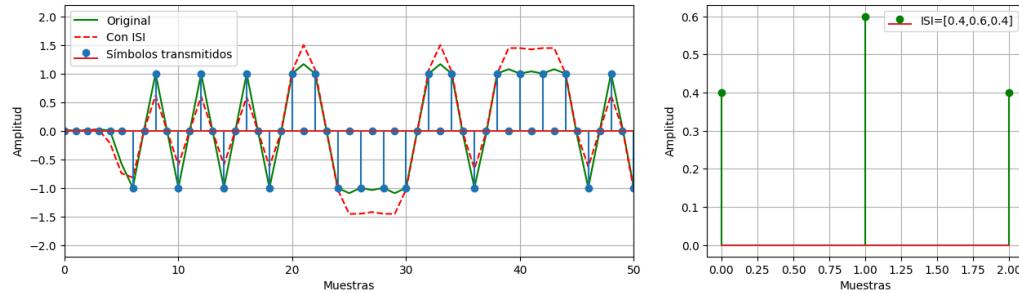


Figura 19 - Señal con y sin ISI (izq.) con valores de ISI moderada (der.)

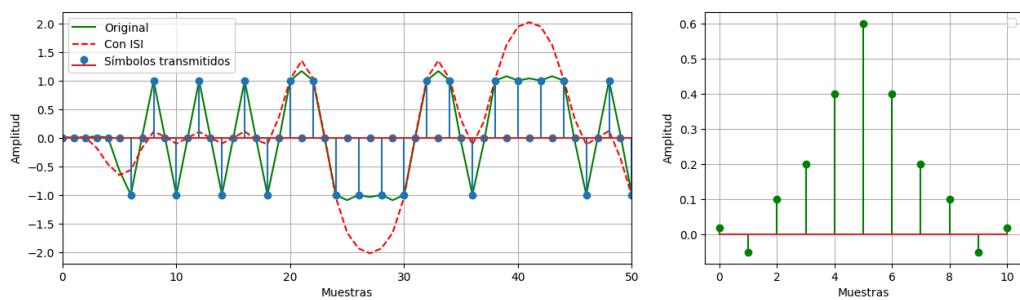


Figura 20 - Señal con y sin ISI (izq.) con valores de ISI severa (der.)

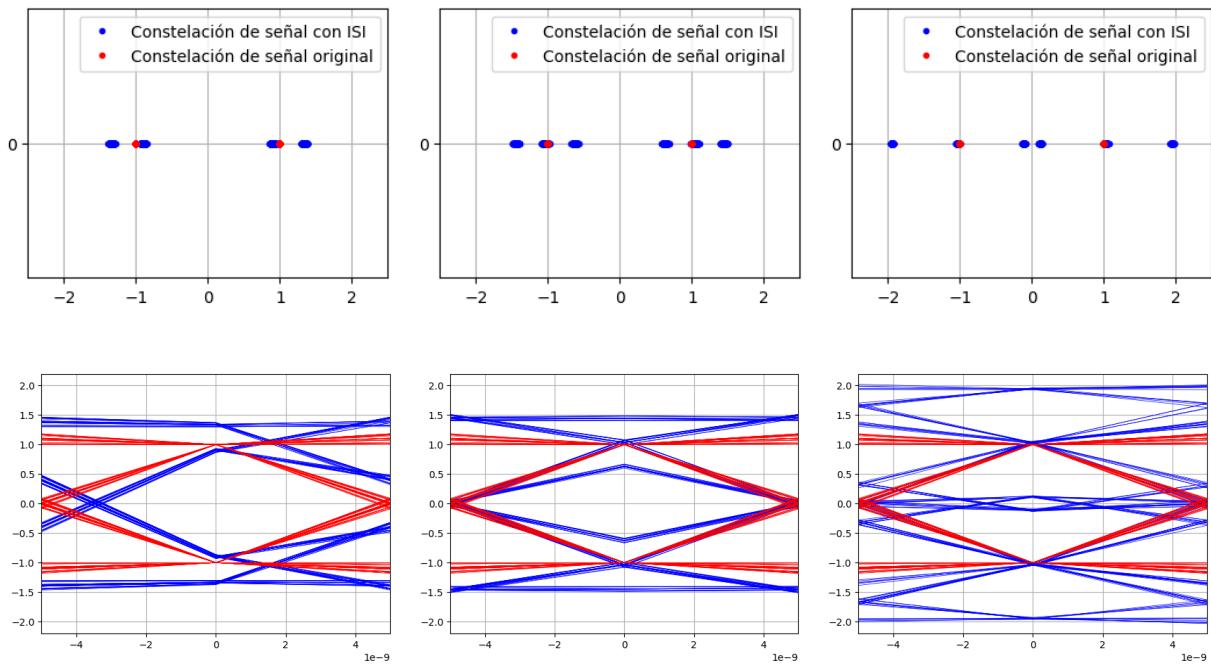


Figura 21 - Constelación (arriba) y diagrama de ojo (abajo) de la señal con ISI leve (izq.), moderada (cen.) y severa (der.)

Vemos que el diagrama de ojo corta en el punto de muestreo (en el centro) en los mismos puntos que indica la constelación, corroborando que ambas herramientas son útiles para representar las señales en el dominio del tiempo.

## 2) Ecualizador y adaptación de coeficientes

Una vez obtenida la señal a la salida del canal, es momento de ecualizarla, de modo que se puedan mitigar las distorsiones. Se selecciona la ISI moderada de la Figura 19, que será nuestra entrada del FSE guiándonos con el diagrama de la Figura 5 y buscando que la salida se vaya aproximando a la señal original de la Figura 17. Hay algunos puntos importantes del sistema a tener en cuenta en una simulación más real, como lo son el *Slicer*, la constante de aprendizaje, la cantidad de coeficientes y los coeficientes iniciales.

### Slicer

Como en este caso utilizamos una simulación realista del sistema, la señal deseada ya no puede ser la original transmitida, debido a que no hay forma de traerla limpia hasta el receptor. En cambio, se usa un Slicer que se encarga de tomar una decisión  $d(n)$  sobre la salida del ecualizador, es decir, nos da 1 o 0 dependiendo de si la señal está por encima o por debajo de un umbral respectivamente, que en este caso sería el punto medio entre +1 y -1, o sea, 0. Entonces, si la señal es positiva tendremos un 1 (+1) y si es negativa, un 0 (-1), y la señal de error utilizada para calcular el LMS queda:  $e(n)=d(n)-y(n)$ .

### Constante de aprendizaje

Dentro del algoritmo LMS, otro parámetro a tener en cuenta es la constante de aprendizaje  $\mu$ , ya vista en simulaciones anteriores. Sabemos que un  $\mu$  muy pequeño nos dará más precisión pero su convergencia será más lenta que un  $\mu$  mayor, aunque esta última puede tener problemas de estabilidad. En el desarrollo de este proyecto se elegirá

la mejor constante de paso mediante prueba y error, ya que el  $\mu$  ideal cambia para cada tipo de señal y distorsión del canal. Una técnica muy útil es variar el  $\mu$  a lo largo de la convergencia para mejorar el funcionamiento del algoritmo. Por ejemplo, es posible arrancar con un  $\mu$  alto para aproximarse rápidamente a los valores buscados y, una vez que esté cerca, disminuir  $\mu$  para hacer una adaptación más fina.

### Cantidad de coeficientes

Sabemos que, para obtener la respuesta total del sistema, es necesario que el ecualizador compense el canal. Esta respuesta no es más que el conjunto de coeficientes del filtro, que van evolucionando hasta alcanzar los valores que hagan que la salida se aproxime a la señal buscada. La cantidad de taps del FSE debe ser capaz de cubrir toda la ventana de la respuesta al impulso del canal. En pocas palabras, necesitamos los suficientes coeficientes para poder “dibujar” la forma del filtro. Ahora, tampoco debemos aumentar excesivamente esta cantidad ya que puede provocar inestabilidad o divergir si el paso de adaptación no se ajusta correctamente, además de la complejidad computacional que adiciona.

### Coeficientes iniciales

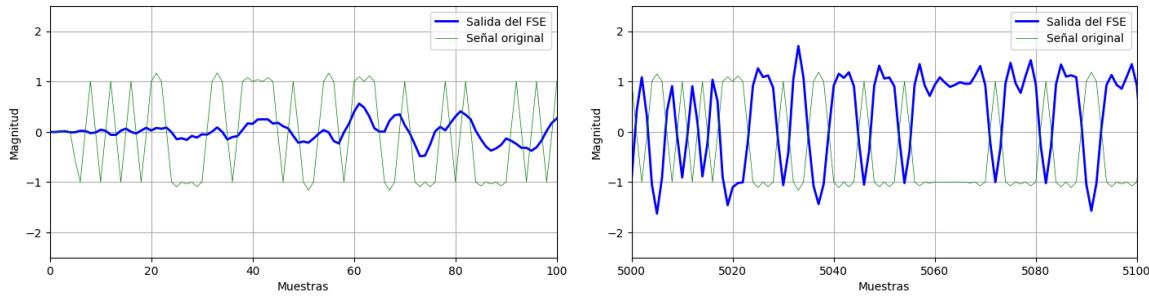
Hemos visto que la respuesta esperada del ecualizador  $h_I(n)$  es aquella que al ser convolucionada con el canal  $h(n)$  nos dé idealmente un impulso unitario o delta de dirac  $\delta(n)=h(n)*h_I(n)$ . Para ello, normalmente la respuesta al impulso del ecualizador debe

tener su energía concentrada en el punto del símbolo (impulso) a transmitir, o sea, en el centro. Es posible facilitar esta evolución y converger más rápido configurando los coeficientes iniciales (respuesta inicial del FSE) en valores que nos convengan, en vez de iniciar en cero. En este caso, esto se logra colocando el tap central en 1 y el resto en 0, un impulso unitario.

Una vez desarrollados estos conceptos y técnicas importantes, podemos llevar a cabo la simulación del sistema tratado variando distintos parámetros, con el modelo de ISI moderada [0.4, 0.6, 0.4] de la Figura 19, donde se establece, luego de varias pruebas, un  $\mu=0.005$  inicial, el tap central en 1 y 17 taps de ecualización. Se adjuntan a continuación algunas medidas tomadas en el proceso, partiendo de los parámetros recién mencionados y variando de a uno.

### 3) Pruebas y resultados a la salida

#### A- Variación de taps iniciales el ecualizador



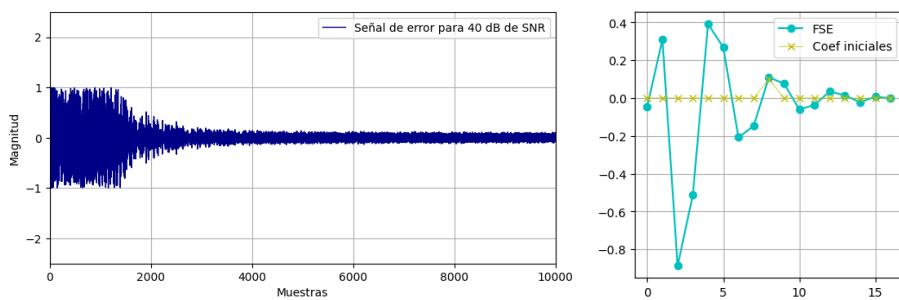


Figura 22 - Todos los taps iniciales en 0: salida del FSE al inicio de la adaptación (arriba izq.), una vez convergido (arriba der.), señal de error del LMS (abajo izq.) y respuesta al impulso del FSE (abajo der.)

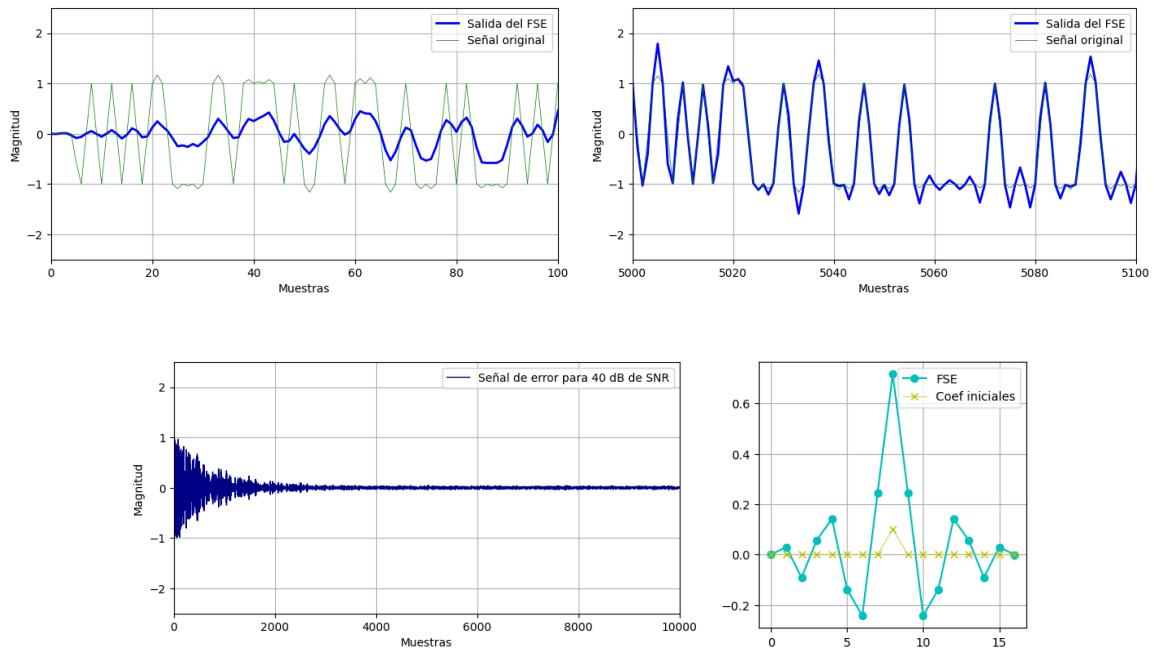


Figura 23 - Tap central inicial en 0.1 y el resto en 0: salida del FSE al inicio de la adaptación (arriba izq.), una vez convergido (arriba der.), señal de error del LMS (abajo izq.) y respuesta al impulso del FSE (abajo der.)

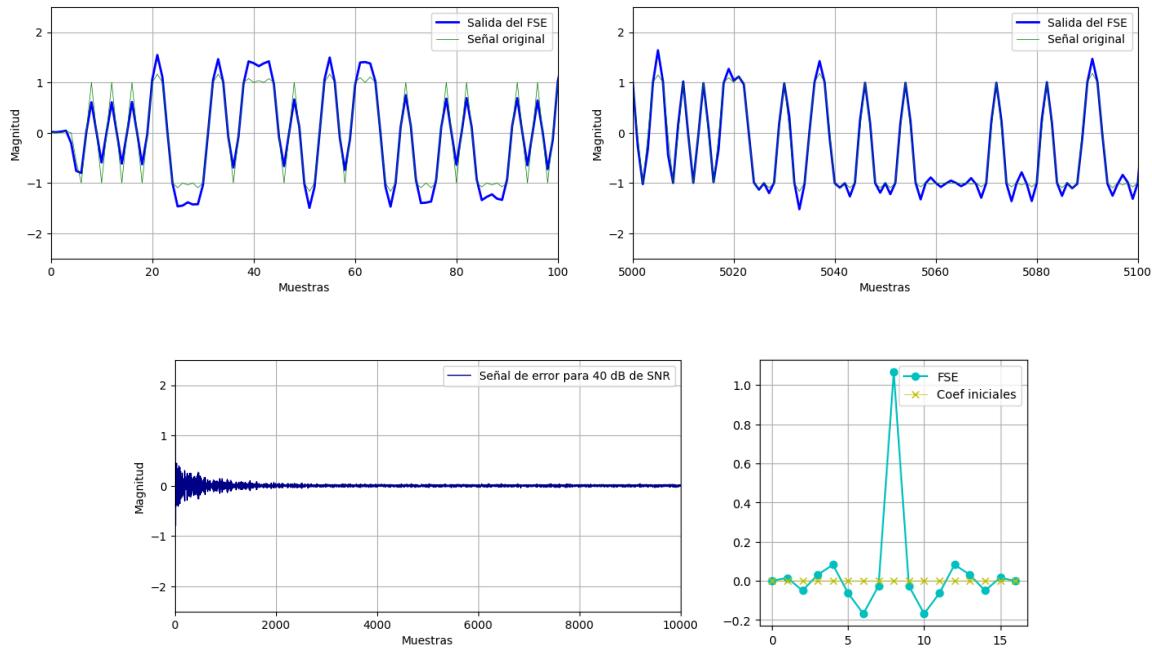


Figura 24 - Tap central inicial en 1 y el resto en 0: salida del FSE al inicio de la adaptación (arriba izq.), una vez convergido (arriba der.), señal de error del LMS (abajo izq.) y respuesta al impulso del FSE (abajo der.)

Podemos apreciar que colocando todos los taps iniciales en cero, con esta configuración, el sistema no converge correctamente y el error deja de decrecer en un punto. Se ve que la señal de salida es similar a la deseada pero con  $180^\circ$  de desfase (signo cambiado). Poniendo el tap central inicial en 0.1 vemos como mejora la respuesta y el error se hace mínimo. Aún así su energía no queda del todo centrada y se ven lóbulos grandes a los costados. Para el tap central en 1, ya la respuesta es mucho mejor y converge más rápidamente (el error llega más rápido al mínimo).

## B- Variación de constante de aprendizaje

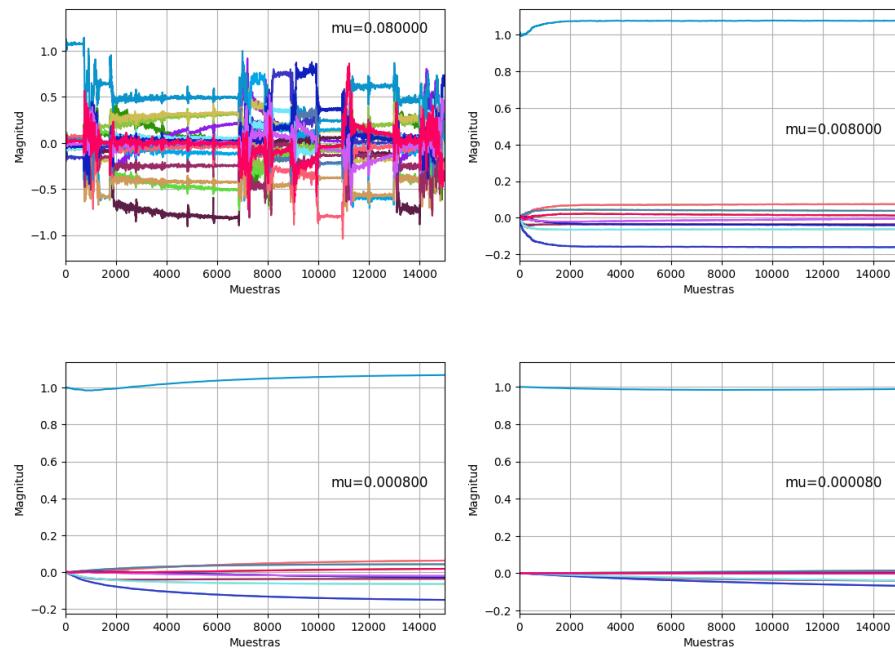


Figura 25 - Evolución de los coeficientes para distintos pasos de adaptación ( $\mu$ )

Mediante estas gráficas, es posible corroborar lo visto en simulaciones anteriores. Mientras mayor es  $\mu$ , más rápido llega a la convergencia (coeficientes estables), pero si el paso es muy grande puede dejar el sistema inestable, como en el primer gráfico.

## C- Variación de cantidad de coeficientes del ecualizador

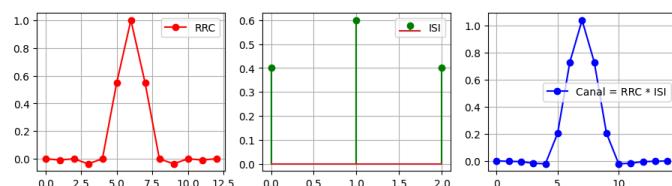


Figura 26 - Respuestas al impulso antes del FSE

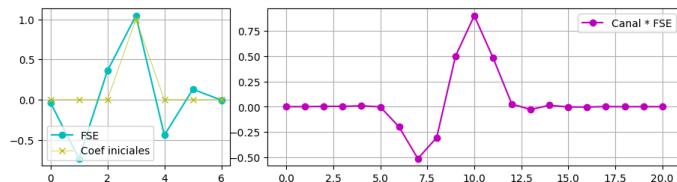


Figura 27 - Respuesta al impulso del FSE y total (canal\*FSE) para 7 taps

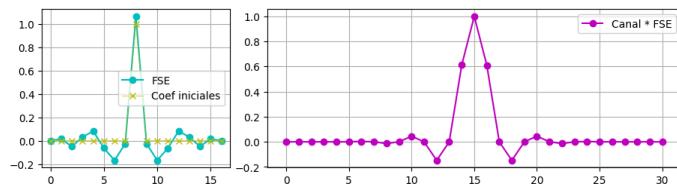


Figura 28 - Respuesta al impulso del FSE y total (canal\*FSE) para 17 taps

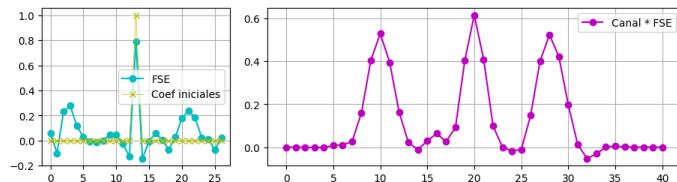


Figura 29 - Respuesta al impulso del FSE y total (canal\*FSE) para 27 taps

Se probaron tres cantidades de coeficientes distintas y se vió que la más óptima es la de 17, ya que con 7 taps nos quedamos cortos y el sistema evoluciona hacia una respuesta no centrada del todo y con mucha ISI residual. Para 27 taps es aún peor ya que la respuesta total queda distribuida en tres puntos afectando a tres símbolos distintos, lo cual es inaceptable.

## Simulación realista del FSE con algoritmo LMS en punto fijo

Una vez estudiado y analizado el comportamiento del sistema en punto flotante, es momento de realizar la representación en punto fijo, simulando las operaciones que realizaría el hardware “uno a uno”, pero en software.

### 1) Esquemático

Lo primero es armar el esquemático final o a bajo nivel, es decir, a nivel de RTL, ya que eso es lo que se simulará en Python. Luego se debe hacer un estudio de la cuantización, analizando cuál es la cantidad de bits necesaria para cada variable del sistema, teniendo en cuenta el error proporcionado por dicha cuantización, en relación a la representación ideal (punto flotante).

Nos metemos dentro de los bloques de la Figura 7, donde el FSE está compuesto por un filtro FIR en su forma directa, en este caso de N=17 coeficientes, guiándonos con la Figura 6. Para el bloque LMS, tendremos la operación  $w_{[n+1]} = w_{[n]} + \mu * e_n * x_{[n]}$  (en corchetes los subíndices de las variables que son arreglos correspondientes a los N registros) repetida N veces por cada coeficiente. Los coeficientes se actualizan a tasa 1/T, es decir, una vez cada dos muestras de entrada, o cada un símbolo. Los esquemas se muestran en las Figuras 33 y 34.

## 2) Cuantización

En cuanto al pasaje a punto fijo, se prueban distintas cuantizaciones y se analiza el error producido por cada una, para determinar cuál es la mejor opción. Para la constante de aprendizaje, como debemos utilizar un paso pequeño, seleccionamos un  $s(16;15)$  (*16 bits con signo en total, con 15 de parte fraccionaria*), ya que es necesario utilizar mayor cantidad de bits que en el resto de variables. Para el resto, lo importante será definir los bits (decimales y fraccionarios) de la entrada y la salida, y a partir de ellos se calculan los bits para otras operaciones.

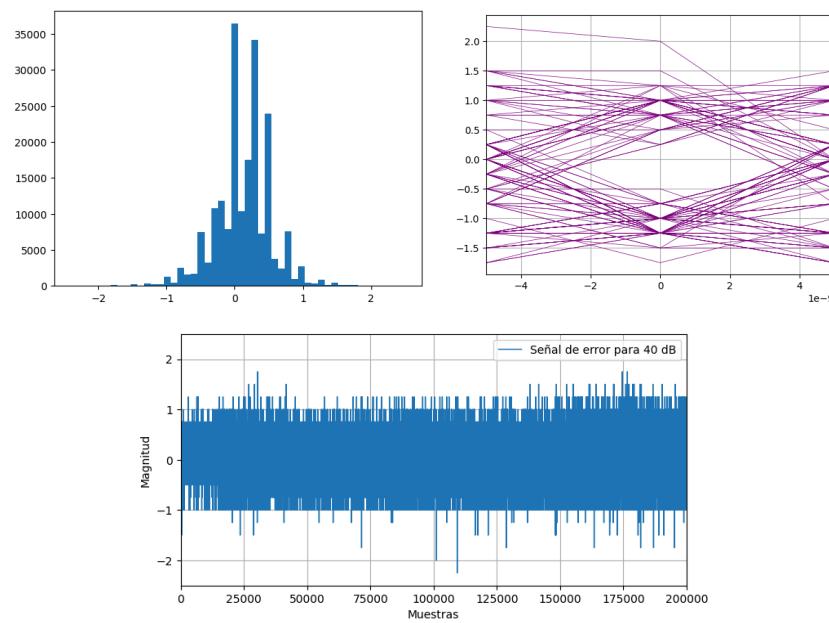


Figura 30 - Cuantización  $s(5;2)$  para  $x(n)$  e  $y(n)$ : Histograma de error absoluto  $y - y_{fp}$  (arriba izq.) y en diagrama de ojo (arriba der.), y señal de error  $e(n)$  en el tiempo (abajo)

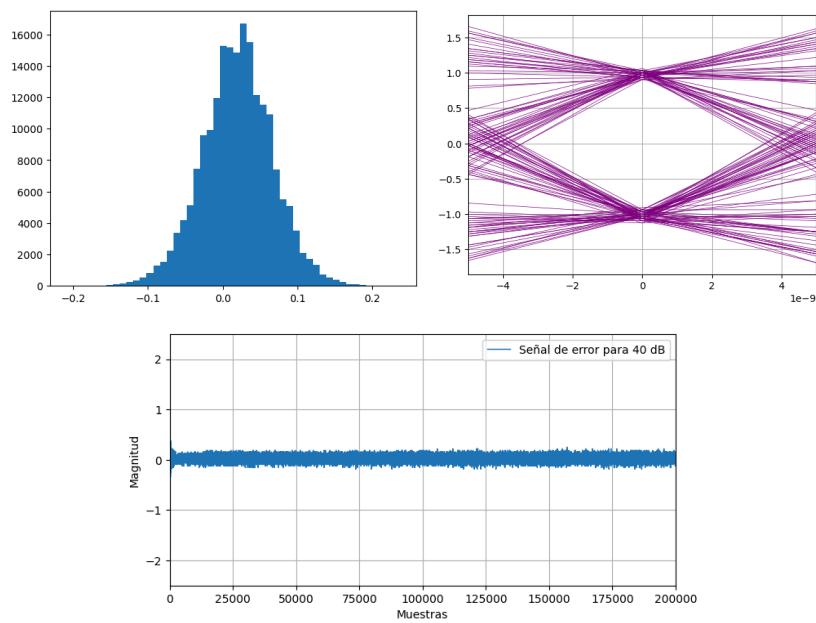


Figura 31 - Cuantización  $s(8;5)$  para  $x(n)$  e  $y(n)$ : Histograma de error absoluto  $y - y_{fp}$  (arriba izq.) y en diagrama de ojo (arriba der.), y señal de error  $e(n)$  en el tiempo (abajo)

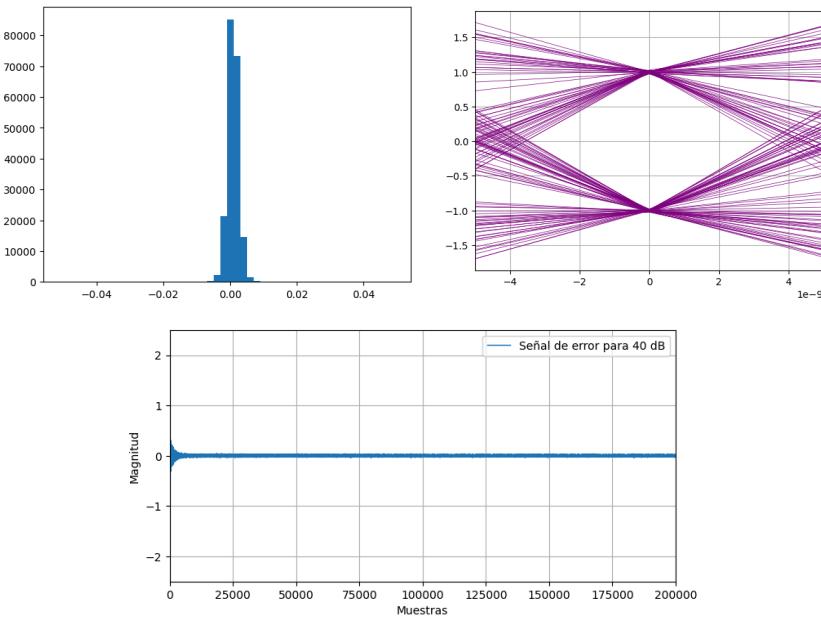


Figura 32 - Cuantización  $s(13;10)$  para  $x(n)$  e  $y(n)$ : Histograma de error absoluto  $y - y_{fp}$  (arriba izq.) y en diagrama de ojo (arriba der.), y señal de error  $e(n)$  en el tiempo (abajo)

Comparando las Figuras 30, 31 y 32, vemos que para  $s(5;2)$  el error de cuantización es muy elevado e inadmisible en un contexto como este. Para  $s(8;5)$ , el error mejora y el diagrama de ojo es más claro, con puntos de muestreo más definidos, pero se elige  $s(13;10)$  debido a que el error absoluto no sobrepasa el 1% y el diagrama de ojo tiene puntos bien definidos, asemejándose a la representación en punto flotante.

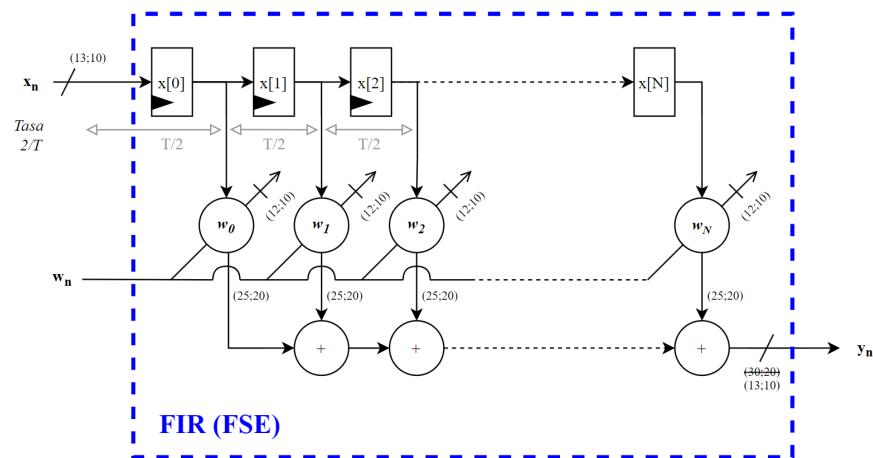


Figura 33 - Bloque FSE

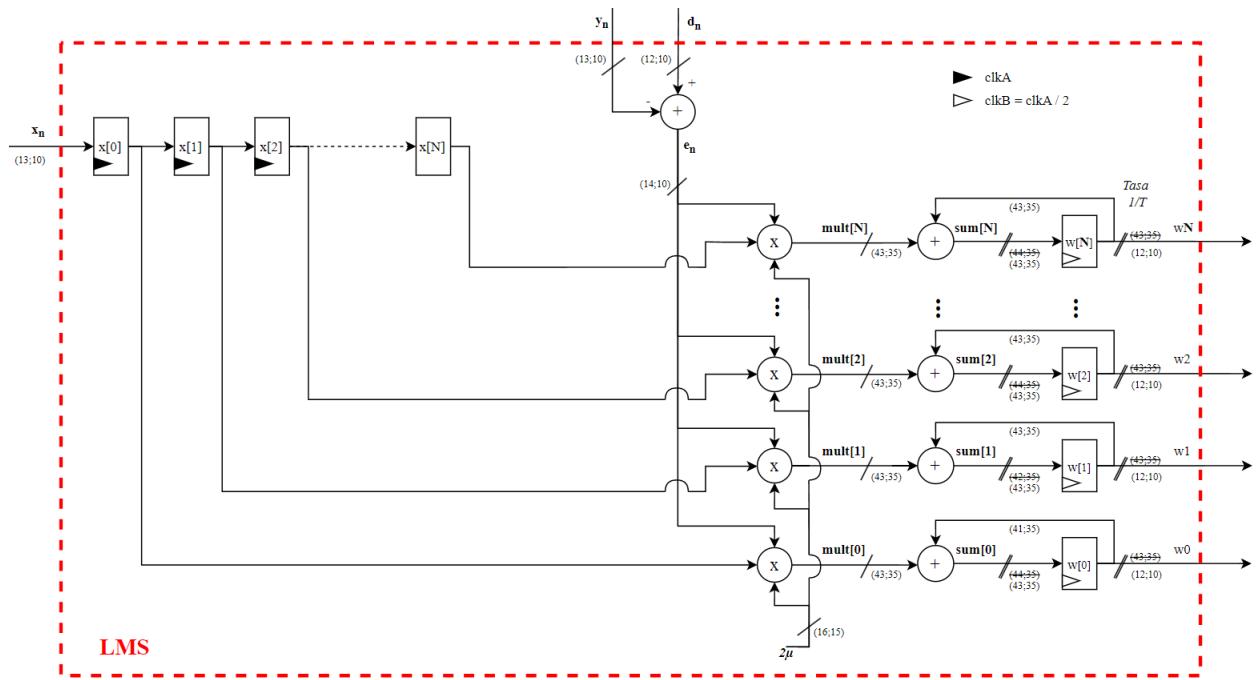


Figura 34 - Bloque LMS

### 3) Tasa de error de bits

Antes de avanzar en el proyecto y realizar el diseño de hardware en RTL, es necesario tomar alguna medida que permita evaluar el rendimiento del sistema. Una muy utilizada es la *Bit Error Rate* (BER), que cuantifica la frecuencia con la que se reciben bits erróneos en comparación con el número total de bits transmitidos en un intervalo de tiempo. Esta suele graficarse en una curva versus la relación señal-ruido (SNR) y son inversamente proporcionales, es decir, a mayor SNR, mayor calidad de señal recibida (más diferenciada del ruido) y menor probabilidad de error en la transmisión. Existen curvas típicas para cada tipo de modulación, nosotros nos centramos en la QPSK/BPSK.

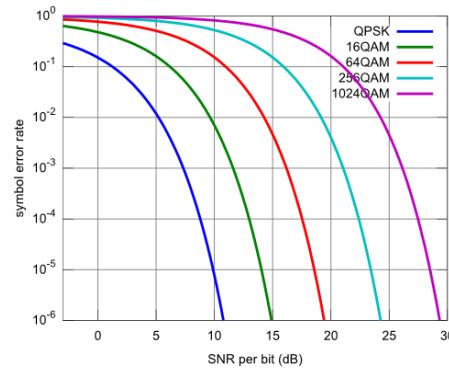


Figura 35 - Curva BER vs. SNR para distintas modulaciones digitales

Para graficar correctamente nuestra curva, debemos ir aumentando el AWGN mediante el aumento de la SNR, e ir tomando los valores para cada valor de BER. Se hace el barrido de 0 a 10 dB y se contrasta con la curva ideal.

$$BER_{BPSK} = Q\left(\sqrt{2 \frac{E_b}{N_0}}\right)$$

$$Q = \frac{1}{2} \operatorname{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

$E_b/N_0$  es la relación energía/ruido por bit, en escala lineal.

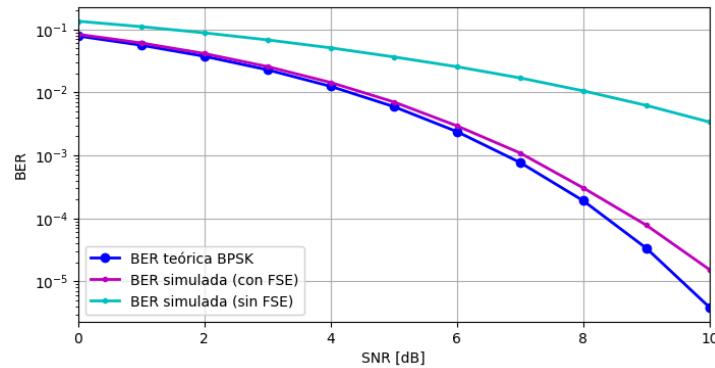


Figura 36 - Curva BER vs. SNR simulada y teórica

## Implementación en hardware

Hasta este punto todo fue simulación. Primero se realizaron pruebas varias para hacer un análisis básico de distintos bloques del sistema, con representaciones poco realistas del mismo. Se fue complejizando en punto flotante hasta alcanzar algo que se asemeja más a una aplicación real y, por último, se trasladó a una simulación en punto fijo. En el camino se realizó un análisis de las distintas señales, medidas y variables a tener en cuenta para representar el proceso.

Ahora es momento de llevar todo eso a una descripción de hardware real para luego implementarlo en una placa, teniendo de referencia el modelo simulado.

### Descripción de bloques en Verilog y Análisis RTL

Al igual que en las simulaciones, el código se escribe en la plataforma de Visual Studio Code, gracias a ventajas ya enumeradas. El flujo de diseño de hardware, Testbench, síntesis e implementación, se realizan en la plataforma de Vivado, de AMD (anteriormente Xilinx).

Como buena práctica, el código estará dividido en distintos bloques, con un top principal y sus respectivas instancias. Esta forma de realizar la descripción nos da muchos beneficios, como escalabilidad, modularidad, reutilización de código, facilidad de testing, y legibilidad, pudiendo incluso hasta lograr una síntesis más eficiente.

### 1) LMS con ecualizador convencional

Lo primero fue describir el bloque LMS de la Figura 34, pero en una versión simplificada y con una senoidal de entrada similar a las primeras simulaciones, para corroborar que el algoritmo se lleve a cabo correctamente. Se aplica junto a un FIR convencional (no fraccionado).

Para dar los estímulos se crea un archivo de *Testbench*, donde se llama al módulo a estimular, en este caso el LMS y FIR, y se configuran los impulsos de entrada, para luego observar su salida. En este banco de pruebas se simula el comportamiento y solo se verifica funcionalidad, antes de sintetizar el hardware físico.

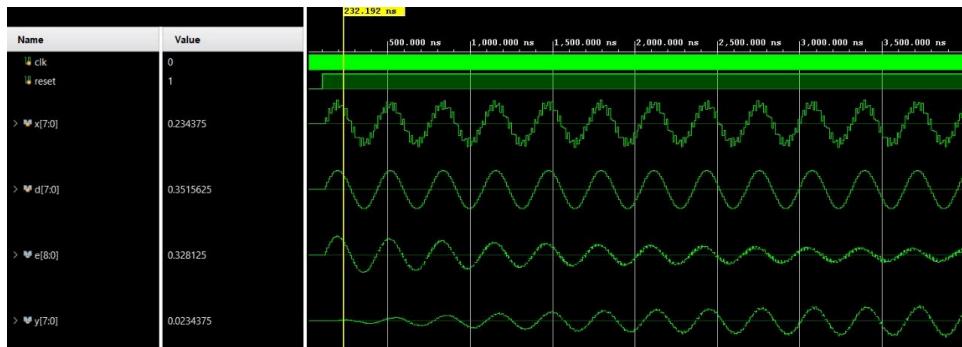


Figura 37 - Testbench para senoidal de entrada

Vemos que a la entrada se tiene una senoidal que tiene otro seno montado, es decir, son dos tonos en frecuencia. La señal deseada es la senoidal principal, de menor frecuencia. Al ejecutar el algoritmo se ve como el error va decayendo con el tiempo y la salida va tiendiendo hacia la senoidal pura, alcanzando la señal buscada.

## 2) LMS con FSE

Luego de corroborar que el LMS (*LMS.v*) funciona correctamente, podemos aplicarlo a nuestro ecualizador fraccionalmente espaciado (*FIR.v*), instanciando ambos bloques en un módulo principal (*top.v*).

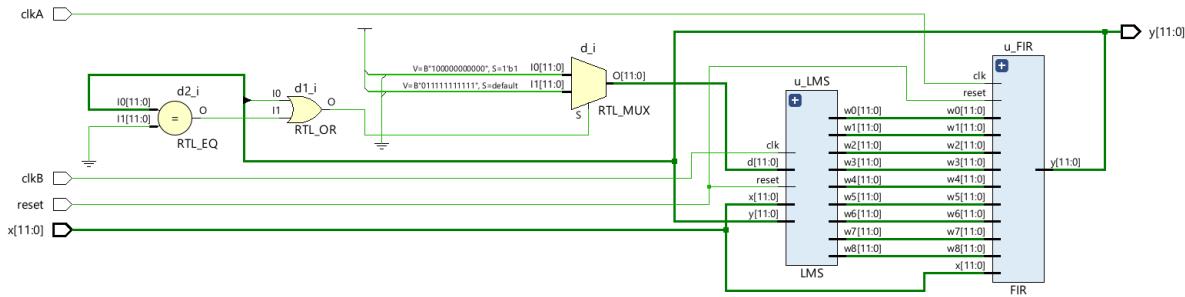


Figura 38 - Esquemático de “RTL Analysis” de Vivado

Luego, con otro archivo (*tb\_top.v*) damos los impulsos necesarios para hacer pruebas funcionales sobre el diseño. En este caso, la entrada provendrá de un archivo creado en Python, donde se guarda exactamente la misma señal de entrada utilizada para la simulación para más tarde hacer el Vector Matching y que los resultados sean iguales.

```
#Generación de estímulos desde Python
folder_path = r"C:\VivadoProjects\TrabajoFinalDDA\TrabajoFinalDDA.sim\sim_1\behav\xsim"
os.makedirs(folder_path, exist_ok=True)
filename = os.path.join(folder_path, 'ENTRADA.hex')

//Lectura de estímulos desde el Testbench de Verilog
$readmemh("ENTRADA.hex", entrada);
for (j=0; j<(TAM_MEM); j=j+1) begin
    x = entrada[j];
    #10;
end
```

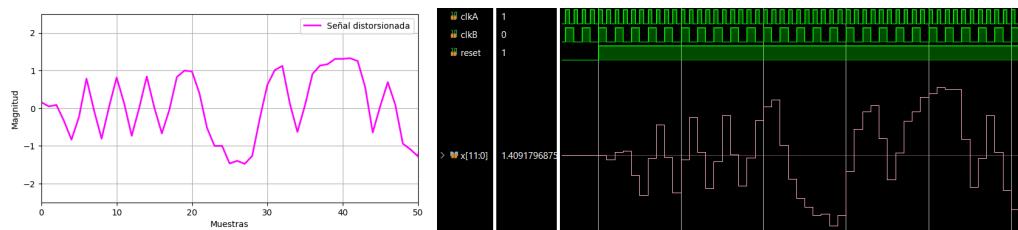


Figura 39 - Generación de estímulos de Python (izq.) a Verilog (der.)

### 3) Vector Matching: simulación vs. RTL

Para corroborar que lo descripto en hardware se condiga con lo simulado anteriormente en alto nivel, es necesario comparar una secuencia representativa de valores de uno y otro. Una forma cómoda de hacerlo es extrayendo los datos de salida del Testbench y llevándolos a Python para poder manejarlos con más facilidad.

Luego de reiteradas pruebas y retoques en ambos códigos, se alcanzó el comportamiento buscado y la coincidencia de los datos entre los dos modelos.

```
//Se guardan los datos de salida del Testbench en memoria
file = $fopen("resultados.txt", "w");
for (j=0; j<(TAM_MEM); j=j+1) begin
    $fwrite(file, "%0h\n", y);
    #10;
end

#Se lee el archivo desde Python
with open ("C:\VivadoProjects\TrabajoFinalDDA\TrabajoFinalDDA.sim\sim_1\behav\xsim\resultados.txt", "r") as f:
    hex_values = [line.strip() for line in f if line.strip()]
```



Figura 40 - Extracción de resultados de Verilog (izq.) a Python (der.)

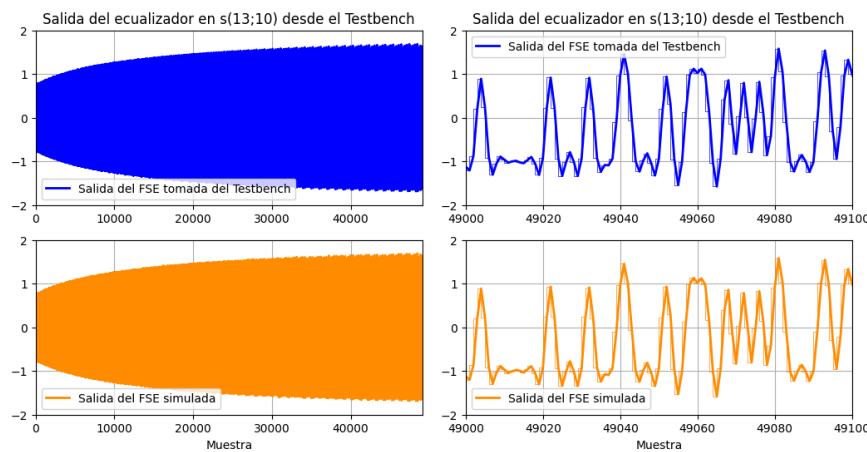


Figura 41 - Vector Matching: comparación entre salida del FSE traída del Testbench (arriba) y simulada (abajo)

#### 4) Adición de otros bloques para validar funcionamiento en placa

Luego de comprobar que lo realizado en RTL coincide con lo simulado en las pruebas funcionales, se puede llevar el diseño a la placa. Pero en este punto se necesita tener en cuenta un aspecto importante: no se dispone de la placa en mano, sino que nos conectamos en forma remota. Debido a esto, no es posible introducir estímulos desde fuera ni corroborar el funcionamiento analizando las salidas. Para ello necesitamos crear un contexto que simule las entradas y nos permita analizar las salidas remotamente. Acá entra en juego el concepto de IP Cores, que son bloques lógicos prediseñados y reutilizables que encapsulan funcionalidades específicas y pueden ser integrados en la FPGA o diseños *System on Chip* (SoC). Dos muy importantes son el *Virtual Input/Output* (VIO) y el *Integrated Logic Analyzer* (ILA). El VIO permite controlar señales internas del diseño de forma virtual, como forzar valores o controlar el comportamiento de la lógica, acá se utilizará para poner el reset en alto o bajo. Por otro lado, el ILA permite

visualizar señales internas del diseño, como un osciloscopio, muy útil para observar la salida y extraer los valores para ser comparados.

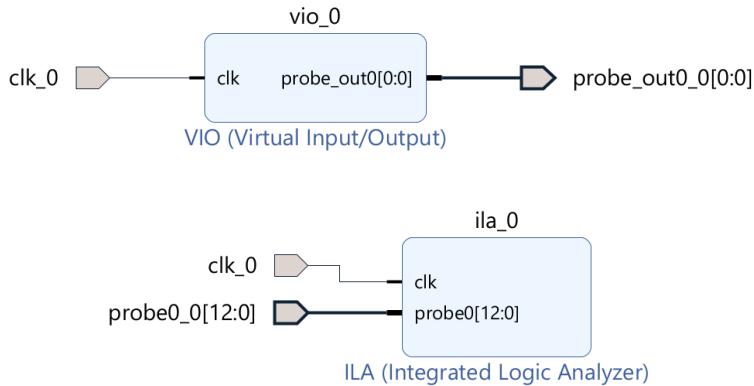


Figura 42 - IP Cores de VIO (arriba) e ILA (abajo)

Además de estos bloques (`vio.v` e `ila.v`), será necesario otro que nos permita generar nuestra entrada  $x(n)$ , que hasta ahora había sido introducida mediante el Testbench, pero es solo de prueba. Entonces, creamos un nuevo bloque (`ROM_signal.v`) que contenga los valores de la señal de entrada para así estimular el diseño. Pero no es necesario que tenga todos los valores, ya que solo bastará con 511 (1022 por el oversampling) muestras de la PRBS9, y se repetirán en bucle.

En un principio se hizo de forma “hardcodeada” escribiendo los valores duros de las muestras en registros, copiando y pegando directamente desde Python. Esto funcionó y es válido, pero existe una manera más eficiente de realizarlo: utilizando la IP Core de *Block Memory Generator*. A este bloque se le carga un archivo con la secuencia y se accede a él con señal de address, clock, reset y enable.

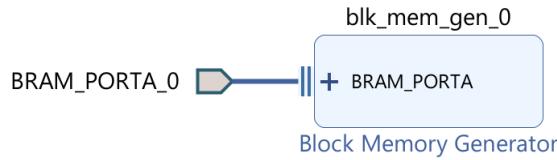


Figura 43 - IP Core de Block Memory Generator

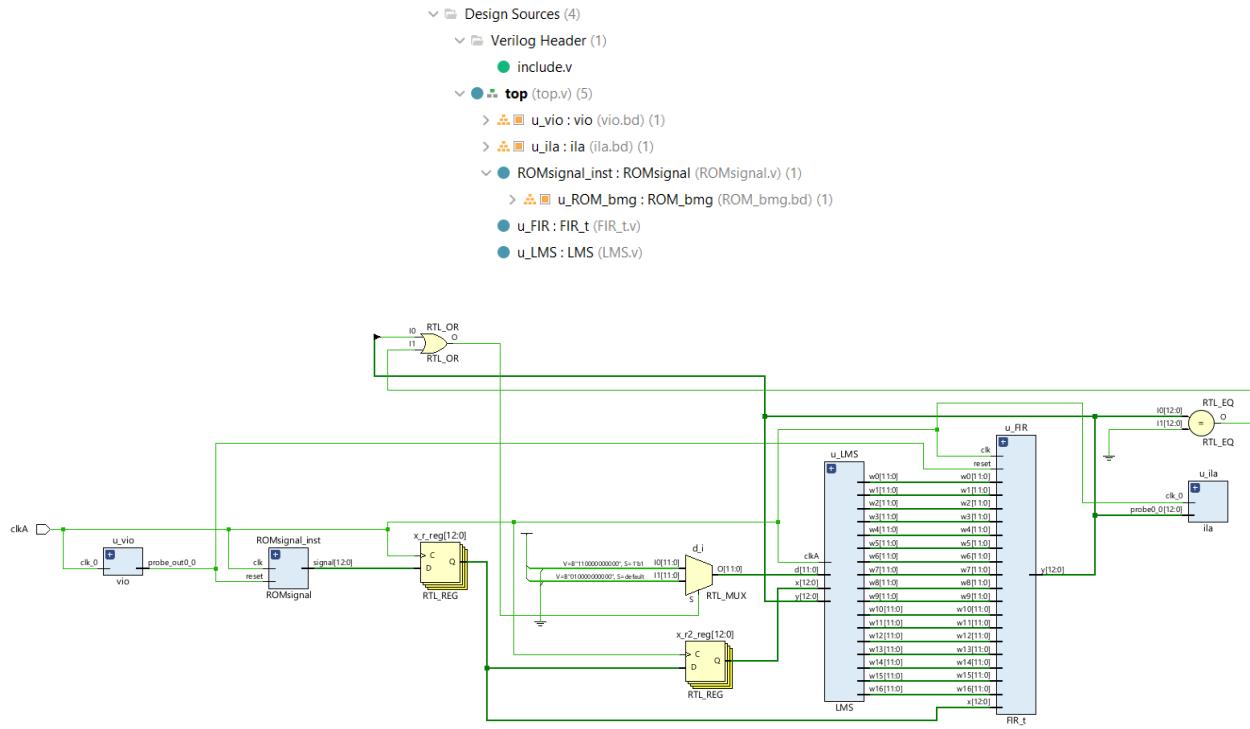


Figura 44 - Esquemático completo pre-síntesis

Por último, se agrega el archivo *instances.v*, llamado desde el módulo principal y que es encargado de contener las constantes y parámetros de todo el diseño, aportando claridad y un mejor manejo del código. Allí se encuentran las cantidades de bits de distintas variables, como entradas o salidas, y algunos *define* utilizados para activar o desactivar algunos fragmentos de código.

```
////////! Parámetros y constantes
```

```
//Descomentar el que quiera usar, con VIO o sin VIO (usa reset del testbench)
`define VIO      //Descomentar si uso VIO
//`define noVIO   //Descomentar si no uso VIO

//Descomentar el que quiera usar, con ROM o sin ROM(usa entrada del testbench)
`define ROM     //Descomentar si uso ROM
//`define noROM   //Descomentar si no uso ROM

//Cantidad de bits
`define NBin      13
`define NBFin     10
`define NBOut     13
`define NBFout    10
`define NBCoeff   12
`define NBFcoeff  10
`define NBD       12
`define NBFd      10

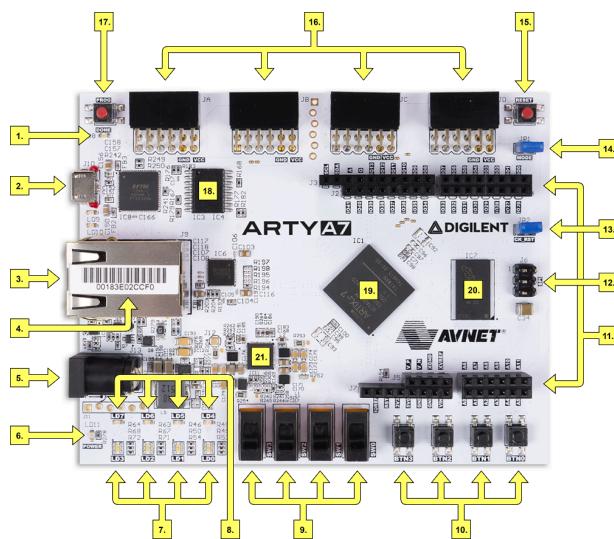
//ROM
`define PRBSx    9
`define os        2
```

## Síntesis e Implementación

Hasta ahora se realizaron verificaciones, simulaciones y optimizaciones preliminares para asegurar que el diseño RTL cumpla con los requisitos funcionales y de rendimiento antes de la conversión a un diseño a nivel de compuerta. Acá se aplican optimizaciones básicas en el código RTL para mejorar el rendimiento o reducir el tamaño del diseño antes de la síntesis. Esto incluye la eliminación de redundancias, la simplificación de expresiones lógicas o la aplicación de ciertas técnicas de optimización.

Lo que sigue es correr la síntesis. Este es el proceso de convertir una descripción de hardware en HDL a nivel de registro, en una representación a nivel de compuerta mediante una lista de conexiones. Esta lista de conexiones es lo que se utiliza para implementar el diseño en un circuito físico.

Luego vendría la implementación, que traduce el diseño HDL sintetizado, en una configuración específica para un dispositivo FPGA, en este caso la placa Arty A7-35 (Artix-7). La implementación incluye la síntesis antes nombrada, la asignación de recursos, el enrutamiento y a partir de ello se genera el archivo de flujo de bits (bitstream), que es el que se monta finalmente en la FPGA.



Callout	Description	Callout	Description	Callout	Description
1	FPGA programming DONE LED	8	User LEDs	15	chipKIT processor reset
2	Shared USB JTAG / UART port	9	User slide switches	16	Pmod connectors
3	Ethernet connector	10	User push buttons	17	FPGA programming reset button
4	MAC address sticker	11	Arduino/chipKIT shield connectors	18	SPI flash memory
5	Power jack for optional external supply	12	Arduino/chipKIT shield SPI connector	19	Artix FPGA
6	Power good LED	13	chipKIT processor reset jumper	20	Micron DDR3 memory
7	User RGB LEDs	14	FPGA programming mode	21	Dialog Semiconductor DA9062 power supply

Figura 45 - FPGA Arty A7-100T (nueva versión de A7-35T)

### 1) Primeras implementaciones de prueba

A modo de prueba, antes del diseño final (sin ROM, VIO e ILA) se corrió la síntesis e implementación para ver la diferencia del sistema para distintas cantidades de coeficientes del ecualizador, ya que debe realizarse la misma operación por cada coeficiente, intuyendo que hay cierta linealidad entre área y cantidad de taps. Se hace la prueba entonces para 9 y 17 coeficientes inicialmente.

Worst Negative Slack (WNS):	7.934 ns	Worst Negative Slack (WNS):	7.946 ns
Total Negative Slack (TNS):	0 ns	Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	241	Total Number of Endpoints:	577

Figura 46 - Resultados de timing para 9 coeficientes (izq.) y 17 coeficientes (der.)

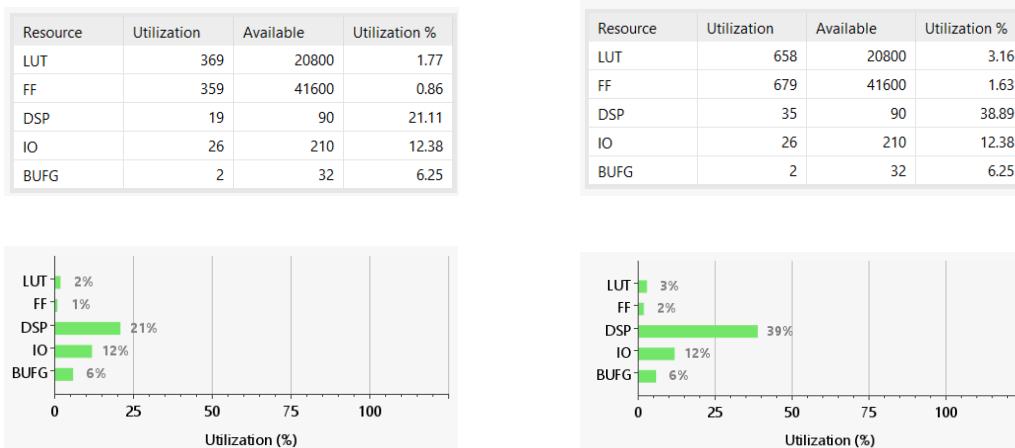


Figura 47 - Reporte de utilización para 9 coeficientes (izq.) y 17 coeficientes (der.)

Vemos que, si bien el timing es prácticamente igual para ambos casos, existe una relación aproximadamente lineal entre la cantidad de coeficientes y el área utilizada, siendo los elementos críticos los *Digital Signal Processors* (DSP). Estos son componentes

optimizados para realizar cálculos numéricos a alta velocidad. Estas operaciones matemáticas están presentes en el cálculo del algoritmo LMS, es decir, en la actualización de los coeficientes, por lo que se utilizarán más DSP mientras más coeficientes tengamos.

Con los datos obtenidos podemos hacer la siguiente estimación:

- 9 coeficientes → 19 bloques DSP (21,11%)
- 17 coeficientes → 35 bloques DSP (38,89%)
- Se estiman menos de 45 coeficientes para ocupar el 100% de los DSP (90 bloques).

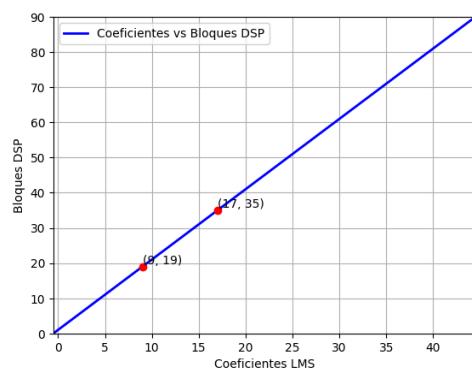


Figura 48 - Estimación coeficientes vs. bloques DSP

Con esto, procedemos a correr la implementación para 45 coeficientes y ver los resultados:

Worst Negative Slack (WNS):	7.515 ns
Total Negative Slack (TNS):	0 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	1753

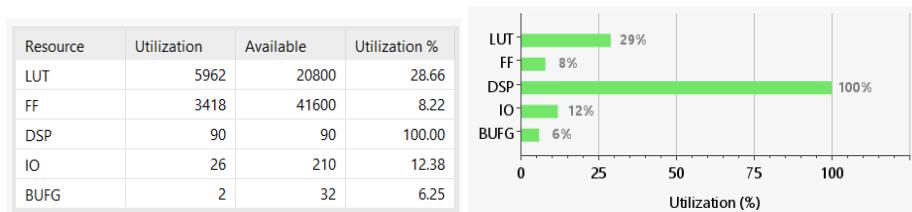


Figura 49 - Modelo de 45 coeficientes: Resultados de timing (arriba) y reporte de utilización (abajo)

Nuevamente, el timing cierra con el mismo tiempo que los otros casos, simplemente aumenta el número total de endpoints, como se puede deducir. Lo importante está en el área, o la utilización de componentes, ya que vemos que alcanza a usar el 100% de los bloques DSP disponibles, como se predijo, e intenta resolver el excedente con *Look-Up Tables* (LUT) y *Flip-Flops* (FF), por lo que estos últimos también crecen en gran medida.

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF	DSP	BRAM	URAM
synth_1	constrs_1	Synthesis Out-of-date									547	359	19	0	0
impl_1	constrs_1	Implementation Out-of-date	7.934	0.000	0.166	0.000		0.000	0.107	0	369	359	19	0	0
synth_2	constrs_1	Synthesis Out-of-date									987	679	35	0	0
impl_2	constrs_1	Implementation Out-of-date	7.946	0.000	0.158	0.000		0.000	0.139	0	658	679	35	0	0
synth_3 (active)	constrs_1	<b>synth_design Complete!</b>									6027	3418	90	0	0
impl_3 (active)	constrs_1	<b>route_design Complete!</b>	7.515	0.000	0.052	0.000		0.000	0.323	0	5962	3418	90	0	0

Figura 49 - Resumen de las 3 implementaciones

Este análisis nos permite saber las dimensiones de nuestro diseño y hasta qué punto es capaz de correr en la placa brindada. También nos sirve para verificar cuándo actúan los distintos bloques constitutivos de la FPGA, y cómo resuelve algunas operaciones del sistema.

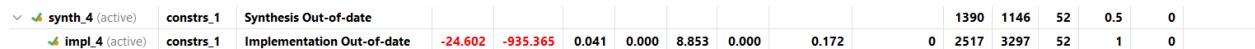
## 2) Implementación de diseño completo

Como se explicó en la sección de hardware, fueron necesarios otros bloques para correr el diseño en la placa y que pueda ser testeado remotamente. Estos son la ROM, el VIO y el ILA. Además de ello, se necesita el archivo de *constraints.xdc*, que es un archivo de texto utilizado para indicarle al software qué pines físicos de la placa se van a usar en relación con las entradas y salidas del HDL, además de restricciones de diseño para el proyecto, como parámetros para la temporización y la ubicación de los componentes.

En este caso, al no utilizar las entradas y salidas físicas (se usa VIO e ILA), solo queda configurar el clock, que será para 100 MHz.

```
set_property -dict {PACKAGE_PIN E3 IOSTANDARD LVCMOS18} [get_ports clkA]
create_clock -period 10.000 -name sys_clk_pin -waveform {0.000 5.000} -add [get_ports clkA]
```

Con todo ello incluído se procede a correr la implementación para más tarde generar el bitstream que nos permita subirlo a la FPGA. Pero existe un problema inicial, y es que no cierran los resultados de Timing. El *Worst Negative Slack* (WNS) nos indica la falla de Timing más severa del diseño, o camino crítico, y en este caso es de -24.602 ns, cuando debería dar positivo (cuando ningún camino viola restricciones de tiempo).



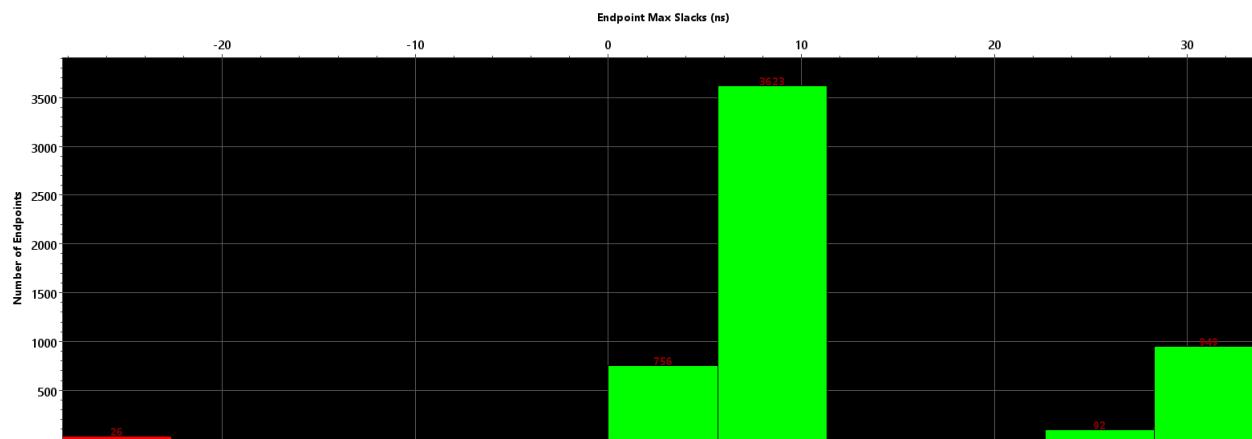


Figura 51 - Resultados de la síntesis e implementación (falla timing), y Slack Histogram

### 3) Corrección de diseño para requisitos de tiempo

En este punto deben aplicarse medidas o técnicas necesarias para cerrar el Timing y que el WNS sea positivo. Como se trata de un sistema con realimentación, no es tan sencilla la tarea de agregar registros pipeline o hacer retiming manual para acortar el camino crítico, así que luego de unas pequeñas pruebas de este estilo de forma manual, los problemas continúan.

Existe una forma de aplicar retiming automático dentro del Vivado, en la configuración de la herramienta, dentro de la sección de Síntesis. Al tildar esta opción, la herramienta busca mejorar el desempeño del circuito mediante *Register Balancing*, que es básicamente mover registros a través de compuertas combinacionales o LUTs, pero manteniendo el comportamiento y latencia sin requerir cambios en las fuentes de RTL.

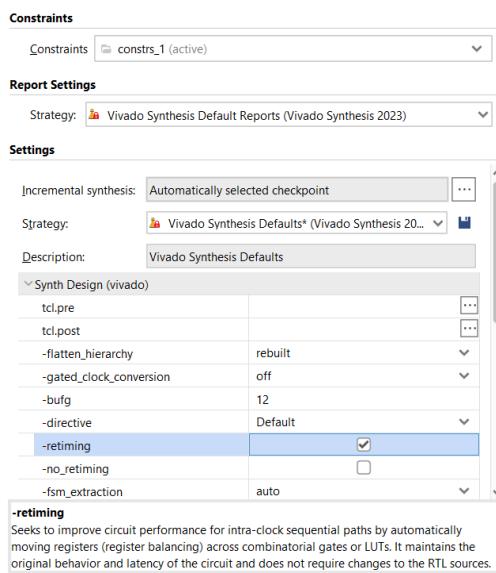


Figura 52 - Configuración de retiming automático

Nuevamente, la implementación no pasó el Timing. Fue momento, entonces, de analizar más en profundidad los caminos críticos que indica la propia plataforma, y se encontró que la mayoría se encontraban dentro del bloque correspondiente al FSE (*FIR.v*). Tomando esto de punto de partida, se atacó esa parte del circuito y se llegó a la idea de cambiar la estructura del filtro FIR, de modo que el camino crítico a través de él sea más corto. Entonces se pasó de una estructura directa a una transpuesta (*FIR\_t.v*), con los registros entre las sumas de los productos, y no entre los productos.

Finalmente, este tipo de estructura es la óptima a la hora de cerrar los requisitos de Timing, con un WNS de 2.944 ns.

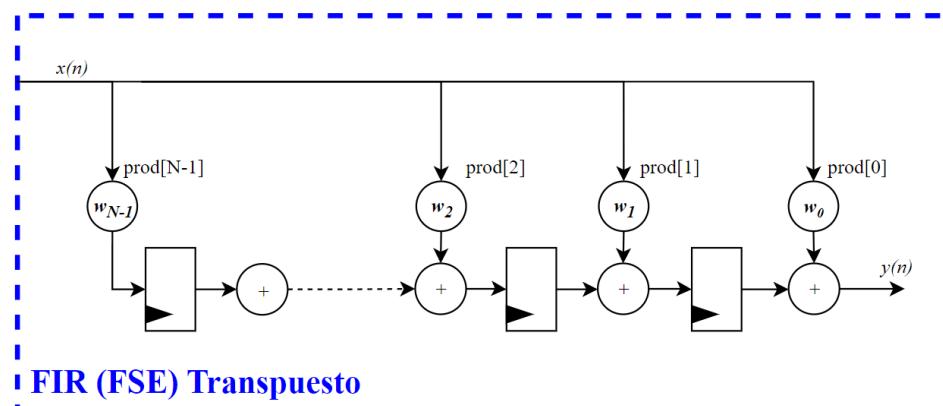


Figura 53 - Bloque FSE con FIR transpuesto

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	LUT	FF	DSP	BRAM	URAM
synth_4 (active)	constrs_1	Synthesis Out-of-date									1764	1003	52	0.5	0
impl_4 (active)	constrs_1	Implementation Out-of-date	2.944	0.000	0.038	0.000	9.140	0.000	0.080	0	1227	2108	17	1	0

Figura 54 - Resultados de la síntesis e implementación (cierra timing)

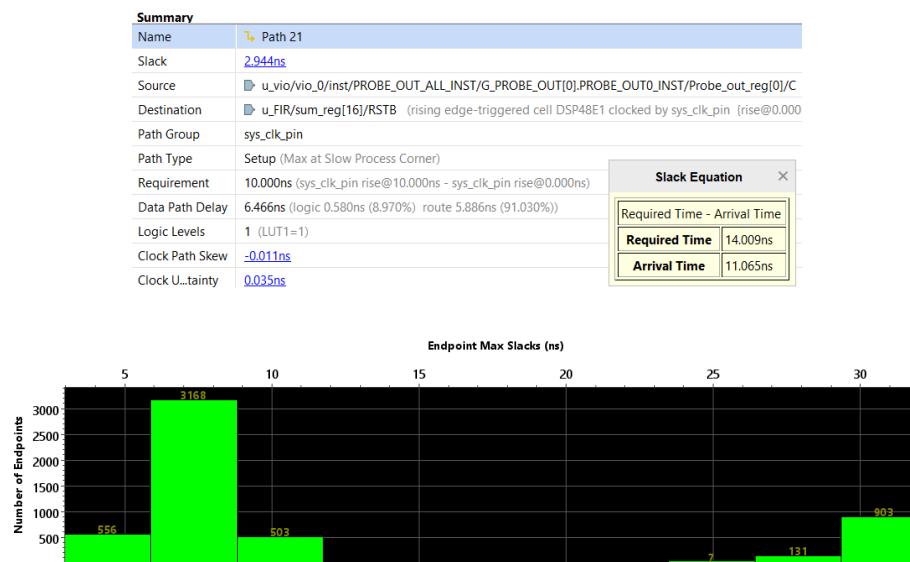


Figura 55 - Resumen de WNS (arriba) y Slack Histogram (abajo)

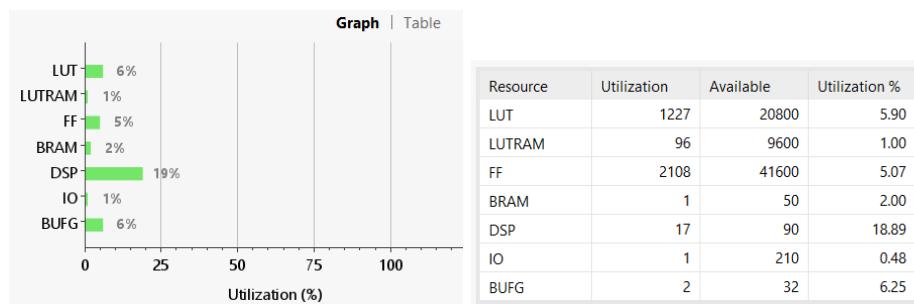


Figura 56 - Reporte de utilización

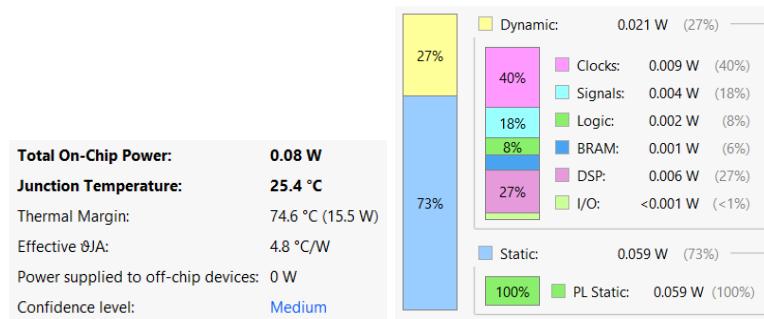


Figura 57 - Potencia disipada

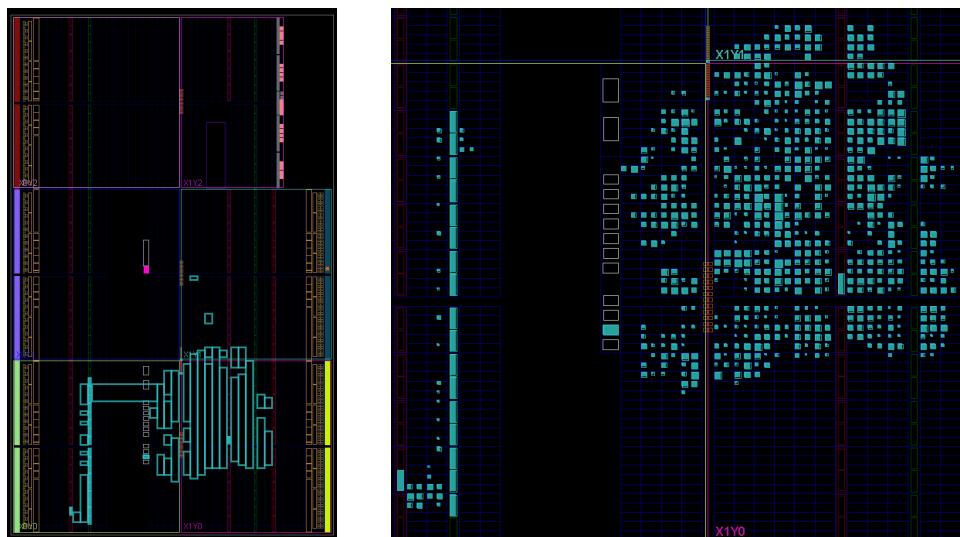


Figura 58 - Dispositivo post-síntesis

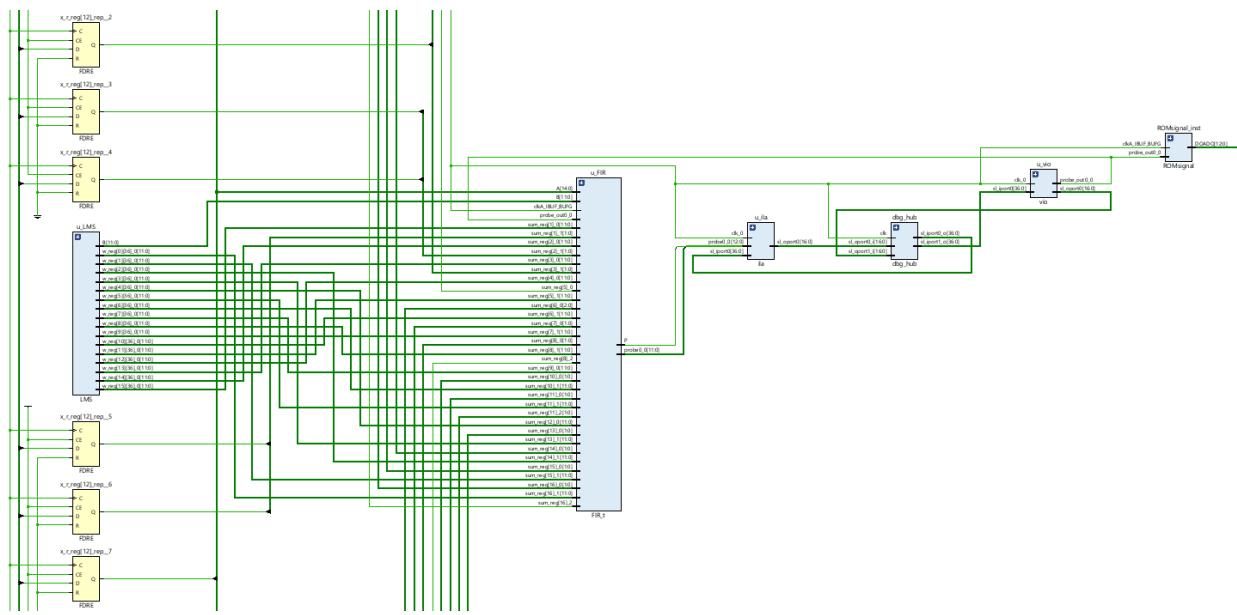


Figura 59 - Esquemático de implementación

## Generación de bitstream e implementación en FPGA

El último paso del flujo es aplicar todo lo realizado hasta el momento, al hardware real.

Ya sabemos que cumple los requisitos de área, tiempo y potencia. Simplemente nos queda generar el binario, el archivo que contiene toda la información para correr el diseño en la placa. Una vez corrida la implementación exitosamente, se genera el bitstream y la plataforma nos da un archivo *.bit*, el cual cargaremos en la placa mediante la sección Hardware Manager de Vivado, habiéndonos conectado previamente de forma remota. Allí accedemos al VIO, lo que permite poner el reset en alto o bajo, y el ILA, que nos deja ver la señal de salida y dispararla con un Trigger, con distintas opciones de visualización. También podemos extraer un archivo *.csv* con los datos de la ventana

mostrada del ILA, lo que nos permitiría hacer un segundo Vector Matching directamente con los datos extraídos del sistema en la placa.



Figura 60 - VIO (arriba) e ILA (abajo)

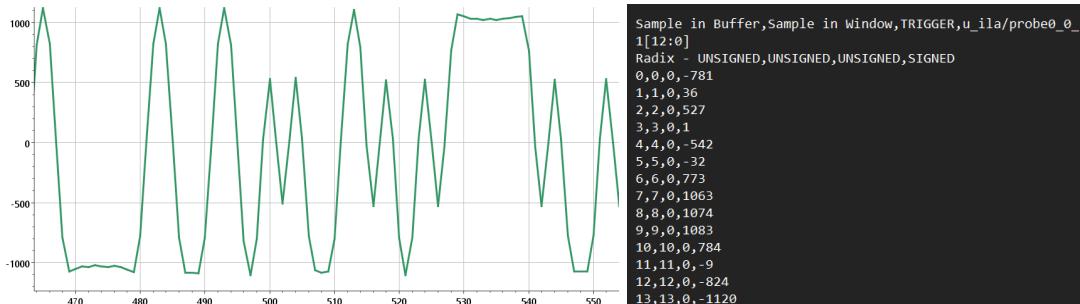


Figura 61 - Datos de la ventana del ILA: Gráfico de Vívado (izq.) y archivo .csv (der.)

Como última comprobación, se lee el archivo .csv extraído del ILA con un script de Python. Allí se acomoda y transforma los valores según sea necesario y se realiza operación de correlación entre la ventana de 1024 valores del ILA, con la señal del Testbench antes utilizada para hacer el primer Vector Matching. Esta correlación, suponiendo que se toman 20000 muestras de la señal del Testbench, nos daría una sucesión de picos de distintos tamaños, con uno por cada ciclo de PRBS9 (cada 1022

valores), indicando que la ventana de 1024 muestras se repite cada ciclo pero con distinta amplitud, es decir, son los mismos símbolos pero van siendo modificados debido al avance del algoritmo que los adapta a la forma buscada. El pico más alto indica el ciclo correcto.

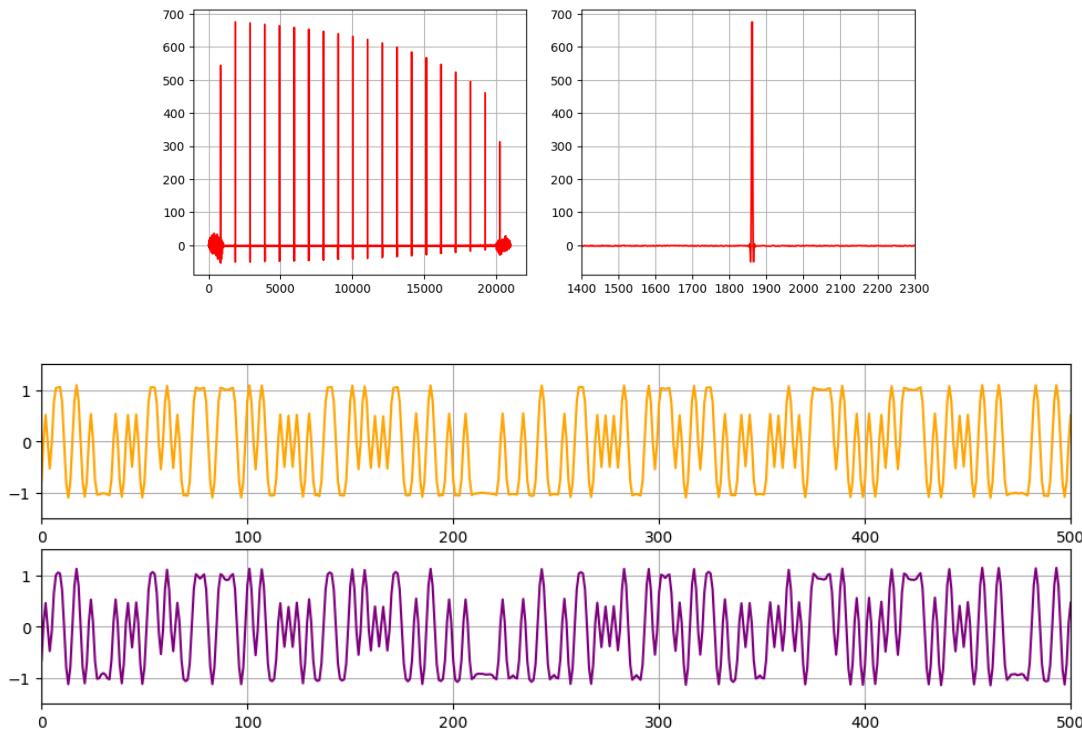


Figura 62 - Vector Matching ILA vs. Testbench: correlación (arriba) y comparación en el tiempo (abajo)

## Bibliografía

Artículos y ensayos:

- [1] S. H. Leung., B. L. Chan and S. M. Lau (1992) “AN EFFICIENT FRACTIONALLY SPACED EQUALIZER WITH LOW COMPUTATIONS FOR DATA TRANSMISSION”

*Department of Electronic Engineering City Polytechnic of Hong Kong*

- [2] J. M. Cioffi and Thomas Kailath (1984) “An Efficient Exact-Least-Squares Fractionally Spaced Equalizer Using Intersymbol Interpolation” *IEEE Journal on Selected Areas in Communications*

- [3] J. A. Castellanos Hernández, C. E. Sandoval Ruiz, M. A. Azpúrua Auyanet (2014) “A FPGA implementation of a LMS adaptative algorithm for smart antenna arrays”

*Rev. Téc. Ing. Univ. Zulia vol.37 no.3 Maracaibo*

- [4] Sanquan Song and Vladimir Stojanovic (2011) “A 6.25 Gb/s Voltage-Time Conversion Based Fractionally Spaced Linear Receive Equalizer for Mesochronous High-Speed Links” *IEEE Journal of Solid-State Circuits*

- [5] Brignone, Matías Nicolás y Rodríguez, Lucía Fernanda (2018) “SIMULACIÓN E IMPLEMENTACIÓN EN FPGA DE ECUALIZADOR FRACCIONALMENTE ESPACIADO EN EL DOMINIO DE LA FRECUENCIA CON ALGORITMO LMS PARA ADAPTACIÓN DE COEFICIENTES” *Proyecto integrador para la obtención del título de grado de ingeniero electrónico*

Links de internet:

- [5] <https://cioffi-group.stanford.edu/doc/book/chap3.pdf>

- [6] <https://aholab.ehu.eus/users/inma/psc/tema4.pdf>
- [7] [http://lapsyc.ingelec.uns.edu.ar/Juan/PSC/PSC\\_aux/Part1\\_5.pdf](http://lapsyc.ingelec.uns.edu.ar/Juan/PSC/PSC_aux/Part1_5.pdf)

Libros:

- [8] Edward A. Lee and David G. Messerschmitt “Digital Communication”
- [9] John G. Proakis and Dimitris G. Manolakis “Digital Signal Processing”