

# Análisis e Implementación de un Transmisor Digital con Modulación Variable y Filtrado SRRC Polifásico

Gabriel Garcés  
[garcesgabrielivan@gmail.com](mailto:garcesgabrielivan@gmail.com)

16 de agosto de 2025

## Resumen

Este informe presenta el análisis técnico completo de un transmisor digital implementado en Verilog que soporta múltiples esquemas de modulación (QPSK, 16-QAM, 32-QAM) con filtrado de pulso conformador SRRC (Square Root Raised Cosine) mediante arquitectura polifásica. El sistema integra generadores PRBS-9 duales, mapeadores de constelación configurables y filtros digitales optimizados para maximizar la eficiencia computacional. La arquitectura polifásica de 8 fases reduce el uso de multiplicadores en un 87.5 % comparado con implementaciones directas, manteniendo el rendimiento y la calidad de señal. El diseño demuestra competencia avanzada en procesamiento de señales digitales y optimización de recursos hardware para sistemas FPGA.

# Índice

|  |           |
|--|-----------|
| <b>1. Introducción</b>   | <b>4</b>  |
| 1.1. Objetivos del Proyecto . . . . .                                      | 4         |
| 1.2. Contribuciones Principales . . . . .                                  | 4         |
| <b>2. Simulación del Transmisor en Punto Flotante</b>                      | <b>4</b>  |
| 2.1. Introducción a las Bibliotecas de Simulación . . . . .                | 5         |
| <b>3. Arquitectura del Sistema</b>   | <b>7</b>  |
| 3.1. Diagrama de Bloques General . . . . .                                 | 7         |
| <b>4. Análisis de Módulos</b>  | <b>8</b>  |
| 4.0.1. Generador PRBS-9 ( <code>prbs_x.v</code> ) . . . . .                | 8         |
| 4.0.2. Generador de Control de Timing ( <code>tx_fcsg.v</code> ) . . . . . | 8         |
| 4.0.3. Mapeador de Constelación . . . . .                                  | 9         |
| <b>5. Filtro SRRC Polifásico (<code>tx_srrc_polyphase.v</code>)</b>        | <b>9</b>  |
| 5.1. Análisis del Diseño de Filtro Polifásico SRRC . . . . .               | 9         |
| 5.1.1. Reorganización de Coeficientes . . . . .                            | 11        |
| 5.1.2. Algoritmo de Procesamiento . . . . .                                | 11        |
| 5.1.3. Ventajas Computacionales . . . . .                                  | 12        |
| 5.1.4. Gestión de Precisión Numérica . . . . .                             | 12        |
| <b>6. Simulación en Punto Fijo:</b>  | <b>12</b> |
| 6.1. Formatos de Punto Fijo . . . . .                                      | 13        |
| 6.1.1. Formato Q4.12 (Señal de entrada y salida) . . . . .                 | 13        |
| 6.1.2. Formato Q2.12 (Coeficientes del filtro) . . . . .                   | 13        |
| 6.2. Algoritmo de Convolución . . . . .                                    | 13        |
| 6.3. Implementación 1: . . . . .   | 13        |
| 6.4. Operaciones Clave . . . . .   | 15        |
| 6.5. Implementación 2: . . . . .   | 16        |
| 6.6. Mapeadores de Modulación . . . . .                                    | 18        |
| 6.6.1. Caracterización de Rendimiento: Cálculo de $SNR_Q$ . . . . .        | 21        |
| <b>7. Vector Matching:</b>   | <b>22</b> |
| 7.1. Verificación mediante Vector Matching . . . . .                       | 22        |
| <b>8. Análisis de Timing y Camino Crítico</b>                              | <b>23</b> |
| 8.1. Caracterización del Camino Crítico . . . . .                          | 24        |
| <b>9. Análisis de Timing y Camino Crítico con VIO e ILA instanciados</b>   | <b>24</b> |
| 9.1. Caracterización del Camino Crítico . . . . .                          | 26        |
| <b>10. Reporte de utilización sin VIO e ILA</b>                            | <b>27</b> |
| 10.1. Lógica de Slice . . . . .  | 28        |
| 10.2. Procesadores de Señal Digital (DSP) . . . . .                        | 28        |
| 10.3. Recursos de Memoria y Clock . . . . .                                | 28        |
| 10.3.1. Distribución de Primitivas . . . . .                               | 28        |

|   |           |
|---|-----------|
| <b>11. Reporte de Utilización con VIO e ILA instanciados</b>                      | <b>29</b> |
| 11.1. Lógica de Slice . . . . .   | 29        |
| 11.2. Recursos de Memoria . . . . .   | 29        |
| 11.3. Procesadores de Señal Digital (DSP) . . . . .                               | 29        |
| 11.4. Recursos de Clock y Debug . . . . .   | 29        |
| 11.5. Impacto de los Módulos de Debug . . . . .                                   | 30        |
| <b>12. Implementación y control mediante el uso de las herramientas VIO e ILA</b> | <b>30</b> |
| 12.1. Constelaciones con ILA a 4096 Muestras . . . . .                            | 33        |
| <b>13. Conclusión</b>   | <b>33</b> |

# 1. Introducción

En el contexto actual de las comunicaciones digitales, la eficiencia en el uso de recursos hardware y la flexibilidad de modulación son aspectos críticos para el diseño de transmisores modernos. Este proyecto implementa un transmisor digital completo que aborda estos desafíos mediante una arquitectura polifásica y soporte multi-modulación.

## 1.1. Objetivos del Proyecto

- Implementar un transmisor digital flexible con soporte para QPSK, 16-QAM y 32-QAM
- Optimizar el uso de recursos hardware mediante arquitectura polifásica
- Desarrollar una solución escalable y modular

## 1.2. Contribuciones Principales

- Arquitectura polifásica de 8 fases que reduce multiplicadores en 87.5 %
- Mapeador de constelación unificado para múltiples modulaciones
- Sistema de control de timing centralizado y sincronizado

# 2. Simulación del Transmisor en Punto Flotante

Esta etapa inicial establece los fundamentos teóricos y las especificaciones funcionales del sistema:

- Definir arquitectura conceptual del transmisor
- Establecer especificaciones de rendimiento objetivo
- Diseñar algoritmos de procesamiento de señales

Las ecuaciones fundamentales se establecen en esta etapa:

$$\text{PRBS: } x[n] = x[n - 9] \oplus x[n - 5] \quad (1)$$

$$\text{Filtro SRRRC: } y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n - k] \quad (2)$$

$$\text{Polifásico: } y_m[n] = \sum_{k=0}^{N/M-1} h_{m,k} \cdot x[n - k] \quad (3)$$

## Configuración del Entorno

- Simulador: Python
- Precisión: Punto flotante
- Condiciones: Canal ideal

## 2.1. Introducción a las Bibliotecas de Simulación

Para la implementación y validación del transmisor variable polifásico, se desarrolló un conjunto de bibliotecas en Python que permiten la simulación en punto flotante de todos los esquemas de modulación soportados. Estas bibliotecas constituyen una referencia matemática contra la cual se compara la implementación en hardware.

A partir de estás se logró comprender que una manera eficiente de transmitir las distintas modulaciones sería normalizando las constelaciones. Esto se debe al hecho que vincular las amplitudes máximas de todas las constelaciones (QPSK, 16-QAM, 32-QAM) al coeficiente más alto del filtro SRRC, se está implementando una estrategia de normalización que resuelve varios problemas simultáneamente. Primero, se garantiza que todas las modulaciones utilicen completamente el rango dinámico disponible de los DACs de 16 bits, maximizando la relación señal-ruido de cuantización. Segundo, se establece una referencia común que mantiene la coherencia entre el filtro conformador de pulso y las constelaciones, asegurando que cualquier ajuste en el filtro se propague automáticamente a todas las modulaciones. La característica más interesante de esta implementación es cómo maneja la potencia. Aunque todas las constelaciones tienen la misma amplitud pico, sus potencias medias son diferentes debido a la distribución de sus puntos. QPSK, con solo 4 puntos equidistantes, mantiene una potencia media alta y por tanto una buena eficiencia energética. En contraste, 32-QAM distribuye sus 32 puntos de manera más dispersa, resultando en menor potencia media pero mayor PAPR (relación potencia pico a potencia media). Esto significa que mientras QPSK será más eficiente para el amplificador de potencia, 32-QAM requerirá mayor ofrecerá mayor eficiencia espectral. Esta normalización te permite comparar el rendimiento de diferentes modulaciones bajo condiciones equitativas.

De esta manera, la Figura 1 muestra un ejemplo del efecto de la normalización aplicada a las tres constelaciones. En las gráficas superiores se observan las constelaciones originales con diferentes escalas y distribuciones de amplitud, donde cada modulación presenta rangos distintos en los ejes I y Q. Tras aplicar la normalización vinculada al coeficiente máximo del filtro SRRC (gráficas inferiores), todas las constelaciones quedan confinadas al mismo rango dinámico representativo [-1, 1], manteniendo sus proporciones geométricas originales pero con amplitudes máximas uniformes.

Posteriormente, a través de la función SRRC implementada en código, se generan 64 coeficientes del filtro SRRC (*Square Root Raised Cosine*) al cual se le realizó el diseño polifásico que se pueden observar en las gráficas mostradas. La función calcula matemáticamente la respuesta impulsiva del filtro basándose en los parámetros de diseño: factor de roll-off  $\beta = 0,35$ , valor típico en comunicaciones ópticas, con 64 taps totales.

La Respuesta al Impulso se observa en la Figura 2, en donde en la parte superior se muestra los coeficientes sin escalar, en la parte inferior se presenta los mismos coeficientes escalados y cuantizados a 14 bits usando formato de punto fijo Q2.12, donde el valor máximo alcanza aproximadamente 1600 unidades.

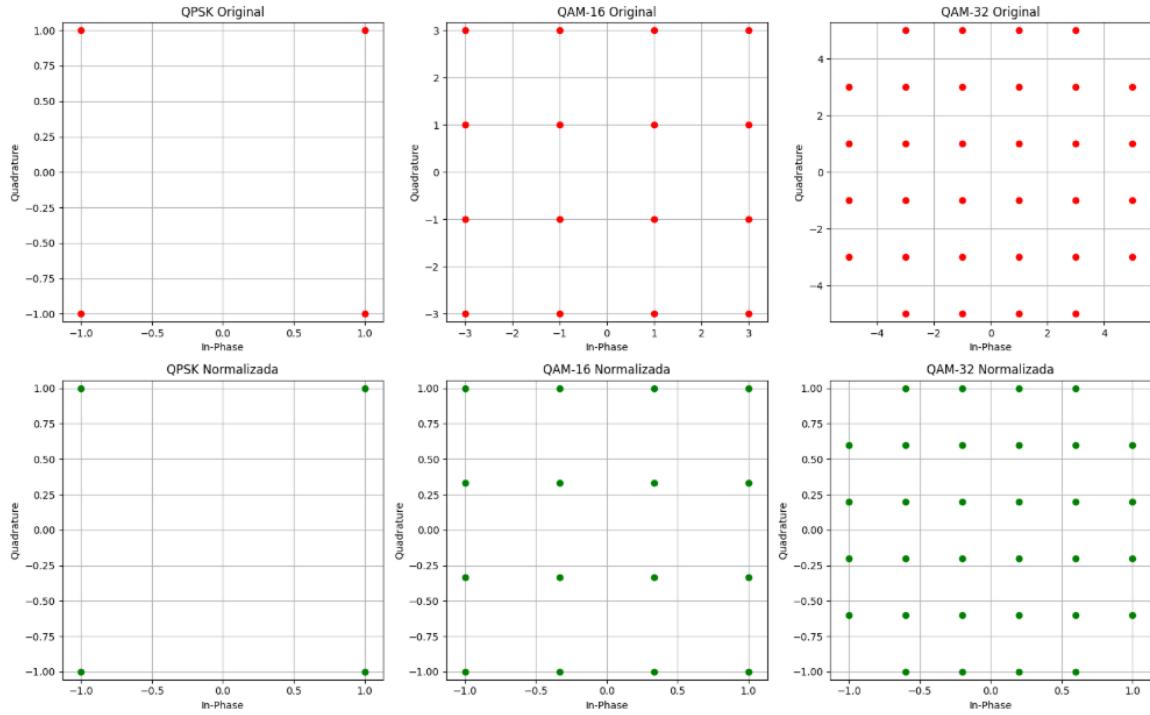


Figura 1: Constelaciones QPSK, 16-QAM y 32-QAM antes (superior) y después (inferior) de la normalización basada en el coeficiente máximo del filtro SRRC.

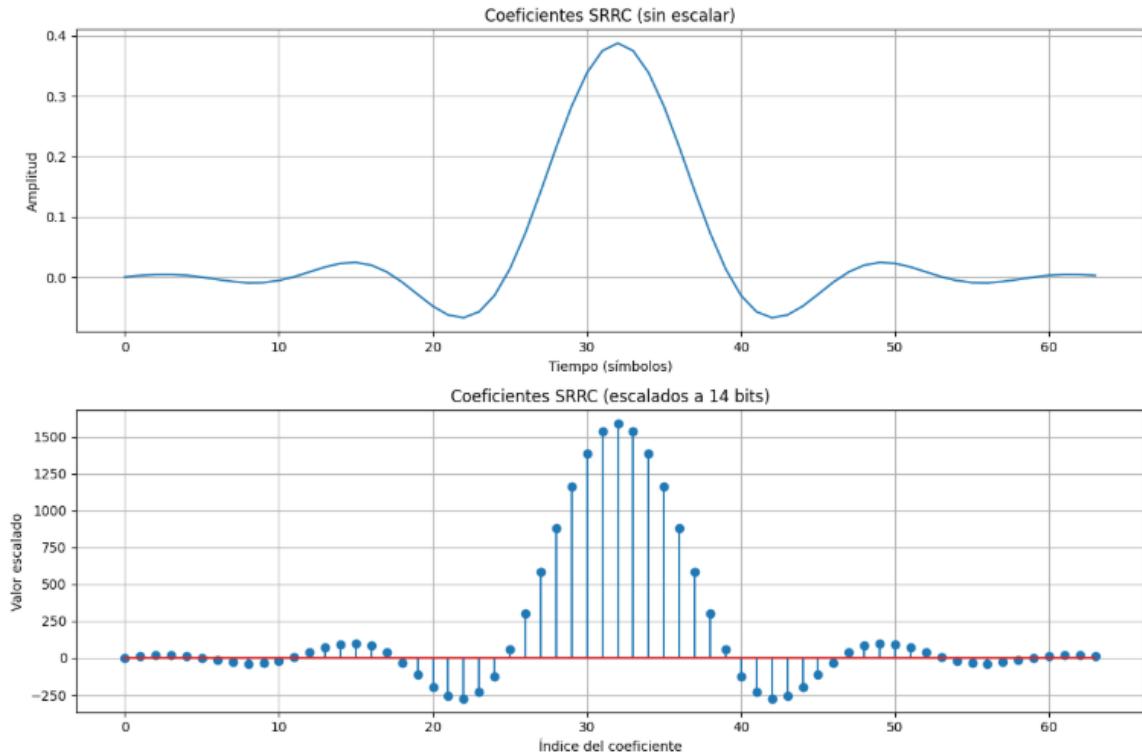


Figura 2: Respuesta al impulso del filtro polifásico.

### 3. Arquitectura del Sistema

#### 3.1. Diagrama de Bloques General

La arquitectura del transmisor se compone de varios módulos interconectados que procesan la señal desde la generación pseudoaleatoria hasta la salida filtrada, como se muestra en la Figura 3.

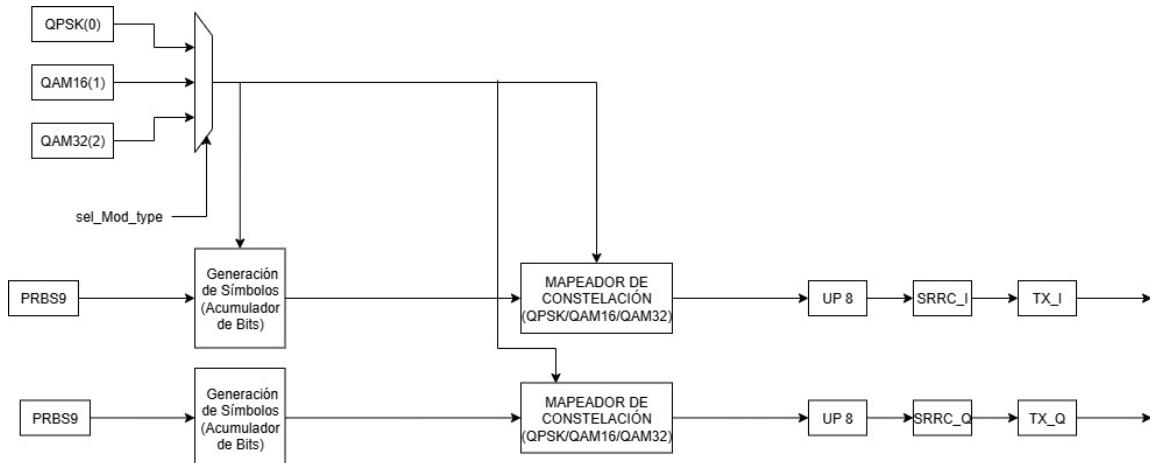


Figura 3: Arquitectura general del transmisor variable polifásico

Este transmisor variable se encuentra conformado por una arquitectura de procesamiento digital en cuadratura que permite la transmisión de señales moduladas con diferentes esquemas de modulación. La entrada del sistema consiste en tres tipos de modulación digital: QPSK, QAM16 y QAM32, las cuales son seleccionadas mediante una señal de control denominada `sel_Mod_type` que actúa como un multiplexor para determinar el esquema de modulación a utilizar. La generación de símbolos se realiza a través de dos bloques idénticos que reciben como entrada una secuencia pseudoaleatoria PRBS9, luego se incluye un acumulador de bits que permite formar los símbolos digitales de acuerdo al tipo de modulación seleccionado. Posteriormente, los símbolos generados son procesados por dos mapeadores de constelación que operan con los esquemas QPSK, QAM16 o QAM32 según la selección realizada, estos mapeadores se encargan de convertir los símbolos digitales en puntos específicos de la constelación correspondiente, generando las componentes en fase (I) y en cuadratura (Q) de la señal. El conformado de pulso se implementa mediante filtros polifásicos que combinan el sobremuestreo UP 8 con filtros de coseno alzado de raíz cuadrada (SRRC), esta implementación polifásica permite realizar simultáneamente el incremento de la tasa de muestreo por un factor de ocho y el filtrado de conformado espectral, optimizando las características de transmisión y minimizando la interferencia intersimbólica de manera eficiente. Finalmente, las salidas del sistema corresponden a las señales `TX_I` y `TX_Q`, las cuales representan las componentes en cuadratura de la señal transmitida.

## 4. Análisis de Módulos

### 4.0.1. Generador PRBS-9 (prbs\_x.v)

Los generadores de secuencias pseudoaleatorias binarias (PRBS) se utilizan en sistemas de comunicación para generar patrones de prueba y datos de transmisión. Un PRBS-9 genera una secuencia de longitud  $2^9 - 1 = 511$  bits antes de repetirse.

El generador utiliza un registro de desplazamiento con retroalimentación lineal (LFSR) según el polinomio:

$$P(x) = x^9 + x^5 + 1 \quad (4)$$

```

1      always @ (posedge clock) begin
2          if (in_reset)
3              data <= PRBS_SEED;
4          else if (in_enable)
5              data <= {data[NB_PRBS-2 -: NB_PRBS-1] ,
6                      data[PRBS_HIGH_ORDER-1] ^ data[
7                          PRBS_LOW_ORDER-1]};
8          else
9              data <= data;
10         end

```

Para garantizar decorrelación entre los canales I y Q, se implementan dos generadores con configuraciones diferentes:

- **Canal I:** Semilla = 9'h1AA, taps en posiciones 9 y 5
- **Canal Q:** Semilla = 9'h1FE, taps en posiciones 9 y 5

### 4.0.2. Generador de Control de Timing (tx\_fcsg.v)

Este módulo genera las señales de sincronización temporal críticas para el sistema:

- **ctrl\_valid\_8MHz:** Control de frecuencia de muestreo de salida
- **ctrl\_valid\_1oTMHz:** Control de frecuencia de símbolo

Luego, la implementación del Contador se calcula como:

$$\text{limit\_counter} = 2^{\text{LOG2\_1oTMHZ\_QPSK}} - 1 = 31 \quad (5)$$

```

1      always@ (posedge clock) begin
2          if (in_reset) begin
3              out_ctrl_valid_8MHz <= 1'b0;
4              out_ctrl_valid_1oTMHz <= 1'b0;
5          end
6          else begin
7              out_ctrl_valid_8MHz <= (period_counter[LOG2_8MHz
8                  -1:0] == 0) ? 1'b1 : 1'b0;
9              out_ctrl_valid_1oTMHz <= (period_counter[
10                 LOG2_1oTMHZ_QPSK-1:0] == 0) ? 1'b1 : 1'b0;

```

```

9      end
10     end

```

#### 4.0.3. Mapeador de Constelación

El mapeador convierte secuencias de bits en puntos de constelación compleja. Para cada modulación:

**QPSK** Utiliza 1 bit por componente:

$$\text{Niveles} = \begin{cases} +H_{\text{MAX}} & \text{si bit} = 1 \\ -H_{\text{MAX}} & \text{si bit} = 0 \end{cases} \quad (6)$$

**16-QAM** Utiliza 2 bits por componente:

$$\text{Niveles} = \left\{ -\frac{3H_{\text{MAX}}}{3}, -\frac{H_{\text{MAX}}}{3}, +\frac{H_{\text{MAX}}}{3}, +\frac{3H_{\text{MAX}}}{3} \right\} \quad (7)$$

**32-QAM** Implementa constelación rectangular:

$$\text{Niveles I} = \left\{ -\frac{3H_{\text{MAX}}}{5}, -\frac{H_{\text{MAX}}}{5}, +\frac{H_{\text{MAX}}}{5}, +\frac{3H_{\text{MAX}}}{5} \right\} \quad (8)$$

$$\text{Niveles Q} = \left\{ -H_{\text{MAX}}, -\frac{3H_{\text{MAX}}}{5}, -\frac{H_{\text{MAX}}}{5}, +\frac{H_{\text{MAX}}}{5}, +\frac{3H_{\text{MAX}}}{5}, +H_{\text{MAX}} \right\} \quad (9)$$

Como se observa, todas las constelaciones están normalizadas respecto a:

$$H_{\text{MAX}} = 2^{15} - 1 = 32767 \text{ (formato Q4.12)} \quad (10)$$

## 5. Filtro SRRC Polifásico (tx\_srrc\_polyphase.v)

Un filtro polifásico descompone un filtro FIR de  $N$  coeficientes en  $M$  subfiltros de  $N/M$  coeficientes cada uno. Para nuestro caso:

$$N = 64 \text{ coeficientes totales} \quad (11)$$

$$M = 8 \text{ fases} \quad (12)$$

$$N/M = 8 \text{ coeficientes por fase} \quad (13)$$

### 5.1. Análisis del Diseño de Filtro Polifásico SRRC

El diseño implementado corresponde a un filtro polifásico SRRC (*Square Root Raised Cosine*) para interpolación de símbolos, organizado en una arquitectura de dos dominios claramente diferenciados como se muestra en la Figura 4. La estructura superior representa el **dominio lento (Slow Domain)**, donde opera la lógica de símbolos a la frecuencia de entrada, mientras que la sección inferior constituye el **dominio rápido (Fast Domain)**, que procesa las muestras interpoladas a una frecuencia 8 veces mayor.

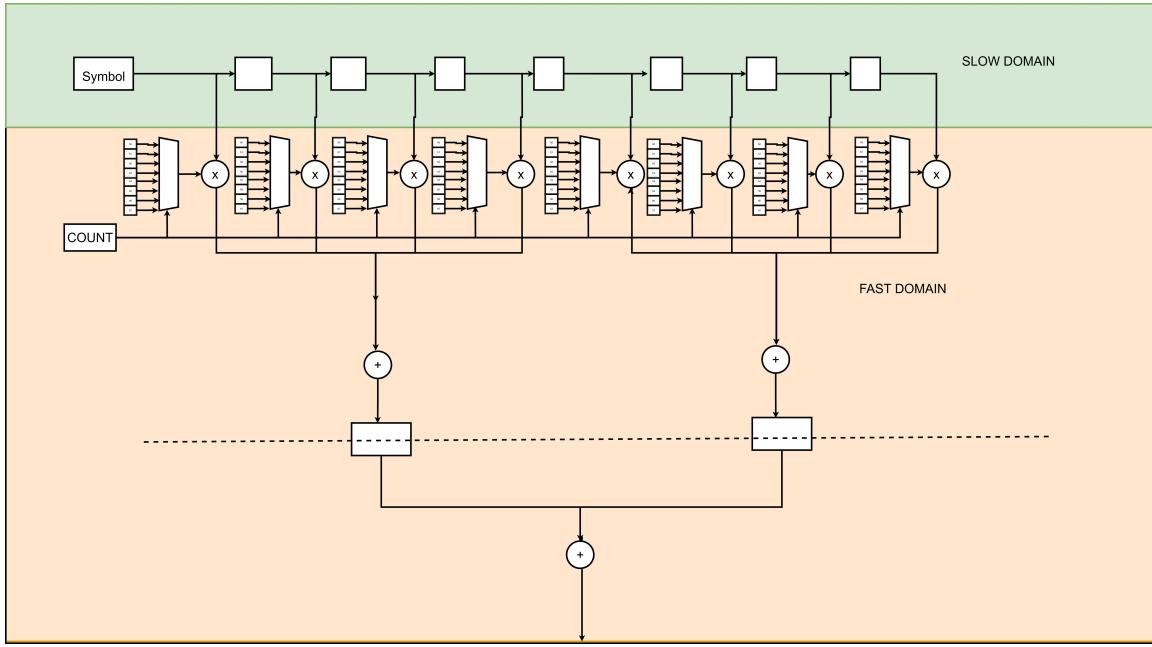


Figura 4: Arquitectura del filtro polifásico SRRRC implementado, mostrando la separación entre dominio lento y dominio rápido.

En el dominio lento, cada símbolo de entrada se almacena en un registro de desplazamiento compuesto por múltiples etapas, representadas por los bloques blancos en la parte superior del diagrama. Este registro mantiene un historial de 8 símbolos, implementando la línea de retardo necesaria para el cálculo de la convolución. El contador (**COUNT**) controla la selección de fase, alternando entre las 8 fases disponibles (0 a 7) en cada ciclo del dominio rápido. Cada símbolo nuevo ingresa en cada tiempo de símbolo que es 8 veces mayor respecto al tiempo de contador que actualiza los coeficientes de cada fase a multiplicar.

El procesamiento en el dominio rápido se ejecuta mediante 8 unidades de cálculo en paralelo, cada una correspondiente a una fase específica del filtro. Cada unidad se la puede imaginar que contiene un multiplexor que selecciona los coeficientes apropiados para la fase actual, seguido de un multiplicador (representado por 'x' en el diagrama). Los coeficientes están organizados en una matriz bidimensional `coef_mem[fase][coeficiente]`, donde cada fase contiene 8 coeficientes específicos que han sido extraídos del filtro SRRRC original de 64 coeficientes. Esto se puede observar en la Figura 5

La arquitectura de acumulación implementa una etapa de *pipeline* para optimizar el *timing* y permitir el funcionamiento a altas frecuencias. Para ello los 8 productos se dividen en dos grupos de 4, realizando sumas parciales en paralelo (`phase_accum_pipe1` y `phase_accum_pipe2`). Posteriormente, se combinan ambas sumas parciales para obtener la acumulación total, que a su vez se le aplicará la saturación para generar la salida final.

Este enfoque polifásico permite que el sistema procese símbolos a baja frecuencia (dominio lento) mientras genera muestras interpoladas a alta frecuencia (dominio rápido), logrando una interpolación por factor de 8. La separación de dominios, claramente visible en el diagrama, facilita el análisis de *timing* y permite que cada sección opere a su frecuencia óptima, maximizando la eficiencia del diseño en términos de recursos de hardware y consumo de potencia.

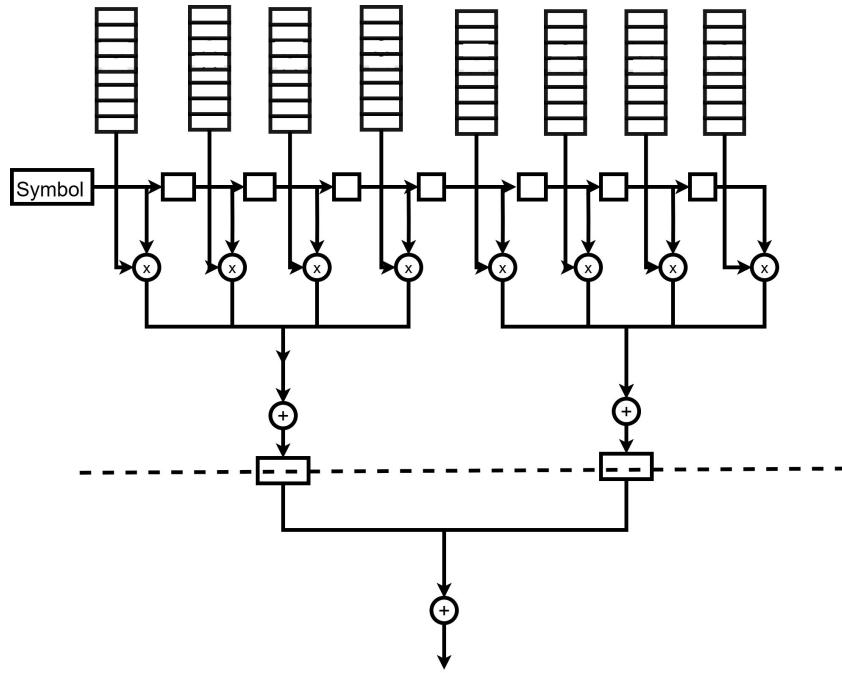


Figura 5: Estructura matricial del filtro polifásico donde se denota la organización de coeficientes en una matriz bidimensional `coef_mem[fase][coeficiente]`. Las columnas verticales representan las 8 fases del filtro, cada una conteniendo 64 coeficientes. El registro de desplazamiento horizontal almacena los símbolos de entrada, mientras que la estructura de pipeline inferior realiza las operaciones de multiplicación y acumulación en etapas separadas.

### 5.1.1. Reorganización de Coeficientes

Los coeficientes se reorganizan según:

$$h_m[k] = h[k \cdot M + m], \quad m = 0, 1, \dots, M - 1 \quad (14)$$

donde  $h_m[k]$  es el  $k$ -ésimo coeficiente de la  $m$ -ésima fase.

### 5.1.2. Algoritmo de Procesamiento

Para cada muestra de salida:

1. Seleccionar fase actual  $m$
2. Calcular productos:  $p_k = x[k] \cdot h_m[k]$  para  $k = 0, 1, \dots, 7$
3. Acumular:  $y[n] = \sum_{k=0}^7 p_k$
4. Incrementar contador de fase:  $m = (m + 1) \bmod 8$

### 5.1.3. Ventajas Computacionales

Cuadro 1: Comparación de recursos: implementación directa vs polifásica

| Implementación | Multiplicadores | Ahorro | Frecuencia           | Throughput |
|----------------|-----------------|--------|----------------------|------------|
| Directa        | 128             | -      | $f_{\text{símbolo}}$ | $f_s$      |
| Polifásica     | 16              | 87.5 % | $f_{\text{salida}}$  | $f_s$      |

### 5.1.4. Gestión de Precisión Numérica

#### Formatos Utilizados:

- Coeficientes: Q2.12 (14 bits)
- Entrada: Q4.12 (16 bits)
- Producto: Q6.24 (30 bits)
- Acumulador: Q12.24 (36 bits)
- Salida: Q4.12 (16 bits)

#### Saturación:

```

1 assign saturated_val = ( ~|phase_accum[NB_ACCUM-1
2   -: NB_SAT+1] ||
3     &phase_accum[NB_ACCUM-1 -: NB_SAT+1] ) ?
4   // No saturación
5   phase_accum[28:12] :
6   // Saturación
7   (phase_accum[NB_ACCUM-1]) ?
8   {1'b1,{NB_OUT_SRRC-1{1'b0}}} :
9   {1'b0,{NB_OUT_SRRC-1{1'b1}}};

```

## 6. Simulación en Punto Fijo:

Para la verificación del funcionamiento del filtro polifásico se diseña un filtro FIR convencional que utiliza aritmética de punto fijo para el procesamiento digital de señales, empleando formato Q4.12 para la señal de entrada y Q2.12 para los coeficientes. El mismo se diseña, por un lado, en una primera implementación mediante aritmética entera pura y, por otro lado, en una segunda implementación utilizando la biblioteca de Python denominada "fxpMath".

## 6.1. Formatos de Punto Fijo

### 6.1.1. Formato Q4.12 (Señal de entrada y salida)

$$\text{Valor} = \frac{\text{Entero signado de 16 bits}}{2^{12}} \quad (15)$$

$$\text{Rango} = [-8,0, 7,999755859375] \quad (16)$$

$$\text{Resolución} = 2^{-12} \approx 0,000244140625 \quad (17)$$

### 6.1.2. Formato Q2.12 (Coeficientes del filtro)

$$\text{Valor} = \frac{\text{Entero signado de 14 bits}}{2^{12}} \quad (18)$$

$$\text{Rango} = [-2,0, 1,999755859375] \quad (19)$$

$$\text{Resolución} = 2^{-12} \approx 0,000244140625 \quad (20)$$

## 6.2. Algoritmo de Convolución

La función implementa la convolución discreta mediante la ecuación fundamental, en donde para su correcto uso en este caso, se necesita que los símbolos de entrada posean un previo Up-Sampling igual a 8 dada la naturaleza de equivalencia en comportamiento entre un filtro FIR y un filtro polifásico de 8 fases.

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n - k] \quad (21)$$

donde:

- $y[n]$  es la señal de salida
- $h[k]$  son los coeficientes del filtro
- $x[n]$  es la señal de entrada
- $N$  es el número de coeficientes (taps)

## 6.3. Implementación 1:

La siguiente implementación realiza el filtrado FIR utilizando operaciones con enteros, la señal de entrada y los coeficientes se encuentran en formato de punto fijo Q4.12 y Q2.12 respectivamente. Esta implementación se realizó con el objetivo de validar tanto la implementación del RTL como el diseño realizado con la librería fxpMath, que se mostrará mas adelante.

```

1      def fixed_point_fir_filter(signal):
2          """
3              Filtro FIR completamente en enteros.
4              Ambas entradas ya están en formato de punto fijo.
5

```

```

6     Args:
7         signal_q412: Señal de entrada (array de enteros
8             signados en formato Q4.12)
9         coeffs_q212: Coeficientes del filtro en formato Q2
10            .12 (enteros signados)
11
12     Returns:
13         Array con la señal filtrada en formato de enteros
14             signados Q4.12
15     """
16
17     signal_q412 = signal #upsampled_signal
18     taps = len(coeffs_q212)
19
20     print(f"Procesando {len(signal_q412)} símbolos con
21           {taps} coeficientes...")
22     print("Modo: Aritmetica entera pura Q4.12 × Q2.12")
23
24     # Padding con ceros (0 en cualquier formato Q es
25     # simplemente 0)
26     # signal_padded = [0] * (taps-1) + signal_q412 +
27     # [0] * (taps-1)
28     signal_padded = [0] * (taps-1) + list(signal_q412)
29     + [0] * (taps-1)
30
31     # Array de salida
32     full_output = []
33
34     # Invertir coeficientes una sola vez para la
35     # convolución
36     coeffs_reversed = coeffs_q212[::-1]
37
38     for i in range(len(signal_padded) - taps + 1):
39         if i % 1000 == 0:
40             print(f"  Procesando símbolo {i}/{len(signal_padded)
41                   } - taps + 1}...")
42
43         # Acumulador de 64 bits para evitar overflow
44         accumulator = 0
45
46         # Convolución: cada multiplicación es Q4.12 × Q2.12
47         # = Q6.24
48         for j in range(taps):
49             product = signal_padded[i + j] * coeffs_reversed[j]
50             accumulator += product
51
52             result_q412 = int(accumulator) >> 12 # de Q6.24 a
53             Q4.12 manteniendo signo
54
55             # Saturación a los límites de Q4.12: [-32768 ,
56             32767]
```

```

45     if result_q412 > 32767:
46         result_q412 = 32767
47     elif result_q412 < -32768:
48         result_q412 = -32768
49
50     full_output.append(result_q412)
51
52     # Aplicar offset para compensar el retardo del
53     # filtro
54     offset = taps // 2
55     vector_output = full_output[offset:offset+len(
56         signal_q412)]
57
58     print(f"Filtrado completado. Salida: {len(
59         vector_output)} símbolos")
60
61     return vector_output

```

Listing 1: Implementación del filtro FIR con aritmética entera pura

La implementación utiliza los siguientes formatos:

- **Señal de entrada:** Q4.12 (4 bits enteros, 12 bits fraccionarios)
- **Coeficientes:** Q2.12 (2 bits enteros, 12 bits fraccionarios)
- **Acumulador interno:** Q6.24 (6 bits enteros, 24 bits fraccionarios)
- **Señal de salida:** Q4.12 (4 bits enteros, 12 bits fraccionarios)

#### 6.4. Operaciones Clave

1. **Multiplicación:**  $Q4.12 \times Q2.12 = Q6.24$
2. **Escalado:** Desplazamiento de 12 bits para conversión  $Q6.24 \rightarrow Q4.12$
3. **Saturación:** Limitación a rango  $[-32768, 32767]$  para Q4.12
4. **Compensación de retardo:** Offset de  $taps//2$  muestras

Luego, el padding resuelve el problema de bordes en la convolución discreta. Para calcular:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k] \quad (22)$$

se requieren  $N$  muestras consecutivas. Sin padding, las primeras  $(N - 1)$  muestras no tendrían datos suficientes para la convolución completa.

Por otra parte, los filtros FIR introducen un retardo de grupo constante:

$$\tau_{\text{grupo}} = \frac{N - 1}{2} \text{ muestras} \quad (23)$$

Este retardo surge porque cada muestra de salida depende de  $N$  muestras de entrada, con la contribución central ocurriendo en el coeficiente medio.

Por este motivo, la compensación alinea temporalmente la salida con la entrada, descartando las primeras  $\lfloor N/2 \rfloor$  muestras que contienen respuesta transitoria del padding inicial.

### Longitud final:

$$L_{\text{salida final}} = M + (N - 1) - \left\lfloor \frac{N}{2} \right\rfloor - \left\lfloor \frac{N}{2} \right\rfloor = M \quad (24)$$

## 6.5. Implementación 2:

El código siguiente muestra la implementación mediante el uso de 'fpxMath':

```

1      def fixed_point_fir_filter(signal, coeffs):
2          """
3              Implementa un filtro FIR tap por tap en punto fijo.
4              Args:
5                  signal: Señal de entrada (array de valores
6                      flotantes)
7                  coeffs: Coeficientes del filtro (array de valores
8                      flotantes)
9              Returns:
10                 Array con la señal filtrada en formato de enteros
11                     signados
12
13
14                 Q_NUM_COEF      = 64
15                 Q_NB_COEF       = 2
16                 Q_NBF_COEF      = 12
17
18                 Q_NB_IN_SRRC    = 4
19                 Q_NBF_IN_SRRC   = 12
20                 Q_NB_IN_ACCUM   = 6
21                 Q_NBF_IN_ACCUM  = 24
22                 Q_NB_OUT_SRRC   = 4
23                 Q_NBF_OUT_SRRC  = 12
24
25                 # Convertir la señal de entrada a punto fijo (Q4
26                     .12)
27                 signal_fixed = [FixedPoint(sample, signed=True, m=
28                     Q_NB_IN_SRRC, n=Q_NBF_IN_SRRC) for sample in
29                     signal]
30
31                 # Convertir los coeficientes a punto fijo (Q2.12)
32                 coeffs_fixed = [FixedPoint(coeff, signed=True, m=
33                     Q_NB_COEF, n=Q_NBF_COEF) for coeff in coeffs]
34
35                 # Crear padding con ceros en punto fijo (una sola
36                     vez)

```

```

32         zero_pad = FixedPoint(0.0, signed=True, m=
33                                     Q_NB_IN_SRRC, n=Q_NBF_IN_SRRC)
34
35         # Aplicar padding al inicio y al final
36         signal_padded_fixed = [zero_pad] * (taps-1) +
37                                     signal_fixed + [zero_pad] * (taps-1)
38
39         print("Aplicando convolución en punto fijo...")
40
41         # Inicializar array de salida con el tamaño
42         # adecuado
43         full_output = []
44
45         # Invertir coeficientes
46         coeffs_fixed_reversed = coeffs_fixed[::-1]
47
48         for i in range(len(signal_padded_fixed) - taps + 1):
49             :
50             if i % 1000 == 0:
51                 print(f"    Procesando muestra {i}/{len(
52                     signal_padded_fixed) - taps + 1}...")
53
54         # Extraer la ventana actual
55         window = signal_padded_fixed[i:i+taps]
56
57         # Crear acumulador con precisión extendida para
58         # evitar desbordamientos
59         sum_value = FixedPoint(0.0, signed=True, m=
60                                     Q_NB_IN_ACCUM, n=Q_NBF_IN_ACCUM)
61
62
63         for j in range(taps):
64             # Producto en punto fijo
65             product = window[j] * coeffs_fixed_reversed[j]
66
67             # Acumular en punto fijo
68             sum_value += product
69
70             # Convertir resultado al formato de salida deseado
71             # (Q12.12)
72             output_sample = FixedPoint(float(sum_value), signed
73                                         =True, m=Q_NB_OUT_SRRC, n=Q_NBF_OUT_SRRC)
74
75             # Convertir a entero signado escalando por 2^NBF y
76             # redondeando
77             output_integer = int(floor(float(output_sample) *
78                               (2**Q_NBF_OUT_SRRC)))
79             if output_integer > 0:
80                 output_integer = output_integer - 1
81             elif output_integer < 0:
82                 output_integer = output_integer

```

```

72     print("Simbolo", j, " :", output_integer, "\n")
73
74     # Guardar el valor entero en lugar del FixedPoint
75     full_output.append(output_integer)
76
77     # Aplicar el mismo offset que en el método polifásico
78     offset = len(coeffs) // 2
79     vector_output = full_output[offset:offset+len(
80         signal)]
81
82     # La función ahora devuelve los enteros signados
83     return vector_output

```

Listing 2: Implementación en punto fijo

## 6.6. Mapeadores de Modulación

Los mapeadores para convertir los símbolos generados por el generador de símbolos en niveles de amplitud apropiados para cada esquema de modulación. El sistema implementa tres tipos de mapeadores distintos, cada uno de ellos introducirá los símbolos necesarios al filtro polifásico a partir del cuál se realizará el vector mathtt{mathing}.

El primer mapeador implementado corresponde a una modulación PAM adaptada, diseñada para procesamiento binario simple. Este mapeador opera sobre bits individuales y realiza una conversión directa donde el bit '1' se mapea al valor máximo positivo (32767) y el bit '0' al valor máximo negativo (-32767), generando efectivamente una modulación PAM-2 equivalente a BPSK en banda base.

```

1      def ModulationMapperAdapted(input_file: str,
2                                     output_file: str):
3          """
4              Lee bits desde un archivo de texto, realiza mapeo
5                  PAM (1 -> 32767, 0 -> -32767)
6              y guarda el resultado en otro archivo de texto.
7
8          Parámetros:
9          - input_file: Ruta del archivo de entrada con los
10             bits.
11          - output_file: Ruta del archivo de salida para
12             guardar los valores mapeados.
13
14          # Leer los bits desde el archivo
15          with open(input_file, "r") as f:
16              bit_lines = f.readlines()
17
18              # Convertir a enteros y eliminar saltos de línea
19              bits = [int(line.strip()) for line in bit_lines]
20
21              # Mapeo PAM: 1 -> 32767, 0 -> -32767
22              pam_mapeado = [32767 if bit == 1 else -32767 for
23                             bit in bits]

```

```

19
20     # Guardar el resultado en un nuevo archivo
21     with open(output_file, "w") as f:
22         for val in pam_mapeado:
23             f.write(f"{val}\n")
24
25         print(f"Mapeo PAM completado y guardado en '{output_file}'")
26
27     return pam_mapeado

```

Listing 3: Mapeador PAM Adaptado

El segundo mapeador corresponde a una implementación QAM-16 que procesa símbolos de 2 bits para generar las componentes en fase (I) y cuadratura (Q) de la constelación. Este mapeador utiliza una tabla de lookup directa donde cada combinación de 2 bits se mapea a uno de cuatro niveles de amplitud posibles (PAM-4).

```

1      import numpy as np
2
3      # Ruta al archivo de entrada
4      input_path = "phase_buffer.txt"
5
6
7      # Leer valores enteros (0, 1, 2, 3)
8      with open(input_path, 'r') as f:
9          phase_buffer_vals = np.array([int(line.strip()) for
10              line in f], dtype=np.uint8)
11
12      # Definición de niveles QAM16 (enteros)
13      H_MAX = 32767
14      QAM16_P1 = H_MAX // 3           # 10922
15      QAM16_P3 = QAM16_P1 * 3        # 32766
16
17      # Crear tabla de mapeo
18      mapper_table = np.array([
19          -QAM16_P3,    # 0b00 -> -P3
20          -QAM16_P1,   # 0b01 -> -P1
21          QAM16_P3,    # 0b10 -> +P3
22          QAM16_P1     # 0b11 -> +P1
23      ], dtype=np.int32)
24
25      # Aplicar mapeo
26      mapped_i = mapper_table[phase_buffer_vals]
27
28      # Guardar resultado
29      output_path = "mapper_qam16_i.txt"
30      np.savetxt(output_path, mapped_i, fmt='%d')
31
32      print(f"Mapeo QAM16 (I) completado y guardado en '{output_path}'")

```

Listing 4: Mapeador QAM-16

El tercer mapeador implementa una constelación QAM-32 compleja, procesando símbolos de 5 bits que se obtienen de la concatenación de 3 bits de cuadratura y 2 bits de fase. Este mapeador lee dos archivos separados conteniendo los valores de fase (0-3) y cuadratura (0-7), los combina mediante operaciones de desplazamiento y OR lógico para generar 32 símbolos posibles, y posteriormente los mapea a una constelación rectangular QAM-32 con tres niveles de amplitud por componente.

```

1      import numpy as np
2
3          # Rutas de entrada
4      phase_path = "phase_buffer.txt"
5      quad_path  = "quad_buffer.txt"
6
7          # Rutas de salida
8      output_i_path = "mapper_qam32_i.txt"
9      output_q_path = "mapper_qam32_q.txt"
10
11         # Leer archivos como arrays de enteros
12     with open(phase_path, 'r') as f:
13         phase_vals = np.array([int(line.strip()) for line
14             in f], dtype=np.uint8)
15
16     with open(quad_path, 'r') as f:
17         quad_vals = np.array([int(line.strip()) for line in
18             f], dtype=np.uint8)
19
20         # Concatenar: {quad, phase} = (quad << 2) | phase
21         symbols = (quad_vals << 2) | phase_vals # Valores
22             entre 0 y 31
23
24         # Constantes enteras QAM32
25         H_MAX = 32767
26         QAM32_P1 = H_MAX // 5           # 6553
27         QAM32_P3 = QAM32_P1 * 3        # 19659
28         QAM32_P5 = QAM32_P1 * 5        # 32765
29
30         # Tabla de mapeo (32 combinaciones posibles)
31         qam32_map = np.zeros((32, 2), dtype=np.int32)
32
33         qam32_map[ 0] = [-QAM32_P3 ,   QAM32_P5]
34         qam32_map[ 1] = [-QAM32_P1 ,   QAM32_P5]
35         qam32_map[ 2] = [ QAM32_P1 ,   QAM32_P5]
36         qam32_map[ 3] = [ QAM32_P3 ,   QAM32_P5]
37
38         qam32_map[ 4] = [-QAM32_P5 ,   QAM32_P3]
39         qam32_map[ 5] = [-QAM32_P3 ,   QAM32_P3]
40         qam32_map[ 6] = [-QAM32_P1 ,   QAM32_P3]
41         qam32_map[ 7] = [ QAM32_P1 ,   QAM32_P3]
42         qam32_map[ 8] = [ QAM32_P3 ,   QAM32_P3]
43         qam32_map[ 9] = [ QAM32_P5 ,   QAM32_P3]
44
45         qam32_map[10] = [-QAM32_P5 ,   QAM32_P1]
```

```

43     qam32_map[11] = [-QAM32_P3, QAM32_P1]
44     qam32_map[12] = [-QAM32_P1, QAM32_P1]
45     qam32_map[13] = [ QAM32_P1, QAM32_P1]
46     qam32_map[14] = [ QAM32_P3, QAM32_P1]
47     qam32_map[15] = [ QAM32_P5, QAM32_P1]
48
49     qam32_map[16] = [-QAM32_P5, -QAM32_P1]
50     qam32_map[17] = [-QAM32_P3, -QAM32_P1]
51     qam32_map[18] = [-QAM32_P1, -QAM32_P1]
52     qam32_map[19] = [ QAM32_P1, -QAM32_P1]
53     qam32_map[20] = [ QAM32_P3, -QAM32_P1]
54     qam32_map[21] = [ QAM32_P5, -QAM32_P1]
55
56     qam32_map[22] = [-QAM32_P5, -QAM32_P3]
57     qam32_map[23] = [-QAM32_P3, -QAM32_P3]
58     qam32_map[24] = [-QAM32_P1, -QAM32_P3]
59     qam32_map[25] = [ QAM32_P1, -QAM32_P3]
60     qam32_map[26] = [ QAM32_P3, -QAM32_P3]
61     qam32_map[27] = [ QAM32_P5, -QAM32_P3]
62
63     qam32_map[28] = [-QAM32_P3, -QAM32_P5]
64     qam32_map[29] = [-QAM32_P1, -QAM32_P5]
65     qam32_map[30] = [ QAM32_P1, -QAM32_P5]
66     qam32_map[31] = [ QAM32_P3, -QAM32_P5]
67
68 # Aplicar mapeo
69 mapped_i = qam32_map[symbols, 0]
70 mapped_q = qam32_map[symbols, 1]
71
72 # Guardar resultados
73 np.savetxt(output_i_path, mapped_i, fmt='%.d')
74 np.savetxt(output_q_path, mapped_q, fmt='%.d')
75
76 print(f"{'Mapeo QAM32 completado:\n- I: '}{output_i_path}{'\n- Q: '}{output_q_path}'")

```

Listing 5: Mapeador QAM-32

### 6.6.1. Caracterización de Rendimiento: Cálculo de $SNR_Q$

Esta etapa cuantifica el rendimiento del sistema implementado calculando la relación señal-ruido de cuantización:

**Signal-to-Noise Ratio (SNR):**

$$SNR = 10 \log_{10} \left( \frac{P_{señal}}{P_{ruido}} \right) \text{ dB} \quad (25)$$

La Figura 6 presenta los resultados del análisis del  $SNR$  en función del número de bits fraccionarios para ambos canales I y Q. Se observa que la SNR crece de manera casi lineal, con una pendiente cercana a 6 dB por bit fraccionario adicional, hasta aproximadamente los 16 bits. A partir de ese punto, la SNR comienza a saturarse, alcanzando un

valor máximo cercano a los 86 dB. Este comportamiento indica que, para este sistema, configuraciones superiores a 16 bits fraccionarios no aportan mejoras significativas en el rendimiento, estableciendo así un punto de referencia para un diseño eficiente en cuanto a complejidad y precisión. No obstante, también se observa que a partir de los 12 bits fraccionarios ya se alcanza un SNR superior a 65 dB, valor que puede considerarse más que suficiente. Por tanto, utilizar 12 bits permite alcanzar una buena relación de compromiso entre rendimiento y consumo de recursos.

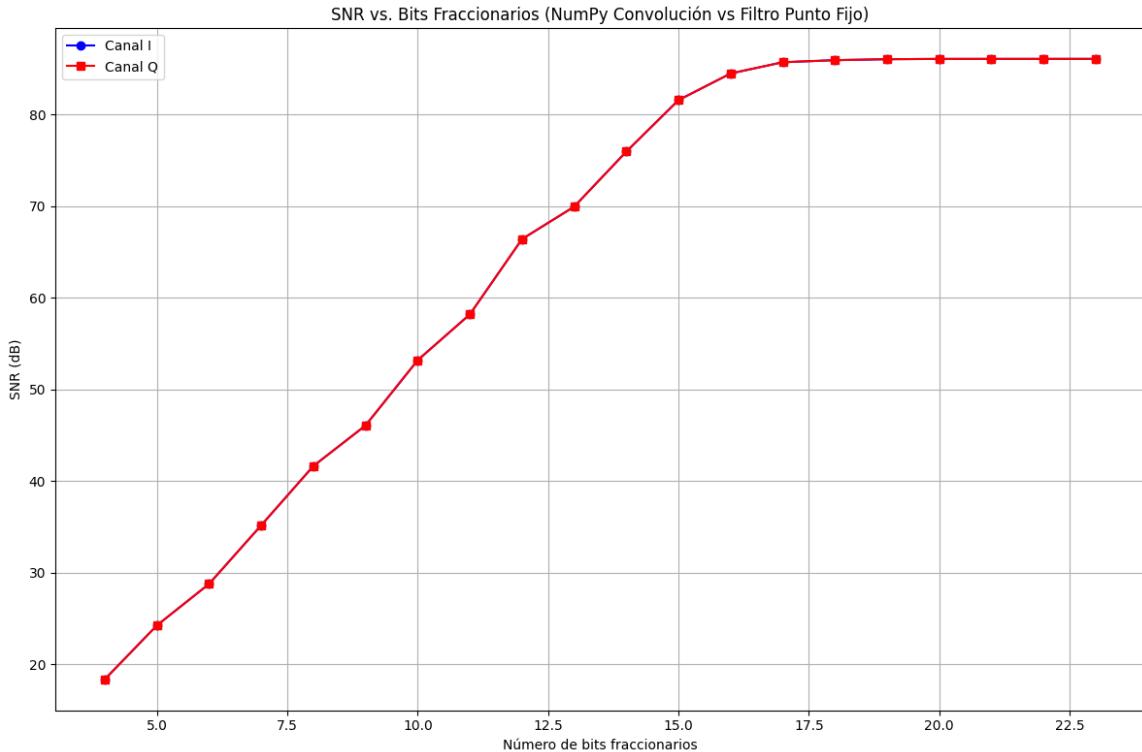


Figura 6: Relación señal-ruido de cuantización ( $SNR_Q$ ) en función del número de bits fraccionarios para la comparación entre implementaciones de punto fijo y punto flotante utilizando secuencia PRBS.

## 7. Vector Matching:

### 7.1. Verificación mediante Vector Matching

Para validar la correcta implementación de los diferentes bloques del sistema, se llevó a cabo el vector matching de cada uno con su modelado en punto fijo. Esta metodología permitió comparar bit a bit las salidas de los módulos implementados en hardware.

El proceso de vector matching se aplicó al mapeador de símbolos, a la respuesta del filtro polifásico y el sistema transmisor completo. Para cada uno de estos bloques, se generaron vectores de prueba representativos que cubrieran el rango completo de operación esperado.

En el caso del mapeador, se verificó la correcta asignación de bits de entrada a símbolos complejos de la constelación. Para el filtro polifásico, se validó que la respuesta impulsional implementada en aritmética de punto fijo mantuviera la fidelidad requerida respecto a su contraparte FIR en punto fijo.

La verificación final se realizó sobre el sistema transmisor completo, donde se implementó en principio una operación de correlación cruzada entre las señales de salida del transmisor implementado y las generadas por el filtro FIR de referencia en punto fijo. La Figura ?? presentada muestra los resultados de esta correlación normalizada, donde se observa claramente un pico de correlación unitario con un determinado *delay* y, subpicos equiespaciados dada la naturaleza periódica de la secuencia pseudoaleatoria.

Se arroja como resultado *vector matching* del 100 % en cada uno de los casos, lo que confirma la correcta implementación funcional del sistema transmisor y valida la precisión de la aritmética de punto fijo utilizada en todos los bloques componentes.

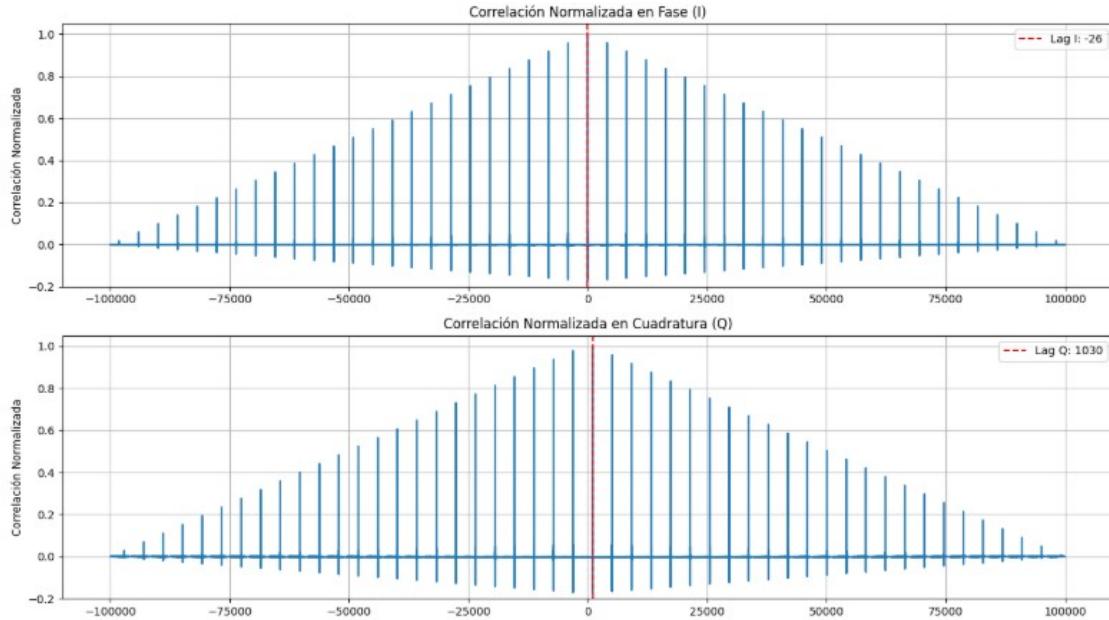


Figura 7: Resultados de correlación normalizada para verificación por vector matching. (a) Correlación en fase, (b) Correlación en cuadratura, (c) Correlación combinada normalizada. Nota: para las PRBS de transmisión de los canales de fase y cuadratura se utilizaron las mismas semillas generadoras .

## 8. Análisis de Timing y Camino Crítico

El análisis del reporte de tiempos que se observa en la implementación del filtro transmisor variable con modulación SRRC muestra que el diseño cumple satisfactoriamente con todas las restricciones temporales especificadas para operación a 100 MHz. Los parámetros críticos obtenidos revelan un Worst Negative Slack (WNS) de 2,390 ns, un Worst Hold Slack (WHS) de 0,162 ns y un Worst Pulse Width Slack (WPWS) de 4,500 ns, todos valores positivos que confirman la ausencia de violaciones de tiempo en las categorías de setup, hold y ancho de pulso respectivamente.

La frecuencia máxima alcanzable por el diseño se determina a partir del camino crítico identificado en el análisis temporal. Considerando que el período actual de reloj es de 10,000 ns (correspondiente a 100 MHz) y el slack de setup más restrictivo es de 2,390 ns, el período mínimo efectivo del sistema resulta de la diferencia entre el período actual y el slack disponible. Por tanto, el período mínimo se calcula como:

$$T_{min} = T_{actual} - WNS = 10,000 \text{ ns} - 2,390 \text{ ns} = 7,610 \text{ ns}$$

La frecuencia máxima alcanzable se determina mediante la ecuación:

$$F_{max} = \frac{1}{T_{min}} = \frac{1}{7,610 \times 10^{-9}} \approx 131,4 \text{ MHz}$$

Este resultado indica que la implementación actual del filtro SRRC posee un margen de frecuencia significativo, siendo capaz de operar aproximadamente 31,4 MHz por encima de la frecuencia objetivo de 100 MHz. Esta reserva "de performance temporal proporciona robustez ante variaciones de proceso, tensión y temperatura, así como flexibilidad para futuras optimizaciones o incrementos en los requerimientos de throughput del sistema.

## 8.1. Caracterización del Camino Crítico

El camino crítico identificado en el análisis temporal se origina en el registro `phase_counter_reg[1]` ubicado en el módulo `u_tx_srrc_polyphase_i` y concluye en el puerto de entrada B[3] del multiplicador DSP48E1 denominado `mul_results_pipe_reg[3]` dentro del mismo módulo de procesamiento. Esta ruta representa la restricción temporal más severa del diseño completo, estableciendo efectivamente el límite superior de frecuencia de operación del sistema.

La composición del delay total de 3,890 ns en el camino crítico revela características importantes sobre la implementación física del diseño. El componente lógico contribuye con 0,805 ns, representando el 20,69 por ciento del delay total, mientras que el componente de ruteo domina con 3,085 ns, equivalente al 79,31 por ciento del delay total. Esta distribución se debe a que utilizan recursos de DSP. En las Figuras (8,9) se observan el esquemático del camino crítico y, el histograma de timming, respectivamente.

El clock path skew de 0,054 ns entre la fuente y destino del camino crítico es relativamente pequeño, indicando una distribución balanceada del reloj del sistema. La incertidumbre del reloj de 0,035 ns incluye componentes de jitter del sistema y tolerancias de distribución, valores que están dentro de rangos aceptables para la frecuencia de operación especificada.

La eficiencia de la implementación se evidencia también en la utilización optimizada de los recursos DSP48E1, elementos especializados que en esta aplicación operan con márgenes temporales holgados. El período mínimo requerido por estos bloques de 3,884 ns contrasta favorablemente con el período actual de 10,000 ns, confirmando que la limitación temporal no reside en estos elementos críticos sino en las interconexiones entre módulos.

El análisis del timing y camino crítico de la implementación del filtro transmisor variable SRRC confirma una realización exitosa que excede los requerimientos temporales especificados. La capacidad de operación a 131,4 MHz frente al objetivo de 100 MHz establece un margen de seguridad robusto que garantiza operación estable bajo condiciones variables y proporciona flexibilidad para futuras optimizaciones.

## 9. Análisis de Timing y Camino Crítico con VIO e ILA instanciados

El análisis del reporte de tiempos correspondiente al diseño `modulation_transmitter_wrapper` implementado sobre un dispositivo Artix-7 XC7A35T revela que el sistema cumple satis-

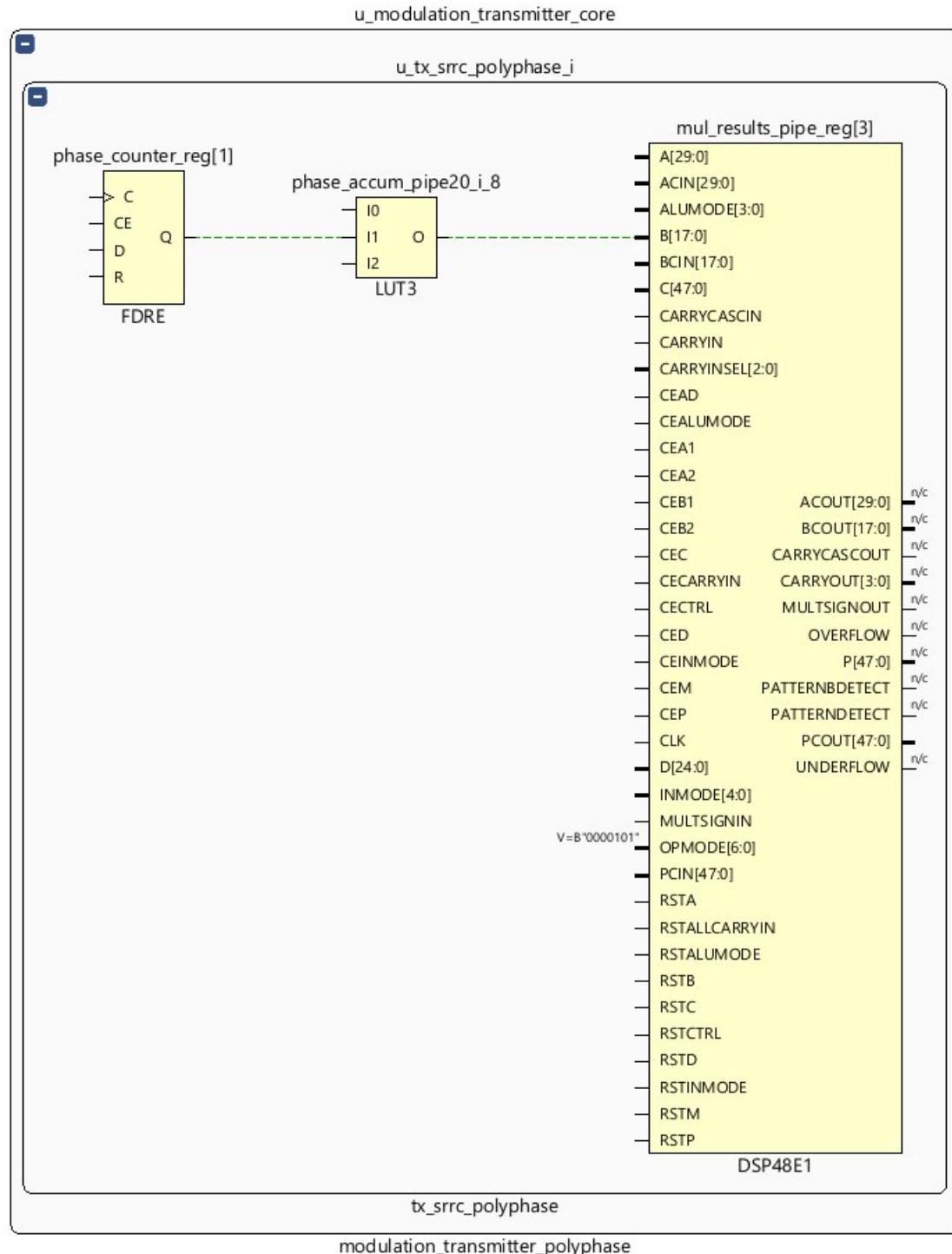


Figura 8: Esquemático del camino crítico del transmisor variable.

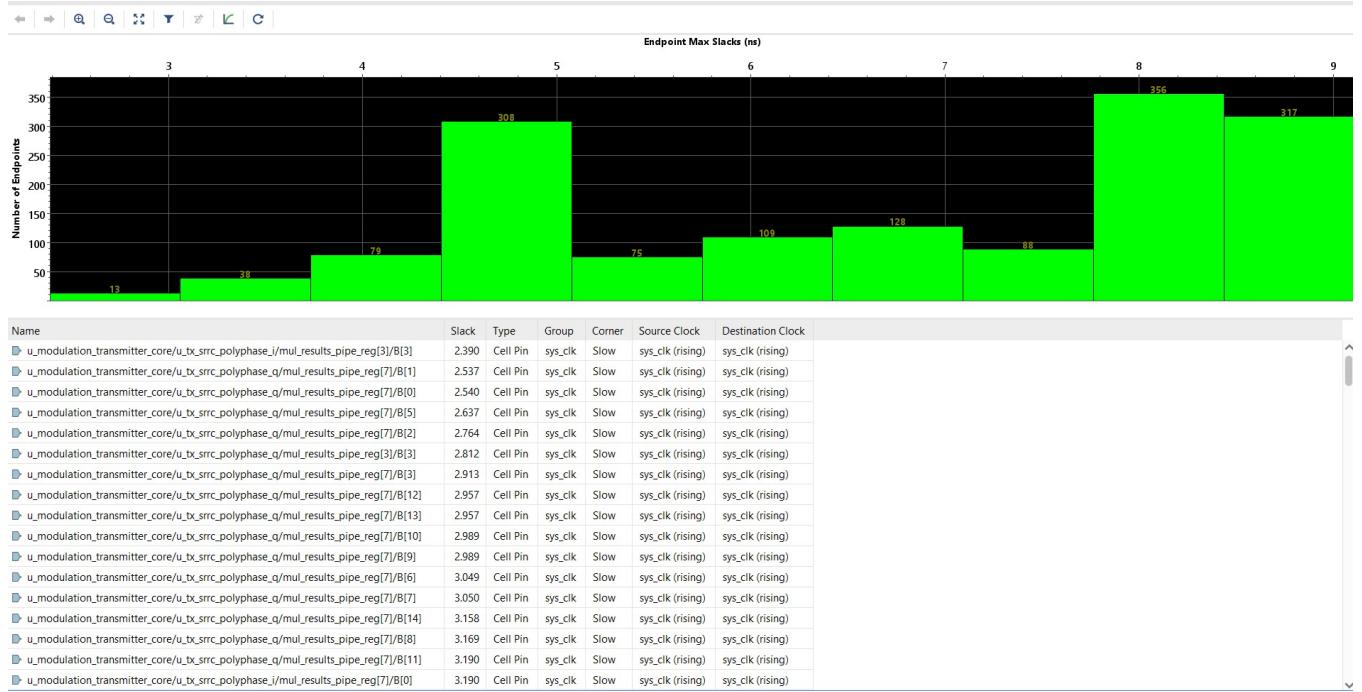


Figura 9: Histograma de Timming del transmisor variable a  $f_c = 100 \text{ MHz}$ .

factorialmente con todas las restricciones temporales impuestas para operación a 100 MHz. El informe sintetizado presenta un Worst Negative Slack (WNS) de 3,130 ns, un Worst Hold Slack (WHS) de 0,061 ns y un Worst Pulse Width Slack (WPWS) de 3,750 ns, todos ellos con valores positivos, lo que confirma la ausencia de violaciones temporales en las categorías de *setup*, *hold* y *pulse width*, respectivamente.

A partir de la información del slack más restrictivo (WNS), puede deducirse la frecuencia máxima teórica alcanzable por el sistema. Dado que el período de reloj actual es de 10,000 ns (equivalente a 100 MHz), el período mínimo efectivo se determina como:

$$T_{\min} = T_{\text{actual}} - \text{WNS} = 10,000 \text{ ns} - 3,130 \text{ ns} = 6,870 \text{ ns}$$

Lo cual implica una frecuencia máxima teórica de operación de:

$$F_{\max} = \frac{1}{T_{\min}} = \frac{1}{6,870 \times 10^{-9}} \approx 145,5 \text{ MHz}$$

Este resultado establece que el diseño posee una reserva temporal del orden de 45,5 MHz por encima de la frecuencia objetivo, proporcionando un margen de robustez ante variaciones de proceso, temperatura y tensión, además de flexibilidad para posibles futuras optimizaciones o incrementos de throughput.

## 9.1. Caracterización del Camino Crítico

El camino crítico identificado en el análisis temporal se origina en el flip-flop `state_reg[2]` y finaliza en el flip-flop `portno_temp_reg[2]`, ambos ubicados dentro del módulo de escaneo boundary `bscan_switch`. Este camino se encuentra temporizado por el reloj TCK, asociado al bloque de depuración JTAG y no forma parte del dominio de reloj principal del sistema (`sys_clk_pin`), sino de lógica auxiliar de depuración.

La demora total del camino crítico asciende a 6,949 ns, desglosada en un componente lógico de 2,129 ns (30,64 %) y un componente de ruteo de 4,820 ns (69,36 %). Esta proporción evidencia que la mayor parte del retardo se debe al enrutamiento de señales a través del tejido programable del dispositivo, y no a la complejidad lógica de la función implementada, que solo involucra seis niveles lógicos (incluyendo elementos tipo LUT3, LUT4, LUT5, LUT6 y bloques CARRY4).

Cabe destacar que el camino crítico analizado se encuentra dentro de un dominio de reloj auxiliar (TCK), por lo que la lógica principal del diseño (asociada al reloj `sys_clk_pin`) presenta incluso un slack superior de 3,130 ns, reforzando la robustez del diseño frente a exigencias temporales. En la Figura 10 se puede observar el histograma de *timming* respectivo a ésta implementación.

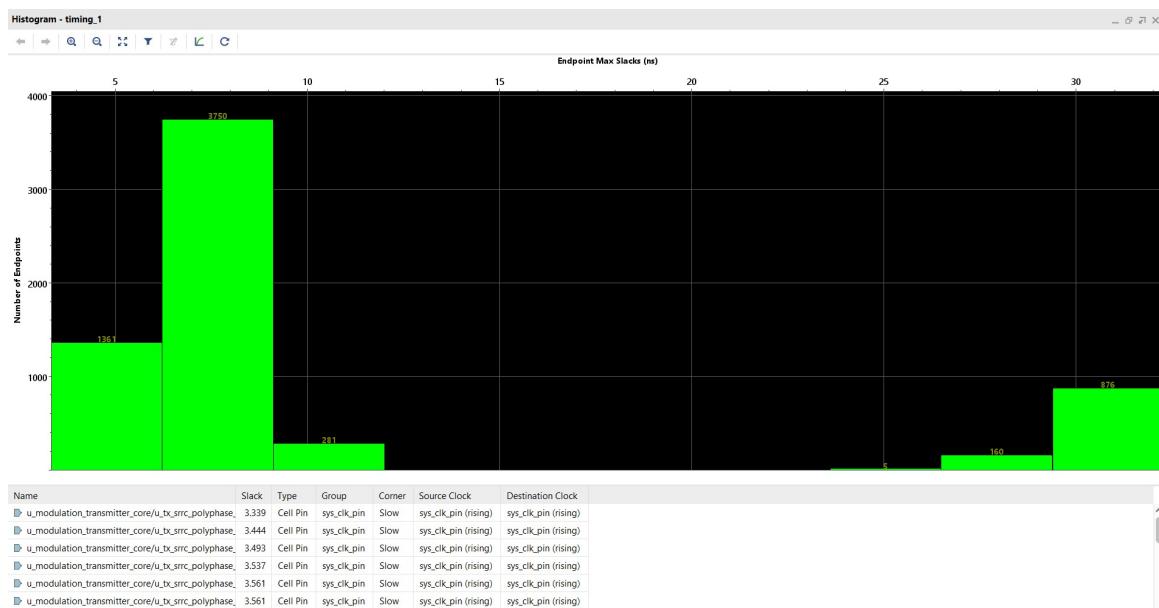


Figura 10: Histograma de *timming* correspondiente a la implementación con VIO e ILA instanciados.

En conclusión, el análisis de timing del diseño `modulation_transmitter_wrapper` confirma que la implementación es temporalmente robusta, con márgenes significativos tanto en la lógica principal como en dominios auxiliares. La posibilidad de operación a frecuencias superiores a los 145 MHz establece una base sólida para futuras expansiones del diseño y garantiza su funcionamiento confiable bajo condiciones variables.

## 10. Reporte de utilización sin VIO e ILA

El diseño `modulation_transmitter_wrapper` ha sido exitosamente sintetizado e implementado en un dispositivo Artix-7 XC7A35T. La utilización general de recursos es muy eficiente, con un uso menor al 1 % en la mayoría de los recursos lógicos, lo que indica un diseño compacto que deja amplio margen para futuras expansiones.

### 10.1. Lógica de Slice

| Tipo de Recurso  | Usado | Disponible | Util. % |
|------------------|-------|------------|---------|
| Slice LUTs       | 142   | 20,800     | 0.68 %  |
| LUT como Lógica  | 142   | 20,800     | 0.68 %  |
| LUT como Memoria | 0     | 9,600      | 0.00 %  |
| Slice Registers  | 181   | 41,600     | 0.44 %  |
| Flip Flops       | 181   | 41,600     | 0.44 %  |
| Latches          | 0     | 41,600     | 0.00 %  |

**Análisis:** La utilización de recursos lógicos es extremadamente baja, indicando un diseño muy eficiente en términos de área.

### 10.2. Procesadores de Señal Digital (DSP)

| Recurso | Usado | Disponible | Util. % | Estado   |
|---------|-------|------------|---------|----------|
| DSP48E1 | 19    | 90         | 21.11 % | Moderado |

El uso de DSPs es el más significativo del diseño, representando más del 20 % de los recursos disponibles.

**Análisis:** La utilización de pines I/O es moderada, sugiriendo que el diseño tiene una interfaz externa bien definida pero no excesivamente compleja.

### 10.3. Recursos de Memoria y Clock

| Recurso        | Usado | Disponible | Util. % | Estado       |
|----------------|-------|------------|---------|--------------|
| Block RAM Tile | 0     | 50         | 0.00 %  | No utilizado |
| BUFGCTRL       | 1     | 32         | 3.13 %  | Mínimo       |

**Análisis:** El diseño no utiliza memoria en bloque, sugiriendo que los requerimientos de almacenamiento son mínimos o se satisfacen con memoria distribuida. Se utiliza un solo buffer de clock global, indicando un diseño de clock simple.

#### 10.3.1. Distribución de Primitivas

Las primitivas más utilizadas en el diseño son:

| Primitiva                   | Cantidad |
|-----------------------------|----------|
| FDRE (Flip-flops con reset) | 173      |
| LUT3 (LUTs de 3 entradas)   | 107      |
| LUT6 (LUTs de 6 entradas)   | 56       |
| OBUF (Buffers de salida)    | 32       |
| LUT2 (LUTs de 2 entradas)   | 22       |
| DSP48E1 (Bloques DSP)       | 19       |

## 11. Reporte de Utilización con VIO e ILA instanciados

El diseño implementado incluye módulos de debug (VIO e ILA) que incrementan la utilización de recursos comparado con el diseño base.

### 11.1. Lógica de Slice

| Tipo de Recurso  | Usado | Disponible | Util. % |
|------------------|-------|------------|---------|
| Slice LUTs       | 1,435 | 20,800     | 6.90 %  |
| LUT como Lógica  | 1,322 | 20,800     | 6.36 %  |
| LUT como Memoria | 113   | 9,600      | 1.18 %  |
| Slice Registers  | 2,516 | 41,600     | 6.05 %  |
| Flip Flops       | 2,516 | 41,600     | 6.05 %  |
| Latches          | 0     | 41,600     | 0.00 %  |

**Análisis:** El incremento en la utilización respecto al diseño base es notable, pasando de 0.68 % a 6.90 % en LUTs y de 0.44 % a 6.05 % en registros. Este aumento es principalmente atribuible a los módulos de debug instanciados.

#### Distribución de Registros por Tipo de Control:

| Tipo de Control       | Cantidad |
|-----------------------|----------|
| Reset síncrono (FDRE) | 2,256    |
| Reset asíncrono       | 192      |
| Set asíncrono         | 43       |
| Set síncrono          | 25       |

### 11.2. Recursos de Memoria

| Recurso        | Usado | Disponible | Util. % |
|----------------|-------|------------|---------|
| Block RAM Tile | 1     | 50         | 2.00 %  |
| RAMB36E1       | 1     | 50         | 2.00 %  |

**Análisis:** Se observa la utilización de una Block RAM (36 Kb), probablemente asociada con el buffer de datos del módulo ILA para el almacenamiento de muestras capturadas.

### 11.3. Procesadores de Señal Digital (DSP)

| Recurso | Usado | Disponible | Util. % |
|---------|-------|------------|---------|
| DSP48E1 | 19    | 90         | 21.11 % |

**Análisis Crítico:** Este es el recurso con mayor utilización, que respecto del caso anterior se mantiene constante. Esto confirma que los DSPs pertenecen a la lógica funcional principal y no a los módulos de debug.

### 11.4. Recursos de Clock y Debug

| Recurso  | Usado | Disponible | Util. % |
|----------|-------|------------|---------|
| BUFGCTRL | 2     | 32         | 6.25 %  |
| BSCANE2  | 1     | 4          | 25.00 % |

**Análisis:** El incremento en buffers de clock (de 1 a 2) y la utilización de BSCANE2 confirman la presencia activa de la infraestructura de debug JTAG.

Luego, las primitivas más utilizadas en el diseño son:

| Primitiva                            | Cantidad |
|--------------------------------------|----------|
| FDRE (Flip-flops con reset y enable) | 2,256    |
| LUT6 (LUTs de 6 entradas)            | 558      |
| LUT3 (LUTs de 3 entradas)            | 417      |
| LUT4 (LUTs de 4 entradas)            | 233      |
| DSP48E1 (Bloques DSP)                | 19       |

## 11.5. Impacto de los Módulos de Debug

| Recurso   | Diseño Base  | Con Debug      | Incremento |
|-----------|--------------|----------------|------------|
| LUTs      | 142 (0.68 %) | 1,435 (6.90 %) | +1,293     |
| Registros | 181 (0.44 %) | 2,516 (6.05 %) | +2,335     |
| Block RAM | 0 (0 %)      | 1 (2 %)        | +1         |
| Pines I/O | 38 (18 %)    | 1 (0.48 %)     | -37        |
| DSPs      | 19 (21 %)    | 19 (21 %)      | 0          |

Al comparar ambos diseños, se observa que la inclusión de los módulos de debug (VIO e ILA) genera un aumento considerable en la utilización de recursos. Las LUTs pasan de 0.68 % a 6.90 % y los registros de 0.44 % a 6.05 %, lo cual refleja una diferencia importante provocada por la instrumentación agregada para facilitar el monitoreo y control del sistema.

También se utiliza una memoria Block RAM en el diseño con debug, que no está presente en el diseño base. En contraste, el uso de DSPs no cambia, lo que indica que estos bloques son parte de la lógica principal del transmisor y no están relacionados con los módulos de debug. Por último, se reduce el uso de pines de entrada/salida, lo cual sugiere que el diseño con debug está pensado para funcionar principalmente dentro del FPGA, sin requerir una interfaz externa compleja.

En resumen, los módulos de debug incrementan el uso de recursos, pero permiten una mejor visualización y control del sistema durante la etapa de pruebas.

## 12. Implementación y control mediante el uso de las herramientas VIO e ILA

Para la depuración y validación del diseño en hardware, se integraron los núcleos de depuración *VIO* (Virtual Input/Output) e *ILA* (Integrated Logic Analyzer).

El *VIO* permite la “inyección” y el monitoreo de señales internas en tiempo real. En este diseño, el *VIO* se utiliza para:

- Controlar el tipo de modulación mediante la señal `sel_mod_type`.
- Habilitar el generador *PRBS* mediante la señal `in_enable_prbs`.
- Activar el filtrado *SRRRC* a través de `in_enable_srrc`.
- Gestionar la señal de reinicio global `in_reset`.

- Monitorizar las salidas moduladas `tx_out_i` y `tx_out_q`.

Por otro lado, el núcleo *ILA* permite la captura y visualización de formas de onda internas, facilitando el análisis temporal de las señales moduladas `tx_out_i` y `tx_out_q` durante la operación del transmisor. Este análisis es fundamental para evaluar la respuesta del sistema en condiciones reales y verificar la correcta implementación de las etapas de modulación.

En las Figuras (11,12 ,13) se muestran la salida del transmisor para el caso de una modulación QPSK, QAM16 y QAM32, respectivamente. Estas figuras permiten observar la correcta generación de las formas de onda en cuadratura para cada esquema de modulación.

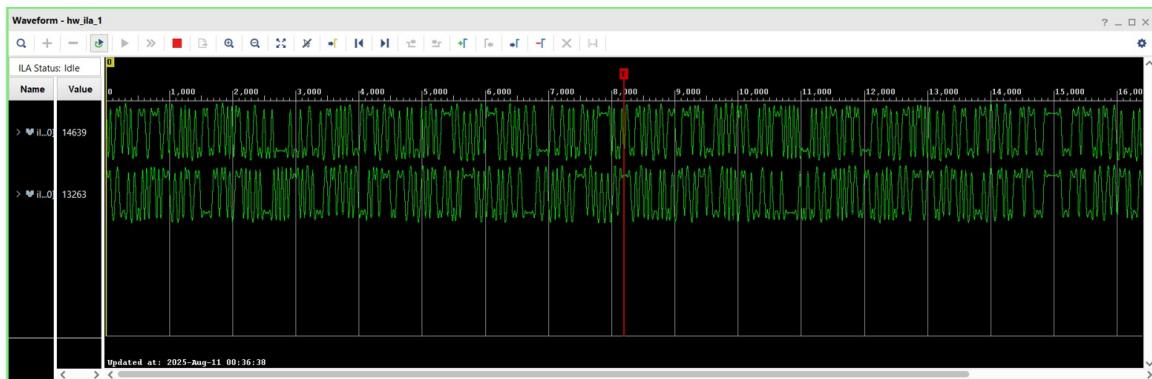


Figura 11: Salida del transmisor para modulación QPSK.

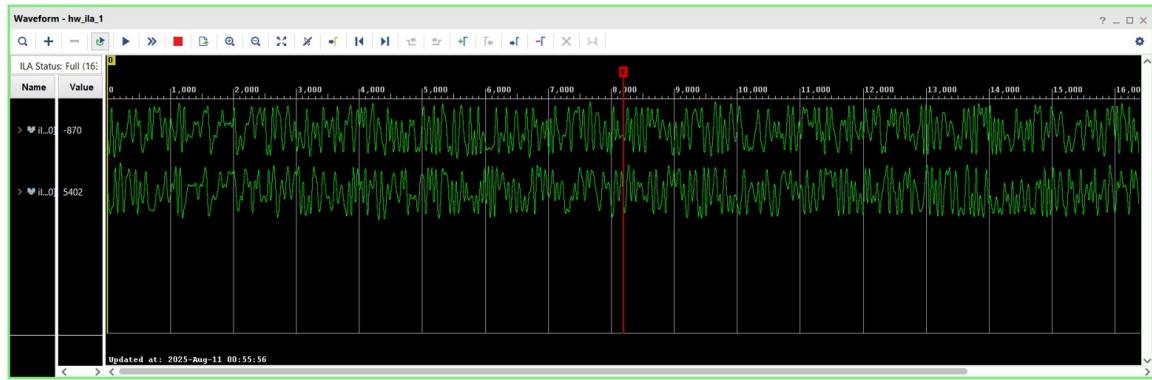


Figura 12: Salida del transmisor para modulación 16-QAM.

El *trigger setup* del *ILA* se configuró con la condición:

$$\text{probe0\_0} \neq 0 \quad \text{y} \quad \text{probe1\_0} \neq 0$$

Esto garantiza que la captura se realice únicamente cuando ambas componentes de la señal modulada (`tx_out_i` y `tx_out_q`) estén activas, evitando capturas en períodos de inactividad o valores nulos (por ejemplo, durante una etapa de reset o inicialización).

La instanciación de los núcleos *VIO* e *ILA* se muestra a continuación:

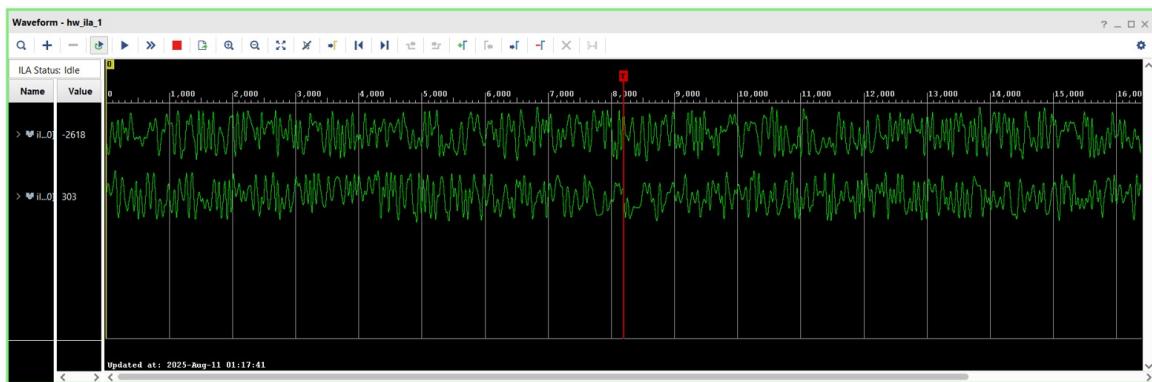


Figura 13: Salida del transmisor para modulación 32-QAM.

```

1      // Instancia del VIO
2      vio vio_0
3      (
4          .clk_0(clock),
5          .probe_in0_0(tx_out_i),
6          .probe_in1_0(tx_out_q),
7          .probe_out0_0(sel_mod_type),
8          .probe_out1_0(in_enable_prbs),
9          .probe_out2_0(in_enable_srrc),
10         .probe_out3_0(in_reset)
11     );
12
13     // Instancia del ILA
14     ila ila_0 (
15         .clk_0(clock),           // Reloj de
16         .muestreo,
17         .probe0_0(tx_out_i),    // Señal a
18         .probe1_0(tx_out_q)     // Señal a
19     );

```

Listing 6: Instanciación de los núcleos VIO e ILA

### 12.1. Constelaciones con ILA a 4096 Muestras.

Se configuró el núcleo Integrated Logic Analyzer (ILA) en Vivado con una profundidad de captura (Capture Depth) de 4096 muestras. Esta configuración permitió capturar más símbolos por adquisición, mejorando la visualización de las constelaciones.

Se obtuvieron las siguientes constelaciones experimentales:

1. **QPSK**: Cuatro puntos, correspondientes a las fases  $0^\circ$ ,  $90^\circ$ ,  $180^\circ$  y  $270^\circ$  (Figura 14).
2. **16-QAM**: Conjunto de 16 puntos distribuidos en una cuadrícula  $4 \times 4$ , representando combinaciones de amplitud y fase (Figura 15).
3. **32-QAM**: Conjunto de 32 puntos distribuidos de acuerdo a la modulación QAM de 32 símbolos, con múltiples niveles de amplitud (Figura 16).

## 13. Conclusión

El desarrollo e implementación del transmisor digital con modulación variable y filtrado SRRC polifásico ha permitido mostrar la potencial viabilidad de un diseño eficiente y escalable para aplicaciones en FPGA. Respecto de este último punto, cabe destacar que este diseño dada su naturaleza parametrizable es escalable/*customizable* tanto en cantidad de coeficientes, como en el número de bits enteros y fraccionales. La arquitectura propuesta logra un potencial equilibrio entre flexibilidad y optimización de recursos, incorporando soporte para múltiples esquemas de modulación (QPSK, 16-QAM y 32-QAM) sin comprometer el rendimiento ni la calidad de la señal transmitida.

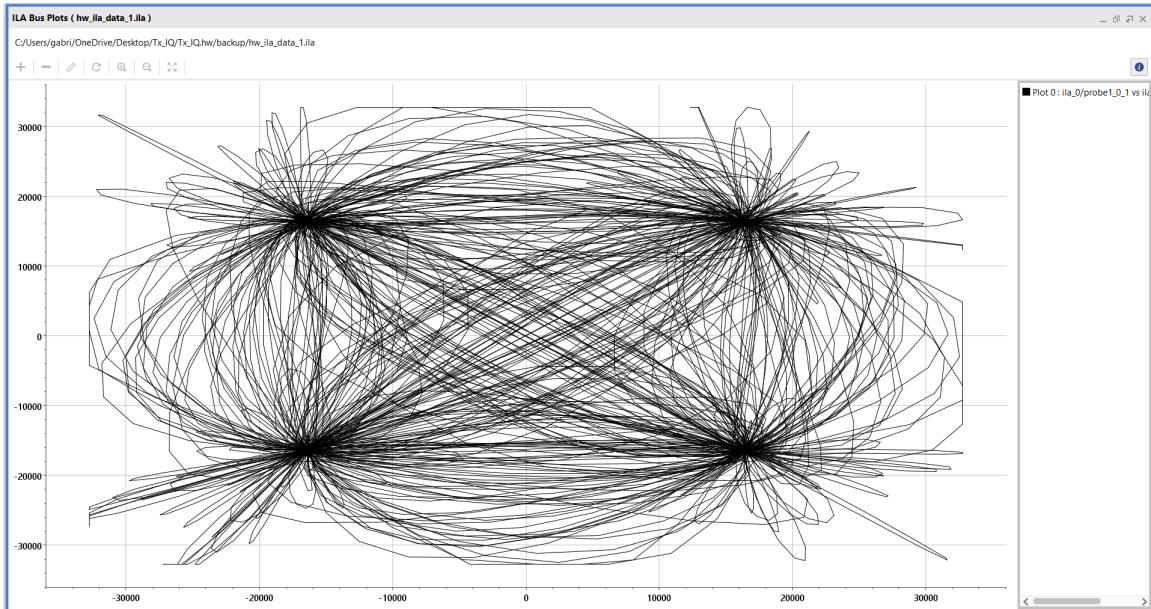


Figura 14: Constelación QPSK capturada con ILA configurado a 4096 muestras.

El uso de un filtro SRRC polifásico de 8 fases permitió una reducción significativa en el número de multiplicadores (87.5 % menos en comparación con una implementación directa), manteniendo la conformación de pulsos y el control de la interferencia intersimbólica. Asimismo, las simulaciones en punto flotante y punto fijo confirmaron la correcta funcionalidad del sistema.

Los resultados de temporización evidencian un margen holgado lo que garantiza robustez frente a variaciones de proceso y posibles pequeñas escalas de frecuencia en futuros desarrollos. Por otro lado, la baja utilización de LUTs, registros y slices indica que el diseño es apto para dispositivos de menor capacidad o para futuras expansiones, como la incorporación de modulaciones adicionales o filtros con mayor número de taps.

Finalmente, las herramientas de depuración VIO e ILA conforman una etapa en la que se logra observar una correcta generación y transmisión de las señales moduladas y el cumplimiento de las especificaciones teóricas y funcionales. En conjunto, el proyecto forma parte de una etapa de aprendizaje que muestra la practicidad de la arquitectura polifásica como una solución eficiente.

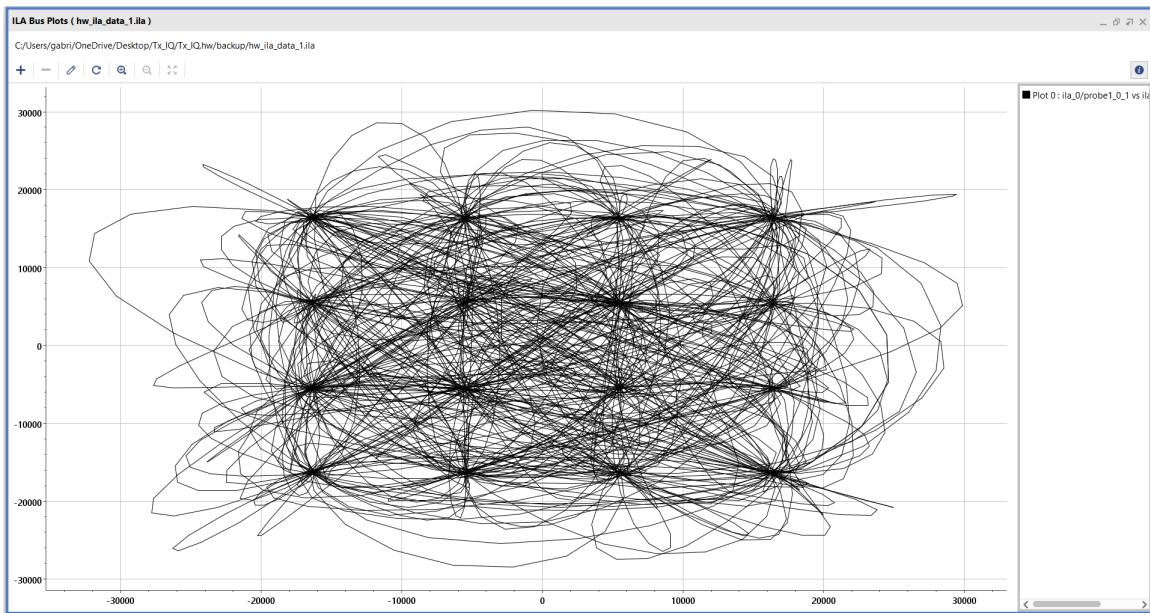


Figura 15: Constelación 16-QAM capturada con ILA configurado a 4096 muestras.

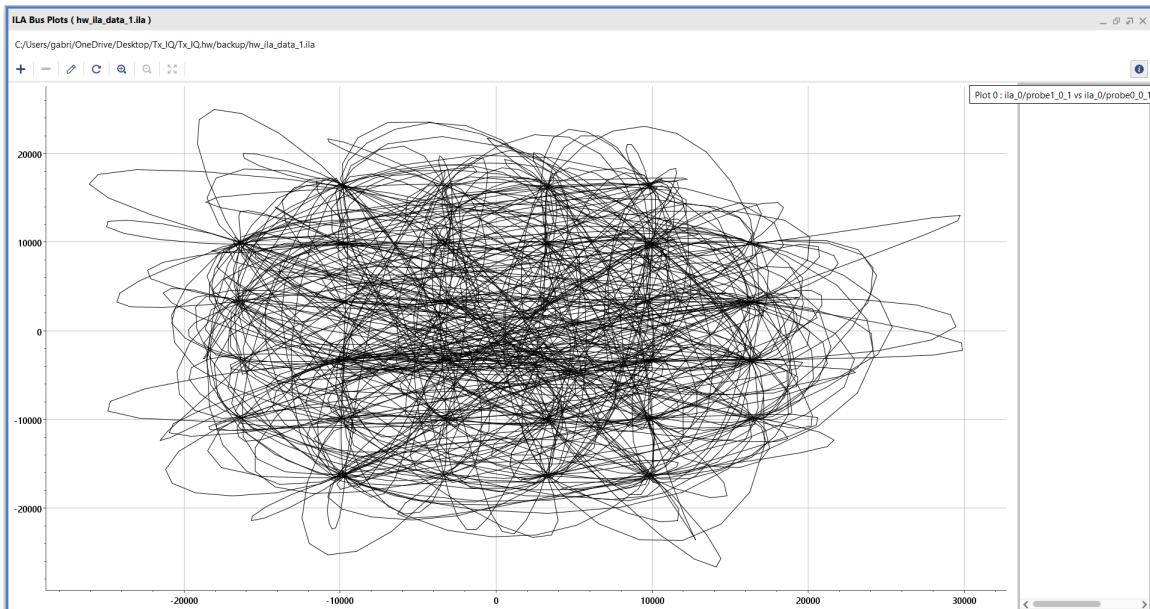


Figura 16: Constelación 32-QAM capturada con ILA configurado a 4096 muestras.

## Referencias

- [1] J. G. Proakis and M. Salehi, *Digital Communications*, 5th ed. New York: McGraw-Hill, 2008.
- [2] B. Sklar, *Digital Communications: Fundamentals and Applications*, 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2001.
- [3] Xilinx Inc., “LogiCORE IP Digital Upconverter v6.0,” Product Guide PG016, 2020.
- [4] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [5] F. J. Harris, *Multirate Signal Processing for Communication Systems*. Upper Saddle River, NJ: Prentice Hall, 2004.
- [6] M. Meyer-Baese, *Digital Signal Processing with Field Programmable Gate Arrays*, 3rd ed. Berlin: Springer, 2007.